Algoritmi & Strutture dati

Gabriele Frassi

A.A 2019-2020 - Secondo semestre

Dispensa inizialmente pubblicata in copisteria. Ho deciso di caricarla nella repository visto l'addio della professoressa De Francesco. Usatela con cura, tenendo a mente che il docente è cambiato.

Indice

Capitolo 1

Unimap

1. **Mar 03/03/2020 10:30-13:30 (3:0 h) lezione**: Introduzione al corso. Nozione di algoritmo. Algoritmo di Euclide per MCD, crivello di Eratostene. Definizione di complessità computazionale. Notazione O-grande. Classi di comlessità.
(NICOLETTA DE FRANCESCO)
2. **Gio 05/03/2020 13:30-15:30 (2:0 h) non tenuta**: lezione di laboratorio non tenuta per il blocco della didattica dovuto al coronavirus (Decreto Rettorale n. 476 del 5 Marzo 2020)
(ANTONIO LUCA ALFEO,NICOLETTA DE FRANCESCO)
3. **Ven 06/03/2020 11:30-13:30 (2:0 h) non tenuta**: lezione non tenuta per il blocco della didattica (Decreto Rettorale n. 476 del 5 Marzo 2020)
(NICOLETTA DE FRANCESCO)
4. **Mar 10/03/2020 10:30-13:30 (3:0 h) lezione**: Notazioni Omega grande e Theta Grande. Complessità dei programmi iterativi. (lezione tenuta in modalità telematica)
(NICOLETTA DE FRANCESCO)
5. **Gio 12/03/2020 13:30-15:30 (2:0 h) laboratorio**: Presentazione prova pratica d'esame, strutture dati basilari e modalita' di debugging. Nello specifico: • Debug Manuale • Gestione Dinamica Input e Liste • Debug Assistito • Soluzioni della Standard Template Library • Gestione Stringhe e Vectors (lezione tenuta in modalità telematica - primo gruppo di studenti)
(ANTONIO LUCA ALFEO,NICOLETTA DE FRANCESCO)
6. **Ven 13/03/2020 11:30-13:30 (2:0 h) lezione**: Programmazione ricorsiva: principi e regole. Programmi ricorsivi su liste. Principio di induzione naturale. Esercizi sul calcolo della complessità dei programmi iterativi. (lezione tenuta in modalità telematica)
(NICOLETTA DE FRANCESCO)
7. **Mar 17/03/2020 10:30-13:30 (3:0 h) lezione**: complessità dei programmi ricorsivi. Relazioni di ricorrenza divide et impera e lineari. Ricerca binaria. Quicksort. Torre di Hanoi. Esercizi su algoritmi ricorsivi. (lezione tenuta in modalità telematica)
(NICOLETTA DE FRANCESCO)
8. **Gio 19/03/2020 13:30-15:30 (2:0 h) laboratorio**: Presentazione prova pratica d'esame, strutture dati basilari e modalita' di debugging. Nello specifico: • Debug Manuale • Gestione Dinamica Input e Liste • Debug Assistito • Soluzioni della Standard Template Library • Gestione Stringhe e Vectors (lezione tenuta in modalità telematica - secondo gruppo di studenti)
(ANTONIO LUCA ALFEO,NICOLETTA DE FRANCESCO)
9. **Mar 24/03/2020 10:30-13:30 (3:0 h) lezione**: Numeri di Fibonacci. Algoritmo di ordinamento mergesort. Mergesort su liste semplici. Esercizi di calcolo della complessità dei programmi. (lezione tenuta in modalità telematica)
(NICOLETTA DE FRANCESCO)
10. **Gio 26/03/2020 13:30-15:30 (2:0 h) laboratorio**: Presentazione prova pratica d'esame, strutture dati basilari e modalita' di debugging. Nello specifico: • Debug Manuale • Gestione Dinamica Input e Liste • Debug Assistito • Soluzioni della Standard Template Library • Gestione Stringhe e Vectors (lezione tenuta

in modalità telematica - terzo gruppo di studenti)
(ANTONIO LUCA ALFEO,NICOLETTA DE FRANCESCO)

11. **Ven 27/03/2020 11:30-13:30 (2:0 h) lezione**: Alberi binari. Definizione. Memorizzazione. Visite. Alcuni programmi. (lezione tenuta in modalità telematica)
(NICOLETTA DE FRANCESCO)

12. **Mar 31/03/2020 10:30-13:30 (3:0 h) lezione**: Alberi binari di ricerca. Limiti inferiori mediante alberi di decisione. Counting sort. Radix sort. Esercizi su alberi binari. (lezione tenuta in modalità telematica)
(NICOLETTA DE FRANCESCO)

13. **Gio 02/04/2020 13:30-15:30 (2:0 h) laboratorio**: - Soluzione esercizi volta precedente e relativa complessità - Insertion Sort - Merge Sort - Altri Esercizi (lezione tenuta in modalità telematica - primo gruppo di studenti)
(ANTONIO LUCA ALFEO,NICOLETTA DE FRANCESCO)

14. **Mar 07/04/2020 10:30-13:30 (3:0 h) lezione**: Tipo di dato heap. Algoritmo di ordinamento heapsort. Alberi generici: definizione, vistite, memorizzazione, esrcizi. (lezione tenuta in modalità telematica)
(NICOLETTA DE FRANCESCO)

15. **Gio 16/04/2020 13:30-15:30 (2:0 h) laboratorio**: - Soluzione esercizi volta precedente e relativa complessità - Insertion Sort - Merge Sort - Altri Esercizi (lezione tenuta in modalità telematica - secondo e terzo gruppo di studenti)
(ANTONIO LUCA ALFEO,NICOLETTA DE FRANCESCO)

16. **Ven 17/04/2020 11:30-13:30 (2:0 h) lezione**: esercizi su alberi binari, alberi generici e heap. (lezione tenuta in modalità telematica)
(NICOLETTA DE FRANCESCO)

17. **Mar 21/04/2020 10:30-13:30 (3:0 h) lezione**: Metodo di ricerca ricerca hash. Algoritmo PLSC per trovare la più lunga sottosequenza comune fra due sequenze. Esercizi. (lezione tenuta in modalità telematica)
(NICOLETTA DE FRANCESCO)

18. **Gio 23/04/2020 13:30-15:30 (2:0 h) laboratorio**: - Alberi Binari di Ricerca - Progettazione e Inizializzazione - Esempi Funzioni con Alberi Binari di Ricerca - Alberi binari con etichette complesse - Esercizi (lezione tenuta in modalità telematica - primo gruppo di studenti)
(ANTONIO LUCA ALFEO,NICOLETTA DE FRANCESCO)

19. **Mar 28/04/2020 10:30-13:30 (3:0 h) lezione**: Metodologia greedy. Algoritmo di compressione di Huffman. Grafi orientati e non orientati: definizioni, memorizzazione, visita in profondità. Esercizi. (lezione tenuta in modalità telematica)
(NICOLETTA DE FRANCESCO)

20. **Gio 30/04/2020 13:30-15:30 (2:0 h) laboratorio**: - Alberi Binari di Ricerca - Progettazione e Inizializzazione - Esempi Funzioni con Alberi Binari di Ricerca - Alberi binari con etichette complesse - Esercizi (lezione tenuta in modalità telematica - secondo gruppo di studenti)
(ANTONIO LUCA ALFEO,NICOLETTA DE FRANCESCO)

21. **Mar 05/05/2020 10:30-13:30 (3:0 h) lezione**: Algoritmo di Kruskal per trovare il minimo albero di copertura. (lezione tenuta in modalità telematica)
(NICOLETTA DE FRANCESCO)

22. **Gio 07/05/2020 13:30-15:30 (2:0 h) laboratorio**: - Alberi Binari di Ricerca - Progettazione e Inizializzazione - Esempi Funzioni con Alberi Binari di Ricerca - Alberi binari con etichette complesse - Esercizi (lezione tenuta in modalità telematica - terzo gruppo di studenti)
(ANTONIO LUCA ALFEO,NICOLETTA DE FRANCESCO)

23. **Ven 08/05/2020 11:30-13:30 (2:0 h) lezione**: Algoritmo di Dijkstra per trovare il cammino minimo da un nodo a tutti gli altri. (lezione tenuta in modalità telematica)
(NICOLETTA DE FRANCESCO)

24. **Mar 12/05/2020 10:30-13:30 (3:0 h) lezione**: Complessità degli algoritmi di teoria dei numeri. Algoritmo della moltiplicazione veloce. Cenni alla NP-completezza: problemi difficili, gli insiemi P e NP, il teorema di Cook, problemi non calcolabili. (lezione tenuta in modalità telematica)
(NICOLETTA DE FRANCESCO)

25. **Gio 14/05/2020 13:30-15:30 (2:0 h) laboratorio**: - Heap - Ordinamento tramite Heap - Hashing - Hashing e tipi di input - Esercizi (lezione tenuta in modalità telematica - primo gruppo di studenti) (ANTONIO LUCA ALFEO,NICOLETTA DE FRANCESCO)
26. **Mar 19/05/2020 10:30-13:30 (3:0 h) lezione**: Complementi di c++: funzioni e classi modello (template), derivazione. (lezione tenuta in modalità telematica) (NICOLETTA DE FRANCESCO)
27. **Gio 21/05/2020 13:30-15:30 (2:0 h) laboratorio**: - Heap - Ordinamento tramite Heap - Hashing - Hashing e tipi di input - Esercizi (lezione tenuta in modalità telematica - secondo gruppo di studenti) (ANTONIO LUCA ALFEO,NICOLETTA DE FRANCESCO)
28. **Ven 22/05/2020 11:30-13:30 (2:0 h) lezione**: complementi di c++: ereditarietà delle classi, funzioni virtuali e classi astratte. (lezione tenuta in modalità telematica) (NICOLETTA DE FRANCESCO)
29. **Mar 26/05/2020 10:30-13:30 (3:0 h) lezione**: Complementi di c++: gestione delle eccezioni. Esercizi. (lezione tenuta in modalità telematica) (NICOLETTA DE FRANCESCO)
30. **Gio 28/05/2020 13:30-15:30 (2:0 h) laboratorio**: - Heap - Ordinamento tramite Heap - Hashing - Hashing e tipi di input - Esercizi (lezione tenuta in modalità telematica - terzo gruppo di studenti) (ANTONIO LUCA ALFEO,NICOLETTA DE FRANCESCO)

Parte I

Lezioni della De Francesco

Capitolo 2

Lunedì 03/03/2020

Registro 1 (*Mar 03/03/2020 10:30-13:30 (3:0 h) lezione*)

Introduzione al corso. Nozione di algoritmo. Algoritmo di Euclide per MCD, crivello di Eratostene. Definizione di complessità computazionale. Notazione O-grande. Classi di comlessità.

2.1 Cosa si fa nel corso?

- Complessità degli algoritmi: degli algoritmi già visti e di quelli che vedremo studieremo la complessità. Valuteremo l'efficienza dei vari algoritmi, li metteremo a confronto e sceglieremo quello più adatto per le nostre esigenze.

  Esempio bubblesort e selectionsort, qual è il migliore? (di certo non ci metteremo a generare n fattoriale combinazioni e a cercare quella ordinata). Studieremo altri algoritmi di ordinamento migliori di quelli già visti.

  Vedremo che esistono alcuni algoritmi (NP-Complesso) in cui non è possibile fare i ragionamenti appena citati.

- Nuove strutture dati. Fino ad ora abbiamo studiato solo strutture dati lineari, in cui gli elementi sono posti in fila. Introdurremo gli alberi, esempio per eccellenza di struttura dati non lineare (ogni elemento non ha per forza un solo successore come, per esempio, gli elementi di una lista). Essi risulteranno per descrivere strutture gerarchiche: in un'azienda, nella nostra università... Analizzeremo metodi per la manipolazione di questi alberi.

- Approfondiremo il concetto di ricorsività: essa risulterà utile, per esempio, nella creazione di funzioni per manipolare gli alberi. Le corrispondenti funzioni iterative non sono così semplici.

- Introdurremo i grafi, caratterizzati da nodi e archi. Analizzeremo degli algoritmi per manipolare questi grafi.

- Svilupperemo delle metodologie per affrontare problemi (metodo *divide et impera*, divisione del problema in sottoproblemi per poi ritornare al problema di partenza; ricerca delle combinazioni...).

- Le classi derivate, utili nel caso in cui si abbiano campi e strutture dati molto simili (Esempio: persone, studenti, docenti, borsisti...). Analizzeremo le regole di visibilità già introdotte applicandole alle classi.

- classi e funzioni modello.

2.2 Nozione di algoritmo

La formalizzazione del concetto di algoritmo risale al Novecento ma è un qualcosa che ha origini lontane. Nelle slide viene citato un arabo che già nell'800 scrisse dei libri, in particolare introdusse la notazione posizionale dei numeri e il concetto di 0.
Personaggio degno di nota è Alan Turing: la macchina omonima è stata la base per la creazione dei primi computer.

Concetto Un algoritmo è un insieme finito di istruzioni teso a risolvere un problema. Si ha un input e un output, le proprietà sono quelle viste anche a FdP. Si ha la possibilità di memorizzare risultati intermedi.

Algoritmi aritmetici I primi algoritmi sono quelli aritmetici, inventati da babilonesi, egizi e greci. Nelle slide sono introdotti alcuni algoritmi:

- quello *di Euclide*[1], che permette di individuare il MCD di due numeri. Di questo individuiamo una versione alternativa, non di Euclide, in cui si svolgono delle divisioni.
- il *setaccio di Eratostene*, un algoritmo che mi permette di individuare tutti i numeri primi fino a un dato numero n. Normalmente potrei pensare di dividere ogni numero per i suoi predecessori: ragionamento giusto, ma estremamente lungo soprattutto con numeri elevati. Col setaccio di Eratostene individuiamo un procedimento molto più semplice! La cybersecurity è strettamente collegata agli algoritmi sui numeri primi.

Nozione di algoritmo ripresa dagli appunti di *Fondamenti di programmazione*

*Alla base dell'informatica abbiamo il concetto di algoritmo: una **sequenza precisa** (non ambigua) **e finita di operazioni**, comprensibili e perciò eseguibili da uno strumento informatico, che portano alla realizzazione di un compito).*
Queste operazioni possono essere del seguente tipo:

- ***Sequenziali**: viene svolta una singola azione. Dopo questa si passa all'operazione successiva*
- ***Condizionali**: si controlla una condizione e si sceglie quale sarà l'operazione successiva in base al valore della condizione*
- ***Iterative**: un blocco di operazioni può essere eseguito ripetutamente finchè non si verifica una determinata condizione*

***Il compito dell'esperto informatico** è quello di individuare e codificare sottoforma di un programma l'algoritmo giusto per risolvere un determinato problema. Risolvere un problema significa produrre dei risultati a partire da dati in ingresso.*
L'algoritmo dovrà essere applicabile a qualunque dato appartenente al dominio di definizione dell'algoritmo e ai sottoinsiemi di questo: se l'algoritmo si applica ai numeri interi esso dovrà funzionare sia con interi positivi che con interi negativi!

Proprietà dell'algoritmo *L'algoritmo presenta le seguenti proprietà:*

- ***Correttezza**: un algoritmo è corretto se privo di difetti in ogni passo fondamentale (non posso ignorare casi particolari all'interno di un problema, l'algoritmo deve occuparsi anche di loro)*
- ***Efficienza**: si ottiene la soluzione del problema nel modo più veloce possibile (tenendo conto della correttezza)*
- ***Eseguibilità**: ogni azione è eseguita in un tempo finito*

[1] *Euclide non l'ha scritto in C++, io sì!*

- ***Non-ambiguità**: ogni azione è interpretata in maniera univoca dall'esecutore*
- ***Finitezza**: il numero totale di azioni da eseguire è finito*

2.2.1 Algoritmo di Euclide

L'algoritmo di Euclide è uno dei primi algoritmi della storia: risale al 300 a.C! Esso ci permette di trovare il massimo comun divisore fra due numeri interi non negativi.

- Verifico che i numeri non siano negativi, l'algoritmo in quel caso non è applicabile
- Se i due numeri sono uguali abbiamo già il risultato!
- In tutti gli altri casi procedo mediante una serie di sottrazioni: confronto ogni volta i due numeri e sottraggo al numero più grande quello più piccolo
- Continuo finchè i due numeri non saranno uguali: il valore ottenuto è il MCD dei numeri da cui abbiamo iniziato!

```
int MCD_prima_versione(int n1, int n2) {
    while(n1 != n2) {
        if(n1 < n2)
            n2 -= n1;
        else
            n1 -= n2;
    }
    return n1;
}
```

2.2.1.1 Versione con il resto della divisione

Esiste una versione alternativa in cui si coinvolge il resto della divisione. Sappiamo che il massimo comun divisore tra due numeri x ed y è uguale al massimo comun divisore tra y e il resto tra i numeri x ed y:

$$MCD(x, y) = MCD(y, x\%y)$$

- Verifico che i numeri non siano negativi
- Se i numeri sono uguali il corpo del while sarà eseguito una volta sola
- Ogni volta che eseguo il corpo del while pongo $x = y$ e $y = x\%y$ (utilizzando una variabile ausiliaria)
- Ripeto le istruzioni finchè $y \neq 0$. Il valore di x ottenuto è il MCD dei numeri da cui abbiamo iniziato!

```
int MCD_seconda_versione(int x, int y) {
    while(y != 0) {
        int k = x;
        x = y;
        y = k%y;
    }
    return x;
}
```

2.2.2 Setaccio di Eratostene

Contesto Vogliamo trovare tutti i numeri primi fino a un dato numero n. Analizzare ogni singolo numero verificando il resto della divisione tra questo e tutti i numeri precedenti è inefficiente. Vogliamo un algoritmo che ci permetta di ottenere i numeri primi fino ad n con il minor numero di mosse possibile.

Il setaccio di Eratostene è l'algoritmo soluzione:

- Elenco tutti i numeri fino ad n (lo faccio mediante un vettore di booleani)
- Escludo direttamente lo zero e l'uno dalla lista (pongo i primi due booleani *false* e tutti gli altri *true*)
- Rimuovo dalla lista i multipli dei numeri primi iniziali.
  - Parto sempre dal doppio prodotto del numero (i numeri precedenti non cancellati sono primi)
  - Scorro tutti i multipli (sommo, non utilizzo l'operatore resto) e pongo i corrispondenti booleani falsi
  - Continuo finchè non incontro un multiplo maggiore o uguale ad n
- Continuo finchè il doppio prodotto del numero primo analizzato risulta inferiore ad n

```
int setaccio(int n) {
    bool primi[n]; // Definisco un vettore di booleani

    // Pongo il valore degli elementi del vettore
    primi[0] = primi[1] = false;
    for(int i = 2; i < n; i++) primi[i] = true;

    // Scorro i numeri primi facendo quanto detto prima
    for(int i = 1; i*i <n; i++) {
        // Incremento i finche non incontro un numero primo
        while(!primi[i]) i++

        // Scorro i multipli del numero primo i
        for(int k = i*i; k < n; k += i) primi[k] = false;
    }

    // Stampo i numeri primi minori di n
    for(int i = 2; i < n; i++)
        if(primi[i])
            cout<<i<<endl;
}
```

2.3 Complessità computazionale degli algoritmi

Con complessità computazionale si intende una funzione matematica sempre positiva che ha per dominio le dimensioni del problema ($n \geq 0$). Al codominio appartiene il costo della risoluzione del problema. Ciò che appartiene al dominio varia in base agli algoritmi analizzati: è ovvio che parlando di algoritmi di ordinamento la dimensione consiste nel numero di elementi da ordinare.
Il costo, nel 99% dei casi che vedremo, consiste nel tempo (in altre circostanze può trattarsi della memoria occupata). Per confrontare l'efficienza di due algoritmi confronterò le corrispondenti funzioni matematiche di complessità. Vediamo l'esempio semplificato presente nelle slide. Per ottenere il tempo sommerò i

tempi per svolgere le singole istruzioni.

Supponiamo che ogni singola istruzione richieda un secondo e che sia costante (cioè non dipende da un valore n)

- ho l'operazione di assegnamento prima del for.
- ho l'istruzione *return*
- ho il for, un tantino più complesso:
  - ho l'inizializzazione della variabile di controllo
  - ad ogni iterazione ho un confronto. Se il confronto restituisce true eseguo anche l'operazione di assegnamento. Considero il caso peggiore, quindi ogni iterazione richiede due secondi.
  - ho lo step che comporta l'incremento della variabile di controllo
  - ho la verifica della condizione in cui è coinvolta la variabile di controllo.
  - considero che avrò uno step in più che mi porterà ad avere la condizione citata prima falsa.

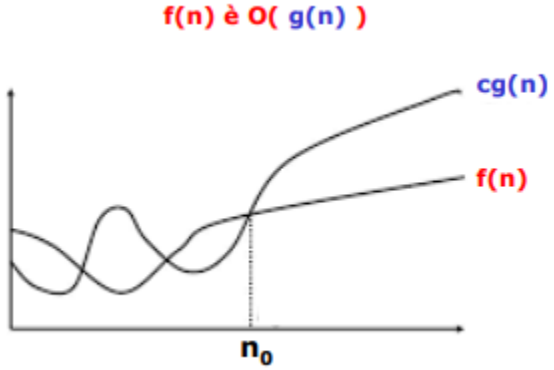
Tenendo conto di tutti questi aspetti ottengo un $T_{max}(n) = 4n$. Analizzare l'efficienza mediante queste formule ci permette di svolgere un'analisi oggettiva, svincolata dal dispositivo utilizzato e dal linguaggio di programmazione adottato.
Metto a confronto due equazioni A e B: devo confrontarle e individuare quando $A \leq B$. Supponiamo di avere due algoritmi: uno è sicuramente più efficiente dell'altro. Si osserva che nel "primo pezzetto" un algoritmo è più efficiente, mentre nel "secondo pezzetto" risulta più efficiente il secondo algoritmo. Segue che il secondo algoritmo è più efficiente.

Vediamo le formule di complessità sulle slide $T_P(n) = 2n^2$, $T_Q(n) = 100n$, $T_R(n) = 5n$

- q ed r hanno complessità minore o uguale a p
  - $T_q(n) \leq T_p(n)$ se $n \geq 50$. Quindi la complessità della prima è minore o uguale a quella della seconda. Non vale il contrario
  - $T_r(n) \leq T_p(n)$ se $n \geq 3$. Stessa cosa di prima
- q ed r hanno la stessa complessità. Se analizzo individuo la complessità della prima minore o uguale a quella della seconda. La cosa vale anche al contrario.

2.4 Notazione O-grande (limite asintotico superiore)

Definizione $f(n)$ è di ordine $O(g(n))$ se esistono un interno n_0 e una costante $c > 0$ tali che $n \geq n_0$: $\boxed{f(n) \leq cg(n)}$



Appartenenza di una funzione a un certo ordine O-grande Se voglio dimostrare che una funzione è O dell'altra devo trovare una coppia di valori $[n_0, c]$. Se esiste almeno una coppia (non è necessario trovare quella con valori più bassi) che soddisfa la definizione allora la funzione è O dell'altra! In tutte le coppie di formule messe a confronto poco fa è possibile individuare la coppia di valori $[n_0, c]$.

Notazione Una funzione si dice che è di un certo ordine ($f(n)$ è $O(g(n))$) o che appartiene a un certo ordine ($f(n) \in O(g(n))$). In alcuni casi è possibile leggere la notazione $f(n) = O(g(n))$, non proprio corretta (da evitare).
Solitamente non si utilizza la notazione $f(n)$ o $g(n)$ ma si pone direttamente l'espressione!

2.5 Regole

2.5.1 Regola dei fattori costanti

Per ogni costante positiva k, posso dire che $O(f(n)) = O(kf(n))$

2.5.2 Regola della somma - IMPORTANTE

Se $f(n)$ è di ordine $O(g(n))$ allora posso dire che $f(n) + g(n)$ è di ordine $O(g(n))$. Ho preso, tra le due, quella più grande!

2.5.3 Regola del prodotto

Se $f(n)$ è di ordine $O(f_1(n))$ e $g(n)$ è di ordine $O(g_1(n))$ posso dire che $f(n)g(n)$ è di ordine $O(f_1(n)g_1(n))$.

2.5.4 Transitività

Se $f(n) \in O(g(n))$ e $g(n) \in O(h(n))$ allora $f(n) \in O(h(n))$.

2.5.5 Complessità di costanti

Ogni costante k è di ordine $O(1)$.

2.5.6 Complessità di funzioni polinomiali

Regola dei polinomi Un polinomio di grado m ha complessità $O(n^m)$. La cosa è dimostrata facendo riferimento alla regola dei fattori costanti e alla regola della somma.

Confronto Dati due valori n, m, se $n \leq p$ allora $n^m \in O(n^p)$

Esempi

- $2n + 3n + 2 \in O(n)$
- $(n + 1)^2 \in O(n^2)$
- $10n^2 + 2n \in O(n^2)$

2.6 Classi di complessità

Possiamo immaginare la notazione O-grande come la rappresentazione di un insieme di funzioni di un certo ordine. Per esempio:

- $O(n) = \{k, n, 4n, 300n, 100 + n, \dots\}$
- $O(n^2) = O(n) \cup \{n^2, 300n^2, n + n^2, \dots\}$

Abbiamo le seguenti classi:

- **Complessità costante** ($O(1)$): ottime poichè non dipendenti dalla dimensione n.
- **Complessità logaritmica** ($O(\log n)$): tra le migliori che analizzeremo
- **Complessità lineare** ($O(n)$): anch'essa molto buona
- **Complessità** $n \log n$ ($O(n \log n)$)
- **Complessità quadratica** ($O(n^2)$)
- **Complessità cubica** ($O(n^3)$)
- **Complessità polinomiale** ($O(n^p)$)
- **Complessità esponenziale** ($O(2^n)$ o $O(n^p)$)

Individueremo programmi con complessità sottolineare, cioè compresa tra 1 ed n: non sono nè costanti nè lineari.

2.6.1 Complessità esponenziale

Le classi di complessità esponenziali sono le peggiori. Possono essere scritte[2] ma non eseguite: con n grande si avrebbero tempi elevatissimi. Esistono numerosi problemi ancora irrisolti in modo non esponenziale.

Teorema conseguente $\forall k, n^k \in O(a^n), \forall a > 1$... Una qualsiasi funzione polinomiale ha minore complessità di una qualsiasi funzione esponenziale.

Leggenda degli scacchi La prof ha parlato della *leggenda degli scacchi*: un principe ricco ma annoiato promise qualunque cosa richiesta a colui che fosse riuscito a farlo divertire. Vincitore tra tanti fu un mercante, che presentò al principe il gioco degli scacchi (una tavola con 64 caselle). Egli chiese in cambio un chicco di grano per la prima, due per la seconda, raddoppiando il numero di chicchi presenti fino a raggiungere la sessantaquattresima casella.
I funzionari di corte, tuttavia, si resero conto che la richiesta non poteva essere soddisfatta: la quantità richiesta era superiore ai raccolti di grano di tutto il mondo! Il mercante, conseguentemente, fu giustiziato.

$$2^{64} - 1 \textit{ chicchi} = 18.446.744.073.709.551.615 \textit{ chicchi}$$

[2] *Se lo scrivete licenziamento immediato!*

Capitolo 3

Martedì 10/03/2020

Registro 2 (*Mar 10/03/2020 10:30-13:30 (3:0 h) lezione*)

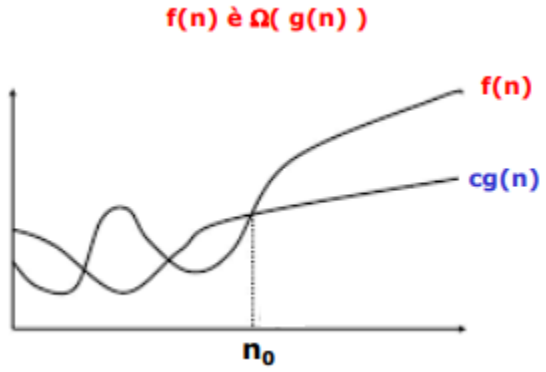
Notazioni Omega grande e Theta Grande. Complessità dei programmi iterativi.

La scorsa volta abbiamo iniziato a parlare della complessità degli algoritmi: funzioni sempre positive che associa alla dimensione del problema il costo della sua risoluzione. Un confronto tra algoritmi, ricordiamo, avviene tra funzioni di complessità. Il tutto ci permette di svolgere un confronto oggettivo svincolato dal dispositivo utilizzato e dal linguaggio di programmazione adottato. A tal proposito ricordiamo la notazione O-grande e tutte le regole connesse.

Sappiamo che $n \in O(n)$, vediamo che vale anche $n \in O(n^2)$, $n \in O(n^3)$. Dire una di queste cose in un esercizio in cui si chiede la complessità della funzione è cosa valida!

3.1 Notazione Ω grande (limite asintotico inferiore)

Si ha un limite asintotico inferiore, quindi l'esatto opposto della notazione O-grande. Sia $f(n) \in \Omega(g(n))$: abbiamo un intero n_0 e una costante $c > 0$ tale che per ogni $n \geq n_0$ si ha $\boxed{f(n) \geq cg(n)}$

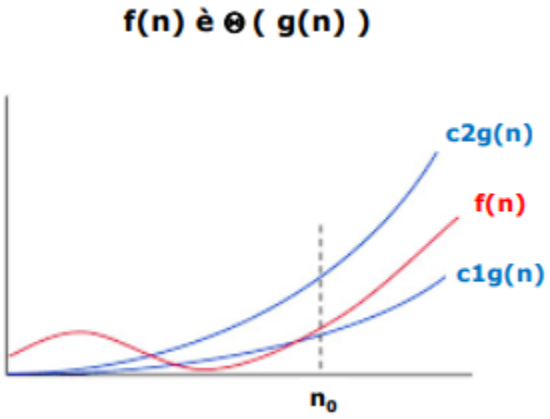


Esempi

- $2n^2 + 3n + 2 \in \Omega(n)$ con $[n_0 = 1, c = 1]$
- $2n^2 + 3n + 2 \in \Omega(n^2)$ con $[n_0 = 1, c = 1]$
- $n^2 \in \Omega(2n^2)$ con $[n_0 = 1, c = \frac{1}{2}]$
- $n^2 \in \Omega(100n)$ con $[n_0 = 101, c = 1]$

3.2 Notazione Θ grande (limite asintotico stretto)

Si dice $f(n) \in \Theta(g(n))$ se $f(n)$ e $g(n)$ hanno lo stesso ordine di complessità. Ciò avviene se nelle definizioni precedenti di O-grande e Ω-grande le funzioni sono interscambiabili nella definizione. Esiste un intero n_0 e due costanti $c_1, c_2 > 0$ tali che per ogni $n \geq n_0$ si ha $\boxed{c_1 g(n) \leq f(n) \leq c_2 g(n)}$



Esempi

- $2n^2 + 3n + 2 \in \Omega(n)$, ma $\notin O(n)$, quindi $\notin \Theta(n)$
- $2n^2 + 3n + 2 \in O(n^2)$ con $[n_0 = 1, c = 7]$, $\in \Omega(n^2)$ con $[n_0 = 1, c = 1]$. Segue che $\in \Theta(n^2)$.

Osservazione dall'esempio Un polinomio di grado m è di ordine $\Theta(n^m)$

3.3 Uniamo tutto quanto

$$\boxed{f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))}$$

$$\boxed{f(n) \in \Theta(g(n)) \Rightarrow g(n) \in \Theta(f(n))}$$

$$\boxed{f(n) \in O(g(n)), f(n) \in \Omega(g(n)) \Rightarrow f(n) \in \Theta(g(n))}$$

Le regole dei fattori costanti, della somma e del prodotto valgono anche per le notazioni introdotte oggi, tenendo conto di piccole differenze dagli esempi.

3.4 Complessità dei programmi

Fino ad ora abbiamo applicato le notazioni precedenti nella matematica. Proviamo ad applicare gli stessi concetti a un programma concentrandoci sul tempo. L'efficienza non si verifica contando i secondi necessari per l'esecuzione. Procedo analizzando il testo dell'algoritmo. Per comodità divideremo i programmi in due categorie, ciascuna con un approccio diverso: iterativi e ricorsivi.

3.4.1 Complessità dei programmi iterativi

Immaginiamo una funzione $C[x]$ in cui associo ad ogni costrutto del linguaggio una classe di complessità. La x può assumere i seguenti valori:

- **V**: costante
- **I**: variabile
- **E**: espressione
- **C**: comando

Analizziamo i vari costrutti partendo dai più semplici:

- $\mathbf{C[V]} = \mathbf{C[I]} = \mathbf{O(1)}$: costanti o espressioni. Si ha un tempo costante: ovviamente l'assegnamento di un valore appartenente alla variabile richiede un tantino di più rispetto a una costante.
- $\mathbf{C[E1\ op\ E2]} = \mathbf{C[E1]} + \mathbf{C[E2]}$: combinazione di due espressioni con un operatore. L'unione di due espressioni ci porta a sommare i tempi di esecuzione di ciascuna espressione.
- $\mathbf{C[I[E]]} = \mathbf{C[E]}$: espressione associata all'indice dell'array. Il tempo di esecuzione è pari a quello di calcolo dell'espressione.
- $\mathbf{C[I = E;]} = \mathbf{C[E]}$: operazione di assegnamento. Il tempo di esecuzione è pari a quello di calcolo dell'*rvalue*.
- $\mathbf{C[I[E1] = E2;]} = \mathbf{C[E1]} + \mathbf{C[E2]}$: operazione di assegnamento con espressione associata all'indice di un array. Il tempo di esecuzione è pari alla somma dei tempi di calcolo per le due espressioni.
- $\mathbf{C[return\ E;]} = \mathbf{C[E]}$: *return-statement*. Il tempo di esecuzione è quello del calcolo dell'espressione.
- $\mathbf{C[if(E)C1\ else\ C2]} = \mathbf{C[E]} + \mathbf{C[C1]} + \mathbf{C[C2]}$: *if-statement*. Il tempo di esecuzione è pari alla somma tra i tempi di esecuzione per la condizione, il *then-statement* e l'eventuale *else-statement*.

La complessità cresce nei programmi ripetitivi

- $\mathbf{C[for(E1;E2;E3)\ C]} = \mathbf{C[E1]} + \mathbf{C[E2]} + (\mathbf{C[C]} + \mathbf{C[E2]} + \mathbf{C[E3]})\ \mathbf{O(g(n))}$: *for-statement*. Il tempo di esecuzione è pari alla somma tra l'espressione dell'inizializzatore, l'espressione della condizione (entrambe prima esecuzione). Si somma, in aggiunta, la somma dei tempi di esecuzione del corpo, dello *step* e della condizione moltiplicata per O-grande di $g(n)$, la complessità del numero di iterazioni svolte.
- $\mathbf{C[while(E)\ C]} = \mathbf{C[E]} + (\mathbf{C[C]} + \mathbf{C[E]}\ \mathbf{O(g(n))}$: *while-statement*. Il tempo di esecuzione è pari a quello di calcolo della condizione (prima volta), più la somma del tempo di esecuzione del corpo del while e nuovamente della condizione moltiplicata per O-grande di $g(n)$, la complessità del numero di iterazioni svolte.
- $\mathbf{C[\{C1,\dots,Cn\}]} = \mathbf{C[C1]} + \cdots + \mathbf{C[Cn]}$: comando composto. I comandi vengono eseguiti in fila (primo, secondo,..., fino all'ultimo). Il tempo di esecuzione è pari alla somma di tutti i tempi di esecuzione dei vari comandi
- $\mathbf{C[F(E1,\dots,En)]} = \mathbf{C[E1]} + \cdots + \mathbf{C[En]} + \mathbf{C[\{C1,\dots,Cn\}]}$: chiamata di funzione. Ho una una funzione con tot argomenti e un corpo che viene eseguito. Considero il caso peggiore passando tutti i parametri per valore: ciò significa che ho un'espressione per ogni argomento attuale. Il tempo di esecuzione consiste nella somma tra i tempi di calcolo delle espressioni associate ai vari argomenti e il tempo di esecuzione del corpo della funzione.

Esempi di esercizi Vedere gli appunti sulle diapositive da pag.118 a pag.120

3.5 Esempio di calcolo: complessità del *selecionSort*

Consideriamo il *selectionSort*. Per scambiare gli elementi utilizziamo la *swap* della *STL*.

```
void selectionSort(int A[], int n) {
        for(int i = 0; i < n-1; i++) {
                int min = i;
                for(int j = i +1; j < n; j++)
                        if(A[j] < A[min]) min = j;
                swap(A[i], A[min]);
        }
}
```

- Osservo che gli assegnamenti presenti hanno complessità $O(1)$
- L'if-statement ha complessità $O(1)$
- La swap ha complessità $O(1)$
- Il for più interno ha complessità $O(n)$:
  - L'inizializzazione ha complessità $O(1)$
  - La condizione ha complessità $O(n)$: tengo conto che nel caso peggiore ho $j = 1$, quindi $n - 1$ verifiche di condizione, più la verifica di condizione finale che determina la conclusione del ciclo.
  - $j++$ ha complessità $O(n)$ (incremento n volte)
  - Il corpo ha complessità $O(1)$ ed è eseguito $n - 1$ volte, quindi ho complessità del numero di iterazioni pari a $O(n)$.
  - Quindi $O(1) + O(n) + O(n) + O(1) * O(n) = O(n)$.
- Il for più esterno, quindi il contenuto della funzione, ha complessità $O(n^2)$:
  - L'inizializzazione ha complessità $O(1)$
  - La condizione ha complessità $O(n)$: è verificata n volte ($n - 1$ volte + il controllo finale)
  - L'incremento ha complessità $O(n)$: è eseguito n volte
  - Il corpo ha complessità $O(n)$ ed è eseguito $n - 1$ volte, quindi ho complessità del numero di iterazioni pari a $O(n)$.
  - Quindi: $O(1) + O(n) + O(n) + O(n)O(n) = O(n^2)$

3.6 Chiamate di funzione all'interno di cicli

Quando la calcoliamo la complessità di una funzione, dipendente da un argomento n, potrebbe essere utile individuare non solo la complessità ma anche il risultato. Prendiamo per esempio le seguenti funzioni:

```
int f(int x) {
        return x;
}
```

Ho complessità $O(1)$, risultato $O(n)$.

```
int k(int x) {
        int a = 0;
        for(int i = 1; i <= x; i++) {
                a++;
        }
        return a;
}
```

Ho complessità $O(n)$, risultato $O(n)$.

Attenzione Se abbiamo una chiamata di funzione, per esempio, nelle condizioni di un while o di un for dovremo tenere conto sia della complessità (di una funzione chiamata più volte) sia del risultato per individuare la complessità totale. Ragioniamo sulle seguenti funzioni:

```
void g(int n) {
        for(int i = 1; i <= f(n); i++)
                cout<<f(n);
}
```

Le regole per il calcolo della complessità sono le stesse ma devo stare attento al numero di iterazioni e alla complessità derivante dalla condizione. Tengo conto che $i <= f(n)$

- Viene eseguito un numero di volte pari al risultato della funzione, più 1.
- Ogni esecuzione del programma presenta una certa complessità.

Tenendo conto che la funzione f ha complessità $O(1)$ e risultato $O(n)$ individuo che la complessità della funzione sarà

$$O(1) + \boxed{O(n+1) * O(1)} + O(n) + \boxed{O(n) * O(1)} = O(n)$$

```
void g(int n) {
        for(int i = 1; i <= k(n); i++)
                cout<<k(n);
}
```

Le regole sono le stesse, tenendo conto che la funzione $k(n)$ ha complessità e risultato $O(n)$ individuo che la complessità della funzione sarà

$$O(1) + \boxed{O(n+1) * O(n)} + O(n) + \boxed{O(n) * O(n)} = O(1) + O(n^2) + O(n) + O(n^2) = O(n^2)$$

Possiamo ridurre la complessità di questa funzione scrivendola nella seguente forma

```
void p(int n) {
        int b = k(n);
        for(int i = 1; i <= b; i++)
                cout<<b;
}
```

L'esecuzione di $k(n)$ è unica e si ottiene lo stesso risultato. Segue che la complessità della funzione sarà

$$O(n) + O(1) + \boxed{O(n+1)} + O(n) + \boxed{O(n) * O(1)} = O(n)$$

Capitolo 4

Venerdì 13/03/2020

Registro 3 (*Ven 13/03/2020 11:30-13:30 (2:0 h) lezione*)

> *Programmazione ricorsiva: principi e regole. Programmi ricorsivi su liste. Principio di induzione naturale. Esercizi sul calcolo della complessità dei programmi iterativi.(lezione tenuta in modalità telematica)*

Necessario fare ragionamento generale prima di affrontare il calcolo della complessità.

Come esempio di funzione ricorsiva possiamo prendere il fattoriale. Ad esso associamo una definizione iterativa da cui possiamo trarre un algoritmo iterativo e una definizione induttiva (o ricorsiva) da cui possiamo trarre un algoritmo ricorsivo.

Supponiamo di non avere il prodotto moltiplicazione: creiamo una funzione. Individuiamo anche in questo caso una definizione ricorsiva da cui creiamo un algoritmo ricorsivo.

Un altro esempio è l'algoritmo di Euclide: noi lo abbiamo già visto scritto in forma iterativa, adesso lo vedremo anche in forma ricorsiva.

Si osserva la scomparsa dei cicli negli algoritmi ricorsivi, sostituiti da chiamate ricorsive di funzioni.

4.1 Regole per la programmazione ricorsiva

Il primo rischio quando si scrive un algoritmo ricorsivo è finire in loop in caso di errore. Ricordiamo che quando avviene una chiamata ricorsiva si interrompe l'esecuzione del programma attuale per avviare l'istanza della nuova funzione, al termine di questa si ritorna al programma precedentemente interrotto. Quindi:

- dobbiamo individuare i casi basi in cui la funzione è definita immediatamente
- ogni volta che si effettua una chiamata questa avviene su un insieme più piccolo. Il principio è che gli argomenti attuali di ogni chiamata saranno diversi.
- dobbiamo fare in modo che la sequenza di chiamate ricorsive venga interrotta facendoci cadere in uno dei casi base

4.2 Esempi con le liste

Vediamo dei programmi ricorsivi che permettono di manipolare le liste. La lista è una struttura dati adatta per algoritmi ricorsivi. La lista può essere definita come: NULL (lista vuota) è una lista, un elemento seguito da una lista è una lista.

- **lunghezza di una lista**. Prendiamo una lista con due elementi: ho una chiamata ricorsiva e un caso base (che determina la somma di 0 al conto già fatto e la conclusione del ciclo)

```
int lenght(Elem* p) {
        if(p == NULL) return 0;
        return 1 + lenght(p->next);
}
```

- **quante volte incontro un intero nella lista**. Utilizzo un'espressione booleana che restituisce 0 o 1 in base alla non corrispondenza o corrispondenza di un numero con quello indicato tra gli argomenti

```
int howMany(Elem* p, int x) [
        if(p == NULL) return 0;
        return (p->inf == x) + howMany(p->next, x);
}
```

- **individuare presenza di un certo numero in una lista**. Chiamo la funzione finchè non si ha corrispondenza. Mi fermo, ovviamente, se arrivo alla fine della lista: in questo caso restituirò 0 per indicare che non ci sono elementi

```
int belongs(Elem* l, int x) {
        if(l == NULL) return 0;
        if(l->inf == x) return 1;
        return belongs(l->next, x);
}
```

- **eliminazione dell'ultimo elemento della lista**. Scorro la lista con chiamate ricorsive finchè non raggingo l'ultimo elemento. L'ultimo elemento ha puntatore a NULL: quando individuo un puntatore con quel valore elimino l'elemento e pongo il puntatore dell'elemento precedente a NULL. **Osservazione**: l'assenza della delete non comporta il non funzionamento dell'algoritmo, è una cosa in più che facciamo a supporto del C++. Il delete permette il recupero della cella per poterla riutilizzare in altre istruzioni.

```
void tailDelete(Elem*& l) [
        if(l == NULL) return;
        if(l->next == NULL) {
                delete l;
                l = NULL;
        }
        else tailDelete(l->next);
}
```

- **inserimento di un nuovo elemento in fondo alla lista**. Scorro la lista mediante chiamate di funzione finchè non arrivo in fondo alla lista (anche in questo caso sfruttiamo il puntatore a NULL dell'ultimo elemento).

```
void tailInsert(Elem*& l) [
        if(l == NULL) {
                l = new Elem;
                l->inf = x;
                l->next = NULL;
        }
        else tailInsert(l->next, x);
}
```

- **append**, per unire due liste. Faccio in modo che l'ultimo elemento della prima lista punti al primo elemento della seconda lista!

```
void append(Elem*& l1, Elem* l2) {
        if(l1 == NULL) l1 = l2;
        else append(l1->next, l2);
}
```

- **append**, seconda versione. Faccio la stessa cosa mediante restituzione di puntatore: ogni volta mediante assegnazione pongo l'indirizzo del puntatore all'elemento successivo.

```
Elem* append(Elem* l1, Elem* l2) {
        if(l1 == NULL) return l2;
        l1->next = append(l1-next, l2);
        return l1;
}
```

4.3 Induzione naturale

Quando si lavora sui programmi ricorsivi può essere utile il principio di induzione: ne esistono tanti in matematica, vediamo quello più semplice che può essere utilizzato per dimostrare la validità del programma di una lista.

4.3.1 Dimostriamo una proprietà P su tutti i numeri naturali

Non possiamo svolgere una dimostrazione di tipo esaustivo poichè i numeri naturali sono infiniti. Dobbiamo dimostrare:

- **la base**: P vale per 0.
- **il passo induttivo**. Se la proprietà P è valida con n allora è valida anche per $n+1$. Se vale per un numero allora vale anche per il numero dopo!

Esempio

Vogliamo dimostrare che la sommatoria dei primi n numeri è uguale all'operazione nel secondo membro

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$

Base $n = 0 \quad \sum_{0\ldots0} = \frac{0*1}{2} = 0$

Passo induttivo

- Ipotesi: $\sum_{0\ldots n} = \frac{n(n+1)}{2}$
- Tesi: $\sum_{0\ldots n+1} = \boxed{\frac{(n+1)(n+2)}{2}}$
- Dimostrazione:
  - $\sum_{0\ldots n+1} = \sum_{0\ldots n} + (n+1) = \frac{n(n+1)}{2} + (n+1)$
  - svolgendo i calcoli trovo che

$$\frac{n(n+1)}{2} + (n+1) = \frac{n^2+3n+2}{2} = \boxed{\frac{(n+1)(n+2)}{2}}$$

Attenzione La proprietà dimostrata con l'induzione è estremamente utile negli esercizi sulla complessità. Vedere le ultime pagine della dispensa!

Capitolo 5

Martedì 17/03/2020

Registro 4 (*Mar 17/03/2020 10:30-13:30 (3:0 h) lezione*)

> *Complessità dei programmi ricorsivi. Relazioni di ricorrenza divide et impera e lineari. Ricerca binaria. Quicksort. Torre di Hanoi. Esercizi su algoritmi ricorsivi.*

L'altro giorno ci siamo lasciati introducendo l'induzione naturale. Oggi entriamo nel vivo della complessità dei programmi ricorsivi.

Complessità del fattoriale Vediamo la funzione fact.

```
int fact(int x) {
        if(x == 0) return 1;
        else return x*fact(x-1);
}
```

Abbiamo l'if-statement con espressione di complessità $O(1)$ e il then-statement di complessità $O(1)$. L'else-statement presenta complessità di $O(1)$ per svolgere la moltiplicazione, ma dobbiamo considerare la chiamata ricorsiva di funzione.

Considero i casi base e i passi ricorsivi attraverso una relazione di ricorrenza. Necessario campionare la complessità con tutti i valori n:

$$\begin{cases} T(0) = a \\ T(n) = b + T(n-1) \end{cases}$$

Dove a, b sono generiche costanti di ordine $O(1)$. Con le chiamate ricorsive con argomento diverso a un certo punto avrò $T(1)$ con all'interno la chiamata di $T(0)$. Proviamo a risolvere

$$\begin{aligned} T(0) &= a \\ T(1) &= b + a \\ T(2) &= b + b + a = 2b + a \\ T(3) &= 3b + a \dots \\ T(n) &= n\,b + a \end{aligned}$$

Individuiamo che la relazione di ricorrenza $T(n) \in O(n)$. Il tempo di questa funzione è lineare!

selectionSort Proviamo con un selectionSort ricorsivo.

```
void r_selectionSort(int* A, int m, int i = 0) {
        if(i == m-1) return;
        int min = i;
        for(int j = i+1; j < m; j++)
                if(A[j] < A[min]) min = j;

        swap(A[i], A[min]);
        r_selectionSort(A, m, i+1);
}
```

Nella funzione avremo un argomento in più poichè la struttura dati coinvolta è un array. Abbiamo sostituito il for più esterno con un procedimento di chiamata ricorsiva: in ordine inverso saranno terminate tutte le istanze.

- Il caso base blocca l'algoritmo ricorsivo quando ho compiuto $n-1$ scorrimenti. Ho una situazione del genere quando mi rimane un'area di intervento con un solo elemento. Quindi $T(1) = a$
- Escludendo il caso base e la chiamata ricorsiva di funzione, il blocco di istruzioni rimanente è di complessità $O(n)$.
- Ottengo

$$\begin{cases} T(1) = a \\ T(n) = bn + T(n-1) \end{cases}$$

$$\begin{aligned} T(1) &= a \\ T(2) &= 2b + a \\ T(3) &= 3b + 2b + a \\ T(n) &= (n + n - 1 + n - 2 + \dots 2)b + a = \left(\frac{n(n+1)}{2} - 1\right) b + a \end{aligned}$$

Il *selectionSort* ha complessità $O(n^2)$ anche in forma ricorsiva.

5.1 *quickSort* - Algoritmo di ordinamento

quickSort è un algoritmo di ordinamento utilizzabile sia in forma ricorsiva che iterativa: la prima rè più conveniente in termini di comprensione e realizzazione. Come vedremo la complessità di questo algoritmo è inferiore ad n^2. Gli argomenti della funzione sono il puntatore ad array e gli indici del primo e dell'ultimo elemento appartenenti all'intervallo che stiamo analizzando.
Inizialmente l'estremo inferiore sarà 0 e l'estremo superiore il numero di elementi dell'array meno uno.

1. Scelgo un perno, cioè un elemento dell'array nell'intervallo che stiamo analizzando. Trovo l'indice del perno mediante media tra inf e sup. In alcune versioni di questo algoritmo la scelta del perno è addirittura casuale!
2. Con il perno divido l'array in due parti: a sinistra devo avere elementi minori del perno, a destra elementi maggiori del perno.
3. Scorro l'array sia dall'ultimo elemento che dal primo: individuo il primo elemento a sinistra del perno che dovrebbe stare alla sua destra e il primo elemento a destra del perno che dovrebbe stare alla sua sinistra. Scambio i due numeri di posizione
4. Continuo finchè l'indice incrementato non supererà l'indice decrementato

5. A questo punto effettuerò due chiamate ricorsive: una per analizzare l'intervallo a sinistra del perno, una per analizzare l'intervallo a destra del perno.

   - Effettuo la prima chiamata solo se l'estremo inferiore è più piccolo dell'indice decrementato
   - Effettuo la seconda chiamata solo se l'indice incrementato è minore dell'estremo superiore

   Le due chiamate potrebbero essere svolte parallelamente in quanto una non va ad intaccare l'altra!

6. Ripeto quanto detto con altre chiamate ricorsive. L'algoritmo termina la sua opera quando entrambe le condizioni non sono più soddisfatte! A quel punto mi rimane un solo elemento nell'intervallo analizzato.

5.1.1 Codice

```
void quickSort(int A[], int inf, int sup) {
        int perno = A[(inf+sup)/2], s=inf, d= sup;
        while(s<=d) {
                while(A[s]<perno) s++;
                while(A[d]>perno) d--;

                if(s>d) break;

                exchange(A[s],A[d]);
                s++; d--;
        }

        if(inf < d) quickSort(A,inf,d);
        if(s < sup) quickSort(A,s,sup);
}
```

5.1.2 Complessità

- Considero che $T(1) = a$ poichè l'algoritmo si ferma quando nell'intervallo si ha un solo elemento
- Tengo conto che con le due chiamate ricorsive analizzerò rispettivamente k elementi ed $n - k$ elementi.
- Considero che il while all'interno della funzione *quickSort* ha complessità $O(n)$.
- Ottengo
$$\begin{cases} T(1) = a \\ T(n) = bn + T(k) + T(n-k) \end{cases}$$
Necessario porre un valore k per rendere la risoluzione agevole.
- **Caso peggiore:** $k = 1$ Nel caso peggiore ho una divisione in una parte con un elemento e un'altra con $n - 1$ elementi
$$\begin{cases} T(1) = a \\ T(n) = bn + T(n-1) \end{cases}$$
La relazione di ricorrenza è la stessa del *selectionSort*, quindi posso dire senza fare calcolo che ho complessità $O(n^2)$.

- **Caso migliore:** $k = n/2$ Nel caso migliore si hanno due parti con $n/2$ elementi. Ottengo due chiamate ricorsive con lo stesso argomento:

$$\begin{cases} T(1) = a \\ T(n) = bn + T(n/2) + T(n/2) = bn + 2T(n/2) \end{cases}$$

$$\begin{aligned} T(1) &= a \\ T(2) &= 2b + 2a \\ T(4) &= 4b + 4b + 4a \\ T(8) &= 8b + 8b + 8b + 8a = \mathbf{3}(8b) + 8a \\ T(16) &= 16b + 16b + 16b + 16b + 16a = \mathbf{4}(16b) + 16a \end{aligned}$$

I valori evidenziati in grassetto consistono nel logaritmo in base due del valore n posto come argomento ($4 = \log_2 16, 3 = \log_2 8$). Quindi

$$T(n) = (n \log n)b + na$$

Ho complessità di ordine $O(n \log n)$. L'efficienza è maggiore rispetto a un algoritmo di ordine $O(n^2)$. I casi in cui l'algoritmo è di ordine $O(n^2)$ sono pochi, solitamente quando gli algoritmi sono già ordinati (dettaglio da considerare nella scelta).

5.2 Ricerca lineare ricorsiva

```
int RlinearSearch(int A[], int x, int m, int i = 0) {
        if(i == m) return 0;
        if(A[i] == x) return 1;
        return RlinearSearch(A, x, m, i+1);
}
```

L'algoritmo ricorsivo per la ricerca lineare non è strutturalmente diverso dallo scorrimento della lista. Abbiamo due casi base:

- Quando abbiamo finito di scorrere l'array senza trovare corrispondenze
- Quando individuiamo una corrispondenza tra valore numerico indicato e valore dell'elemento dell'array

Poichè quando termino l'esecuzione dell'algoritmo ricorsivo non ho più elementi, allora $T(0) = a$. Considerando che il tempo di esecuzione delle istruzioni rimanenti è costante e che andando avanti il numero di elementi rimasti si riduce di uno ottengo $T(n) = b + T(n-1)$. Quindi

$$\begin{cases} T(0) = a \\ T(n) = b + T(n-1) \end{cases}$$

Questa relazione di ricorrenza è la stessa del fattoriale, quindi ho una complessità di ordine $O(n)$.

5.3 Ricerca binaria ricorsiva

```
int binSearch(int A[], int x, int i, int j) {
        if(i > j) return 0;
        int k = (i+j)/2;
        if(x == A[k]) return 1;
```

```
        if(x < A[k])
                return binSearch(A, x, i, k-1);
        else
                return binSearch(A, x, k+1, j);
}
```

Ricordare Algoritmo utilizzabile soltanto in array ordinati.

La funzione presenta due argomenti per esprimere la porzione di array analizzata (l'indice iniziale e quello finale). Il meccanismo è quello già visto a Fondamenti di programmazione (vedere se necessario il capitolo dedicato negli appunti).

- Svolgo la media tra i due indici e trovo l'indice del perno, che divide l'array in due parti.
- Se il perno è uguale al valore che sto ricercando mi fermo
- Se non è uguale svolgo la chiamata ricorsiva: analizzo la parte sinistra se il valore da ricercare è minore del perno, la parte destra se il valore da ricercare è maggiore del perno.
- Mi fermo quando l'indice normalmente più piccolo sarà maggiore dell'indice normalmente più grande. Questo avviene perchè ogni volta che si va a chiamare la funzione uno dei due indici viene alterato (o incremento l'indice sinistro, o decremento l'indice destro).

Nella relazione di ricorrenza ricordo che l'algoritmo si ferma quando non ci sono più elementi e ogni volta che chiamo una funzione la porzione analizzata si dimezza. Ottengo

$$\begin{cases} T(0) = a \\ T(n) = b + T(n/2) \end{cases}$$

Calcoliamo

$$\begin{aligned} T(0) &= a \\ T(1) &= b + a \\ T(2) &= b + b + a \\ T(4) &= b + b + b + a \\ T(8) &= b + b + b + b + a \\ &\dots \\ T(n) &= (\log_2 n + 1)b + a \end{aligned}$$

L'algoritmo ha una complessita di ordine $O(\log n)$, inferiore a $O(n)$.

5.4 Ricerca in un insieme (per array non ordinati)

```
int Search(int A[], int x, int i, int j) {
        if(i > j) return 0;
        int k = (i+j)/2;
        if(x == A[k]) return 1;
        return Search(A, x, i, k-1) || Search(A, x, k+1, j);
}
```

Posso applicare gli stessi principi della ricerca binaria per verificare la presenza di un elemento anche all'interno di un array non ordinato. La differenza si ha nella chiamata ricorsiva: non restituisco il valore

ottenuto da una chiamata ricorsiva ma il risultato di un'espressione booleana. Sfrutto il meccanismo di cortocircuito ed effettuo una ricerca prima a sinistra del perno, e dopo a destra del perno (se non ho già trovato l'elemento).
Per scrivere la relazione di ricorrenza tengo conto che nel caso peggiore ho la chiamata di due funzioni: ciascuna si occupa di metà porzione.

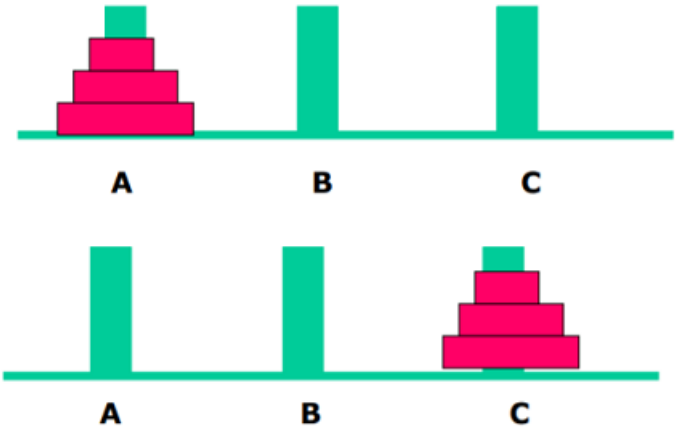
$$\begin{cases} T(0) = a \\ T(n) = b + 2T(n/2) \end{cases}$$

Calcoliamo

$$\begin{aligned} T(0) &= a \\ T(1) &= b + 2a \\ T(2) &= b + 2b + 4a = 3b + 4a \\ T(4) &= b + 6b + 8a = 7b + 8a \\ &\dots \\ T(n) &= (2n-1)b + 2n\,a \end{aligned}$$

La complessità dell'algoritmo è di ordine $O(n)$.

5.5 Torre di Hanoi



La *Torre di Hanoi* è un rompicapo matematico. Ho tot dischi di grandezza decrescente presenti in un paletto, il mio obbiettivo è spostarli in un altro paletto tenendo conto di due regole

- Si sposta un disco alla volta
- Non si pone mai un disco sopra un altro più piccolo

Segue che per spostare i dischi avrò bisogno di un paletto ausiliario, quindi tre paletti in tutto. Analizziamo la funzione:

```
void hanoi(int n, pal A, pal B, pal C) {
        if(n == 1)
                sposta(A,C);
        else {
                hanoi(n-1, A, C, B);
                sposta(A, C);
                hanoi(n-1, B, A, C);
        }
}
```

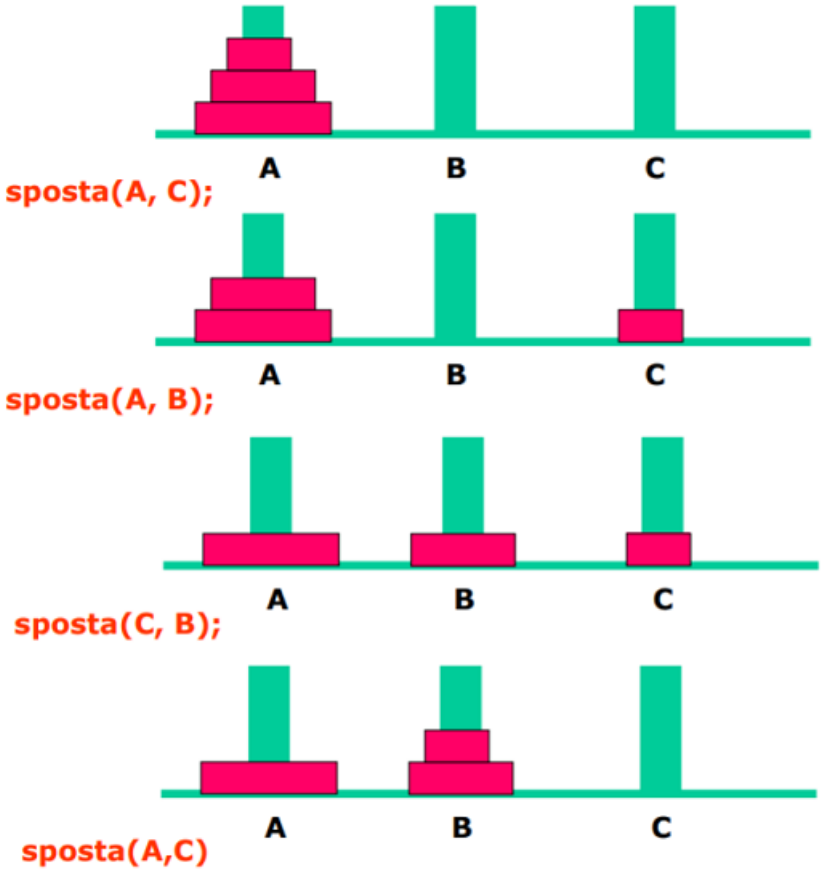
- *pal A* è il paletto di partenza, *pal B* quello di supporto, *pal C* quello di destinazione.
- Attraverso il primo argomento indico il numero di elementi da spostare.
- Supponiamo di avere n cerchi nel paletto A: attraverso la prima chiamata di funzione del caso ricorsivo evidenzio lo spostamento di $n-1$ elementi (tutti tranne quello più grande in basso) dal paletto A al paletto B
- L'elemento rimasto viene spostato sul paletto di supporto C.
- Riprendo la torre messa nel paletto B e la sposto nel paletto C. In questo caso sarà il paletto A quello di supporto.

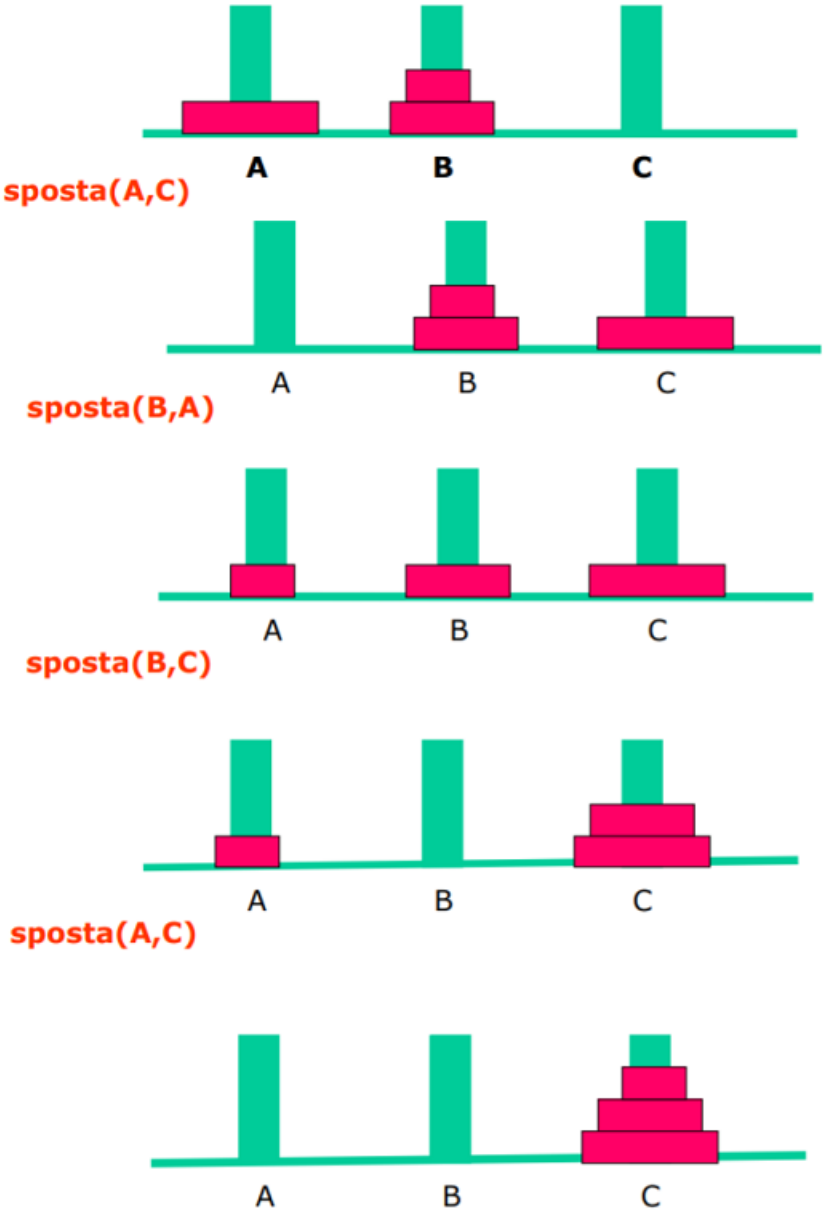
Complessità Ho la seguente relazione

$$\begin{cases} T(1) = a \\ T(n) = b + 2T(n-1) \end{cases}$$

Segue complessità esponenziale $O(2^n)$. Questo algoritmo può essere considerato uno dei problemi ricorsivi più complessi.

Esempio con le immagini




A
B
C
sposta(A,C)
A
B
C
sposta(B,A)
A
B
C
sposta(B,C)
A
B
C
sposta(A,C)
A
B
C

5.6 Metodo *divide et impera*

Prendiamo gli algoritmi ricorsivi dove le chiamate sono fatte su un numero di elementi pari alla metà di quelli a cui si applica la funzione. Questo tipo di algoritmo in cui si divide gli insiemi in parti uguali si basa sulla metodologia del *divide et impera*.

Come è fatto un programma del genere?

- Si applica a un insieme S
- Ho un caso base quando la dimensione dei dati è minore o uguale di una certa costante: in quel caso risolvo il problema
- In caso contrario divido il problema in più sottoinsiemi: svolgo una chiamata ricorsiva per ogni sottoinsieme
- Combino i risultati dopo aver effettuato tutte le chiamate necessarie

5.6.1 Esempi

5.6.1.1 Ricerca binaria

L'insieme è diviso in due sottoinsiemi ma abbiamo effettuato una chiamata di funzione per uno solo dei due. Ho complessità $O(\log n)$.

$$\begin{cases} T(0) = d \\ T(n) = b + T(n/2) \end{cases}$$

5.6.1.2 Ricerca in un insieme

Divido l'insieme in due parti, nel caso peggiore effettuo due chiamate di funzioni (quando non trovo l'elemento desiderato con la prima chiamata di funzione). Ho complessità $O(n)$.

$$\begin{cases} T(0) = d \\ T(n) = b + 2T(n/2) \end{cases}$$

5.6.1.3 *quickSort*

Caso migliore del *quickSort*. Array diviso in due sottoinsiemi con due chiamate di funzioni. Fuori dalle chiamate la complessità dipende dal numero di elementi. Ho complessità $O(n \log n)$.

$$\begin{cases} T(0) = d \\ T(n) = bn + T(n/2) \end{cases}$$

5.6.2 Teorema per la risoluzione immediata della complessità

Questo teorema è estremamente utile negli esercizi d'esame. La difficoltà non sta nell'individuare la complessità ma nell'arrivare alla relazione di ricorrenza. Le relazioni tipiche di algoritmi con metodo *divide et impera* presentano la seguente struttura (con $h > 0$)

$$\begin{cases} T(n) = d & n \leq m \\ T(n) = hn^k + aT(n/b) & n > m \end{cases}$$

Teorema La risoluzione avviene ponendo a confronto il valore di a con il valore di b^k! Individuo che

- Se $a < b^k, T(n) \in O(n^k)$
- Se $a = b^k, T(n) \in O(n^k \log n)$
- Se $a > b^k, T(n) \in O(n^{\log_b a})$

Ricerca binaria $k = 0, a = 1, b = 2 \Longrightarrow 1 = 2^0 \Longrightarrow T(n) \in O(n^0 \log n) \Longrightarrow T(n) \in O(\log n)$

Ricerca in un insieme $k = 0, a = 2, b = 2 \Longrightarrow 2 > 2^0 \Longrightarrow T(n) \in O(n^{\log_2 2}) \Longrightarrow T(n) \in O(n)$

quickSort $k = 1, a = 2, b = 2 \Longrightarrow 2 = 2^1 \Longrightarrow T(n) \in O(n^1 \log n) \Longrightarrow T(n) \in O(n \log n)$

5.7 Relazioni lineari

Le relazioni di ricorrenza si dicono lineari se avviene l'esecuzione di chiamate di funzione su $n-1$ elementi dell'insieme. Gli algoritmi con relazioni lineari sono meno efficienti rispetto a quelli basati sul *divide et impera*: dati n elementi avrò n chiamate di funzione. Le relazioni presentano la seguente struttura:

$$\begin{cases} T(0) = d \\ T(n) = bn^k + a_1 T(n-1) + a_2 T(n-2) + \cdots + a_r T(n-r) \end{cases}$$

Con due chiamate ricorsive possiamo dire subito che la complessità è esponenziale: $a_1 = 1, a_i = 0\ i > 1$.

5.7.1 Caso elementare

$$\begin{cases} T(0) = d \\ T(n) = bn^k + T(n-1) \end{cases}$$

In questo caso ho $T(n) \in O(n^{k+1})$

5.7.2 Esempi del caso elementare

5.7.2.1 Ricerca lineare

$$\begin{cases} T(0) = d \\ T(n) = b + T(n-1) \end{cases}$$

Ottengo $T(n^{0+1})$

5.7.2.2 *selectionSort*

$$\begin{cases} T(0) = d \\ T(n) = bn + T(n-1) \end{cases}$$

Ottengo $T(n^{1+1})$

5.7.2.3 Torre di Hanoi

$$\begin{cases} T(0) = d \\ T(n) = bn + 2T(n-1) \end{cases}$$

Abbiamo due chiamate ricorsive, quindi sappiamo già che la complessità è di tipo esponenziale.
Ottengo $T(n) \in O(2^n)$

Capitolo 6

Martedì 24/03/2020

Registro 5 (*Mar 24/03/2020 10:30-13:30 (3:0 h) lezione*)

Numeri di Fibonacci. Algoritmo di ordinamento mergesort. Mergesort su liste semplici. Esercizi di calcolo della complessità dei programmi.

La scorsa volta abbiamo analizzato la risoluzione della complessità dei programmi ricorsivi mediante una relazione di ricorrenza. Abbiamo analizzato due tipi di relazioni: la *divide et impra* e la *lineare*.

6.1 Serie di Fibonacci

La serie di Fibonacci può essere definita in modo induttivo nel seguente modo

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \end{aligned}$$

Il primo numero è zero, il secondo numero uno, gli altri la somma dei due numeri precedenti.

Motivo della serie (tratto da FdP) Supponiamo di avere una coppia di conigli (maschio e femmina). I conigli sono in grado di riprodursi all'età di un mese. Supponiamo che i nostri conigli non muoiano mai e che la femmina produca sempre una nuova coppia (un maschio ed una femmina) ogni mese dal secondo mese in poi. Il problema posto da Leonardo Fibonacci fu: quante coppie ci saranno dopo un anno?

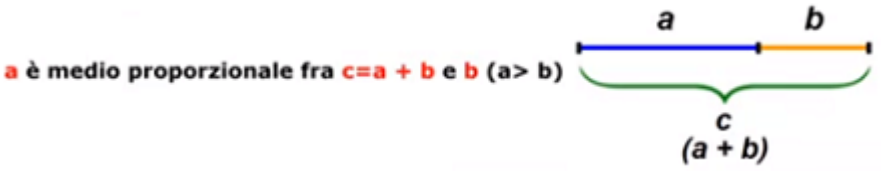
Sezione aurea Osserviamo che il il limite di un rapporto tra ogni numero della serie e il suo precedente tende a un certo valore (man mano che scorro la serie il valore del rapporto si avvicina sempre di più al valore indicato)

$$\lim_{n\to\infty} \frac{f_n}{f_{n-1}} = \frac{1+\sqrt{5}}{2}$$

- In un pentagono regolare il rapporto tra diagonale e lato è uguale al valore aureo.
- Prendiamo una grandezza c e dividiamola in due sottograndezze:a e b.
  Se a è medioproporzionale fra c e b, cioè è valida la seguente proporzione

  $$c : a = a : b$$

  il rapporto è sempre il valore della sezione aurea.



Spirale logaritmica La serie di Fibonacci è la base della spirale logaritmica del Bernoulli.

6.1.1 Programma che calcola l'*n-esimo* numero della serie

6.1.1.1 Prima versione (ricorsiva, vista a *Fondamenti di programmazione*)

Considero i casi base presenti nella definizione induttiva. Possiamo scrivere un programma molto elegante

```
int fibonacci_malvagio(int n) {
        if(n == 0) return 0;
        if(n == 1) return 1;
        return fibonacci_malvagio(n-1)+fibonacci_malvagio(n-2);
}
```

Problema Proviamo a calcolare la complessità. Abbiamo la seguente relazione di ricorrenza

$$\begin{cases} T(0) = T(1) = d \\ T(n) = b + T(n-1) + T(n-2) \end{cases}$$

La linearità e le due chiamate di funzione ci portano a dire che $T(n) \in O(2^n)$. Il programma è assolutamente inefficiente!

Dimostrazione Provate a eseguire questa versione e quelle successive con $n = 500$. Vedrete che nelle successive la risoluzione è immediata, in questa bisogna attendere.

6.1.1.2 Seconda versione

Il programma precedente non può essere assolutamente scritto in contesto lavorativo. Possiamo scrivere una versione iterativa più efficiente? Sì!

```
int fibonacci_iterativo(int n) {
        int n1 = 0;
        int n2 = 1;

        for(int i = 0; i < n; i++) {
                int sup = n1;
                n1 = n2;
                n2 = n1 + sup;
        }
        return n1;
}
```

Il programma è meno elegante ma più efficiente, infatti $T(n) \in O(n)$.

6.1.1.3 Terza versione

Ma non possiamo scrivere una funzione ricorsiva più efficiente? Certamente!

```
int fibonacci_ricorsivo(int n, int n1 = 0, int n2 = 1) {
        if(n == 0) return n1;
        return fibonacci_ricorsivo(n-1, n1, n1 + n2);
}
```

Calcoliamo la complessità Abbiamo la seguente relazione di ricorrenza

$$\begin{cases} T(0) = d \\ T(n) = b + T(n-1) \end{cases}$$

Quindi $T(n) \in O(n)$: la versione alternativa è più efficiente.

6.2 *mergeSort* - Algoritmo di ordinamento

Il *mergeSort* ha una complessità ottima come il *quickSort* ed è applicabile a sequenze generiche: esiste sia la versione ricorsiva che quella iterativa ed esiste sia la versione per liste che quella per array. L'algoritmo è consigliato soprattutto per l'ordinamento delle liste: la versione per array occupa memoria creando un array di appoggio.

Funzionamento

- L'algoritmo restituisce la lista inalterata se vuota o con un solo elemento
- In tutti gli casi:
  - Con *split* divido la lista in due sottoliste aventi lo stesso numero di elementi (ovviamente se ho un numero di elementi dispari avrò una sottolista con un elemento in più)
  - Effettuo una chiamata di funzione per ciascuna sottolista
  - Con *merge* effettuo una fusione ordinata delle due sottoliste.

6.2.1 Funzione principale

Il ciclo ricorsivo termina quando si ha una lista vuota o una lista con un solo elemento. Definisco un puntatore a lista inizialmente vuoto, che diventerà puntatore alla seconda lista creata con *split*. Chiamo due volte la funzione *mergeSort* per intervenire sulle due sottoliste appena create. Successivamente unisco le due liste già ordinate in un'unica liste con *merge*.

Particolarità In questa funzione si effettua un'ulteriore chiamata di funzione dopo aver richiamato ricorsivamente la funzione principale.

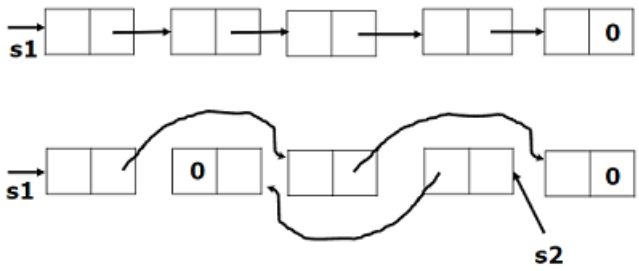
```
void mergeSort(Elem*& lista) {
        if(lista == NULL | lista->next == NULL) return;

        Elem* lista2 = NULL;
        split(lista, lista2);
        mergeSort(lista);
        mergesort(lista2);
        merge(lista, lista2);
}
```

Relazione di ricorrenza e complessità

$$\begin{cases} T(0) = T(1) = d \\ T(n) = bn + 2T(n/2) \end{cases}$$

Con le regole che conosciamo possiamo dire subito che $T(n) \in O(n \log n)$.

6.2.2 *split*



Pensiero Non potrei fare una funzione che scorre una prima volta la lista per contare il numero di elementi e una seconda per arrivare al punto di divisione? La complessità è $O(n)$, ma voglio realizzare qualcosa di ancora più efficiente: la funzione adottata ha sempre complessità $O(n)$ ma effettua un solo scorrimento: crea le due liste man mano che svolge lo scorrimento.

La funzione *split* crea una nuova lista formata solo da elementi in posizione pari e modifica la lista già esistente in modo che contenga soltanto elementi in posizione dispari. Anche qua mi fermo se la lista è vuota o ha un solo elemento.

- Creo un puntatore p a cui assegno come valore l'indirizzo dell'elemento successivo (quello dispari) a quello analizzato (quello pari)
- Collego all'elemento che stiamo analizzando (quello pari) l'elemento successivo a quello puntato prima (il prossimo elemento pari)
- Collego la seconda lista con elementi dispari a $s2$: abbiamo un aggiunta di elementi in testa, segue che l'ordine degli elementi della seconda lista sarà al contrario. Non posso fare l'aggiunta in coda: dovrei scorrere tutte le volte la lista perdendo tempo!
- Richiamo la lista applicandola all'elemento pari successivo e ripeto quanto fatto prima

```
void split(Elem*& lista1, Elem*& lista2) {
        if(lista1 == NULL || lista1->next == NULL) return;

        Elem* sup = lista1->next;
        lista1->next = sup->next;
        sup->next = lista2;
        lista2 = sup;

        split(lista1->next, lista2);
}
```

Relazione di ricorrenza e complessità

$$\begin{cases} T(0) = T(1) = d \\ T(n) = b + T(n-2) \end{cases}$$

Trovo che $T(n) \in O(n)$.

6.2.3 *merge*

La funzione riceve in ingresso due liste già ordinate e le fonde in modo da ottenere una lista ordinata. Scorro in parallelo le due liste confrontando i loro primi elementi e scegliendo ogni volta il minore fra i due. Non faccio nulla se la seconda lista è vuota o se lo è la prima: in quest ultimo caso faccio in modo che il puntatore della prima lista (vuota) punti alla seconda (ha tutti gli elementi).

Riferimento $s1$ è passata per riferimento, $s2$ no. Questo perchè il primo puntatore ospiterà la lista finale. Nella funzione $s2$ non è mai *lvalue*, solo *rvalue*.

Funzionamento

- Confronto il primo elemento della lista $s1$ col primo elemento della lista $s2$
- Se il primo elemento di $s1$ è minore o uguale al primo elemento di $s2$ faccio merge tra la seconda lista e la sottolista di $s1$ formata dagli elementi successivi all'elemento di $s1$ che stiamo analizzando
- Se la condizione precedente non è soddisfatta significa che l'elemento più piccolo è il primo della seconda lista. Faccio merge tra ciò che rimane della seconda lista e la prima e faccio puntare il puntatore della prima lista alla seconda lista (ricordiamo che $s1$ punterà alla lista unita).

```
void merge(Elem*& lista1, Elem* lista2) {
        if(lista2 == NULL) return;
        if(lista1 == NULL) {
                lista1 = lista2;
                return;
        }

        if(lista1->inf <= lista2->inf)
                merge(lista1->next, lista2);
        else {
                merge(lista2->next, lista1);
                lista1 = lista2;
        }
}
```

Relazione di ricorrenza e complessità

$$\begin{cases} T(0) = d \\ T(n) = b + T(n-1) \end{cases}$$

Trovo che $T(n) \in O(n)$.

6.2.4 Esempio di applicazione

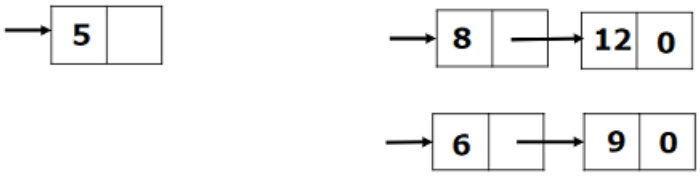
- Applichiamo la funzione alla seguente lista di elementi:

$$5\ \ 8\ \ 1\ \ 9\ \ 2\ \ 4\ \ 10\ \ 7$$

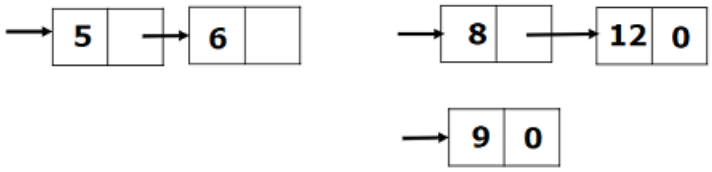
- Divido le liste (ricordare il metodo adottato per ottenere la seconda lista) fino ad ottenere atomi
  - * **Prima lista**: $5, 1, 2, 10$
    * **Seconda lista**: $7, 4, 9, 8$
  - * **Prima lista**: $5, 2$
    * **Seconda lista**: $10, 1$
    * **Terza lista**: $7, 9$
    * **Quarta lista**: $8, 4$
  - * **Prima lista**: 5
    * **Seconda lista**: 2
    * **Terza lista**: 10
    * **Quarta lista**: 1
    * **Quinta lista**: 7
    * **Sesta lista**: 9
    * **Settima lista**: 8
    * **Ottava lista**: 4
- Unifico le liste
  - * Unisco prima e seconda lista: $2, 5$
    * Unisco terza e quarta lista: $1, 10$
    * Unisco quinta e sesta lista: $7, 9$
    * Unisco settima e ottava lista: $4, 8$
  - * Unisco prima, seconda, terza e quarta: $1, 2, 5, 10$
    * Unisco quinta, sesta, settima e ottava: $4, 7, 8, 9$
  - Unisco le due sottoliste e ottengo quella definitiva: $1, 2, 4, 5, 7, 8, 9, 10$

6.2.4.1 Spiegazione più specifica della merge

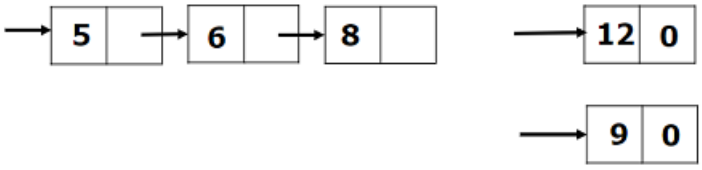
- Supponiamo di avere due liste e di volerle unificare (il fatto che una lista abbia un elemento in più è assolutamente lecito, in particolare se la lista iniziale da ordinare ha un numero dispari di elementi):
  - **Prima lista**: $5, 8, 12$
  - **Seconda lista**: $6, 9$
- Confronto il primo elemento (5) della prima lista col primo elemento (6) della seconda lista. Osservo che $5 < 6$, quindi il primo elemento della lista unita sarà 5



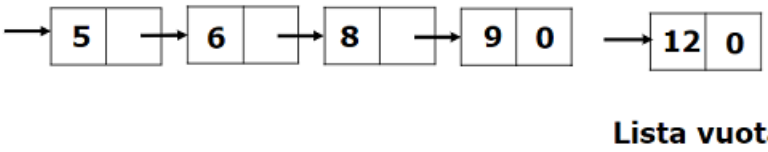
- Confronto il secondo elemento (8) della prima lista col primo elemento (6) della seconda lista. Osservo che $8 > 6$, quindi il secondo elemento della lista unità sarà 6.



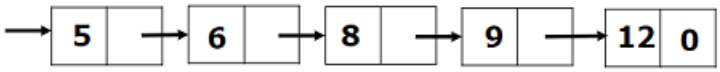
- Confronto il secondo elemento (8) della prima lista col secondo elemento (9) della seconda lista. Osservo che $8 < 9$, quindi il terzo elemento della lista unità sarà 8.



- Confronto il terzo elemento (12) della prima lista col secondo elemento (9) della seconda lista. Osservo che $12 > 9$, quindi il quarto elemento della lista unità sarà 9.



- La seconda lista è vuota. Rimane soltanto il terzo elemento (12) della prima lista. La lista finale è



Capitolo 7

Venerdì 27/03/2020

Registro 6 (*Ven 27/03/2020 11:30-13:30 (2:0 h) lezione*)

Alberi binari. Definizione. Memorizzazione. Visite. Alcuni programmi.

Oggi parliamo di una nuova struttura dati con i relativi algoritmi: gli alberi. Esistono vari tipi di alberi: oggi iniziamo con gli alberi binari. Gli alberi sono una struttura non lineare: non ho una semplice successione di elementi (come nella lista) ma una vera e propria gerarchia.

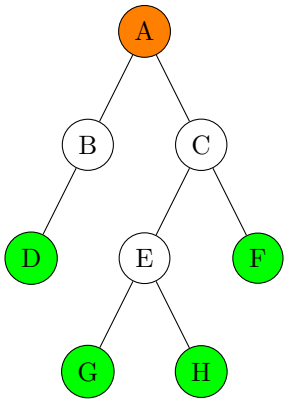
Rappresentazione grafica Un albero è rappresentato alla rovescia rispetto a un albero vero: il nodo più alto è detto **radice**!

7.1 Alberi binari

Gli alberi binari sono molto utilizzati. Un esempio in particolare è quello dei compilatori: dopo aver verificato la sintassi trasforma il nostro programma in un albero binario dove sono rappresentate tutte le istruzioni. Successivamente l'albero viene rivisitato e da quello si costruisce il codice macchina.

Definizione ricorsiva

- NULL è un albero binario
- Un **nodo** p più due alberi binari B_s e B_d formano un albero binario.

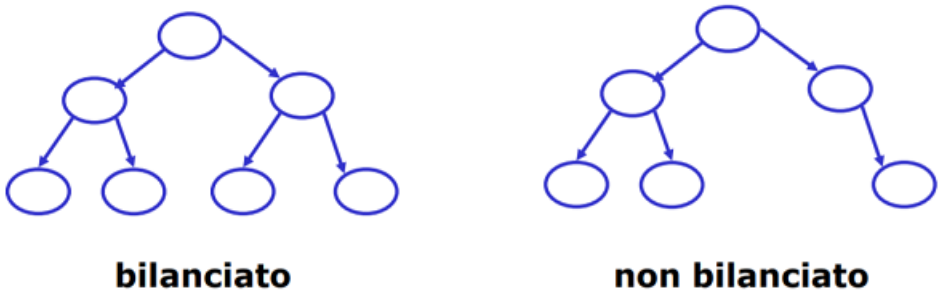


7.1.1 Definizioni

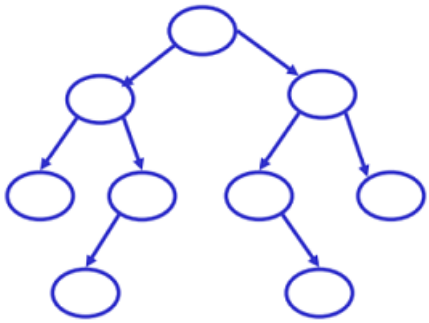
- **Nodo**: elemento costitutivo dell'albero. Ciascun nodo è etichettato.
- **Padre**: nodo a cui sono collegati al più due nodi detti **figli**. Per ogni padre posso avere al più due figli (caratteristica fondamentale dell'albero binario), che distinguo in **figlio sinistro** e **figlio destro**.
- **Antecedenti**: i nodi che si incontrano per raggiungere un certo nodo. L'unico nodo che non ha antecedenti è la radice.
- **Foglia**: nodo che ha entrambi i sottoalberi vuoti.
- **Discendenti**: tutti i nodi sotto un certo nodo. La cosa non è lineare come con gli antecedenti. Gli unici nodi che non hanno discendenti sono le foglie.
- **Livello di un nodo**: numero dei suoi antecedenti. Il livello della radice è il livello zero.
- **Livello dell'albero**: massimo fra i livelli dei suoi nodi. Raggiungere questo livello significa svolgere il percorso più lungo all'interno dell'albero.

7.2 Alberi binari particolari

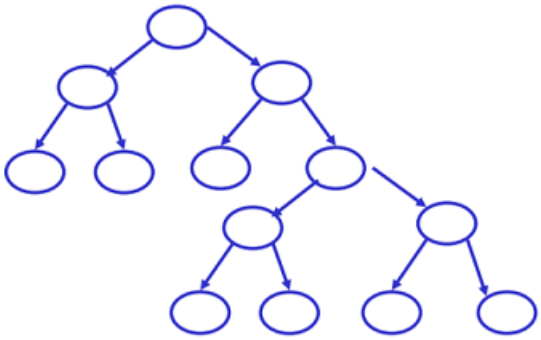
- **Albero binario bilanciato**: un albero binario bilanciato presenta tutti i nodi, ad eccezione di quelli all'ultimo livello, sia con figlio sinistro che con figlio destro. Il sottoalbero sinistro e destro della radice presentano lo stesso numero di nodi! Un albero bilanciato con livello dell'albero k presenta $2^{(k+1)} - 1$ nodi e 2^k foglie. In alcuni casi particolari ottengo complessità migliore applicando le regole del *divide et impera*.



- **Albero binario quasi bilanciato**: un albero binario quasi bilanciato è un albero binario bilanciato fino al penultimo livello. Questo significa che troverò le fogli sull'ultimo e sul penultimo livello. I cammini dalla radice a una foglia possono differenziarsi al massimo di un passo. Un albero quasi bilanciato ha un numero di nodi $\in [2^k; 2^{(k+1)} - 1]$ e un numero di foglie $\in [2^{(k-1)}; 2^k]$.



- **Albero pienamente binario**: un albero binario presenta tutti i nodi, ad eccezione delle foglie, sia con figlio sinistro che con figlio destro. In un albero di questo tipo il numero di nodi (interni) equivale al numero di foglie -1.



Albero pienamente binario $\neq$ Albero binario bilanciato

7.2.1 Scrivere un programma su un albero binario

La struttura dati degli alberi risulta meglio gestibile attraverso algoritmi ricorsivi. In ogni algoritmo consideriamo

- il **caso base**, quando l'albero è vuoto;
- il **caso ricorsivo**, quando l'albero è costituito da una radice più due sottoalberi.

Si ha una *struct* avente tra i membri due puntatori: uno al figlio sinistro e uno al figlio destro.

7.2.2 Perchè programmiamo ricorsivamente?

Il programma risulta più elegante e facile da scrivere. Proviamo a scrivere una funzione *preOrder* in modo iterativo utilizzando una classe per le pile della *STL* (per salvare i vari nodi).

```
void preOrder(Node* tree) {
        stack<Node*> miapila(100);
        for (;;) {
             while (tree) {
                <esamina tree->label>;
                miapila.push(tree);  tree=tree->left;
             }
             if (miapila.empty()) return;    tree=miapila.pop()->right;
        }
}
```

Non c'è bisogno di commentare.

Riflessione L'algoritmo iterativo occupa più memoria con lo stack? No, nell'algoritmo ricorsivo abbiamo le varie chiamate con conseguente occupazione di memoria.

7.3 Linearizzazione dell'albero

Linearizzare significa scorrere (visitare) i vari nodi dell'albero ponendoli, in un certo senso, come se fossero elementi di una lista. Esistono tre modi per linearizzare un albero: in ciascuno varia l'ordine di esaminazione dei vari nodi.

Riferimento Prendiamo come riferimento, nel confronto tra le varie visite, l'albero presente nella pagina precedente.

7.3.1 Visita anticipata (*preOrder*)

```
void preOrder(const nodo* n) {
    if(n) {
        /* Analisi nodo */

        preOrder(n->left);
        preOrder(n->right);
    }
}
```

Spiegazione

- Se l'albero è vuoto non ho nulla da visitare
- Se non è vuoto:
  - Esamino la radice
  - Visito il sottoalbero sinistro
  - Visito il sottoalbero destro
- **Ottengo**: A B D C E G H F
  Il risultato è intuitivo: ogni volta che chiamo la funzione per un nodo non nullo l'analizzo subito. La disposizione delle funzioni fa sì che prima analizzi i sottoalberi sinistri e dopo, come se si tornasse indietro dalla foglia, i sottoalberi destri.

7.3.2 Visita differita (*postOrder*)

```
void postOrder(const nodo* n) {
    if(n) {
        postOrder(n->left);
        postOrder(n->right);

        /* Analisi nodo */
    }
}
```

Spiegazione

- Se l'albero è vuoto non ho nulla da visitare
- Se non è vuoto:
  - Visito il sottoalbero sinistro
  - Visito il sottoalbero destro
  - Esamino la radice
- **Ottengo**: D B G H E F C A
  Osservando la disposizione delle chiamate di funzione vedo che analizzerò un nodo dopo aver visto i nodi di un sottoalbero destro (se presenti). I primi due valori sono facili da intuire, considerando che nel sottoalbero sinistro alla radice dell'albero non si hanno sottoalberi destri. Passo a G, ipotizzando un sottoalbero destro rispetto a G non esistente; poi ad H, sottoalbero destro di E; poi ad E; dopo ad F, sottoalbero destro di C; ancora dopo a C, completando il sottoalbero destro di A; infine A.

7.3.3 Visita simmetrica (*inOrder*)

```
void inOrder(const nodo* n) {
    if(n) {
        inOrder(n->left);

        /* Analisi nodo */

        inOrder(n->right);
    }
}
```

Spiegazione

- Se l'albero è vuoto non ho nulla da visitare
- Se non è vuoto:
  - Visito il sottoalbero sinistro
  - Esamino la radice
  - Visito il sottoalbero destro
- **Ottengo**: D B A G E H C F
  Termino di scorrere i sottoalberi sinistri ed analizzo il nodo. Dopo averlo analizzato passo a quelli destri. Parto da D, foglia del sottoalbero sinistro, poi B ed A; dopo riparto da G, altra foglia di un sottoalbero sinistro, salgo ad E e poi analizzo il sottoalbero destro avente come unico nodo H; vado a C e infine analizzo il sottoalbero destro avente come unico elemento F.

7.3.4 Complessità

Se l'albero è vuoto ho complessità costante, negli altri casi considero le chiamate di funzione per i nodi sinistri e per i nodi destri. Ottengo la seguente relazione di ricorrenza

$$\begin{cases} T(0) = a \\ T(n) = b + T(n_s) + T(n_d) \;\; con\ n_s + n_d = n - 1\ con\ n > 0 \end{cases}$$

Questa relazione è diversa da quelle solite e non siamo in grado di risolverla con le regole che conosciamo. Un caso particolare potrebbe essere avere un albero bilanciato, cioè avere lo stesso numero di nodi sia a sinistra che a destra, dato un qualunque nodo. Ciò significa dividere ogni volta i discendenti di un nodo in due parti. La relazione che otteniamo è la seguente

$$\begin{cases} T(0) = a \\ T(n) = b + 2T((n-1)/2) \end{cases}$$

Questa relazione la sappiamo risolvere: $T(n) \in O(n)$. Negli appunti è presente una dimostrazione (non va saputa, cit) che conferma la stessa complessità anche per alberi non bilanciati.

Perchè le visite sono convenienti? I nodi vengono visitati una volta soltanto! Se io volessi analizzare i nodi nell'ordine alfabetico senza alterare l'albero preso ad esempio dovrei scorrere diversi nodi tante volte attraverso un programma con complessità sicuramente superiore a $O(n)$.

7.3.5 Complessità delle visite in funzione dei livelli (con albero bilanciato)

$$\begin{cases} T(0) = a \\ T(k) = b + 2T(k-1) \end{cases}$$

La complessità in funzione dei livelli è esponenziale! $T(k) \in O(2^k)$.

7.4 Programma ricorsivo per contare i nodi

```
int nodes(Node* tree) {
        if(!tree) return 0;
        return 1 + nodes(tree->left) + nodes(tree->right);
}
```

Spiegazione

- Se l'albero è vuoto ho zero nodi (caso base) e restituisco zero.
- Se l'albero non è vuoto restituisco la somma tra 1, il numero di nodi del sottoalbero sinistro e il numero di nodi del sottoalbero destro (passo ricorsivo)

Complessità La relazione di ricorrenza è la stessa della visita, quindi $T(n) \in O(n)$.

7.5 Programma ricorsivo per contare le foglie

```
int leaves(Node* tree) {
        if(!tree) return 0;
        if(!tree->left && !tree->right) return 1;
        return leaves(tree->left) + leaves(tree->right);
}
```

Spiegazione

- Se l'albero è vuoto ho zero foglie (caso base) e restituisco zero.
- Se il puntatore al sottoalbero sinistro e al sottoalbero destro sono a NULL ho trovato una foglia (altro caso base) e restituisco uno
- Negli altri casi sommo le due chiamate ricorsive, cioè sommo il numero di foglie del sottoalbero sinistro al numero di foglie del sottoalbero destro (passo ricorsivo)

Complessità La relazione di ricorrenza è la stessa della visita, quindi $T(n) \in O(n)$.

7.6 Ricerca di un'etichetta in alberi binari

Si restituisce il puntatore al nodo che contiene l'etichetta n. Se l'etichetta non compare nell'albero ritorna NULL. Se più nodi contengono n, ritorna il primo nodo che si incontra facendo la visita anticipata.

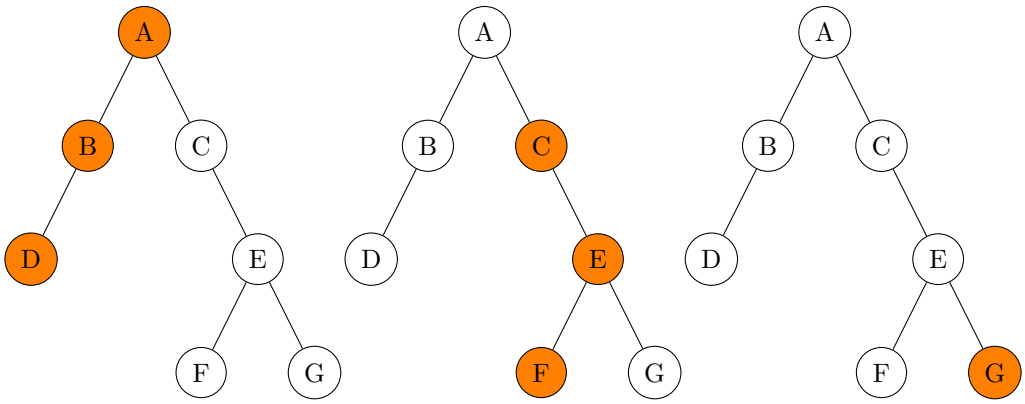
```
Node* findNode(Label n, Node* tree) {
        if(!tree) return NULL;
        if(tree->label == n) return tree;

        Node* a = findNode(n, tree->left);
        if(a) return a;
        else return findNode(n, tree->right);
}
```

Spiegazione

- Se l'albero è vuoto restituisco NULL
- Se la radice è uguale a n restituisco l'indirizzo e fermo il programma
- Se il nodo non corrisponde continuo la ricerca:
  - Se ho trovato il nodo restituisco l'indirizzo e ho finito
  - Se non l'ho trovato devo andare a cercare nel sottoalbero destro e quello che troveremo lì sarà il risultato!
  - Importante salvare l'indirizzo nel puntatore a per rendere il programma più efficiente: nei fatti la complessità non varia perchè il tempo di esecuzione di entrambe le chiamate è lo stesso.

7.6.1 Esempio

Ricerchiamo nell'albero raffigurato nelle varie immagini il nodo con etichetta G.



1. Inizialmente si scorre il ramo sinistro dell'albero: arrivo a D senza aver trovato nulla.
2. Passo al sottoalbero destro: scorro fino ad E i figli destri (A e C non hanno figli sinistri). Quando raggiungo E passo al figlio sinistro
3. Non ho ancora trovato ciò che mi interessa, quindi analizzo il figlio destro di E. Individuo finalmente l'etichetta desiderata e restituisco l'indirizzo del nodo in questione.

Ricordare: la funzione verifica ogni volta il sottoalbero sinistro del nodo sotto analisi, se questo sottoalbero non restituisce nulla di interessante (quindi NULL) si passa al sottoalbero destro.

7.7 Eliminazione di un albero

Per cancellare un albero potrebbe essere sufficiente porre $tree = NULL$ senza scorrere tutto l'albero. Il nostro interesse, tuttavia, è utilizzare l'operatore delete su ogni nodo per liberare spazio e permettere al C++ di utilizzare celle inutilizzate.

```
void delTree(Node* &tree) {
        if(tree) {
                delTree(tree->left);
                delTree(tree->right);
                delete tree;
                tree = NULL;
        }
}
```

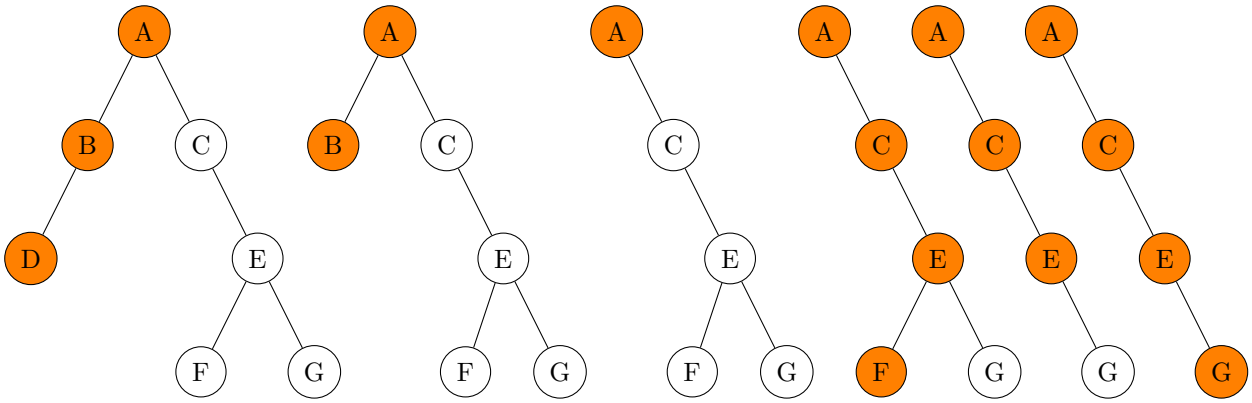
Spiegazione

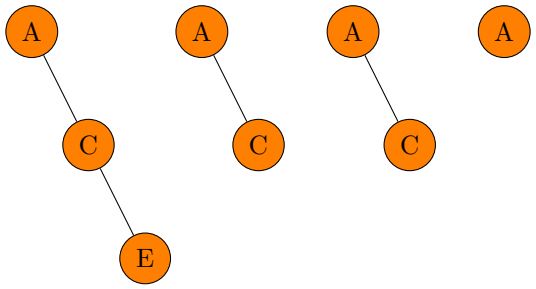
- L'albero viene passato per riferimento, poichè modifichiamo gli indirizzi
- Se l'albero è vuoto non dobbiamo fare niente
- Se l'albero non è vuoto:
  - Cancello il sottoalbero sinistro
  - Cancello il sottoalbero destro
  - Cancello la radice e pongo il puntatore alla radice a NULL

Visita La visita adottata è la postOrder (intuibile dalla disposizione delle istruzioni). Le altre visite non sono corrette poichè non potrei più accedere al sottoalbero sinistro o destro. Nella preOrder ho il problema con entrambi i sottoalberi, nella inOrder solo con il sottoalbero destro.

7.7.1 Esempio

Analizziamo le varie fasi di eliminazione dei nodi che costituiscono l'albero.





7.8 Inserire un nodo in un albero binario

Il programma inserisce un nodo (*son*) come figlio di *father*, sinistro se $c =' l'$, destro se $c =' r'$. Restituisce 1 se l'operazione ha successo, 0 in caso di fallimento. Se l'albero è vuoto inserisce il nodo come radice.

```
int insertNode(Node*& tree, Label son, Label father, char c) {
        if(!tree) {
                tree = new Node;   tree->label = son;
                tree->left = tree->right = NULL;
                return 1;
        }

        Node* a = findNode(father, tree);
        if(!a) return 0;
        if(c == 'l' && !a->left) {
                a->left = new Node;   a->left->label = son;
                a->left->left = a->left->right = NULL;
                return 1;
        }

        if(c == 'r' && !a->right) {
                a->right = new Node;   a->right->label = son;
                a->right->left = a->right->right = NULL;
                return 1;
        }
        return 0;
}
```

Spiegazione

- Se l'albero è vuoto aggiungo il nodo come radice e restituisco 1 (l'operazione ha avuto successo)
- Se l'albero non è vuoto:
  - Svolgo una ricerca all'interno dell'albero mediante la funzione findNote spiegata nella sezione precedente
  - Se la funzione restituisce NULL significa che non ho trovato corrispondenze, quindi l'operazione è fallita e restituisco 0
  - Se salto lo step precedente ho trovato corrispondenze, quindi inserisco il nodo come figlio destro o sinistro del padre trovato. Se ciò avviene l'operazione ha avuto successo e restituisco 1.
  - Se dopo aver trovato corrispondenze non ho effettuato aggiunte restituisco 0. Questo può succedere se il valore c è diverso dai valori indicati (l ed r) o se il padre ha il puntatore richiesto già occupato.

Capitolo 8

Martedì 31/03/2020

Registro 7 (*Mar 31/03/2020 10:30-13:30 (3:0 h) lezione*)

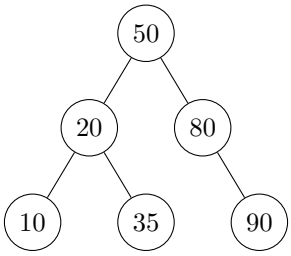
Alberi binari di ricerca. Limiti inferiori mediante alberi di decisione. Counting sort. Radix sort. Esercizi su alberi binari.

Continuiamo sugli alberi binari seguendo un percorso diverso da quello delle diapositive.

8.1 Alberi binari di ricerca

Un albero binario di ricerca è un albero binario che presenta proprietà aggiuntive sul contenuto dei vari nodi. Dato un qualunque nodo p:

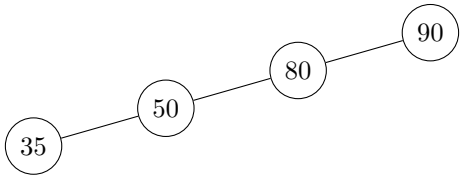
- i nodi del sottoalbero sinistro di p hanno etichetta minore di quella di p
- i nodi del sottoalbero destro di p hanno etichetta maggiore di quella di p



Attenzione a domande nei compiti Il principio appena introdotto non vale solo per la radice, ma per ogni singolo nodo presente nell'albero. Scrivere "radice" e basta è estremamente sbagliato.

Non univocità Non esiste un albero binario univoco che rappresenta un particolare insieme di elementi. Si possono avere più versioni

Alberi binari degeneri In certi casi gli alberi binari diventano degeneri. Nel seguente, per esempio, abbiamo una serie di elementi che vanno a formare una lista.



Proprietà

- Non sono presenti doppioni
- Con la visita simmetrica *inOrder* le etichette vengono elencate in ordine crescente

Operazioni possibili Le operazioni possibili sono:

- Ricerca
- Inserimento
- Eliminazione

8.1.1 Ricerca

La ricerca da il nome a questa versione particolare degli alberi. Voglio individuare un nodo con una certa etichetta all'interno dell'albero. Prima guardo nella radice, se non è lì tengo conto delle proprietà dell'albero binario di ricerca e continuo la ricerca analizzando soltanto uno dei sottoalberi della radice.

```
Node* findNode(Label n, Node* tree) {
        if(!tree) return 0;
        if(n == tree->label) return tree;

        if(n < tree->label)
                return findNode(n, tree->left);

        return findNode(n, tree->right);
}
```

Funzionamento

- Se l'albero è vuoto restituisco 0
- Se ho corrispondenza tra etichetta del nodo ed etichetta indicata restituisco il puntatore della radice
- Se non ho corrispondenza
  - Se l'etichetta è minore faccio ricorsione a sinistra
  - Se l'etichetta è maggiore faccio ricorsione a destra

Complessità della ricerca Se l'albero è vuoto la complessità è costante, se l'albero non è vuoto si ha il caso buono (corrispondenza) o il caso cattivo (non corrispondenza). Come argomento della funzione si ha un valore $k < n$ (noi non sappiamo quanti nodi formano il sottoalbero)

$$\begin{cases} T(0) = a \\ T(n) = b + T(k) \quad k < n \end{cases}$$

Il caso migliore è il seguente, in cui l'albero binario di ricerca è anche bilanciato (quindi ho, più o meno, lo stesso numero di nodi sia a destra che a sinistra)

$$\begin{cases} T(0) = a \\ T(n) = b + T(n/2) \end{cases} \qquad O(\log n)$$

Il caso cattivo è la forma degenere vista prima (o comunque i casi in cui i sottoalberi sono estremamente sbilanciati)

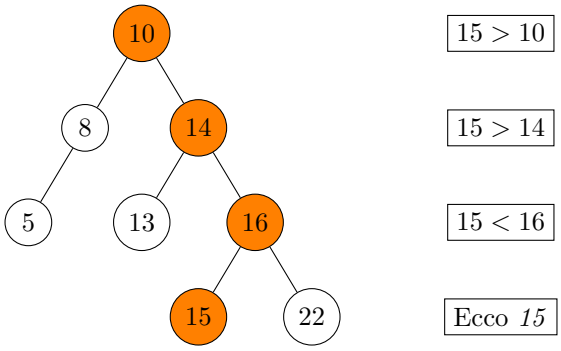
$$\begin{cases} T(0) = a \\ T(n) = b + T(n-1) \end{cases} \qquad O(n)$$

Alla fine la complessità media è $O(\log n)$, poichè i casi peggiori hanno minore probabilità di manifestarsi.

Vantaggio Se ho bisogno di una ricerca veloce quanto visto è estremamente utile. Tuttavia dobbiamo tener conto della complessità delle altre operazioni.

8.1.1.1 Esempio

Ricerchiamo nell'albero un nodo con etichetta 15.



8.1.2 Inserimento

Voglio inserire un nodo all'interno di un albero binario di ricerca. Confronto l'etichetta del nodo con il nodo della radice: se è uguale mi fermo e non faccio niente poichè non sono permessi doppioni, se non è uguale ricerco un albero vuoto a destra o a sinistra (in base al valore dell'etichetta).

```
void insertNode(Label n, Node*& tree) {
        if(!tree) {
                tree = new Node;
                tree->label = n;
                tree->left = tree->right = NULL;
                return;
        }
        if(n < tree->label)
                insertNode(n, tree->left);
        if(n > tree->label)
                insertNode(n, tree->right);
}
```

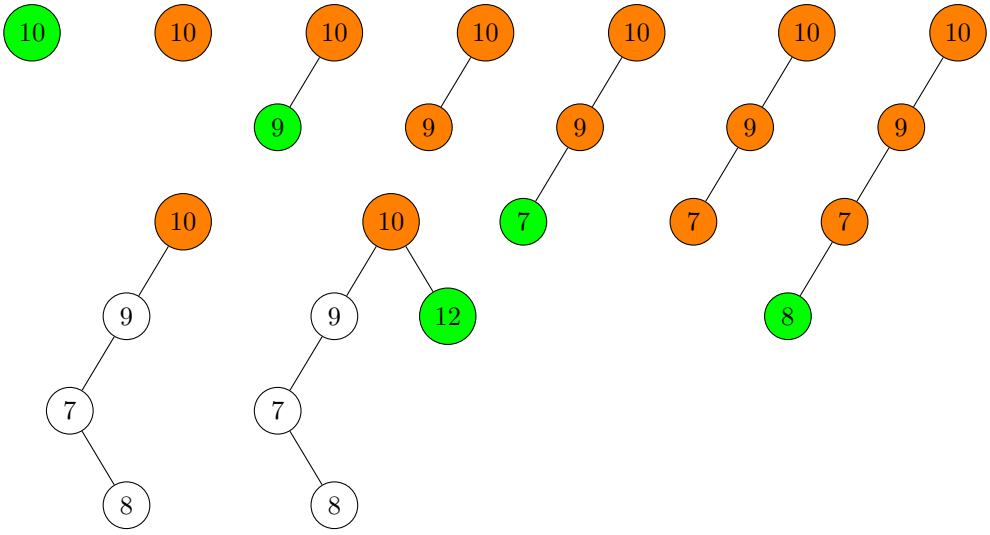
Funzionamento

- Se l'albero è vuoto creo il nodo
- Se l'albero non è vuoto:
  - Se il valore dell'etichetta è minore del nodo radice continuo a ricercare l'albero vuoto a sinistra
  - Se il valore dell'etichetta è maggiore del nodo radice continuo a ricercare l'albero vuoto a destra
  - Se il valore dell'etichetta è uguale mi fermo poichè i doppioni non sono ammessi

Complessità Valgono gli stessi discorsi fatti prima. Quindi ho complessità media di $O(\log n)$.

8.1.2.1 Esempio

Vediamo l'aggiunta a un albero vuoto dei numeri $10, 9, 7, 8, 12$ (nell'ordine indicato)



8.1.3 Eliminazione

L'eliminazione risulta un procedimento decisamente più complesso: oltre ad eliminare un nodo dobbiamo fare in modo che le proprietà tipiche dell'albero binario di ricerca non vengano violate. Procediamo nel seguente modo:

1. Verifico l'esistenza del nodo analizzato: fermo l'esecuzione dell'istanza se arrivo alla fine di un ramo senza aver trovato nulla o se ho un albero vuoto
2. Se scorrendo un ramo non c'è corrispondenza tra la mia etichetta e quella del nodo analizzato continuo il percorso muovendomi nel sottoramo sinistro o nel sottoramo destro (in base al valore delle etichette)
3. Se ho corrispondenza:
   - Verifico se uno dei due sottoalberi è inesistente. In questo caso l'unico sottoalbero presente del nodo da eliminare diventa sottoalbero del padre dello stesso. Sarà sottoalbero dello stesso tipo del nodo eliminato (sinistro o destro)
   - Se esistono due sottoalberi ricerco in quello destro il nodo avente etichetta minore: lo cancello e sostituisco l'etichetta del nodo con corrispondenza con quella di questo nodo. Il ragionamento è giusto poichè la nuova etichetta posta al nodo con corrispondenza risulterà sicuramente maggiore di quelle presenti nel sottoalbero sinistro e minore di quelle presenti nel sottoalbero destro.

Il numero di operazioni richieste ci porta a dividere il tutto in due funzioni.

Complessità Anche *deleteNode* ha complessità logaritmica $O(\log n)$.

8.1.3.1 *deleteNode*

Scorro ricorsivamente l'albero. Verifico l'esistenza del nodo per fermarmi in caso di albero vuoto o termine di rami, decido quali sottoalberi scorrere in base ai valori delle etichette, in caso di corrispondenza verifico se si ha un solo sottoalbero o due sottoalberi. In quest'ultimo caso chiamo la funzione *deleteMin* per individuare il minimo nel sottoalbero destro ed effettuare lo scambio di etichette.

```
void deleteNode(const int num, nodo*& n) {
    if(n) {
        if(num < n->num) deleteNode(num, n->left);
        else if(num > n->num) deleteNode(num, n->right);
        else if(!n->left) {
                nodo* a = n;  n = n->right;  delete a;
        }
        else if(!n->right) {
                nodo* a = n;  n = n->left;  delete a;
        }
        else deleteMin(n->right, n->num);
    }
}
```

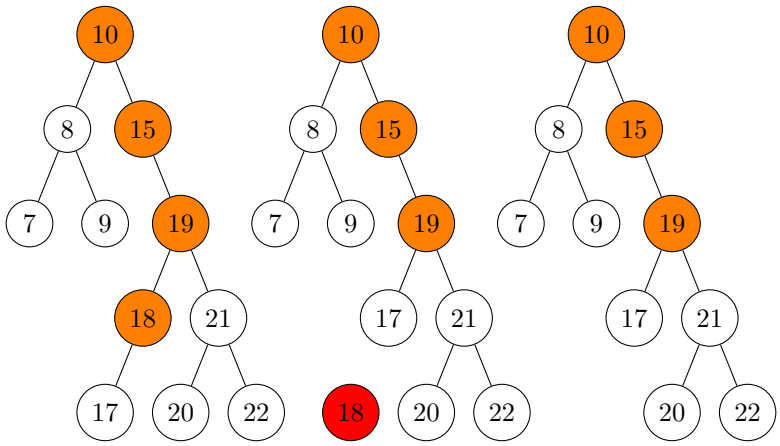
8.1.3.2 *deleteMin* - utilizzata in *deleteNode*

Per individuare il nodo giusto nel sottoalbero è necessario andare "più a sinistra possibile": ogni volta che esiste un sottoalbero sinistro scorro quello. Questo perchè ogni sottoalbero sinistro contiene etichette con valore minore rispetto al nodo analizzato. Lo scorrimento del sottoalbero termina quando si individua un nodo privo di sottoalbero sinistro (quello che cercavamo).

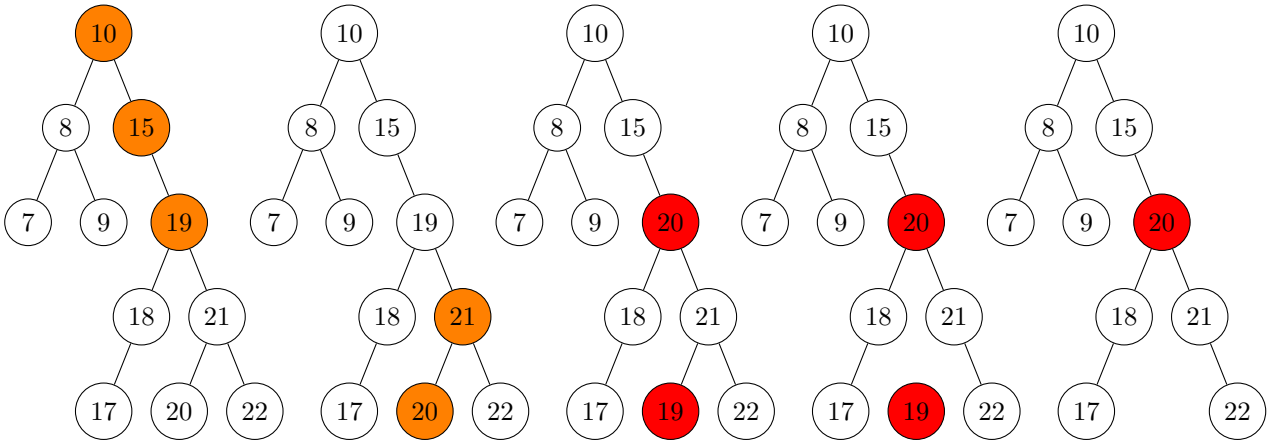
```
void deleteMin(nodo*& n, int& num) {
    if(n->left)
        deleteMin(n->left, num);
    else {
        num = n->num; // Modifico etichetta (int&)
        nodo* a = n;  n = n->right;
        delete a; // Elimino nodo con etichetta min
    }
}
```

8.1.3.3 Esempi

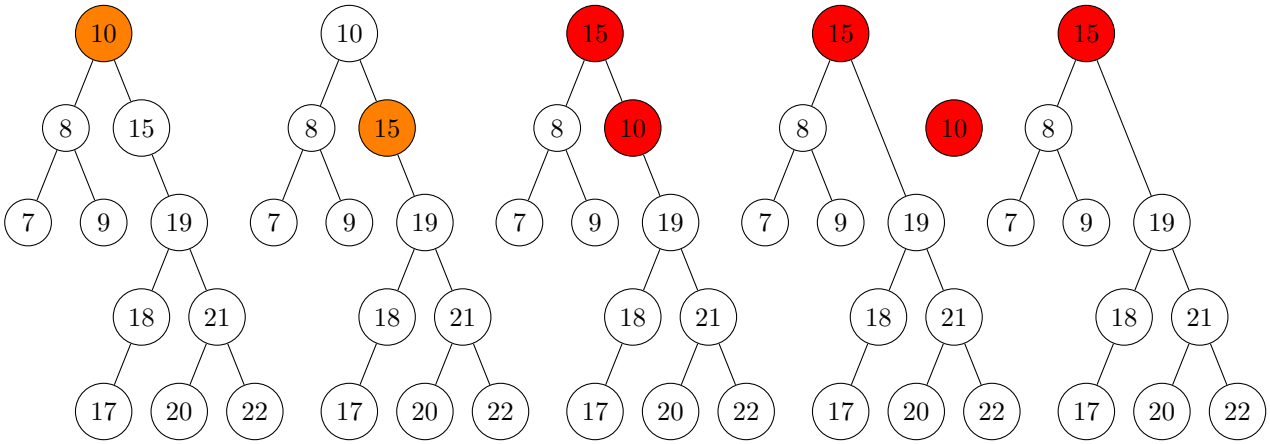
Rimozione di 18 **con un solo ramo**



Rimozione di 19 con due rami



Rimozione di 10 - radice



8.2 Limiti inferiori per i problemi

Noi abbiamo dato una definizione di Ω-grande in riferimento alle funzioni. Il concetto di limite inferiore può essere applicato anche ai problemi!
Si dice che un problema è di ordine $\Omega(f(n))$ se non è possibile trovare un algoritmo che lo risolva con complessità minore di $f(n)$: questo significa che tutti gli algoritmi che lo risolvono hanno complessità $\Omega(f(n))$.

Ordinamento Il concetto è interessante. Prendiamo gli algoritmi di ordinamento e chiediamoci: si possono creare algoritmi con complessità migliore di $O(n \log n)$? Vedremo che esistono algoritmi con migliore complessità.

8.2.1 Alberi di decisione

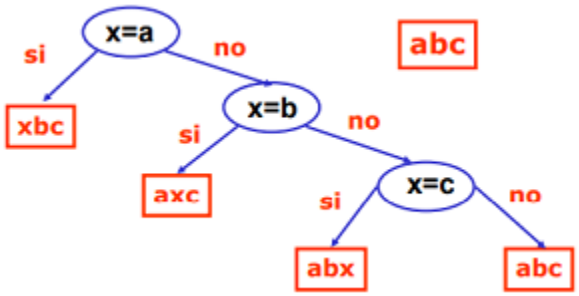
Nell'informatica è facile dire che una cosa può essere fatta, meno facile dire che non può essere fatta! Per dimostrare che non è possibile ottenere migliore complessità possiamo ricorrere, in certi casi, al metodo degli *alberi di precisione*.

Requisiti Il metodo può essere utilizzato solo con programmi basati su confronti che hanno complessità proporzionale al numero di confronti effettuati durante l'esecuzione dell'algoritmo. Se prendiamo come riferimento gli algoritmi di ordinamento individuiamo che effettivamente non è possbile ottenere un ordinamento con migliore complessità attraverso meccanismi basati sul confronto.

Elementi di un albero

- **Foglia**: ogni foglia rappresenta una possibile soluzione del problema
- **Cammino**: ogni cammino dalla radice ad una foglia rappresenta una possibile esecuzione dell'algoritmo per giungere alla soluzione

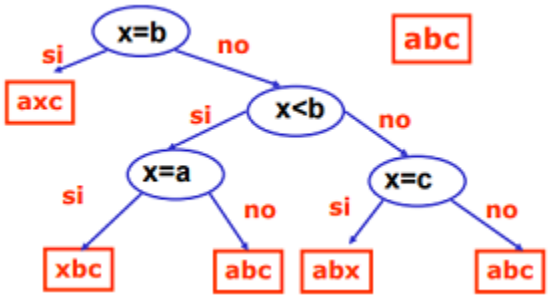
Ricerca lineare Supponiamo che il mio array sia formato dagli elementi a, b, c, voglio trovare il valore di x nell'array.



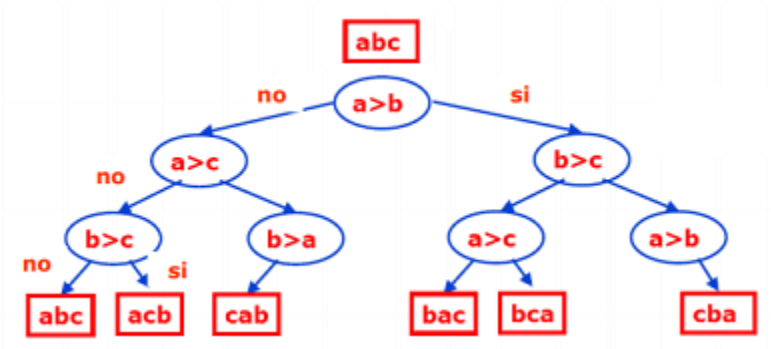
- Se $x = a$ ho finito trovando una soluzione, altrimenti continuo
- Se $x = b$ ho finito trovando una soluzione, altrimenti continuo
- Se $x = c$ ho finito trovando una soluzione, altrimenti ho finito senza aver trovato corrispondenze.

Per ogni nodo si ha un confronto e due possibili uscite. Dati tre elementi osservo che ho quattro foglie, e quindi n+1 soluzioni possibili.

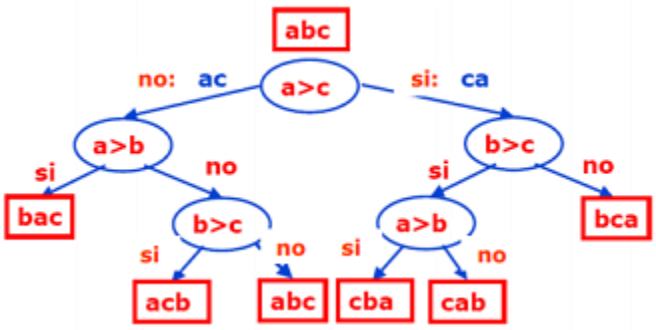
Ricerca binaria Oltre ad uguaglianze verifico disuguaglianze. Se c'è corrispondenza mi fermo, se non c'è verifico una disuguaglianza e poi di nuovo un'uguaglianza. In questo caso ho cinque soluzioni possibili.



selectionSort Abbiamo un array di tre elementi. Abbiamo una serie di confronti in cui vogliamo individuare il valore minimo o massimo di un certo intervallo. Osservo che con tre elementi posso avere sei soluzioni possibili.



mergeSort Abbiamo sempre un array di tre elementi. Il numero di soluzioni è sempre lo stesso: con qualsiasi *sort* avrò sempre $n!$ soluzioni.

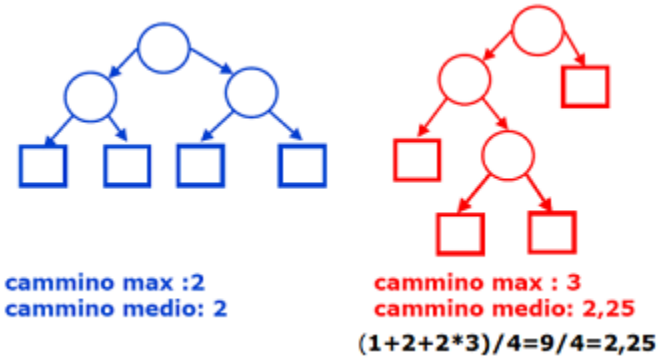


Come si confrontano due alberi? Ho due algoritmi che risolvono lo stesso problema e che hanno lo stesso numero di foglie. Osservo il caso peggiore, quindi il percorso più lungo possibile: un algoritmo ottimo è quello che ha il più corto cammino medio dalla radice alle foglie.

Come deduciamo il tempo? La lunghezza del cammino determina il tempo di esecuzione necessario. Analizzando i cammini medi degli algoritmi di ordinamento citati si vede che nel *mergeSort* si hanno due casi con due confronti e altri casi con tre confronti, mentre nel *selectionSort* si fanno sempre tre confronti. Maggiore è il numero di elementi, maggiore è il numero di cammini con percorso minore: segue la maggiore efficienza del *mergeSort* rispetto al *selectionSort*.

Osservazioni Bisogna ridurre la profondità degli alberi: più gli alberi sono ridotti, più l'algoritmo è buono. Dobbiamo tener conto che:

- Un albero binario con k livelli ha al più 2^k foglie (si hanno 2^k foglie quando l'albero è bilanciato)
- Se ho un albero binario con s foglie non potrò avere meno di $\log_2 s$ livelli.
- Gli alberi, se bilanciati, minimizzano sia il caso peggiore che quello medio: si hanno $\log s(n)$ livelli. **Meglio non si può fare**.
- Dobbiamo tener conto dell'equiprobabilità: le conclusioni derivanti dagli alberi sono perfette se tutte le soluzioni presentano la stessa probabilità. La maggiore frequenza di una certa soluzione potrebbe portarmi a scegliere un algoritmo invece di un altro.



Conclusione sugli algoritmi di ordinamento L'algoritmo di ordinamento ha sempre $n!$ soluzioni, quindi avremo $\log(n!) = \boxed{n \log n}$. Un algoritmo basato sui confronti con numero di confronti uguale ad $n \log n$ è un algoritmo ottimo. Segue che *mergeSort* è ottimo in ogni caso mentre *quickSort* solo nel caso medio. *selectionSort* non è ottimo.
Vediamo adesso degli algoritmi con complessità minore di $O(n \log n)$: essi non sono basati sul confronto e non possono essere applicati ad ogni caso (presenti dei vincoli).

8.3 *countingSort* - Algortmo di ordinamento

Il *countingSort* mi permette di ordinare una sequenza di interi. L'algoritmo può essere utilizzato soltanto se conosciamo il valore minimo e il valore massimo degli elementi da ordinare. La fattibilità dell'algoritmo dipende anche dal numero di elementi distinti presenti: è efficiente se devo ordinare un array di mille elementi con 200 numeri distinti, sconsigliato se i numeri distinti sono 900. Fondamentalmente:

- Conto quante volte un certo valore è presente nell'array salvando il conteggio in un array ausiliario
- Successivamente ordino i valori tenendo conto dell'array ausiliario.

Abbiamo analizzato un *countingSort* dove il valore minimo è sempre zero: negli argomenti della funzione indichiamo il puntatore ad array, il valore massimo e il numero di elementi presenti nell'array.

- Col primo for pongo tutti gli elementi dell'array ausiliario uguali a zero.
- Col secondo for effettuo il conteggio delle presenze di ogni elemento all'interno dell'array
- Successivamente andiamo a "sovrascrivere" l'array. Gli indici, che rappresentano i valori possibili nell'array, sono già ordinati: Scorrendo l'array ausiliario pongo ogni numero tante volte quante indicate dai conteggi.

```
void counting_sort(int A[], int k, int n) {
        int i, j;  int C[k+1];

        for(i = 0; i <= k; i++) C[i] = 0;
        for(j = 0; j < n; j++) C[A[j]]++;

        j = 0;
        for(i = 0; i <= k; i++) {
                while(C[i] > 0) {
                        A[j] = i; C[i]--; j++;
                }
        }
}
```

Nell'ultimo for Con j scorro l'array da ordinare, con i scorro l'array ausiliario.

Debug testuale

```
Array iniziale
[ 7 7 4 4 7 5 4 7 4 5 1 1 0 1 ]

Numero: 0   Presenze: 1
[ 0 7 4 4 7 5 4 7 4 5 1 1 0 1 ]

Numero: 1   Presenze: 3
[ 0 1 1 1 7 5 4 7 4 5 1 1 0 1 ]

Numero: 2   Presenze: 0
[ 0 1 1 1 7 5 4 7 4 5 1 1 0 1 ]

Numero: 3   Presenze: 0
[ 0 1 1 1 7 5 4 7 4 5 1 1 0 1 ]

Numero: 4   Presenze: 4
[ 0 1 1 1 4 4 4 4 4 5 1 1 0 1 ]

Numero: 5   Presenze: 2
[ 0 1 1 1 4 4 4 4 5 5 1 1 0 1 ]

Numero: 6   Presenze: 0
[ 0 1 1 1 4 4 4 4 5 5 1 1 0 1 ]

Numero: 7   Presenze: 4
[ 0 1 1 1 4 4 4 4 5 5 7 7 7 7 ]
```

Complessità Si ha una complessità $O(n + k)$ dove n consiste nel numero totale di elementi presenti nell'array, k nel numero di elementi distinti. La complessità non è legata al confronto ma ad un conteggio ed è minore di $O(n \log n)$: ciò non è un problema poichè il metodo degli alberi riguarda soltanto algoritmi basati sul confronto. L'algoritmo necessita di memoria ausiliaria e può essere usato anche per ordinare sequenze di caratteri.

8.4 *radixSort* - Algoritmo di ordinamento

Anche il *radixSort* è utilizzabile per ordinare una sequenza di interi. Dobbiamo conoscere la lunghezza massima d dei numeri da ordinare: se ho, per esempio, un array con numeri compresi fra 50 e 220 la lunghezza massima sarà 3. Fondamentalmente:

- Effettuo d scorrimenti ripartendo i numeri ogni volta in k contenitori, dove k sono i possibili valori di una cifra.
- Prendiamo il caso con $d = 3$:
  - **Prima passata**: Inserisco i numeri nei contenitori in base al valore delle unità
  - **Seconda passata**: Inserisco i numeri nei contenitori in base al valore delle decine
  - **Terza passata**: Inserisco i numeri nei contenitori in base al valore delle centinaia
- Ogni volta estraggo i numeri dai contenitori: dal primo all'ultimo contenitore, dal primo elemento all'ultimo elemento inserito in ciascun contenitore.

Esempio concreto Abbiamo l'array formato da i seguenti elementi

$$190, 051, 054, 207, 088, 010$$

Abbiamo $k = 10, d = 3$.

- **Prima passata**

010									
190	**051**			**054**			**207**	**088**	
0	1	2	3	4	5	6	7	8	9

Ottengo [**190**, **010**, **051**, **054**, **207**, **088**]

- **Seconda passata**

					054				
207	**010**				**051**			**088**	**190**
0	1	2	3	4	5	6	7	8	9

Ottengo [**207**, **010**, **051**, **054**, **088**, **190**]

- **Terza passata**

088									
054									
051									
010	**190**	**207**							
0	1	2	3	4	5	6	7	8	9

Ottengo [**010**, **051**, **054**, **088**, **190**, **207**], l'array ordinato!

8.4.1 Complessità

Nel radix sort è fondamentale partire dalla cifra meno significativa. Ho una complessità $O(d(n+k))$ dove d è la lunghezza delle sequenze e k è il numero dei possibili valori di una cifra. Ovviamente n è il numero di elementi da ordinare.
L'algoritmo risulta conveniente quando $d << n$. Anche questo algoritmo può essere utilizzato per ordinare sequenze di caratteri.

Capitolo 9

Martedì 07/04/2020

Registro 8 (*Mar 07/04/2020 10:30-13:30 (3:0 h) lezione*)

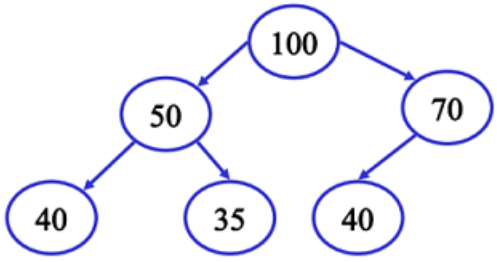
> *Tipo di dato heap. Algoritmo di ordinamento heapsort. Alberi generici: definizione, vistite, memorizzazione, esercizi.*

Oggi parliamo dell'ultimo tipo di albero binario: lo *heap*, che non ha nulla a che vedere con la parte di memoria dinamica introdotta a *Fondamenti di programmazione*.

9.1 Heap

Lo *heap*, noto anche come *coda con priorità*, è un albero binario quasi bilanciato, memorizzabile in array, che presenta le seguenti proprietà.

- I nodi, se presenti, sono tutti addossati a sinistra (osservare soprattutto l'ultimo livello dell'albero)
- Dato un qualunque nodo l'etichetta della radice è maggiore/minore o uguale rispetto a quelle dei discendenti. Distinguiamo due tipi di *heap*:
  - **max-heap**: dato un nodo gli elementi presenti nei sottoalberi saranno tutti minori o uguali. La radice dello heap consiste nell'elemento con valore più grande.
  - **min-heap**: dato un nodo gli elementi presenti nei sottoalberi saranno tutti maggiori o uguali. La radice dello heap consiste nell'elemento con valore più piccolo.



Osserviamo l'esempio Se non ci fosse il 35 l'albero non sarebbe uno heap (tenendo conto della presenza del 40). Non ci devono essere buchi! Questa proprietà permette una memorizzazione più efficiente in cui evitiamo i puntatori salvando il tutto in un array senza perdere informazione sul contenuto dei nodi (quindi dato un array posso capire come disegnare l'albero). Tutto è memorizzato per livelli linearmente.

Requisito Poichè utilizziamo gli array è necessario conoscere una dimensione massima.

Padri e figli Data la posizione i di un certo elemento posso trovare l'indice dei figli o quello del padre.

- **Figlio sinistro**: indice $2i + 1$
- **Figlio destro**: indice $2i + 2$
- **Padre**: indice $(i - 1)/2$

Operazioni possibili Le operazioni possibili sono:

- Inserimento di un nodo in maniera tale da non alterare le proprietà dello heap
- Estrazione della radice, cioè del primo elemento dell'array.

Negli esempi successivi faremo riferimento a un *max-heap* (i confronti cambiano se si ha un *min-heap*). Queste operazioni hanno complessità $O(\log n)$.

Applicazione pratica Tutto questo è utilizzato in particolare nei sistemi operativi nella gestione dei processi. I meccanismi di *scheduling* danno priorità ai processi di un sistema operativo rispetto a quelli aggiuntivi creati dall'utente.

Immaginiamo una classe *Heap* in cui abbiamo un puntatore ad array (allocato dinamicamente) e un intero che consiste nell'indice dell'ultimo elemento presente (sarà -1 in assenza di elementi). Analizziamo le funzioni che permettono di svolgere le operazioni base

9.1.1 Inserimento

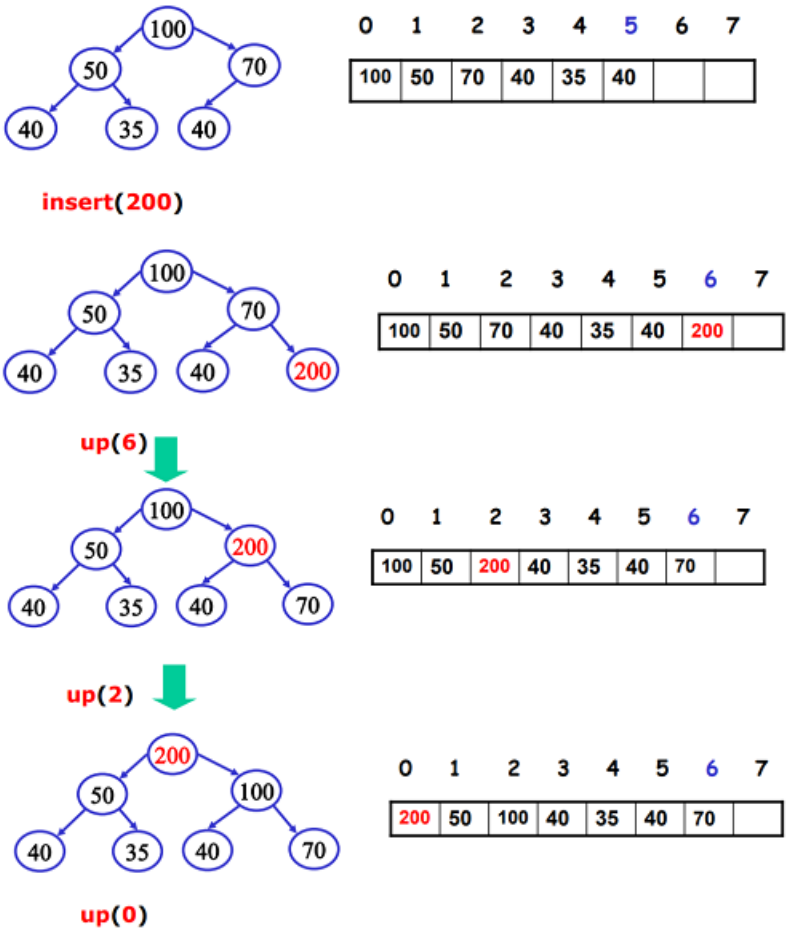
```
void Heap::insert(int x) {
        h[++last] = x;
        up(last);
}
```

Per prima cosa inserisco l'elemento nella prima posizione libera dell'array aggiornando (attraverso incremento) la posizione dell'ultimo elemento, che avrà valore x. Successivamente chiamo la funzione *up*: l'aggiunta di un elemento in fondo all'array potrebbe rendere lo heap inconsistente, effettuo degli scambi *figlio-padre* per rendere nuovamente valide tutte le proprietà tipiche dello heap.

```
void Heap::up(int i) {
        if(i > 0)
                if(h[i] > h[(i-1)/2]) {
                        exchange(i, (i-1)/2);
                        up((i-1)/2);
                }
}
```

Se non sono sulla radice (che non ha padri) verifico se il valore di un certo nodo è maggiore del valore del padre corrispondente (ricordiamo le posizioni introdotte all'inizio): se lo è effettuo uno scambio tra padre e figlio portando l'elemento inserito su di un livello e chiamo ricorsivamente la funzione. Continuo a fare confronti finchè non ho ristabilito la consistenza dell'albero (mi fermo se arrivo alla radice o se il valore appena aggiunto è stato collocato nella posizione giusta).

9.1.1.1 Esempio di inserimento



Inserisco 200 in fondo all'array (posizione 6). Quella non è la posizione corretta: effettuo uno scambio con il padre (in posizione 2). Dopo aver richiamato la funzione individuo che neanche quella è la posizione giusta: effettuo un altro scambio con il padre (che è la radice) portando 200 in cima all'albero. Adesso l'albero è consistente.

9.1.1.2 Complessità

Ogni volta che si sale di livello il numero di elementi considerati si dimezza, segue che ho complessità di $O(\log n)$. Si fanno tante chiamate quanto è la profondità dell'albero.

9.1.2 Estrazione della radice

```
int Heap::extract() {
        int r = h[0];
        h[0] = h[last--];
        down(0);
        return r;
}
```

Osservazione Si parla di estrazione e basta, non di estrazione di un nodo.

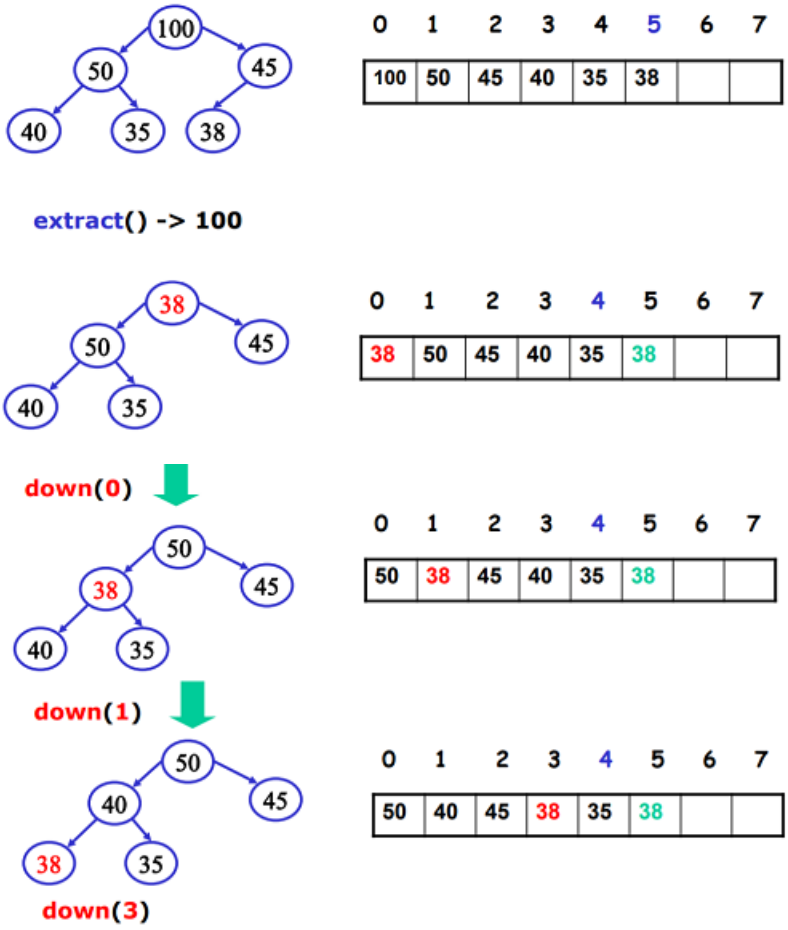
L'elemento più grande si trova nella posizione 0 dell'array. Con la funzione restituisco, dopo averlo estratto, il primo elemento. Pongo al posto del primo elemento l'elemento collocato alla fine dell'array. Successivamente chiamo ricorsivamente la funzione *down* per far scendere l'elemento appena spostato collocandolo nel posto giusto. In questo caso, per ristabilire la consistenza dell'albero, effettuerò scambi *padre-figlio* (parti invertite rispetto all'inserimento).

```
void Heap::down(int i) {
        int son = 2*i+1;
        if(son == last) {
                if(h[son] > h[i])
                        exchange(i, last);
        }
        else if(son < last) {
                if(h[son] < h[son+1]) son++;
                if(h[son] > h[i]) {
                        exchange(i, son);
                        down(son);
                }
        }
}
```

La discesa risulta più complessa da scrivere: mentre nella salita ho solo un confronto tra figlio e padre nella discesa ho il confronto tra il padre e due figli.

- Se il figlio dell'elemento è unico questo figlio è per forza l'ultimo elemento dell'array: effettuo un unico confronto, compio un eventuale scambio e concludo poichè ho già analizzato tutti gli elementi dell'array.
- Negli altri casi un certo elemento ha per forza due figli, quindi devo effettuare un ulteriore confronto.
  - Verifico quale tra i due figli abbia l'etichetta con valore più alto
  - Effettuo l'eventuale scambio con il figlio avente etichetta con valore più alto (l'incremento di son mi permette di effettuare lo scambio con il figlio destro nel caso in cui il figlio sinistro abbia valore più piccolo di lui).
  - In caso di scambio chiamo nuovamente *down*.
- Continuo finchè l'elemento non troverà figli con etichetta minore (o comunque sia fino al termine dello scorrimento dell'array).

9.1.2.1 Esempio di estrazione

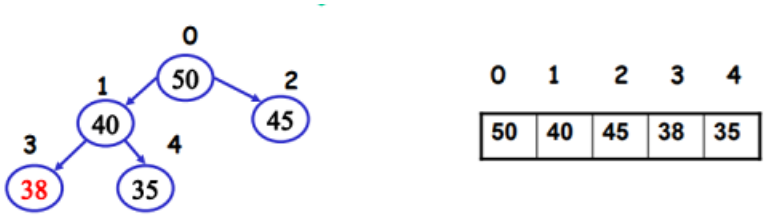


Voglio estrarre 100: decremento l'intero con l'indice dell'ultimo elemento e sostituisco 100 con 38. Il 38 non si trova nel posto giusto: lo faccio scendere due volte chiamando ricorsivamente *down*: prima ho lo scambio tra 38 e 50 ($50 > 45$) e infine con 40 ($40 > 35$).

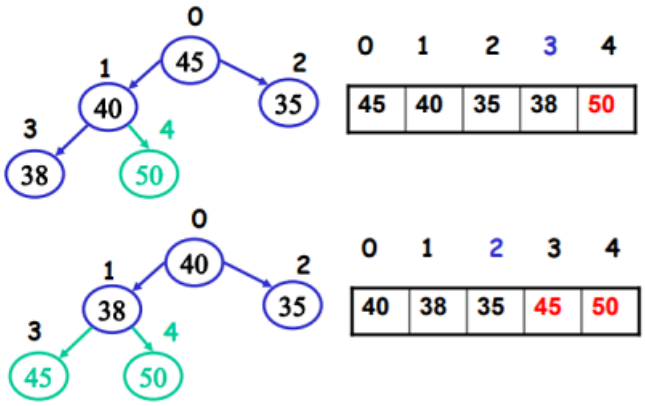
9.1.2.2 Complessità

Scendo sempre di un livello, al massimo scenderò un numero di livelli pari a quelli dell'albero. Segue complessità logaritmica come prima.

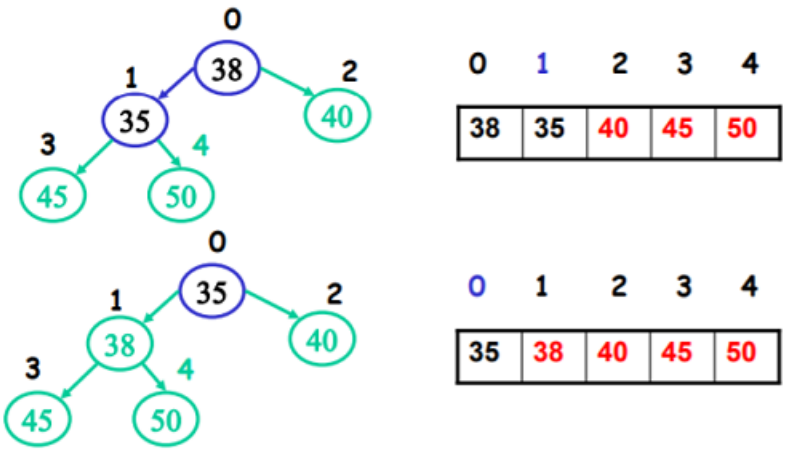
9.2 *heapSort* - Algoritmo di ordinamento



Prendiamo l'array qua sopra e supponiamo di fare un'estrazione: prendo il 50, ma non lo butto via, lo scambio con l'ultimo elemento (il 35). Facciamo un'altra estrazione scambiando 45 con il 38.



A questo punto lo heap avrebbe soltanto tre elementi (ricordando che gli altri elementi sono solo invisibili). Faccio nuovamente un'altra estrazione (scambio tra 40 e 38): rimane uno heap con due elementi. Concludo con l'ultima estrazione: rimane solo il 35.



Osserviamo, considerando anche gli elementi invisibili, che l'array è stato ordinato! Non potrei utilizzare questo sistema per ordinare gli elementi? Sì, ma soltanto con le rappresentazioni di uno heap!

Non è proprio con altri array? Possiamo applicarlo anche ad altri array attraverso un ulteriore step: la trasformazione dell'array in una rappresentazione dello heap!

9.2.0.1 Procedimento

- Trasformo l'array in uno heap con la funzione *buildHeap*

- Eseguo n estrazioni (tante quanti gli elementi presenti nell'array) scambiando ogni volta il primo elemento dell'array con quello puntato da *last*. Alla fine l'estrazione diventa un semplice scambia, ma il meccanismo non cambia (semplicemente mi limito allo scambio senza nascondere l'elemento)

heapSort Con n indichiamo il numero di elementi, che decrementiamo per avere l'indice dell'ultimo elemento. Successivamente con il while eseguo n estrazioni: aggiorno l'indice dell'ultimo elemento attraverso un riferimento. La complessità di *buildHeap* è $O(n)$, la complessità del while è $O(n \log n)$.

```
void heapSort(int* A, int n) {
        buildHeap(A, n-1);
        int i = n-1;
        while(i > 0) {
                extract(A, i);
        }
}
```

down Ho una funzione praticamente uguale: le uniche differenze stanno negli argomenti posti (abbiamo compiuto il passaggio da una funzione membro di una classe a funzione autonoma). Ho complessità $O(\log n)$.

```
void down(int* h, int i, int last) {
        int son = 2*i+1;
        if(son == last) {
                if(h[son] > h[i]) exchange(h, i, last);
        }
        else if(son < last) {
                if(h[son] < h[son+1]) son++;
                if(h[son] > h[i]) {
                        exchange(h, i, son);
                        down(h, son, last);
                }
        }
}
```

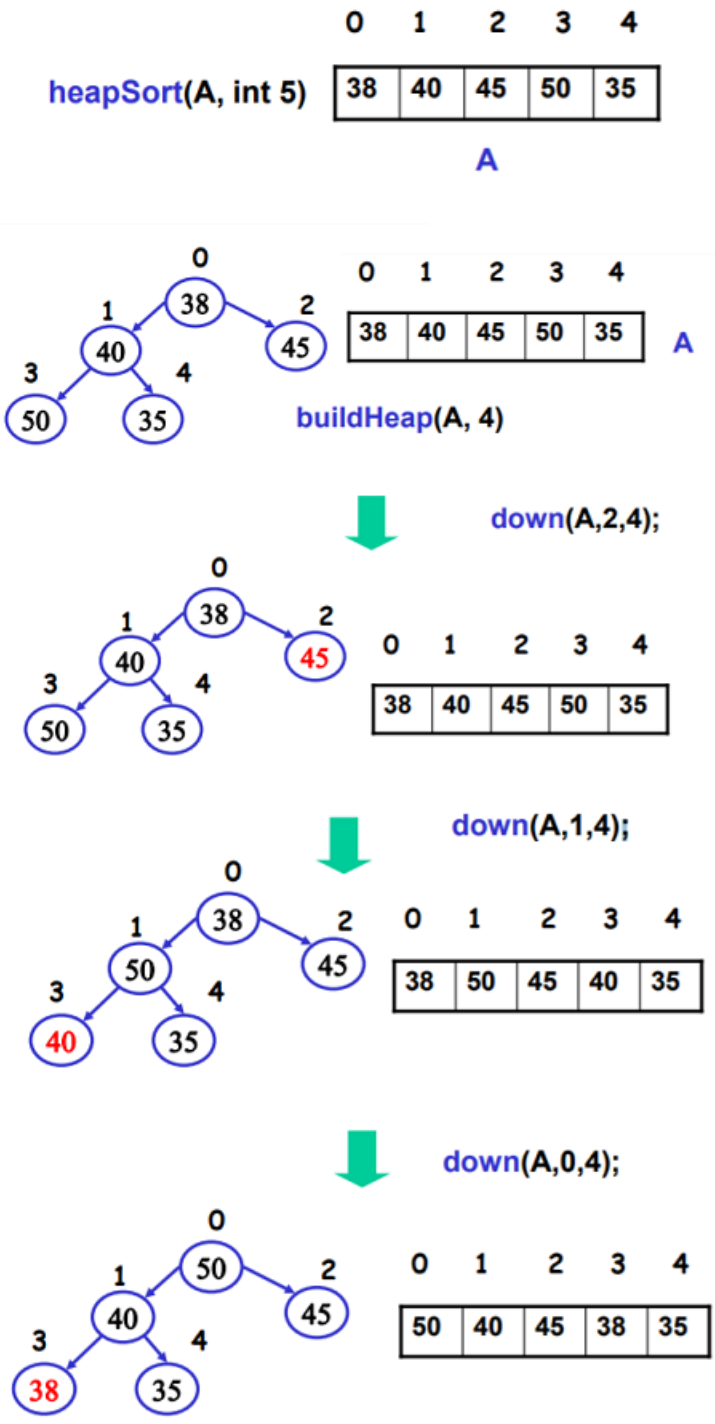
extract Anche l'extract è praticamente una fotocopia. Ho complessità $O(\log n)$.

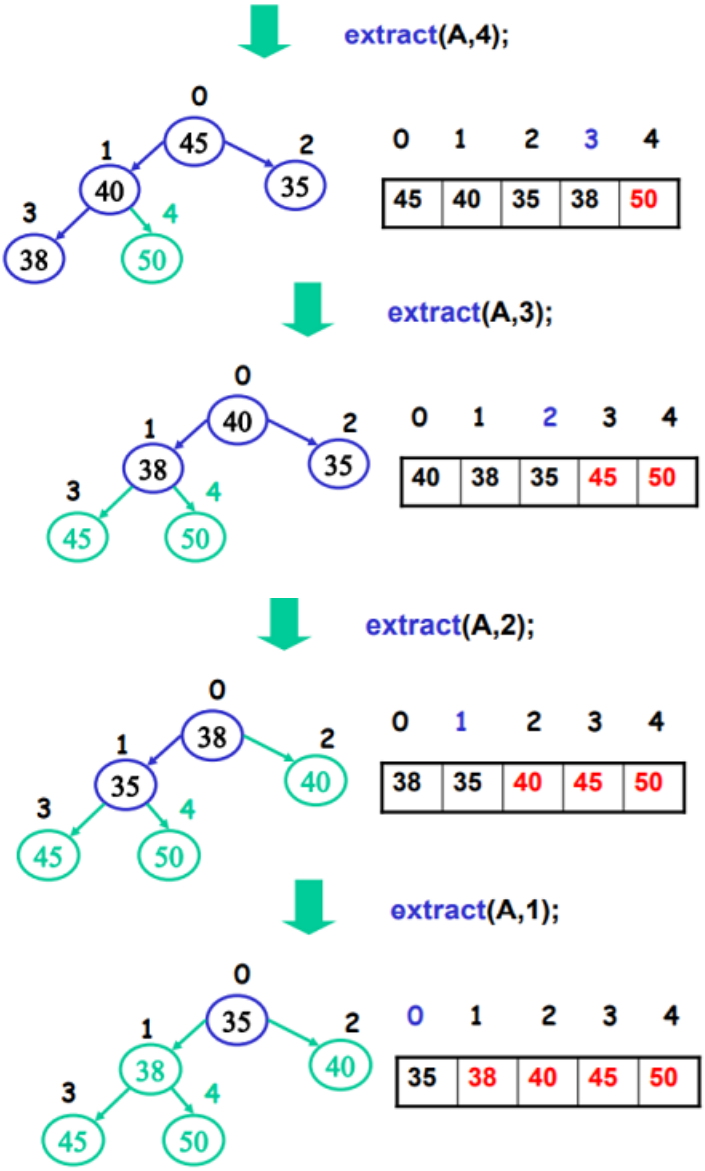
```
void extract(int* h, int& last) {
        exchange(h, 0, last--);
        down(h, 0, last);
}
```

buldHeap Eseguo la funzione *down* sulla prima metà degli elementi dell'array (gli elementi della seconda metà sono foglie): parto dall'elemento centrale e torno indietro arrivando al primo. Il tornare indietro è essenziale: se proviamo a farlo in senso opposto non otteniamo uno heap. Applicare down a tutti gli elementi non è sbagliato ma può portare a un aumento di complessità. Ho complessità $O(n)$.

```
void buildHeap(int* A, int n) {
        for(int i = n/2; i >= 0; i--) down(A, i, n);
}
```

9.2.0.2 Esempio




extract(A,4);
0 1 2 3 4
45 40 35 38 50
extract(A,3);
0 1 2 3 4
40 38 35 45 50
extract(A,2);
0 1 2 3 4
38 35 40 45 50
extract(A,1);
0 1 2 3 4
35 38 40 45 50

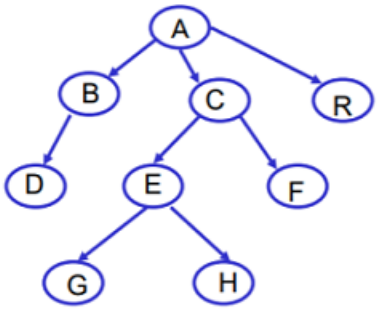
9.3 Alberi generici

Vogliamo rappresentare una gerarchia avente apertura a ventaglio come un qualunque albero binario: emerge la necessità di collegare a più nodi più di due figli. Proprio per questo introduciamo gli *alberi generici* con cui posso rappresentare tutte le possibili gerarchie.

Definizione ricorsiva

- Un nodo p è un albero.
- Un nodo più una sequenza di alberi $A_1, \ldots, A_n$ è un albero.

La definizione induttiva non è accettabile: dato un nodo non posso dire che ha dei sottoalberi vuoti!



All'interno di un albero generico si hanno gli stessi elementi presenti all'interno di un albero binario (radice, padre, i-esimo sottoalbero, i-esimo figlio, livello).

9.3.1 Differenza tra un albero generico e un albero binario

SBAGLIATO Un albero binario è un albero generico con al massimo due figli.

In un albero binario si hanno sempre due sottoalberi. Segue che questi due sottoalberi siano diversi

Mentre in un albero generico i due alberi non sono differenti.

9.3.2 Visite

Il concetto di visita è simile a quello visto in alberi binari. Abbiamo due tipi di visite:

- *preOrder*: esamino la radice prima di scorrere i sottoalberi. Prendendo l'albero generico a inizio sezione ottengo: A B D C E G H F R.
- *postOrder*: esamino la radice dopo lo scorrimento dei sottoalberi. Prendendo l'albero generico a inizio sezione ottengo: D B G H E F C R A.

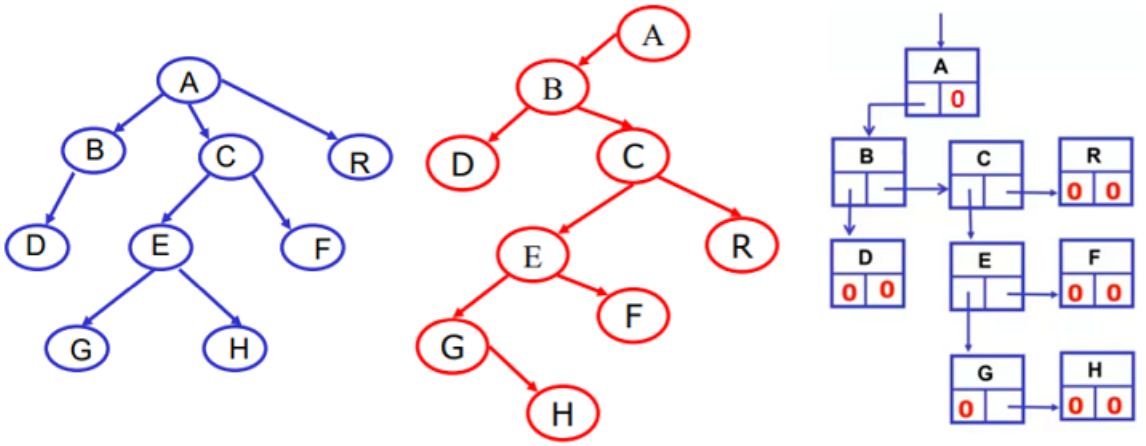
9.3.3 Memorizzazione figlio-fratello

La memorizzazione richiede un attimo di riflessione: memorizzare seguendo le stesse metodiche dell'albero binario (struct con tanti puntatori quanti il numero massimo di sottoalberi possibili per un nodo) comporta un certo consumo di memoria con molte chiamate ricorsive a vuoto.

A tal proposito introduciamo la **memorizzazione figlio-fratello**. Abbiamo una struct identica a quella degli alberi binari:

- il puntatore sinistro punta al primo figlio
- il puntatore destro punta al primo fratello

Vediamo un esempio



è come se avessi trasformato l'albero di partenza in un albero binario. La rappresentazione dell'albero non è isomorfa rispetto a come è fatto l'albero in realtà. Segue la necessità di cambiare alcune funzioni: per esempio non posso utilizzare la funzione per trovare il numero di livelli, restituirebbe un valore errato.

Posso capire se un certo albero è binario o una rappresentazione di un albero generico? No, generalmente non posso saperlo. Solo in una situazione ho la certezza che un certo albero binario non consiste in una rappresentazione di un albero generico: se il sottoalbero destro della radice non è vuoto.

Visite Possiamo utilizzare i soliti programmi:

- la visita *preOrder* del trasformato corrisponde alla visita *preOrder* dell'albero generico
- la visita *inOrder* del trasformato corrisponde alla visita *postOrder* dell'albero generico
- il programma per la visita *postOrder* non può essere usato

Conteggio nodi Il programma rimane uguale

```
int nodes(Node* tree) {
        if(!tree) return 0;
        return 1 + nodes(tree->left) + nodes(tree->right);
}
```

Conteggio foglie Il programma cambia

```
int leaves(Node* tree) {
        if(!tree) return 0;
        if(!tree->left) return 1 + leaves(tree->right);
        return leaves(tree->left)+leaves(tree->right);
}
```

Se il puntatore sinistro punta al primo figlio significa che se non si hanno figli questo sarà nullo. Osserviamo lo schema dell'albero binario con i puntatori: tutte le foglie hanno il puntatore sinistro nullo. Puntatore destro nullo significa avere altre papabili foglie. Quindi proseguo il conteggio analizzando il sottoalbero destro in ogni caso.

- Se il sottoalbero sinistro è vuoto ho una foglia, sommo 1 al numero di foglie del sottoalbero destro.
- Se il sottoalbero sinistro non è vuoto significa che l'elemento non è una foglia, quindi sommo il numero di foglie presenti nel sottoalbero sinistro al numero di foglie presenti nel sottoalbero destro.

Capitolo 10

Martedì 21/04/2020

Registro 9 (*Mar 21/04/2020 10:30-13:30 (3:0 h) lezione*)

Metodo di ricerca ricerca hash.Algoritmo PLSC per trovare la più lunga sottosequenza comune fra due sequenze. Esercizi.

10.1 Metodo di ricerca *hash*

Fino ad ora abbiamo visto metodi di ricerca basati sul confronto. Adesso vedremo il metodo *hash* (dall'inglese *to hash*, rimescolare), basato su una filosofia completamente diversa. Esso è utilizzato, in particolare, nelle implementazioni di basi di dati.

- È un metodo di ricerca in **array**, ho bisogno di accesso diretto a ogni singolo elemento (segue che lo stesso metodo non potrà essere applicato alle liste)
- non è basato su confronti
- è molto efficiente

Funzione *h()* Questo metodo è basato sulla funzione *hash*, una funzione matematica che associa a un elemento dell'insieme un indice dell'array. La funzione restituisce un valore che consiste in uno degli elementi possibili all'interno di un array.

$$h(x) : InfoType \rightarrow indici$$

Questa funzione è sicuramente **surgettiva** poichè la funzione genera tutti e soli gli indici dell'array (Ricordare: una funzione è surgettiva quando ogni elemento del codominio è immagine di almeno un elemento del dominio).

Di cosa dobbiamo tenere conto?

- n: il numero massimo di elementi presenti nell'insieme che vogliamo rappresentare
- k: la dimensione dell'array.

Si osserva che $n \leq k$.

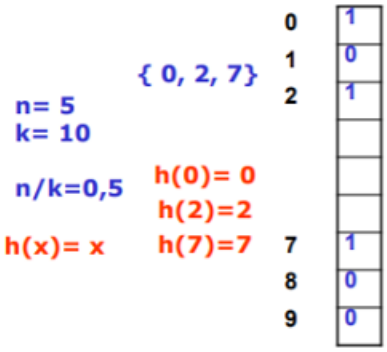
10.1.1 Metodo hash ad accesso diretto

Nel metodo hash *ad accesso diretto* la funzione $h(x)$ restituisce l'*indirizzo hash dell'elemento* che contiene x, cioè l'indice della posizione in cui si trova quell'elemento. In questo caso $h(x)$ è anche **iniettiva** (quindi ogni elemento del dominio è associato ad al più un elemento del codominio)

Esempio Consideriamo un insieme con al massimo $n = 5$ cifre decimali. Ho un vettore con $k = 10$ elementi (poichè si hanno 10 cifre decimali da 0 a 9). Nel metodo hash ad accesso diretto avrò

$$h(x) = x$$

Quindi $h(0) = 0, h(2) = 2, \ldots, h(7) = 7, \ldots$ Il valore degli elementi del vettore deve essere una marca che mi permetta di capire se l'elemento c'è o non c'è (per esempio potrei porre 0 se l'elemento non è presente, 1 se presente).



Codice Considerando $h(x) = x$

```
bool hashSearch(int* A, int k, int x) {
        int i = h(x);
        if(A[i] == 1) return true;
        else return false;
}
```

Problema Attraverso questo metodo abbiamo ottenuto un metodo di ricerca con complessità $O(1)$, tuttavia si ha un consumo di memoria non banale. Con un array di dieci posizioni e un massimo di 5 elementi ho la seguente percentuale di occupazione

$$\frac{n}{k} = 0.5$$

Pur con il massimo numero di elementi una parte dell'array rimarrà inutilizzata (la metà). La cosa è peggiore se con n invariato abbiamo $k = 10000$ (k, ricordiamo, è il valore massimo assumibile dagli elementi). Il metodo richiede un array con 10.000 elementi: un'occupazione di memoria che non possiamo accettare e che ci porta a metodi alternativi all'accesso diretto.

10.1.2 Metodo hash ad accesso non diretto

La funzione $h(x)$, rinunciando all'accesso diretto, non è più iniettiva. Questo significa che due elementi diversi potrebbero avere lo stesso indirizzo hash. Abbiamo una collisione

$$h(x_1) = h(x_2)$$

E ciò implica che due elementi diversi saranno associati allo stesso indice di array. Dovremo svolgere delle istruzioni in più: è ovvio che non posso associare allo stesso indice di array due elementi diversi!

Situazioni da gestire

- Ricerca di un elemento nel caso in cui il suo posto sia occupato da altri elementi
- Inserimento degli elementi
- (Eventuale) eliminazione degli elementi

10.1.2.1 Metodo hash ad indirizzamento aperto

Questa è la soluzione principale che vedremo. Abbiamo una funzione hash modulare

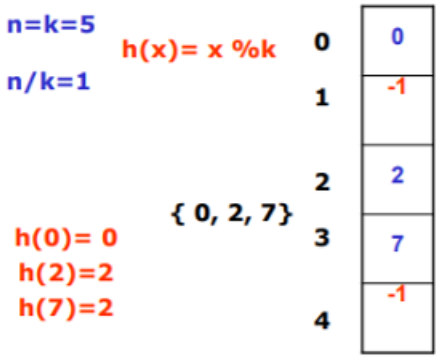
$$h(x) = x\%k$$

Restituisco il resto della divisione: ovviamente dividendo per k ho la certezza di avere tutte le volte un indice dell'array.

Legge di scansione lineare Se non trovo l'elemento al suo posto, lo cerco nelle posizioni successive finchè non lo trovo o incontro una posizione vuota. La cosa vale anche per l'inserimento!

Velocità La velocità viene degradata perchè dovremo effettuare degli scorrimenti dell'array.

Esempio Abbiamo $n = 5$ (numero massimo di elementi salvabili) e $k = 5$ (quindi array con 5 elementi), con $\frac{n}{k} = 1$ (array pienamente occupato). Con la funzione $h(x) = x\%k$ avremo $h(0) = 0, h(2) = 2, \ldots, h(7) = 2, \ldots$.



Troviamo 2

- Prendo il resto della divisione x/k, quindi $2\%5 = 2$
- Non ho da fare altro, ho già corrispondenza in posizione 2

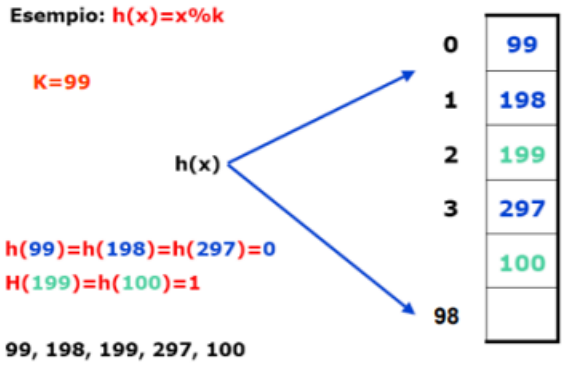
Troviamo 3

- Prendo il resto della divisione x/k, quindi $3\%5 = 3$
- Scorro l'array e arrivo alla posizione 4 con valore -1, mi fermo poichè ho trovato la marca di posto vuoto: con la legge di scansione lineare affermo che l'elemento non è presente nell'array.

Troviamo 7

- Prendo il resto della divisione x/k, quindi $7\%5 = 2$
- In posizione 2 ho l'elemento 2, non è quello che cercavo quindi scorro
- In posizione 3 ho l'elemento 7: mi fermo poichè ho trovato corrispondenza!

Conseguenze Si creano gruppi di elementi, agglomerati, con indirizzi hash diversi. Tutto questo degrada la velocità aumentando il tempo di ricerca. L'esempio lo vediamo nella seguente foto: $99, 198$ e 297 sono associati all'indice 0, mentre 199 e 100 sono associati all'indice 1.



Nelle seguenti funzioni adotteremo l'approccio dell'array circolare, già visto a *Fondamenti di programmazione* parlando di code. Questo ci permette di scorrere tutti gli elementi dell'array data una qualunque posizione dell'array con k elementi. In questi cicli ho almeno un'iterazione (quella con $j = 0$, verifico se l'elemento che mi interessa cade proprio nell'indice ottenuto da $h(x)$).

Funzione per la ricerca con scansione lineare Vediamo una funzione che restituisce un valore booleano

```
bool hashSearch(int* A, int k, int x) {
        int i = h(x);
        for(int j = 0; j < k; j++) {
                int pos = (i+j)%k;
                if(A[pos] == -1) return false;
                if(A[pos] == x) return true;
        }
        return false;
}
```

Funzione per l'inserimento La seguente funzione inserisce un elemento e restituisce un valore booleano

```
bool hashInsert(int* A, int k, int x) {
        int i = h(x);
        bool b = false;
        for(int j = 0; !b && j < k; j++) {
                int pos = (i+j)%k;
                if(A[pos] == -1) {
                        A[pos] = x;
                        b = true;
                }
        }
        return b;
}
```

E se io volessi cancellare degli elementi? Non possiamo porre come valore dell'elemento cancellato -1, altrimenti la ricerca non funzionerebbe: provare a cercare 297 (nell'ultimo array) dopo aver cancellato 199. Segue la necessità di un'altra marca che indica posizione vuota a causa di cancellazione: ciò permetterà alla funzione di distinguere i casi proseguendo nella ricerca. Riscriviamo la funzione per l'inserimento

```
bool hashInsert(int* A, int k, int x) {
        int i = h(x);
        bool b = false;
        for(int j = 0; !b && j < k; j++) {
                int pos = (i+j)%k;
                if(A[pos] == -1 || A[pos] == -2) {
                        A[pos] = x;
                        b = true;
                }
        }
        return b;
}
```

Nulla vieta di porre un nuovo valore in quella cella con la funzione *hashInsert*.

Scansioni Le scansioni possono essere caratterizzate da un *passo di scansione* diverso, che può ridurre gli agglomerati. La cosa essenziale è che siano visitabili tutte le possibili celle vuote dell'array evitando così il fallimento dell'inserimento anche con array non pieno. Abbiamo le seguenti scansioni

Scansione lineare Usata prima.

$$(h(x) + cost * j) \ mod \ k$$

Scansione quadratica

$$(h(x) + cost * j^2) \ mod \ k$$

Tempo medio di ricerca Il tempo medio di ricerca è influenzato dai seguenti elementi:

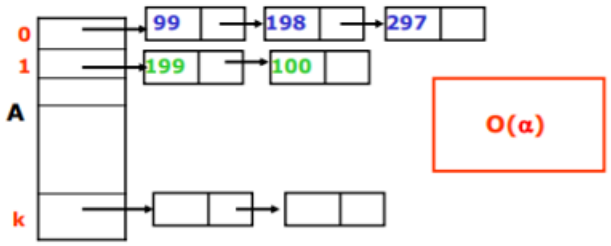
- **Fattore di carico**: dal rapporto $1 > \alpha = \frac{n}{k}$ (numero medio di elementi per posizione) deduciamo che maggiore è la dimensione dell'array migliore sarà la velocità. Questo perchè con un numero maggiore di celle si riducono le probabilità di collisione tra elementi e quindi gli agglomerati (che, ribadiamo, rallentano l'esecuzione della funzione).
- **Legge di scansione**: è dimostrabile che certe leggi di scansioni migliorano la velocità (per esempio la quadratica)
- **Uniformità della funzione hash**: generazione degli indici con uguale probabilità. Se un indice ha maggiore probabilità di essere generato allora esso può diventare con più facilità l'indirizzo di un agglomerato.

Numero medio di accessi con scansione lineare Dipende da α. Si individua che sarà minore al seguente rapporto

$$\leq \frac{1}{1-\alpha}$$

Necessità di risistemare l'array periodicamente Gli inserimenti e le cancellazioni degradano il tempo di ricerca. Risulta necessario sistemare periodicamente l'array ricorrendo ad algoritmi che non tratteremo in questo corso.

Metodo di concatenazione Possiamo velocizzare la ricerca attraverso il *metodo di concatenazione*: creo un array di liste, ciascuna avente al suo interno gli elementi che collidono su un certo indice. Si evitano del tutto gli agglomerati sull'array ma ovviamente la cosa va a discapito della memoria, tenendo conto che le liste occupano più spazio di un semplice array.



Se consideriamo presenti $\alpha = \frac{n}{k}$ elementi in ogni lista individuiamo una complessità $O(\alpha)$.

10.1.3 Algoritmi di ricerca e ordinamento di dati non in memoria interna

Attenzione Queste cose non vengono chieste all'esame (cit.), ma sono importanti per il nostro futuro lavorativo. Vedere le diapositive 230 e 231.

10.2 Programmazione dinamica (o Tabulazione)

Parliamo di metodologie di programmazione. Abbiamo già parlato di *divide et impera*, dividere il problema in parti uguali, risolvere separatamente e trattare i vari risultati insieme. Questa metodologia non è sempre applicabile!

Strategia *bottom-up* Se non sono in grado di dividere il problema in sottoproblemi posso risolvere tutti i possibili sottoproblemi partendo dal basso combinando i risultati passo dopo passo. Il numero di sottoproblemi, che determinerà la complessità, non deve essere esponenziale.

- **sottostruttura ottima**: il metodo è applicabile quando si ha una sottostruttura ottima. Cioè: ho una soluzione ottima per il mio sottoproblema, so che a partire da questa soluzione potrò arrivare alla soluzione ottima del problema iniziale.

Mentre nel *divide et impera* abbiamo una netta suddivisione dei sottoproblemi, nella programmazione dinamica sono presenti sovrapposizioni di sottoproblemi. Un esempio già visto potrebbe essere la risoluzione della serie di Fibonacci in modo iterativo.

10.2.1 Algoritmo PLSC (*Più lunga sottosequenza comune*)

Supponiamo di avere due sequenze di elementi qualsiasi

$$\alpha = a\,b\,c\,a\,b\,b\,a \quad \beta = c\,b\,a\,b\,a\,c$$

Cioè, generalizzando

$$\alpha = \alpha_1\,\alpha_2\,\alpha_3\,\alpha_4\,\alpha_5\,\alpha_6\,\alpha_7 \quad \beta = \beta_1\,\beta_2\,\beta_3\,\beta_4\,\beta_5\,\beta_6$$

Il nostro obbiettivo è individuare le sottosequenze comune più lunghe, cioè sequenze all'interno di quella principale dove l'ordine degli elementi non viene alterato (ottenute cancellando elementi). Nelle sequenze prese ad esempio abbiamo 3 PLSC (tutte di lunghezza quattro):

- baba
- cbba
- caba

Concentriamoci sulla lunghezza Scriviamo una funzione che restituisce semplicemente la lunghezza massima tra le sottosequenze comuni. Abbiamo una funzione $L(i, j)$ che restituisce la lunghezza massima della PLSC di $\alpha_1, \ldots, \alpha_i$ e $\beta_1, \ldots, \beta_j$. Individuiamo che

$$\begin{cases} L(0,0) = L(i,0) = L(0,j) = 0 & \\ L(i,j) = L(i-1,j-1) + 1 & \alpha_i = \beta_j \\ L(i,j) = max(L(i,j-1), L(i-1,j)) & \alpha_i \neq \beta_j \end{cases}$$

- Se mettiamo a confronto una sottosequenza vuota, o una sequenza con un'altra vuota, il risultato sarà 0
- Se $\alpha_i = \beta_j$ ho un elemento comune, quindi la funzione restituirà la somma tra 1 e la lunghezza massima delle PLSC nelle due sottosequenze, rispettivamente fino a $i-1$ e $j-1$ (incremento).
- Se $\alpha_i \neq \beta_j$ non posso sommare uno poichè non ho trovato elementi comuni. A quel punto dovrò confrontare
  - la sequenza α fino all'elemento i con la sequenza β fino all'elemento $j-1$
  - la sequenza α fino all'elemento $i-1$ con la sequenza β fino all'elemento j

  e scegliere tra le lunghezze ottenute quella massima.

Prima versione del problema La seguente funzione rispetta la definizione induttiva

```
int lenght(char* a, char* b, int i, int j) {
        if(i == 0 || j == 0) return 0;

        if(a[i] == b[j])
                return lenght(a, b, i-1, j-1)+1;
        else
                return max(lenght,a,b,i,j-1),lenght(a,b,i-1,j));
}
```

Tuttavia, si ha nella relazione di ricorrenza $T(n) = b + 2T(n-1)$, quindi una complessità esponenziale per noi inaccettabile!

Proviamo ad applicare la strategia *bottom-up* Partendo dagli indici più piccoli costruiamo tutte le sottosequenze possibili

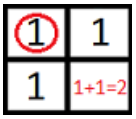
$$L(0,0), L(0,1) \dots L(0,n),$$
$$L(1,0), L(1,1) \dots L(1,n),$$
$$\dots$$
$$L(m,0), L(m,1) \dots L(m,n)$$

Funzione che restituisce la lunghezza massima Costruiamo una matrice avente dimensioni m, n. La seguente funzione ha complessità $O(nm)$, $O(n^2)$ se le stringhe presentano la stessa lunghezza ($m = n$).

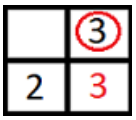
```
const int m = 7; const int n = 6;
int L[m+1][n+1];

int quickLenght(char* a, char* b) {
        for(int j = 0; j <= n; j++) L[0][j]=0;

        for(int i = 1; i <= m; i++) {
                L[i][0] = 0;
                for(int j = 1; j <= n; j++) {
                        if(a[i] != b[j])
                                L[i][j] = max(L[i][j-1], L[i-1][j]);
                        else
                                L[i][j] = L[i-1][j-1]+1;
                }
        }
        return L[m][n];
}
```

La funzione, attraverso una serie di confronti, genera una matrice:

- La prima colonna, i cui elementi hanno $y = 0$, viene posta tutta nulla col primo for
- Col secondo for scorro le righe dall'alto verso il basso, ciascuna dall'elemento più a sinistra all'elemento più a destra
- La prima riga, i cui elementi hanno $x = 0$, viene posta tutta nulla nel tempo attraverso l'assegnamento presente nel secondo for (non è un problema farlo passo dopo passo e non immediatamente)
- Ogni volta confronto il valore dei caratteri presenti nelle sequenze a e b:
  - se si ha corrispondenza assegno come valore alla cella in cui mi trovo quello della cella avente indici $(i-1, j-1)$ incrementato di uno



  - se non si ha corrispondenza assegno come valore alla cella in cui mi trovo il massimo tra i due valori aventi indici $(i-1, j)$ e $(i, j-1)$.



- Al termine restituisco il valore della cella avente indice (m, n), cioè il valore presente nella cella più a destra della riga più in basso.

Il risultato è una matrice in cui, dati due indici, sono in grado di individuare la lunghezza massima tra due sottosequenze qualsiasi presenti all'interno delle sequenze principali. Quindi, con $L[5][3]$ individuerò la lunghezza della PLSC nelle sequenze

$$\alpha = \alpha_1\,\alpha_2\,\alpha_3\,\alpha_4\,\alpha_5 \quad \beta = \beta_1\,\beta_2\,\beta_3$$

Dalle stringhe $\alpha = a\,b\,c\,a\,b\,b\,a$ e $\beta = c\,b\,a\,b\,a\,c$ otteniamo la seguente matrice

		c	b	a	b	a	c
	0	0	0	0	0	0	0
a	0	0	0	1	1	1	1
b	0	0	1	1	2	2	2
c	0	1	1	1	2	2	3
a	0	1	1	2	2	3	3
b	0	1	2	2	3	3	3
b	0	1	2	2	3	3	3
a	0	1	2	3	3	4	4

Osservazione L'algoritmo può essere fatto analizzando le colonne invece delle righe: il risultato è sempre lo stesso, è come se avessi scambiato le sequenze di posizione.

Capitolo 11

Martedì 28/04/2020

Registro 10 (*Mar 28/04/2020 10:30-13:30 (3:0 h) lezione*)

> *Metodologia greedy. Algoritmo di compressione di Huffman. Grafi orientati e non orientati: definizioni, memorizzazione, visita in profondità. Esercizi.*

Oggi analizziamo quei casi in cui la risoluzione di tutti i sottoproblemi non ha senso: in questi casi abbiamo un numero esponenziale di sottoproblemi e la strategia *bottom-up* non è considerabile valida.

11.1 Algoritmi greedy (avido/goloso)

La soluzione ottima si ottiene attraverso una sequenza di scelte. Supponiamo di dover creare un algoritmo e che il metodo divide et impera non sia applicabile. Ad ogni passo dovremo prendere delle decisioni: date delle strade scegliamo quella che pare migliore. Se l'algoritmo funziona la scelta locale sarà in accordo con la scelta globale, quindi non si perdono alternative che potrebbero rivelarsi migliori nel seguito.

Strategia adottata La strategia adottata è completamente diversa da quella della *programmazione dinamica* dove risolviamo il problema in modo *bottom-up*. Nella programmazione greedy abbiamo un metodo *top-down*: si sceglie un sottoproblema da risolvere e si risolve soltanto quello, scartando gli altri.

Attenzione Non è detto che si trovi sempre la soluzione ottima, in certi casi posso limitarmi a una soluzione approssimata.

Esempi I programmi che giocano a scacchi, basati un tempo su intelligenza artificiale, sceglievano una strategia ben precisa che poteva portare alla vittoria o alla sconfitta. Oggi questi programmi adottano una strategia diversa: ricevono moltissimi esempi di partite e attraverso questi apprendono le migliori strade per vincere. Un altro esempio riguardo questa strategia si ha con il problema dello zaino, che incontreremo più avanti.

11.2 Codice di compressione

Abbiamo un alfabeto, cioè un insieme di caratteri (per esempio le lettere dell'alfabeto fino ad f). Vogliamo assegnare ad ogni carattere una stringa binaria. Presupponiamo di avere un testo (per esempio i Promessi Sposi) da trasmettere attraverso un canale di comunicazione: prima di fare ciò dobbiamo codificare il testo in una sequenza binaria. Chi riceverà il codice procederà alla decodifica ricostruendo il testo originario.

Primo sistema Assegnare ai vari caratteri codici binari che presentano la stessa lunghezza. In questo caso avrò bisogno di 2 bit ($2^3 = 8$). Questo metodo non è conveniente in termini di spazio.

Scopo del codice di compressione Il codice di compressione permette di assegnare ai vari caratteri codici di lunghezza variabile: in questo modo il numero di bit occupati sarà minore (*compressione della codifica del testo*). Vediamo la seguente tabella

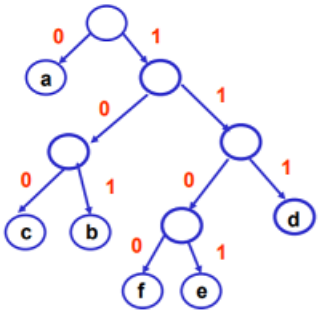
	a	b	c	d	e	f
frequenza	45	13	12	16	9	5
Codice a lunghezza fissa	000	001	010	011	100	101
Codice a lunghezza variabile	0	101	100	111	1101	1100

Abbiamo un carattere che necessita solo di un bit per la codifica, tre che ne utilizzano tre, e due che ne utilizzano quattro. Proviamo a codificare la stringa *abc*

- Con codice a lunghezza fissa: 000 001 010 (9 bit)
- Con codice a lunghezza variabile: 0 101 100 (7 bit)

La codifica si basa sulla frequenza dei caratteri presenti all'interno del testo: il 45% dei caratteri sono a, il 13% sono b, il 12% c e così via... Perchè utilizziamo 4 bit e non tre come nel primo sistema? Si ha un **codice prefisso**: il codice binario associato ad un certo carattere non deve essere prefisso di altri codici binari. Utilizziamo un bit in più, è vero, ma codifichiamo i caratteri con frequenza maggiore in sequenze di minore lunghezza. Segue un codice complessivo più corto.

Rappresentazione dei codici prefissi I codici prefissi possono essere rappresentati attraverso alberi binari.



a	b	c	d	e	f
0	101	100	111	1101	1100

- Ogni arco rappresenta lo zero o l'uno.
- Le foglie consistono nei caratteri.
- Il percorso consiste nella sequenza binaria che rappresenta il carattere nella foglia.

Se l'albero è pienamente binario ho una rappresentazione ottima.

11.2.1 Algoritmo di Huffman

Problema da risolvere Dato un alfabeto e la frequenza dei suoi caratteri, costruire un codice ottimo (che minimizza la lunghezza in bit delle codifiche).

Attenzione all'esame Non minimizzo la codifica dei caratteri, minimizzo la codifica dell'intero testo!

Strategia adottata Costruisco l'albero binario in modo *bottom-up*.

Spiegazione Gestisco una foresta di alberi

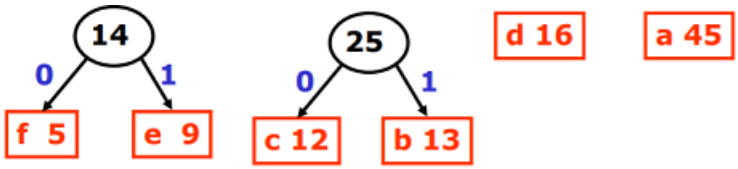
- All'inizio ho n alberi di cui un solo nodo con le frequenze dei caratteri.
- Scelgo i due con la radice minore e li fondo creando un nuovo albero. Introduco in questo una nuova radice avente come etichetta la somma delle due radici.

Esempio

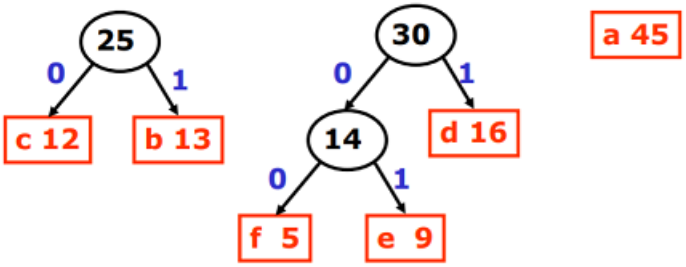
- Ho la seguente foresta di alberi, in tutto sei radici.

- Le radici minori hanno etichetta 5 e 9.

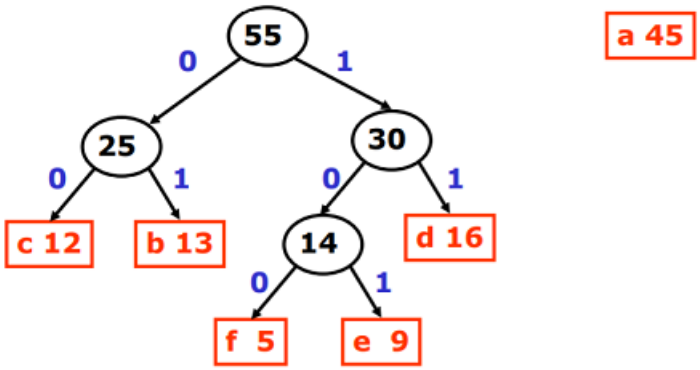
- Scelgo di nuovo i minori, adesso le radici con 12 e 13.



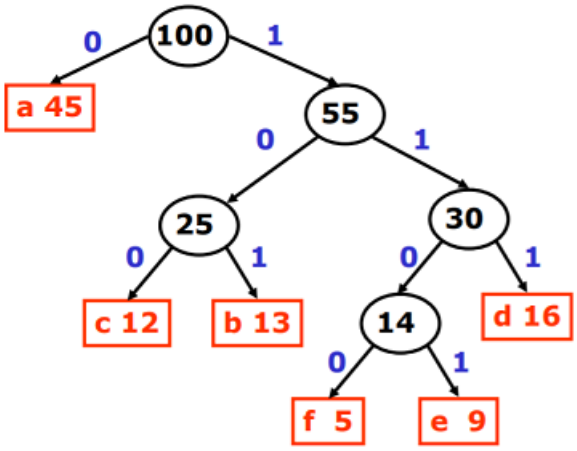
- Ancora. Adesso le etichette minori sono 14 e 16.



- Adesso le etichette minori sono 25 e 30.



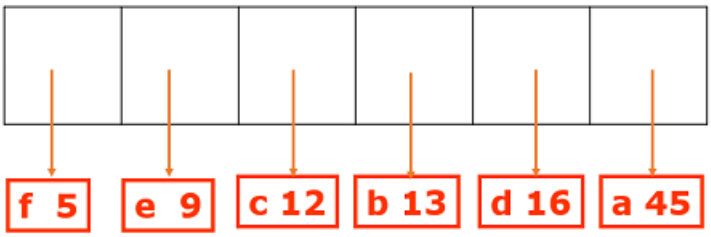
- Infine fondo gli alberi con 45 e 55, gli unici rimasti. Ottengo il risultato finale!



Perchè funziona? La scelta locale è consistente con la scelta globale: sistemo prima i nodi con minore frequenza, questi apparterranno ai livelli più alti dell'albero.

11.2.1.1 Implementazione e complessità

Per avere un'organizzazione efficiente ricorro a un *minheap*, ossia uno *heap* dove l'ordine degli elementi è invertito: dato un nodo i suoi sottoalberi hanno tutti elementi con valore maggiore. Ogni posizione dello heap contiene un albero.

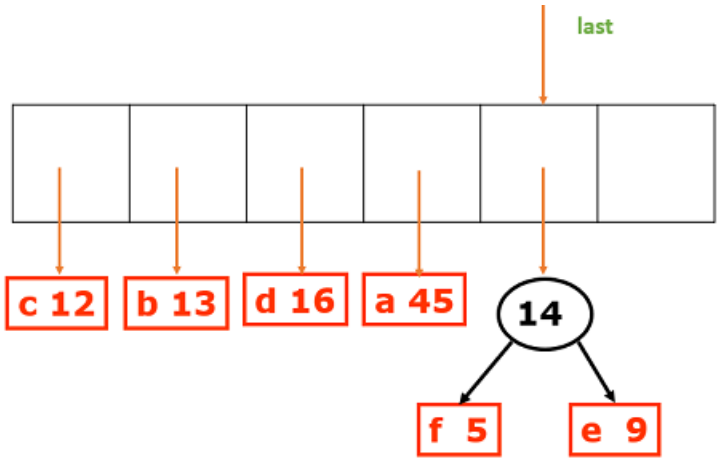



```
struct Node {
char c;
int val;
Node * left;
Node * right;
}
```

Ogni volta:

- Estraggo due elementi dallo heap, quelli con frequenza minore
- Li unisco formando un nuovo albero
- Aggiungo il nuovo albero allo heap, collocandolo con la funzione *up* in base al valore ottenuto dalla somma delle frequenze.

Esempio Estraggo i primi due nodi $\boxed{f\ 5}$ ed $\boxed{e\ 9}$, creo un albero ed inserisco questo nello heap. Ottengo



Complessità Ho un ciclo con n iterazioni dove ciascuna operazione ha complessità $O(\log n)$. Segue complessità $O(n \log n)$.

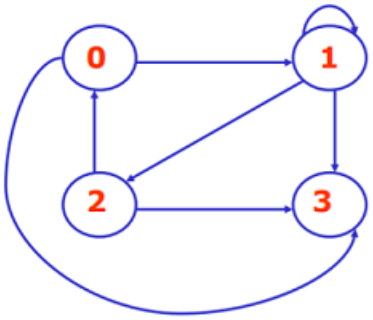
11.3 Grafi

I grafi sono una delle strutture dati più importanti, a cui sono legati molti problemi informatici. Non può essere definita in modo ricorsivo e la sua gestione non è proprio semplicissima.

11.3.1 Grafi orientati

Un grafo orientato è una coppia (N, A) dove

- N è un insieme di nodi
- A un insieme di archi (quindi di coppie ordinate di nodi), un sottoinsieme del prodotto cartesiano $N \times N$.



Definizioni

- **Predecessore di un nodo**: uno dei nodi che puntano al nodo analizzato
- **Successore di un nodo**: nodo raggiungibile dal nodo analizzato attraverso un arco
- **Cammino**: sequenza di nodi. La lunghezza del cammino consiste nel numero di archi!
- **Ciclo**: cammino che parte e finisce nello stesso nodo
- **Grafo aciclico**: grafo privo di cicli (l'esempio non è assolutamente un grafo aciclico)

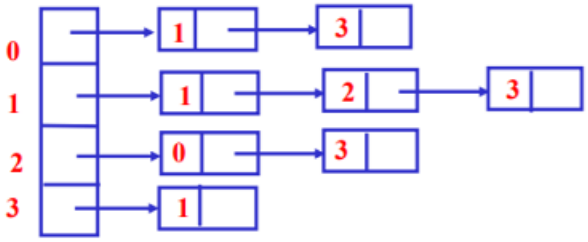
Complessità La complessità di algoritmi che agiscono sui grafi è normalmente dipendente da due variabili:

- n, il numero dei nodi
- m, il numero degli archi

Numero massimo di archi Un grafo orientato con n nodi presente al massimo n^2 archi.

Rappresentazione in memoria: liste di adiacenza Ho un array iniziale con tanti elementi quanti sono i nodi del grafo. Ogni elemento dell'array è puntatore a struct. Abbiamo delle liste, ciascuna contenente gli elementi puntati da un certo nodo.

```
struct Node {
        int NodeNumber;
        Node* next;
};
Node* graph[N];
```



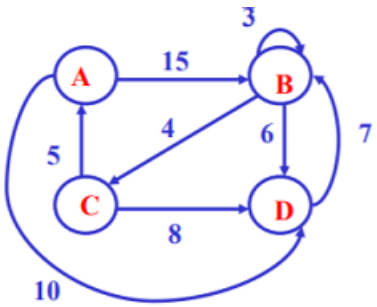
Rappresentazione in memoria: matrici di adiacenza Abbiamo una matrice quadrata dove indico, attraverso una marca (1, per esempio), se è presente un arco tra il nodo della riga e il nodo della colonna.

```
int graph[N][N];
```

	0	1	2	3
0	0	1	0	1
1	0	1	1	1
2	1	0	0	1
3	0	1	0	0

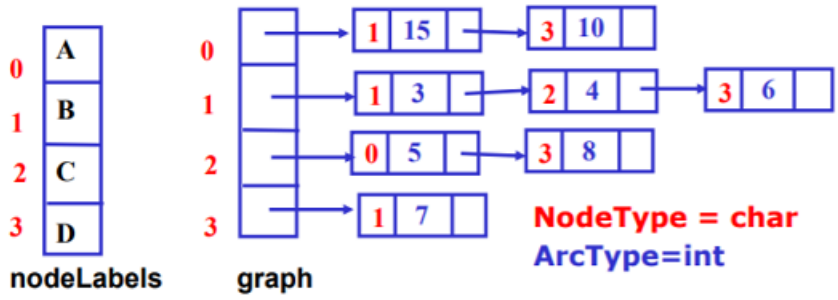
11.3.1.1 Grafi con nodi e archi etichettati

Questi grafi presentano informazioni aggiuntive riguardanti i nodi e gli archi. Le seguenti rappresentazioni in memoria sono simili alle precedenti ma presentano delle differenze.



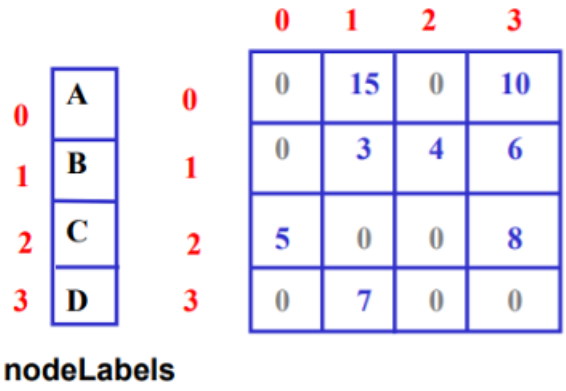
Rappresentazione in memoria: liste di adiacenza Abbiamo un array con cui associamo ad ogni nodo una certa etichetta, e il solito array visto prima che presenterà, in più, la lunghezza dell'arco!

```
struct Node {
        int NodeNumber;
        ArcType arcLabel;
        Node* next;
};
Node* graph[N];
NodeType nodeLabels[N];
```



Rappresentazione in memoria: matrici di adiacenza Abbiamo un array con cui associamo ad ogni nodo una certa etichetta e la matrice di adiacenza dove poniamo non la marca ma la lunghezza dell'arco: se non si hanno archi la lunghezza è zero (ovviamente non devo avere archi con lunghezza zero).

```
ArcType graph[N][N];
NodeType nodeLabels[N];
```



11.3.2 Come scelgo la rappresentazione più adeguata per il grafo?

Dipende dall'algoritmo che si desidera sviluppare. Tengo conto delle operazioni che si vogliono realizzare, della differenza di tempo nello svolgere l'operazione in una rappresentazione e in un altra.

Esempio Voglio trovare i successori di un nodo. Le due rappresentazioni sono analoghe, non si ha differenza tra l'adottare una rappresentazione o un'altra

Esempio 2 Voglio trovare i predecessori di un nodo. La migliore, in questo caso, è la matrice di adiacenza. Scorrendo la colonna trovo il predecessore di un nodo. Nell'altra rappresentazione devo scorrere tutti gli archi del grafo finchè non individuo il nodo che mi interessa.

Occupazione della memoria L'occupazione dipende dal numero degli archi.

- Se gli archi sono tanti (maggiore densità) le liste di adiacenza occupano di più rispetto alle matrici.
- Se ho archi sparsi (minore densità) può convenire una memorizzazione mediante liste di adiacenza.

11.3.3 Visita in profondità di una struttura

L'algoritmo è molto simile a quello per la *visita anticipata* degli alberi. Per non entrare in un ciclo infinito è necessario marcare i nodi già esaminati in modo da poterli saltare. Gli archi vengono esplorati a partire dall'ultimo nodo esaminato che abbia ancora degli archi non esplorati uscenti da esso.
Vengono visitati soltanto i nodi raggiungibili con un cammino dal nodo di partenza (li visiterebbe tutti in un grafo non orientato). L'algoritmo completo prevede un ulteriore funzione (*depthVisit*) per inizializzare un supporto alla marcatura e per applicare la *nodeVisit* a tutti i nodi.

Proviamo a scrivere un'implementazione in C++ sfruttando le classi

```
class Graph {
        struct Node {
                int NodeNumber;
                Node* next;
        };
        Node* graph[N];
        NodeType nodeLabels[N];
        int mark[N];

        void nodeVisit(int i) {
                mark[i] = 1;
                cout<< nodeLabels[i];

                Node* g; int j;
                for(g = graph[i]; g; g = g->next) {
                        j = g->nodeNumber;
                        if(!mark[j]) nodeVisit(j);
                }
        }
        public:
                void depthVisit() {
                        for(int i = 0; i < N; i++) mark[i] = 0;
                        for(int i = 0; i < N; i++)
                                if(!mark[i])
                                        nodeVisit(i);
                }
};
```

Tenendo conto di entrambe le funzioni la complessità della visita in profondità sarà

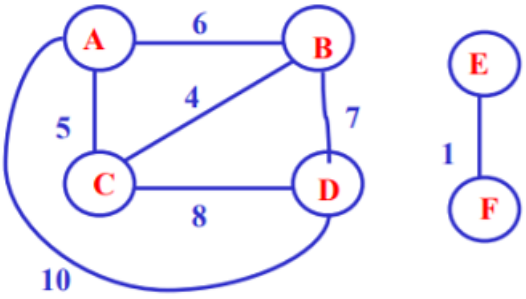
$$O(n) + O(m)$$

Ovviamente se $m \geq n$ avrò $O(m)$ (ricordare *regola della somma*).

Ricordiamo n è il numero di nodi, m il numero degli archi!

11.3.4 Grafi non orientati

Anche un grafo non ordinato consiste in una coppia (N, A), dove N è l'insieme dei nodi e A l'insieme di coppie non ordinate di nodi. Contrariamente ai grafi ordinati A non è un sottoinsieme del prodotto cartesiano e gli archi rappresentano un semplice collegamento tra due elementi: non si ha una direzione ben precisa come prima.



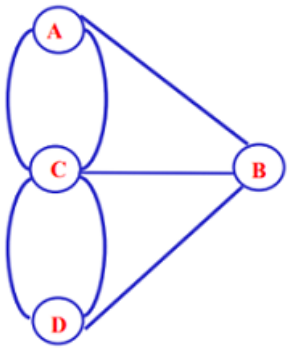
- I nodi sono adiacenti
- Un ciclo è caratterizzato da almeno tre nodi
- Un grafo non orientato con n nodi ha al massimo $n(n-1)/2$ archi

11.3.4.1 Rappresentazione in memoria

Abbiamo una rappresentazione simile a quella dei grafi orientati. Possiamo rappresentare un grafo sia con liste di adiacenza che con matrici di adiacenza. Ovviamente dobbiamo tenere conto che non si hanno direzioni ben precise come in un grafo orientato (quindi possiamo muoverci in entrambi i versi)

11.3.5 Multi-grafi non orientati

Un multi-grafo consiste in una coppia (N, A), dove N è l'insieme dei nodi e A *multi-insieme* di coppie non ordinate di nodi. Posso avere più archi che collegano due nodi: segue che non si ha relazione fra il numero di nodi e il numero di archi. Analogamente si definiscono multi-grafi non orientati.

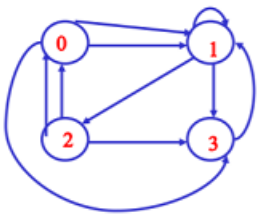


11.3.5.1 Rappresentazione in memoria

Il miglior metodo di rappresentazione è quello delle matrice di adiacenza. Qua di seguito abbiamo un esempio

int graph [N][N];

	0	1	2	3
0	0	2	0	1
1	0	1	1	1
2	2	0	0	1
3	0	1	0	0



Come valore poniamo il numero di archi che collegano un nodo con un altro.

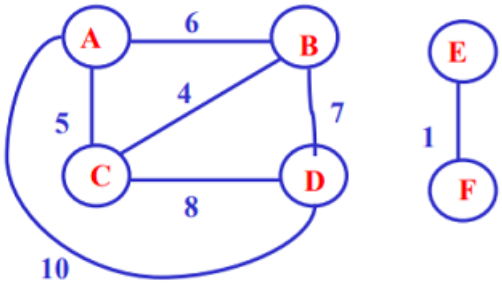
Capitolo 12

Martedì 05/05/2020

Registro 11 (*Mar 05/05/2020 10:30-13:30 (3:0 h) lezione*)

Algoritmo di Kruskal per trovare il minimo albero di copertura.

12.1 Minimo albero di copertura (*minimum spanning tree*)



Nei grafi non orientati è possibile avere un sottografo che connetta tutti i nodi connessi utilizzando il minor numero di archi possibile.

- Questo viene fatto prendendo un sottografo del grafo originario dove non ci sono cicli.
- Tuttavia è massimale perchè tutti gli elementi rimangono connessi.

Grafo connesso Un grafo non orientato si dice connesso se esiste un cammino fra due nodi qualsiasi del grafo. L'esempio qua sopra non è un grafo connesso.

Componente connessa Con componenti connesse si intendono sottografi connessi, gruppi di nodi collegati tra loro. Nell'esempio abbiamo due componenti connesse: quella con $ACBD$ e una con EF.

Componente connessa massimale Componente tale che nessun nodo è connesso ad uno appartenente ad un'altra componente connessa. Questa componente è quella che ci interessa trovare. Supponiamo di avere due nodi collegati tra loro, in modo diretto o indiretto: se si parla di componenti connesse massimali questi due nodi apparterranno per forza alla stessa componente connessa. Nella foto: A, B insieme costituiscono una componente connessa (non massimale); A, B, C, D costituiscono una componente connessa massimale.

Definizione Un *albero di copertura* è un insieme di componenti connesse massimali acicliche.

Perchè albero? Essendo aciclico lo possiamo vedere come un albero (o una foresta di alberi). La radice di ciascun albero sarà scelta in modo arbitrario. La prima componente può essere vista come un albero dove A è la radice, C è figlio di A, B è figlio di C e D è figlio di B. La stessa potrebbe essere vista come un albero che ha B come radice, C e D figli di B ed A figlio di C.

Utilità Non mi interessa conoscere i cammini del grafo, voglio sapere quali nodi sono collegati e quali sono scollegati.

12.1.1 Algoritmo di Kruskal

Per un certo grafo esistono tanti alberi di copertura. Dato un grafo possiamo escludere alcuni grafi realizzando alcune versioni. Nell'esempio potrei porre l'arco 8 al posto dell'arco 4, o l'arco 6 invece dell'arco 5. L'algoritmo di Kruskal trova tra tutti gli alberi di copertura possibili quello **minimo**.

Cosa significa minimo? Sommando le lunghezze di tutti gli archi si ottiene il valore più piccolo possibile.

Algoritmo greedy L'algoritmo di Kruskal è un *algoritmo greedy*.

Prima fase La prima fase del procedimento può essere evitata se non abbiamo interesse nel trovare il minimo albero di copertura (potrei essere interessato a un qualunque albero di copertura, o semplicemente a trovare le componenti connesse).

12.1.1.1 Procedimento

1. Prendo tutti gli archi del grafo e li pongo in ordine crescente.
   Se gli archi sono m avrò complessità $O(m \log m)$.
2. Costruisco le componenti connesse:
   - Inizialmente i nodi sono considerati componenti connesse individuali.
   - Prendo gli archi e li considero uno alla volta:
     - Verifico se l'arco connette due componenti ancora non connesse
     - Se le componenti non sono connesse l'arco sarà parte del minimo albero di copertura finale
     - Se i due nodi sono già connessi scarto l'arco

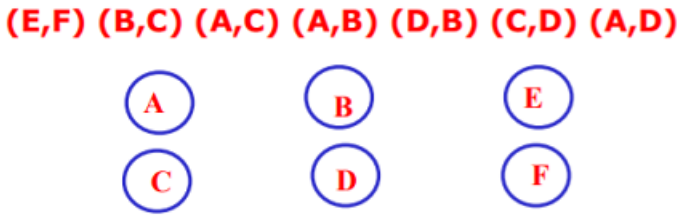
   Rappresentiamo le componenti connesse come liste: complessivamente presentano n nodi. Per scorrere queste liste avrò complessità lineare $O(n)$. Tengo conto che dovrò unire più liste: tengo un puntatore all'ultimo elemento della lista e con complessità $O(1)$ effettuo il collegamento. Se devo connettere n nodi avrò complessità $O(n\,m)$. Ciò non va bene: otterremo con quanto spiegato più avanti (nell'implementazione) complessità $O(m \log n)$.

Alla fine avrò complessità $O(m \log n)$, con $m \in O(n^2)$.

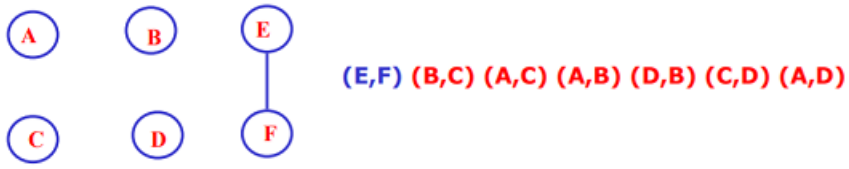
12.1.1.2 Esempio

Prendiamo l'esempio di grafo non ordinato presente a inizio capitolo: vogliamo ottenere il minimo albero di copertura (anch'esso raffigurato a inizio capitolo).

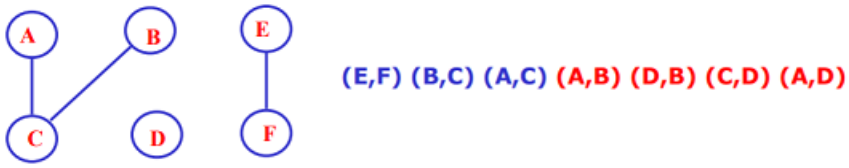
- Abbiamo i seguenti archi, posti in ordine crescente: $(E, F), (B, C), (A, C), (A, B), (D, B), (C, D), (A, D)$. Il primo ha lunghezza 1, l'ultimo ha lunghezza 10!
- Inizialmente abbiamo le seguenti componenti connesse, ciascuna caratterizzata da un solo nodo.



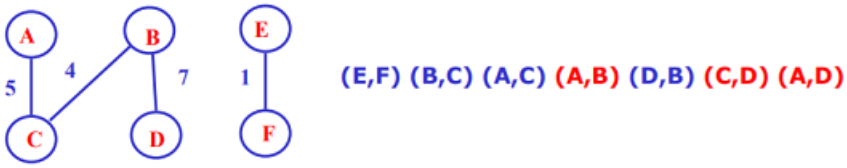
- Prendo l'arco (E, F): esso congiunge componenti ancora non connesse, quindi l'arco sarà parte del minimo albero di copertura



- Stesso discorso per gli archi (B, C) e (A, C).



- Prendo l'arco (A, B): esso congiunge componenti già connesse. Lo ignoro poichè la sua inclusione comporterebbe avere un ciclo (violando quanto detto all'inizio)
- Includo l'arco (D, B).

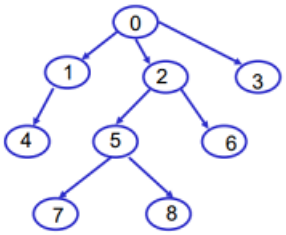


- Scarto gli archi rimanenti: (C, D) e (A, D).

- Ho trovato il minimo albero di copertura (con lunghezza 17) e due componenti massimali connesse.

12.1.1.3 Salvataggio delle componenti connesse in alberi generici mediante array

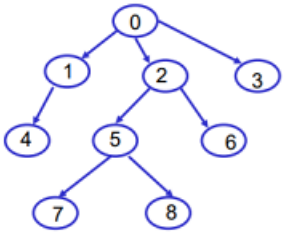
Per ottenere una migliore complessità rappresenteremo le componenti connesse non mediante liste, ma attraverso alberi generici. In questo caso non adotteremo il salvataggio in albero binario secondo la regola *figlio-fratello*, ma ricorreremo a un un array dove ogni nodo è numerato attraverso gli indici. Ogni elemento, ciascuno rappresentante un nodo, ha per valore il nodo padre.



tree[i] è il padre di i

indice	0	1	2	3	4	5	6	7	8
tree	-	0	0	0	1	2	2	5	5

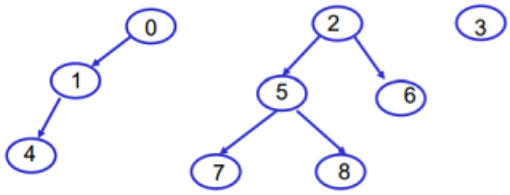
- Trovare il padre di un nodo i è semplice: ho complessità $O(1)$.
- Trovare i figli richiede complessità lineare, poichè devo scorrere tutto l'array.
- Collegare un nodo (già legato a un altro) a un altro ancora, quindi a un nuovo padre, richiede complessità $O(1)$. Vediamo l'esempio seguente



tree[i] è il padre di i

indice	0	1	2	3	4	5	6	7	8
tree	-	0	0	0	1	2	2	5	5

Foresta di alberi Nell'implementazione dell'algoritmo di Kruskal non salverò un albero ma una foresta di alberi! Supponiamo, riprendendo il solito esempio, di avere tre alberi. Conoscendo il numero complessivo di nodi posso includere il tutto in un unico array.



indice	0	1	2	3	4	5	6	7	8
tree	-	0	-	-	1	2	2	5	5
livello	2		2	0					

Come prima il valore di ogni casella rappresenta il padre del nodo. Osservo tre valori indefiniti, quindi tre alberi. Associo agli elementi con valore indefinito il livello di profondità dell'albero.

Unione di due componenti Due componenti vengono unite ponendo la testa di una come figlia di un'altra, il tutto con complessità $O(1)$. Nel compiere la fusione devo mantenere l'altezza dell'albero più bassa possibile, in modo da avere maggiore efficienza.

- **Strategia adottata**: l'albero con livello minore sarà legato alla radice di quello con livello maggiore mantenendo gli alberi meno profondi.

Controllo di appartenenza allo stesso albero Per verificare l'appartenenza verifico gli antecedenti dei due nodi.

- Non appena individuo un antecedente comune posso fermarmi e dire che i due elementi appartengono allo stesso albero.
- Se arrivo alle radici e si hanno due radici diverse i due elementi non appartengono allo stesso albero.

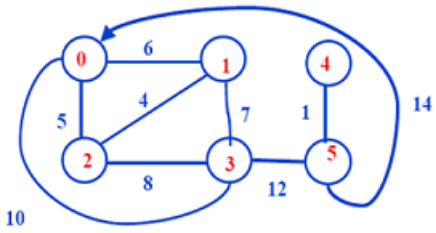
Effettuerò al massimo tanti passi quanti i livelli dell'albero maggiore. Tutto questo ha complessità logaritmica.

12.1.1.4 Implementazione di Kruskal

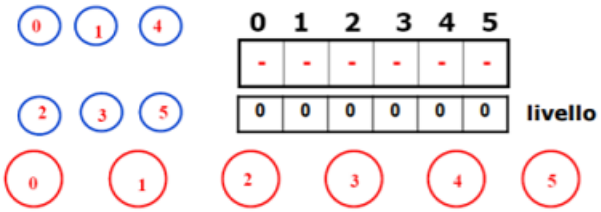
- I nodi sono numerati
- Le componenti connesse sono memorizzate come insieme di alberi generici: inizialmente sono solo radici, nel tempo si costituiscono i vari alberi.
- Sono memorizzate in array: ogni nodo contiene l'indice del padre
- Se due nodi appartengono alla stessa componente risalendo si incontra un antenato comune
- Due alberi sono unificati inserendo quello meno profondo (con livello minore) come sottoalbero della radice di quello più profondo.

Esempio

- Applichiamo Kruskal al seguente grafo



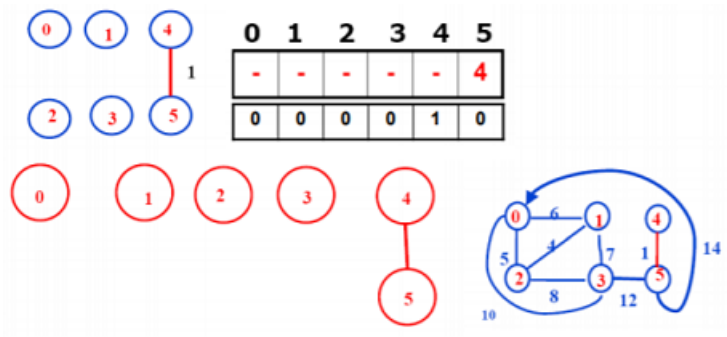
- Inizialmente abbiamo componenti individuali: l'array rappresenta una foresta di alberi, ciascuno con la sola radice

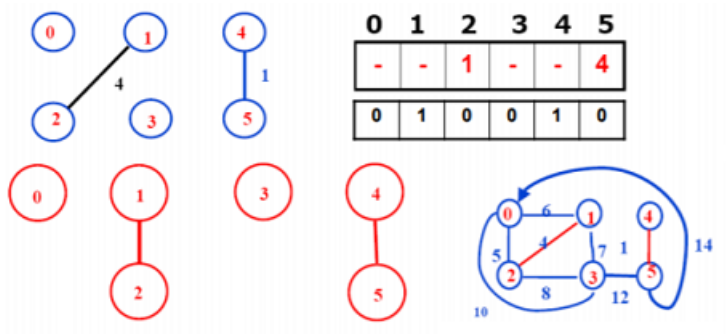


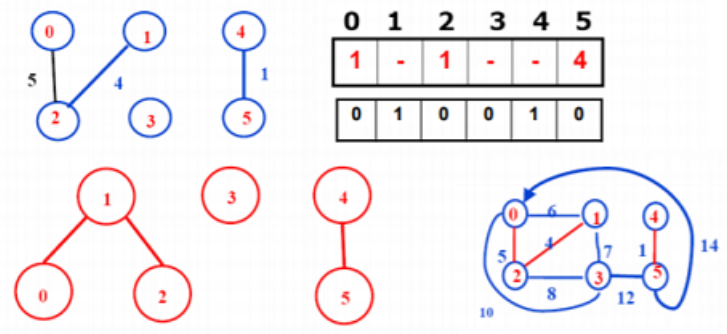
- Ordiniamo gli archi in modo crescente:

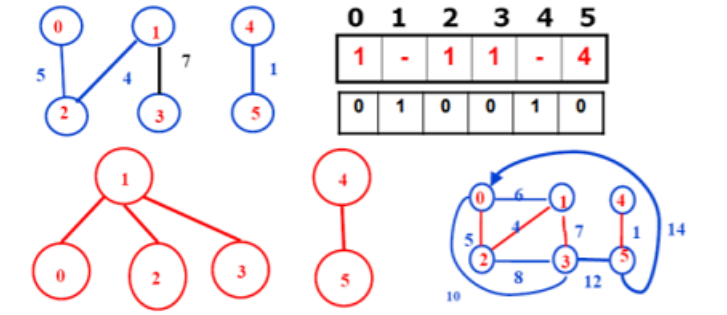
- (4,5) lunghezza 1
- (2,1) lunghezza 4
- (0,2) lunghezza 5
- (0,1) lunghezza 6
- (1,3) lunghezza 7
- (2,3) lunghezza 8
- (0,3) lunghezza 10
- (3,5) lunghezza 12
- (5,0) lunghezza 14

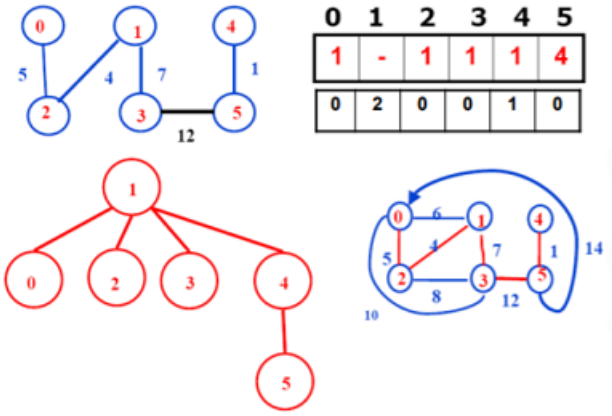
- Pongo gli archi secondo le regole già viste.











12.2 Algoritmo PageRank di Google

I primi motori di ricerca erano basati esclusivamente sul contenuto, su algoritmi artigianali. Nel tempo i meccanismi sono stati raffinati, arrivando all'algoritmo *PageRank*. Io non mi baso sesclusivamente sul contenuto, ma carico una lista di risultati basandomi sul *rank* della pagina (cioè le più importanti, quelle più visitate maggiormente).

Grafo Consideriamo la rete come un grafo: i nodi sono le pagine web, gli archi i link (collegamenti).

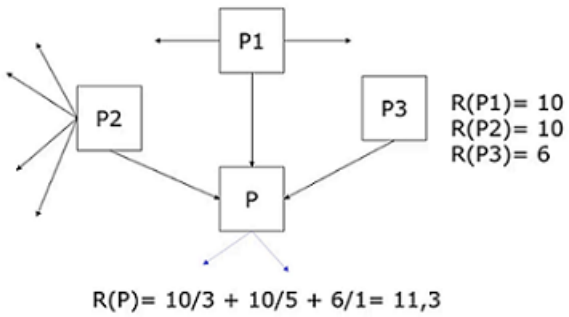
Rilevanza di una pagina web La rilevanza di una pagina web P dipende da quanto sono rilevanti le pagine che puntano a P, cioè quelle che hanno un link che porta a P. Si considera inoltre il numero di link presenti in queste pagine puntanti a P: l'importanza viene penalizzata se una pagina presenta tantissimi link al suo interno.

Formula base

$$R(P) = \sum_{Q \to P} \frac{R(Q)}{|Q|}$$

Dove $|Q|$ consiste nel numero di link uscenti da Q e $Q \to P$ i link da QaP. Si osserva una ricorsività. Il calcolo viene fatto in modo iterativo utilizzando la matrice di adiacenza e partendo da un valore del rango uguale per tutti i nodi. La formula è molto semplificata, in alcuni casi potrebbe essere necessario un aggiustamento per avere convergenza.

Esempio



Versioni recenti La versione vista è una delle prime. Adesso l'algoritmo considera anche altre informazioni, come i giornali, la struttura della pagina web (in particolare se il sito è *mobile-friendly*).

Capitolo 13

Venerdì 08/05/2020

13.1 Algoritmo di Dijkstra

L'algoritmo di Dijkstra può essere applicato ai grafi orientati (e non solo) che hanno peso positivo sugli archi, cioè se ho archi con peso negativo o nullo l'algoritmo non funziona.

Obiettivo Trovare i cammini minimi da un nodo di partenza a tutti gli altri nodi del grafo.

Metodologia adottata Metodologia greedy.

Applicazione pratiche

- **Instradamento di messaggi nella rete**: devo trovare il percorso minimo (ricordiamo che la rete può essere percepita come un grafo).
- **Un navigatore**: si individua il percorso migliore per muoversi da un luogo a un altro.

Filosofia I cammini minimi sono tutti posti a infinito e vengono aggiornati con stime via via sempre più precise. Ciò avviene tramite un ciclo in cui ad ogni iterazione un nuovo nodo viene "sistemato", nel senso che il cammino fino a lui viene stabilizzato. Alla fine tutti i nodi saranno "sistemati".

Spiegazione

- Utilizzo due tabelle: una *distanza* (dist) e una *predecessore* (pred) con n elementi (n consiste nel numero di nodi).
- Per ogni nodo A ho
  - $dist(A)$, che contiene in ogni momento la lunghezza di un cammino dal nodo iniziale ad A.
  - $pred(A)$, il predecessore di A in questo cammino.
- I nodi sono divisi in due gruppi:
  - Quelli già sistemati, per i quali i valori *dist* e *pred* sono definitivi e non cambieranno più. Il primo contiene la lunghezza del minimo cammino: assieme al secondo valore (che punta all'elemento precedente) è possibile ricostruire l'interno percorso.
  - Quelli da sistemare (inizialmente la totalità), che costituiscono il cosiddetto *insieme* Q. Qua i valori *dist* e *pred* sono relativi al cammino trovato fino a quel momento. Questi valori saranno cambiati se si trova un cammino più corto.

- Inizialmente nessun nodo è sistemato: Q contiene tutti i nodi
- Si esegue un ciclo di n passi. Ad ogni passo
  - Si considera sistemato il nodo, fra quelli di Q, con dist minore e lo si toglie da Q
  - Si aggiornano pred e dist per gli immediati successori di questo nodo (li levo da Q, modifico *dist* e *pred*, li reinserisco in Q)
- Il ciclo termina dopo $n-1$ passi, cioè quando Q contiene un solo nodo

13.1.1 Implementazione e complessità

Inizializzazione del programma Vediamo un'implementazione un tantino più precisa.

```
Q = N
per ogni nodo p diverso da p0 {
        dist(p)=infinito
        pred(p) = vuoto
}
dist(p0)=0;
```

- Pongo Q all'insieme degli elementi N di tutti i nodi.
- Uguaglio i valori dist e pred come detto prima: per il primo il valore più alto possibili, per il secondo vuoto.
- Pongo inoltre che il cammino dal nodo iniziale a se stesso è uguale a zero.

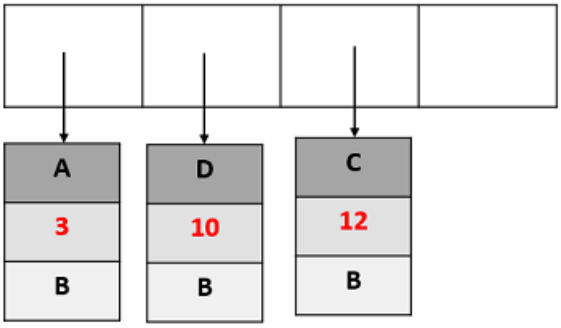
Ciclo Vediamo adesso il ciclo, il cuore dell'algoritmo.

```
while(Q contiene piu di un nodo) {
        estrai da Q il nodo p con minima dist(p);
        per ogni nodo q successore di p {
                lpq = lunghezza arco (p,q);
                if(dist(p)+lpq < dist(q)) {
                        dist(q)=dist(p)+lpq;
                        pred(q)=p;
                        re-inserisci in Q il nodo q modificato;
                }
        }
}
```

- Esco dal ciclo soltanto quando rimane un solo nodo in Q
- Estraggo da Q il nodo p avente distanza minima (guardiamo soltanto i nodi da sistemare). Il nodo estratto è un qualcosa di già sistemato, poichè il cammino è già stato individuato.
- Aggiorniamo i nodi immediatamente successivi a p, ormai sistemato. Per ogni nodo q
  - calcolo la lunghezza dell'arco, cioè quello che collega p e q
  - Ci poniamo la seguente domanda: va bene il percorso passante per q scelto prima o ci conviene invece passare dal q?
  - Se il percorso passante dal nuovo q è più conveniente provvedo aggiornando i valori dist e pred di q
  - Del re-inserimento in Q del nodo q ne riparleremo, la cosa serve soprattutto per ottenere maggiore efficienza.

Complessità

- Effettuo delle operazioni di assegnamento in un ciclo per inizializzare i valori dist e pred di ciascun nodo: complessità $O(n)$.
- Ogni volta devo scorrere la lista di elementi per individuare quello con distanza minore: per ottenere complessità $O(\log n)$ implemento il tutto in un *minheap*. La complessità $O(\log n)$ è dovuta all'estrazione (l'elemento deve essere eliminato da Q, non solo individuato).



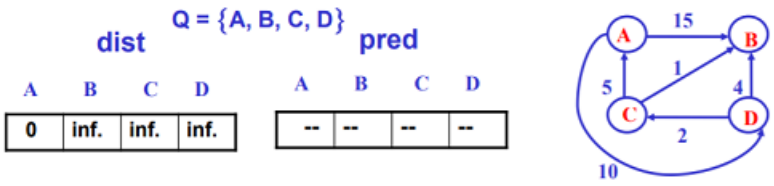

```
struct nodeD {
labeltype label ;
int dist;
int pred;
}
```

- Per ogni nodo q analizzo i suoi successori. La modifica di questi oggetti, collocati nel *minheap*, avviene con complessità logaritmica: estraggo gli elementi da Q (funzione di estrazione con complessità logaritmica), modifico i valori *dist* e *pred* (complessità $O(1)$), reinserisco gli elementi nel minheap (complessità logaritmica, saranno collocati all'interno dello heap in base ai nuovi valori *dist*).
- Ho n iterazioni del ciclo while.
- La complessità dell'iterazione è $C[$ estrazione da Q del nodo p con minima *dist* $] + m/nC[$ reinserimento in Q del nodo q modificato $]) = O(\log n + (m/n)\log n)$
- Segue che la complessità del ciclo sarà

$$O(n(\log n + (m/n)\log n)) = \boxed{O(n\log n + m\log n)}$$

13.1.2 Esempio

Abbiamo il seguente esempio:



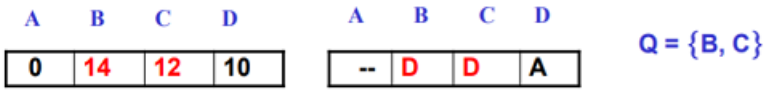
Primo passo

- Il nodo A è quello di partenza (osservare il valore nullo del dist). Gli altri hanno dist infinito. Tutti hanno pred nullo.
- Voglio trovare il cammino minimo da A a B, da A a C e da A a D.
- L'insieme Q contiene tutti gli elementi presenti nel grafo ($Q = \{A, B, C, D\}$)
- La prima volta la cosa è semplice: A è l'unico elemento a non avere dist infinito, ed è il più piccolo pur essendo nullo. Lo estraggo da Q.
- Vado a fare quei confronti spiegati precedentemente. I successori di A sono B e D, C non lo considero.
  - $dist(b)$ ha un certo valore. La distanza migliore è questa o quella che si ottiene sommando $dist(A)$ con la lunghezza dell'arco che collega A e B? Tenendo conto del valore infinito di $dist(b)$ la risposta è semplice. Pongo $dist(B) = 15$) e $pred(B) = A$
  - $dist(A)$ ha un certo valore. La distanza migliore è questa o quella che si ottiene sommando $dist(A)$ con la lunghezza dell'arco che collega A e D? Stessa risposta di prima. Pongo $dist(D) = 10$ e $pred(B) = A$.
- La situazione dei dati è la seguente

Osservando il grafico si può intuire che esistono percorsi migliori. Questi saranno individuati nelle fasi successive!

Secondo passo

- Ho l'insieme $Q = \{B, C, D\}$.
- Estraggo il nodo D, che presenta dist minore.
- Guardiamo i successori di D, cioè B e C:
  - Per andare a B è migliore il percorso trovato prima con lunghezza 15 o quello nuovo in cui sommo a $dist(D)$ la lunghezza dell'arco che collega D e B? Quest'ultimo percorso ha lunghezza 14, quindi è migliore. Pongo $dist(B) = 14$ e $pred(B) = D$.
  - Per andare a C è migliore il percorso già presente (infinito) oppure quello nuovo in cui sommo a $dist(D)$ la lunghezza dell'arco che collega D e C? La risposta è semplice. Pongo $dist(C) = 12$ E $pred(C) = D$.
- La situazione dei dati è la seguente



Anche in questo caso notiamo possibili modifiche.

Terzo passo

- Ho l'insieme $Q = \{B, C\}$.
- Estraggo il nodo C, che presenta dist minore.
- Guardiamo i suoi successori, cioè B ed A.
  - A non lo considero perchè fa parte dei nodi già sistemati. Comunque sia anche facendo il confronto vedo che il percorso già scelto è migliore di quello nuovamente ipotizzato ($12+5 < 0$).
  - Osservo invece che il nuovo percorso proposto per raggiungere il nodo B è migliore ($12+1 < 14$) di quello che avevamo già salvato. Pongo $dist(B) = 13$ e $pred(B) = C$.
- A questo punto ho concluso. Ho $Q = \{B\}$ e la situazione dei dati è la seguente

A	B	C	D
0	13	12	10

A	B	C	D
--	C	D	A

Percorsi individuati

- **Da A a B**: $A \to D \to C \to B$ con lunghezza 13
- **Da A a C**: $A \to D \to C$ con lunghezza 12
- **Da A a d**: $A \to D$ con lunghezza 10

Tabella riassuntiva

		A	B	C	D
	Q = {A, B, C, D}	0 /-	i /-	i /-	i/-
A	Q = {B, C, D}	0 /-	15/A	i /-	10/A
D	Q = {B, C}	0 /-	14/D	12/D	10/A
C	Q = {B}	0/-	13/C	12/D	10/A

13.1.3 Perchè l'algoritmo funziona?

- Possiamo dimostrare che ogni nodo quando viene definito sistemato non sarà più modificato.
- Inoltre il cammino minimo per i nodi già scelti passa soltanto da nodi già scelti.

Capitolo 14

Martedì 12/05/2020

Registro 12 (*Mar 12/05/2020 10:30-13:30 (3:0 h) lezione*)

Complessità degli algoritmi di teoria dei numeri. Algoritmo della moltiplicazione veloce. Cenni alla NP-completezza: problemi difficili, gli insiemi P e NP, il teorema di Cook, problemi non calcolabili.

14.1 Teoria dei numeri

Nella teoria dei numeri non dobbiamo considerare il fatto che quando facciamo operazioni semplici abbiamo complessità $O(1)$. Quando i numeri iniziano ad essere grandi, per esempio possiedono un numero elevato di cifre decimali: a quel punto anche operazioni elementari come l'addizione avranno complessità più elevate. Vedremo che la complessità dipende dal numero di cifre (n) che compongono il numero.

- L'addizione ha complessità $O(n)$
- La moltiplicazione ha complessità $O(n^2)$

Gli algoritmi con quella complessità sono quelli imparati alle scuole elementari. La teoria dei numeri è estremamente importante, soprattutto per quanto riguarda la crittografia.

14.1.1 Moltiplicazione veloce fra interi non negativi

La moltiplicazione[1] può essere effettuata attraverso algoritmi alternativi con complessità leggermente inferiore.

Metodo Supponiamo di dividere un numero A in due parti, ottenendo

$$A = A_s 10^{n/2} + A_d$$

Prendiamo ad esempio $A = 1325$. Posso porre $A_s = 13$ e $A_d = 25$, ottenendo

$$A = 1325 = 13 * 10^2 + 25$$

con $n = 4$ (numero di cifre). La parte sinistra è la più significativa, la parte destra quella meno significativa. Facciamo lo stesso con un numero B che per comodità avrà lo stesso numero di cifre

$$B = B_s 10^{n/2} + B_d$$

[1] *Is this a Berselli reference?*

e svolgiamo la moltiplicazione AB:

$$AB = A_sB_s10^n + \boxed{(A_sB_d + A_dB_s)}10^{n/2} + A_dB_d$$
$$(A_s + A_d)(B_s + B_d) = \boxed{A_sB_d + A_dB_s} + A_sB_s + A_dB_d$$
$$\boxed{A_sB_d + A_dB_s} = (A_s + A_d)(B_s + B_d) - A_sB_s - A_dB_d$$
$$AB = A_sB_s10^n + \boxed{((A_s + A_d)(B_s + B_d) - A_sB_s - A_dB_d)}10^{n/2} + A_dB_d$$

E se volessi moltiplicare per 100? Ovviamente l'applicazione dell'algoritmo, in questo caso, risulta superflua. Mi limito ad aggiungere due zeri a destra! Stessa cosa quando moltiplico due numeri con zeri in fondo: rimuovo gli zeri in fondo, moltiplico i due numeri, riaggiungo gli zeri in fondo al risultato.

Programmino Analizziamo il seguente programma, dove abbiamo per argomento i numeri da moltiplicare e il numero di cifre n (anche qua che i numeri hanno lo stesso numero di cifre)

```
numero mult(numero A, numero B, int n) {
        if(n == 1) return A*B;
        else {
                A_s = parte sinistra di A; A_d = parte destra di A;
                B_s = parte sinistra di B; B_d = parte destra di B;

                int x1 = A_s+A_d; int X2 = B_s+B_d;
                int y1 = mult(x1, x2, n/2);
                int y2 = mult(A_s, B_s, n/2);
                int y3 = mult(A_d, B_d, n/2);

                int z1 = left_shift(y2, n);
                int z2 = left_shift(y1-y2-y3, n/2);
                return z1+z2+y3;
        }
}
```

- Svolgo il prodotto soltanto se il numero di cifre n è uguale a 1 (Complessità $O(1)$ poichè ho una sola cifra).
- Altrimenti:
  - Prendo di entrambi i numeri la parte sinistra e la parte destra (Complessità $O(n/2)$ poichè ho un numero di cifre dimezzato per ciascun elemento)
  - Cerco di eseguire l'espressione ottenuta nella pagina precedente:
    - Trovo le somme $A_s + A_d$ e $B_s + B_d$ (Complessità $O(n/2)$, sommo due volte due numeri di lunghezza $n/2$).
    - Effettuo chiamate ricorsive per svolgere tre moltiplicazioni: A_sB_s, A_dB_d, A_dB_d.
    - Con la funzione $left_shit$ effettuo moltiplicazioni di valori per 10^m.
    - Restituisco la somma $z1 + z2 + y3$, cioè il risultato dell'espressione.

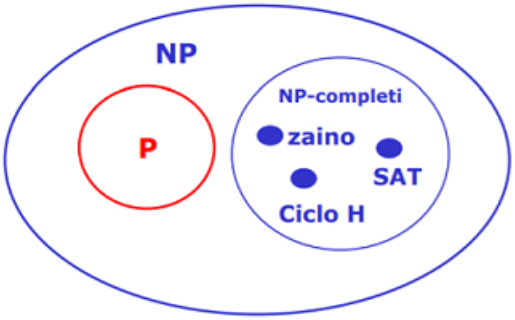
Complessità Ho la seguente relazione di ricorrenza

$$\begin{cases} T(1) = d \\ T(n) = bn + 3T(n/2) \end{cases}$$

Segue che $T(n) \in O(n^{\log_2 3})$, con $\log_2 3 = 1.59$.

shift L'operatore di traslazione visto a *Fondamenti di programmazione* rappresenta la moltiplicazione detta prima.

14.2 Problemi difficili: cenni alla NP-Completezza



Questi problemi sono molto frequenti nell'informatica. I seguenti algoritmi possono essere risolti analizzando tutte (o quasi) le combinazioni presenti. Segue, a causa del numero esponenziale di queste combinazioni, complessità esponenziale!

Problemi campati per aria? Non direi, sono problemi con molte applicazioni pratiche, legati alla vita quotidiana. Immaginiamo a un falegname: come utilizzare in modo più efficiente possibili un blocco di legno? Questa cosa è simile al problema dello zaino.

14.2.1 Problema dello zaino

Vogliamo ottimizzare il riempimento di uno zaino, avente una certa capacità. Ogni oggetto ha un peso e un valore.

- Voglio determinare il numero di oggetti di ogni tipo in modo tale che il peso sia minore o uguale di un dato valore (valore = capacità dello zaino) e il valore totale sia il maggiore possibile.

14.2.2 Problema del commesso viaggiatore

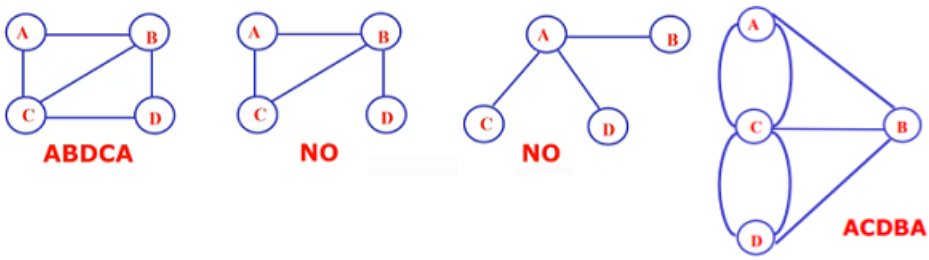
Abbiamo delle città: l commesso viaggiatore deve svolgere un certo percorso.

- Voglio trovare un cammino per visitare tutte le città una e una sola volta per poi tornare alla città di partenza.

14.2.2.1 Ciclo hamiltoniano

Questo ciclo è stato inventato da William Rowan Hamilton per un gioco di società. Dato un multi-grafo (cioè un grafo in cui due nodi possono essere collegati da più archi) voglio trovare, se possibile, un ciclo che tocca tutti i nodi una e una sola volta.

Grafo hamiltoniano Un grafo si dice hamiltoniano se possiede un ciclo hamiltoniano.



14.2.3 Problema SAT: soddisfattibilità di una formula logica

Data una formula F con n variabili trovare, se possibile, una combinazione di valori booleani che, assegnati alle variabili di F, la rendono vera.

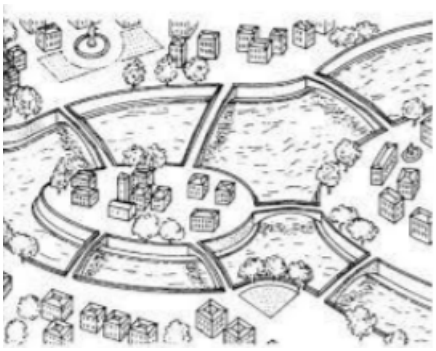
- F = (x AND NOT x) OR (y AND NOT y).
  In questo caso la formula non è soddisfattibile. Ho $n = 2$.
- F = (x AND NOT y) OR (NOT x AND y).
  La formula è soddisfattibile, per esempio ponendo $x = 0, y = 1$. Ho $n = 2$.

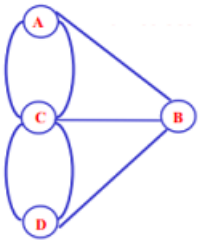
Anche questo è un problema difficile! Se le variabili che compaiono nella formula sono n, si provano tutte le combinazioni di valori delle variabili, che sono 2^n.

14.2.4 Problema del ciclo euleriano [Risolvibile]

Alcune volte i problemi sembrano difficili, ma in realtà risolvibili. Un esempio è il problema del *ciclo Euleriano*. Eulero passeggiava ogni giorno per Konigsberg. Si pose la seguente domanda: è possibile fare una passeggiata attraversando tutti questi ponti una volta soltanto? Il percorso è ciclico, cioè destinazione finale e iniziale coincidono.

Multigrafo Il percorso con i ponti di Konigsberg può essere rappresentato attraverso un multigrafo. Voglio trovare, se possibile, un ciclo che percorre tutti gli archi una e una sola volta. Ecco il risultato





Somiglianze Potrebbe sembrare simile al problema del ciclo hamiltoniano, ma in realtà le cose sono diverse. Nel ciclo hamiltoniano si parla di nodi, nel ciclo euleriano di archi!

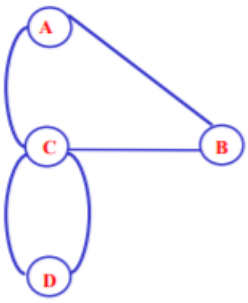
14.2.4.1 Teorema di Eulero

Un multi-grafo non orientato contiene un ciclo Euleriano se e solo se gli archi che partono da ogni nodo **sono in numero pari**.

Verifica Possiamo verificare questa proprietà sul grafo (complessità polinomiale) per sapere se esiste un ciclo. Appena individuo un nodo avente numero dispari di archi possono fermarmi e dire che non esistono cicli euleriani.

Risoluzione del problema dei ponti Se riprendiamo il multigrafo troviamo che ci sono dei nodi che presentano un numero dispari di archi. Segue che non esiste un percorso con le condizioni richieste.

Esempio 2 Quest'altra situazione, invece, ha possibili soluzioni: ogni nodo presenta un numero pari di archi



Un esempio è il ciclo $ABCDCA$.

14.2.5 Teoria della NP-completezza

Alcuni problemi, tipo quella della *Torre di Hanoi*, sono per forza esponenziali! Non c'è speranza che diventino problemi non esponenziali. I problemi classificati come *NP-Completi* non sono considerati pienamente esponenziali e potrebbero essere risolti, in futuro, mediante algoritmi più veloci.
Risulta possibile, per il momento, creare una versione minore di questi algoritmi: una versione decisionale dove la risposta è **sì o no**!

- Ogni problema può essere riformulato come problema decisionale
- Il problema decisionale ha complessità minore o uguale al problema non decisionale corrispondente. Li collochiamo tra i problemi con complessità polinomiale e quelli con complessità esponenziale.
- Se il problema decisionale è difficile, a maggior ragione lo sarà il corrispondente.

Esempi

- **Commesso viaggiatore**: dato un intero k, esiste nel grafo un ciclo senza ripetizione di nodi di lunghezza minore di k?
- **Zaino**: dato un valore v, esiste un riempimento dello zaino con valore maggiore o uguale a v?
- **Ciclo hamiltoniano**: dato un grafo, esiste un ciclo Hamiltoniano?
- **Ciclo euleriano**: dato un grafo, esiste un ciclo Euleriano?
- **Formula logica**: data una formula, esiste un assegnamento alle variabili che rende vera la formula?

Legame tra problemi NP-Completi La cosa interessante è che questi problemi sono tutti collegati tra loro. Questo significa che se in futuro qualcuno troverà un metodo di risoluzione per uno di questi, allora potrà trovarlo anche per tutti gli altri.

14.2.5.1 Algoritmi *nondeterministici*

Un algoritmo non deterministico contiene al suo interno il comando *choice(I)* (una sorta di oracolo). La funzione, che ha per argomento un insieme I, restituisce un elemento dell'insieme che dovrebbe permettere la risoluzione del problema (o almeno avvicinarsi alla stessa). Questi algoritmi hanno un interesse puramente **TEORICO**.

Esempio con la NSAT Scriviamo un problemino

```
int nsat(Formula f, int* a, int n) {
        for(int i = 0; i < n; i++)
                a[i] = choice({0,1});
        if(value(f,a))
                return 1;
        else
                return 0;
}
```

- L'array a raccoglie le variabili della formula.
- La formula è rappresentata mediante albero binario: i nodi consistono in operatori e valori.
- La funzione value (con complessità $O(n)$) permette di valutare la formula f dati i valori a *true/false* generati prima con *choice*.
- Se il responso della valutazione è positivo la funzione NSAT restituisce 1, altrimenti 0.

Il programmino ha complessità $O(n)$: siamo riusciti ad abbattere la complessità esponenziale di NSAT! Tuttavia la funzione choice è **IDEALE**, quindi il problema non è risolto. Una futura implementazione del choice permetterà l'adozione di questo algoritmo con complessità più bassa.

Ricerca nondeterministica Effettuiamo una ricerca in un array con dimensione n.

```
int nsearch(int* a, int n, int x) {
        int i = choice({0 ... n-1});
        if(a[i] == x)
                return 1;
        else
                return 0;
}
```

Individuo un certo indice in modo nondeterministico: se ho corrispondenza restituisco 1, altrimenti 0. Una risoluzione del genere avrebbe complessità $O(1)$.

Ordinamento nondeterministico Immaginiamo l'ordinamento di un array.

```
int nsort(int* a, int n) {
        int b[n];
        for(int i = 0; i < n; i++)
                b[i] = a[i];

        for(int i = 0; i < n; i++)
                a[i] = b[choice({0 ... n-1})];

        if(ordinato(a))
                return 1;
        return 0;
}
```

Creo una copia dell'array a (b), riscrivo l'array a basandomi sui risultati della funzione *choice*, che mi restituisce indici di elementi dell'array b. Il tutto con complessità $O(n)$.

Relazione fra determinismo e non determinismo Per ogni algoritmo nondeterministico ne esiste uno deterministico che lo simula, esplorando lo spazio delle soluzioni, fino a trovare un successo.

14.2.6 Defininizioni: Insieme P e NP

- P (polinomiale): insieme di tutti i problemi decisionali risolubili in tempo polinomiale con un algoritmo deterministico
  Esempi: ricerca, ordinamento, ciclo Euleriano...
  - i problemi decisionali in P sono facili da risolvere
- NP (nondeterministico polinomiale[2]): insieme di tutti i problemi decisionali che ammettono un algoritmo polinomiale di verifica di una soluzione (certificato)[3].
  Esempi: ricerca, ordinamento, fattorizzazione, soddisfattibilità, zaino, commesso viaggiatore, ciclo hamiltoniano
  - i problemi decisionali in NP sono facili da verificare

Esempi di verifica

- **Ciclo hamiltoniano**: dato un cammino verifico con complessità polinomiale $O(n)$ se questo è un ciclo hamiltoniano
- **Formula logica**: dato un assegnamento di valori booleani alle variabili verifico in tempo $O(n)$ se questi rendono vera la formula

Dimostrazione dell'appartenenza a NP Per dimostrare che un problema R appartiene a NP si dimostra che la verifica di una soluzione di R è fatta in tempo polinomiale.

Grande domanda dell'informatica Sappiamo con certezza che $P \subseteq NP$. Possiamo dire che...

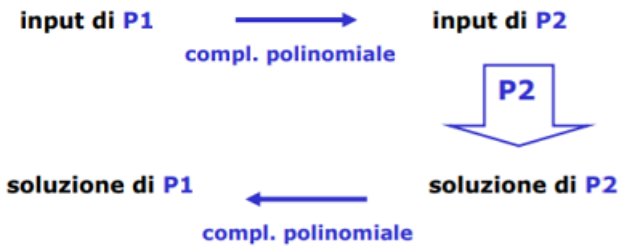
$$P = NP?$$

La cosa avrebbe grandi conseguenze: dati NP e P, sapendo che i primi sono facili da verificare i secondi da risolvere porli uguali significherebbe affermare il principio che a verifica semplice segue soluzione semplice di un problema!
Questa è la grande domanda dell'informatica: un istituto matematico degli USA ha promesso un milione di dollari a chi risponderà a questa domanda.

14.2.7 Riducibilità

La riducibilità è un metodo per ridurre un problema ad un altro: questo ci permette, data la risoluzione di un problema, di risolverne un altro. Abbiamo l'istanza di un problema $P1$ e l'istanza di un problema $P2$.

Affermiamo Un problema $P1$ si riduce a un problema $P2$ se ogni soluzione di $P1$ può ottenersi deterministicamente in tempo polinomiale da una soluzione di $P2$. Cioè



[2]Non *nonpolinomiale*, errore grave dire questo all'esame
[3]Insieme di problemi decisionali risolubili in tempo polinomiale con un algoritmo nondeterministico.

- Converto l'input di $P1$ (con complessità polinomiale) in un input di $P2$
- Risolvo il problema $P2$ ottenendo una soluzione
- Converto la soluzione di $P2$ (complessità polinomiale) in una soluzione di $P1$.

Di principio $P1 \leq P2$: la riducibilità può essere fatta solo se $P1$ ha difficoltà minore rispetto a $P2$.

Conseguenza importante Ho due problemi $P1 \leq P2$. Se $P2$ è riducibile in tempo polinomiale allora $P1$ è risolubile in tempo polinomiale.

14.2.7.1 Teorema di Cook

Qualsiasi problema R in NP è riducibile al problema della soddisfattibilità della formula logica:

$$\forall R \in NP : R \leq SAT$$

Quindi SAT è il più difficile di tutti i problemi in NP? Wait.

14.2.8 Definizione di NP-Completo

Un problema R è NP-completo se

- $R \in NP$
- $SAT \leq R$

Considerando il teorema di Cook affermo che i problemi NP-Completi hanno la stessa difficoltà del problema SAT, cioè $R = SAT$.

Dimostrazione di appartenenza a NP-Completo Possiamo dimostrare che R è un problema NP-completo:

- Individuo un algoritmo polinomiale nondeterministico per risolvere R, o dimostrare che la verifica di una soluzione di R può essere fatta in tempo polinomiale.
- Dimostro che esiste un problema NP-completo che si riduce a R. Scelgo uno dei problemi NP-Completi noti che sia facilmente riducibile a R

Conseguenza Posso utilizzare un altro problema NP-Completo, al posto di quello SAT, nella dimostrazione di NP-Completezza di un altro problema.

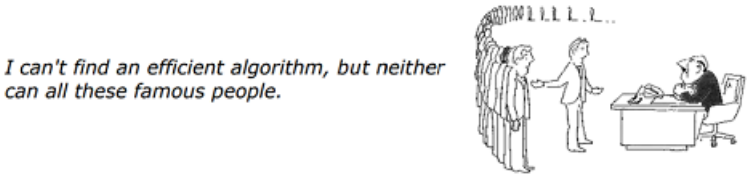
Conseguenza 2 Se SAT è il problema più difficile in assoluto segue che i problemi NP-Completi siano i più difficili della classe NP.

Conclusione Se si trovasse un algoritmo polinomiale per SAT, o per qualsiasi altro problema NP-Completo, allora tutti i problemi in NP sarebbero risolubili in tempo polinomiale e quindi $P = NP$.

Esempi di problemi NP-Completi I seguenti problemi sono NP-Completi

- Commesso viaggiatore
- Zaino
- Ciclo hamiltoniano

A cosa serve dimostrare che un problema è NP-Completo? Non riusciamo a risolvere un problema con algoritmo polinomiale. Vogliamo dimostrare che non ci si riesce a meno che $P = NP$ (problema tuttora irrisolto).



14.3 Problema della fattorizzazione di un numero (FATT)

Vogliamo scomporre un numero in fattori primi. Anche qua, come in tutti i problemi di teoria dei numeri, la complessità si calcola in funzione del numero di cifre da fattorizzare.

Moltiplicazione Prendiamo questi due numeri primi e moltiplichiamoli. Otteniamo il terzo numero, quello dopo l'uguale.

1.634.733.645.809.253.848.443.133.883.865.090.859.841.783.
92.312.181.110.842.389.333.100.104.508.151.212.118.167.511

1.900.871.281.664.822.113.126.851.573.935.413.975.471.896.
15.493.666.638.539.088.027.103.802.104.498.957.191.261.465
=

3.107.418.240.490.043.721.350.750.035.888.567.930.037.346.
27.545.720.161.948.823.206.440.518.081.504.556.346.829.671
782.437.916.272.838.033.415.471.073.108.501.919.548.529.00
.822.783.525.742.386.454.014.691.736.602.477.652.346.609

La moltiplicazione richiede meno di un secondo di calcolo!

Fattorizzazione Proviamo a fare il contrario. Se scomponiamo in fattori primi riotteniamo gli stessi due numeri utilizzati per la moltiplicazione.

Trovare A e B tali che
A * B = 3.107.418.240.490.043.721.350.750.035.888.567.930.037.
346.022.842.727.545.720.161.948.823.206.440.518.081.504.556.346.
829.671.723.286.782.437.916.272.838.033.415.471.073.108.501.919.
548.529.007.337.724.822.783.525.742.386.454.014.691.736.602.477.
652.346.609

A=1.634.733.645.809.253.848.443.133.883.865.090.859.841.783.670
.033.092.312.181.110.842.389.333.100.104.508.151.212.118.167.511
.579
B= 1.900.871.281.664.822.113.126.851.573.935.413.975.471.
896.789.968.515.493.666.638.539.088.027.103.802.104.498.957.191.
261.465.571

Il calcolo richiede più di una decina di anni!

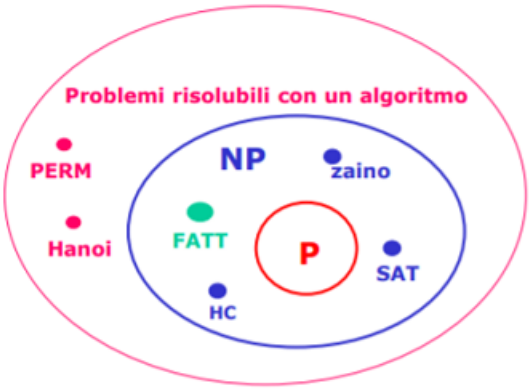
Osservazioni

- Per ora si conoscono soltanto algoritmi esponenziali per FATT
- FATT è in NP: la verifica è polinomiale (moltiplicazione). Non sappiamo se FATT sia in P, sicuramente non è NP-Completo
- Impossibile scomporre un numero di 200 o più cifre decimali con gli attuali algoritmi.

Sulla difficoltà della fattorizzazione si basano i meccanismi della *crittografia a chiave pubblica*: ottengo da un testo due numeri primi molto grandi, che moltiplico. Segue che le operazioni di crittografia sono facili, quelle inverse difficili! Trovando un algoritmo polinomiale per FATT non dimostrerei $P = NP$ ma metterei in forte crisi gli attuali meccanismi di crittografia.

14.4 Problemi risolubili con un algoritmo

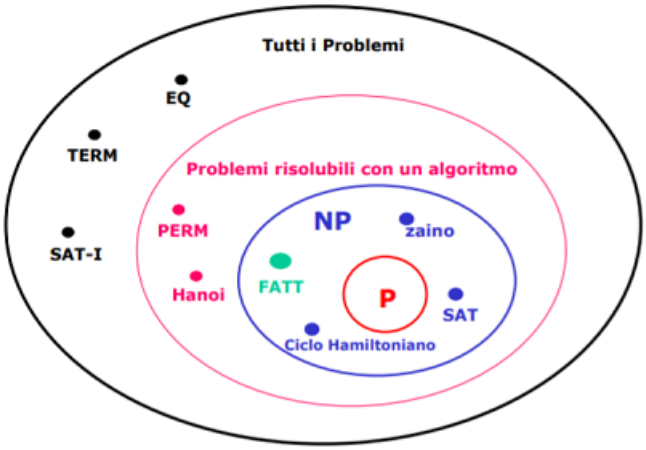
Possiamo espandere la visione degli insiemi vista prima includendo l'insieme NP nell'insieme di problemi risolubili con un algoritmo.



Problemi non considerabili NP sono:

- Trovare tutte le permutazioni di un insieme (PERM)
- Torre di Hanoi.

14.5 Problemi non risolubili con un algoritmo



Allarghiamo ulteriormente la visuale: esistono problemi non risolubili mediante algoritmo!

- **TERM**: dato un programma e un input vogliamo decidere quando termina (unico algoritmo: avviare il programma e vedere quando viene terminato).
- **SAT-I**: soddisfattibilità di una formula nella logica del I ordine (formule con il $\forall$, per esempio).
- **EQ**: decidere l'equivalenza di due programmi (peggiore addirittura di TERM, posso stare ad aspettare ma non potrò mai dire se due programmi sono effettivamente equivalenti).

Capitolo 15

Martedì 19/05/2020

Registro 13 (*Mar 19/05/2020 10:30-13:30 (3:0 h) lezione*)

Complementi di c++: funzioni e classi modello (template), derivazione.

Oggi introduciamo una nuova parte del corso, riguardante il C++. Questa analizzerà:

- funzioni e classi modello
- ereditarietà
- gestione delle eccezioni.

15.1 Vantaggi della programmazione a oggetti

Abbiamo già visto a *Fondamenti di programmazione* una serie di vantaggi della programmazione a oggetti:

- **Incapsulamento** dell'informazione. L'implementazione viene nascosta e può essere aggiornata nel tempo: l'intestazione rimane la stessa! Altro dettaglio è l'*information hiding*, cioè stabilire quali dati sono accessibili e quali no.
- **Decomposizione**. I programmi sono di dimensioni elevate: decompongo la progettazione di un programma di grandi dimensioni in più oggetti. Possiamo stabilire, in un azienda, che certi programmatori si occuperanno di un certo oggetto e non del resto.
- **Riuso**. La cosa è evidente soprattuto quando avremo parlato di funzioni/classi template ed ereditarietà. Vogliamo minimizzare il codice scritto.
- **Manutenzione**. Abbiamo la divisione tra interfaccia e implementazione. Non dobbiamo modificare programmi esterni che ricorrono al nostro tipo.
- **Affidabilità**.

Noi non aumentiamo la capacità di programmazione del nostro linguaggio (attraverso questi strumenti): semplicemente individuiamo metodi più veloci e semplici per poter ottenere il massimo! Togliendo queste funzionalità non vengono meno costrutti significativi.

15.2 Funzioni modello

Grazie alle funzioni possiamo creare modelli di funzioni applicabili a tipi diversi: ciò avviene *parametrizzando* i tipi utilizzati.

Indipendenza degli algoritmi dai dati Un esempio ideale potrebbe essere quello degli algoritmi di ordinamento. Supponiamo di avere il *selectionSort*: in certi casi voglio ordinare interi, in altri double, in altri ancora stringhe. Invece di scrivere l'algoritmo tre volte modificando soltanto il tipo degli elementi da ordinare potrei scrivere tutto una volta soltanto.

Esempio semplice Prendiamo una semplice funzione per individuare il massimo tra due numeri:

```
int max(int n1, int n2) {
    return (n1 > n2) ? n1 : n2;
}
```

Questa funzione può essere utilizzata esclusivamente per mettere a confronto variabili di tipo intero. Per utilizzarla con altre variabili potremo ricorrere al meccanismo dell'overloading, ponendo nello stesso file una funzione uguale ma con tipi diversi

```
double max(double n1, double n2) {
    return (n1 > n2) ? n1 : n2;
}
```

Ciò può essere sufficiente, ma in certe circostanze limitante. Scriviamo la nostra *funzione modello*:

```
template<class tipo>
tipo max(tipo n1, tipo n2) {
    return (n1 > n2) ? n1 : n2;
}
```

In questo modo non dovrò definire più funzioni con argomenti diversi. La funzione introdotta può essere chiamata nei seguenti modi

```
int i_num = max(1,2); // max<int>(1,2);
double d_num = max(1.0,2.2); // max<double>(1.0,2.2);
```

Se il parametro *tipo* è deducibile dalla chiamata allora non è necessario indicarlo tra parentesi angolari (come indicato nei commenti): ci penserà il compilatore. Ovviamente le chiamate non devono essere ambigue: se nella chiamata poniamo due argomenti diversi (per esempio un *int* e un *double*) il compilatore segnalerà un errore.

15.2.1 Puntatori

Proviamo a creare una funzione modello in cui sono coinvolti puntatori. Di principio un puntatore, tipo derivato, è associato a un tipo fondamentale. Possiamo fare questo:

```
template<class tipo>
void primo_elemento_array(tipo*  array) {
    tipo primo_elemento = array[0];
    cout<<primo_elemento<<endl;
}
```

La funzione potrà essere applicata ad array unidimensionali di qualunque tipo: *int, double, char...*

15.2.2 Parametri costanti

Tra i parametri possiamo avere anche valori costanti: *int, double, float, enum, char, bool....* Poniamo come esempio una funzione che stampa un elemento di un array unidimensionale di interi. Vogliamo indicare come parametro il valore default da assegnare, se necessario, all'argomento della funzione *num.*

```
template<int n>
void elemento_array(int* array, int num = n) {
        cout<<array[num]<<endl;
}
```

Si osserva un dettaglio importante: parametri costanti non possono essere impliciti (cioè dedotti dagli argomenti posti nella chiamata di funzione).

```
int i_array[] = {1,2,3,4,5};
double d_array[] = {1.1,2.2,3.3,4.4,5.5};

elemento_array<1>(i_array);
elemento_array<1>(i_array, 3);
```

Tra parentesi angolari posso porre *expressions*: identificatori, costanti, operatori...

15.2.3 Attenzione ai parametri *per forza* espliciti

Attenzione al tipo del valore restituito: se ho una situazione del genere

```
template<class tipo1, class tipo2, class tipo3>
tipo1 massimo(tipo2 x, tipo3 y) {
        return (x>y) ? x : y;
}
```

il parametro *tipo1* non può essere dedotto dagli argomenti posti nella chiamata di funzione. Dovrò quindi indicarlo tra parentesi angolari nella chiamata di funzione!

15.2.4 Conversioni di parametri

Indicando esplicitamente i parametri possiamo compiere, in alcune circostanze, delle conversioni.

```
template<class tipo>
tipo max(tipo n1, tipo n2) {
        return (n1 > n2) ? n1 : n2;
}
```

Vediamo le seguente chiamate di funzioni

```
cout<<max<int>(1,2.4);
cout<<max<double>(2,3.2);
```

Nel primo caso ho la conversione del secondo parametro *double* in int, nel secondo ho la conversione del primo parametro *int* in *double*. Se io pongo

```
int num;
num = max(1, 2.4);
```

otteniamo la conversione di 2.4 in un intero.

15.2.5 Più parametri

Una funzione modello può avere più parametri. Immaginiamoci una funzione che stampa tutti gli elementi appartenenti ad un array unidimensionale. Dovremo indicare il numero di elementi dell'array e il tipo degli elementi dell'array.

```
template<int n, class tipo> // (*)
void stampa_array(tipo*  array) {
    for(int i = 0; i < n ; i++) {
        cout<<array[i]<<endl;
    }
}
```

In questo caso la chiamata di funzione dovrà includere per forza, tra parentesi angolari, il parametro n, che non può essere dedotto in altri modi.

```
int i_array[] = {1,2,3,4,5};
double d_array[] = {1.1,2.2,3.3,4.4,5.5};
stampa_array<5>(i_array); // stampa_array<5, int>(i_array);
stampa_array<5>(d_array); // stampa_array<5, double>(d_array);
```

L'ordine dei parametri è importante quanto negli argomenti attuali delle funzioni. Se vogliamo evitare di indicare ogni volta il tipo tra parentesi angolari dobbiamo porre i parametri nell'ordine visto (∗).

15.2.6 Variabili statiche

Se all'interno del corpo di una funzione pongo una variabile statica non avrò la stessa variabile per tutte le istanze: l'oggetto statico con cui interagiamo dipenderà dal parametro.

```
template<class tipo>
tipo max(tipo x, tipo y) {
    static int a; a++; cout << a << endl;
    return (x>y) ? x : y;
}
```

Avrò, per esempio, una variabile static per le istanze con tipo *int* e un'altra per le istanze con tipo *double*.

15.2.7 Dichiarazione e definizione di template

Una funzione modello non può essere compilata senza conoscere le chiamate: segue che non si può fare una compilazione separata. Il file incluso nel main non deve avere soltanto le intestazioni ma anche l'implementazione.

```
// file templ.h

template<class tipo>
void boh(tipo x){
        // ... definizione
}
```

```
// file main

#include"templ.h"

void main() {
        //..
    boh(6);

      // ..
    }
```

15.3 Classi modello

Possiamo creare anche *classi modello*. I parametri non possono essere definiti in modo implicito.

Esempi Individuiamo un'applicazione pratica nelle classi *coda* e *pila*: mediante una classe modello posso stabilire il tipo degli elementi che formano la pila o la coda.

Versione senza modelli della pila

```
class stack{
        int size;
        int * p;
        int top;
public:
  stack(int n){
    size = n;
    p = new int [n];
    top = -1;
  };

  ~stack() { delete [] p; };

  int empty(){
        return (top==-1);  };

  int full(){
        return (top==size-1); };   }
```

```
int push(int s){
  if (top==size-1) return 0;
  p[++top] = s;
  return 1;
};

  int pop(int& s){
  if (top==-1) return 0;
  s =p[top--];
  return 1;
}
```

Versione modello della pila

```
//file stack.h

template<class tipo>
class stack {
        int size;
        tipo* p;
        int top;
public:
  stack(int n){
    size = n;
    p = new  tipo [n];
    top = -1; };

   ~stack() { delete [] p; };

   int empty() {
        return (top==-1);};

  int full() {
        return (top==size-1);};
```

```
int push(tipo s) {
  if (top==size-1) return 0;
  p[++top] = s;
  return 1; };

int pop(tipo& s){
   if (top==-1) return 0;
   s =p[top--];
   return 1;}
};
```

main.cpp Vediamo il main.

```
#include ''stack.h''

void main() {
        stack<int> s1 (20), s2 (30);
        stack<char> s3 (10);
        stack<float> s4 (20);

        s1.push(3);
        s3.push('');
```

```
        s4.push(4.5);
}
```

Dettagli Il vantaggio sta, senza ombra di dubbio, nella possibilità di ridurre il numero di righe di codice: se io sono certo che quel codice è funzionante allora posso ignorare il tipo e realizzare un qualcosa utilizzabile universalmente.

- quando l'implementazione della classe è divisa dalla sua dichiarazione è necessario porre nell'intestazione di ciascuna funzione membro il solito costrutto

  `template<parameters-list>`

- anche con le classi modello non è possibile tenere dichiarazioni e definizioni in pagine diverse: dobbiamo includere, nella pagina inclusa nel main, sia la dichiarazione che la definizione.

15.3.1 Puntatori

Nei puntatori dobbiamo tenere conto che oggetti con parametri diversi sono considerati tipi diversi. Prendiamo il seguente esempio:

```
stack<int, 100> pila1;
stack<int, 300> pila2;

stack<int,100>* ptr = &pila1;

//   ptr = &pila2;     errore
```

dove *stack<int,300>* e *stack<int,100>* sono tipi diversi.

15.3.2 Membri statici

Quanto detto per le funzioni modello vale anche per le classi modello. All'interno delle classi possiamo porre delle variabili static (come visto a *Fondamenti di programmazione*).

Cosa dobbiamo ricordare Abbiamo istanze diverse per parametri diversi. Se io ho una classe persona con parametro che mi indica il sesso allora posso conteggiare il numero di istanze del sesso maschile e il numero di istanze del sesso femminile (non si ha un conteggio unico poichè ho due tipi diversi: *persona<M>* e *persona<F>*).

Esempio della prof Vediamo un ulteriore esempio:

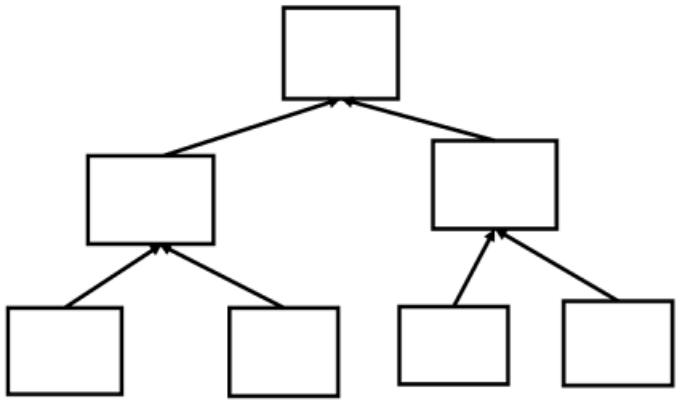
```
template<int n>
class cmod{
        static int istanze;
        int m;
public:
        cmod();
        void stampa();
};
template<int n>
int cmod<n>::istanze=0;

template<int n>
cmod <n>::cmod(){
        m=n;
        istanze ++;
        };
template<int n>
void cmod <n>::stampa(){
  cout << istanze  << '\t'  << m << endl;
}
```

```
void main(){
    cmod<9> nove_a;
    nove_a.stampa();

        // 1  9

    cmod<7> sette;
    sette.stampa();

        // 1  7

    cmod<9> nove_b;
    nove_b.stampa();

        // 2  9
}
```

15.4 Derivazione semplice (o Ereditarietà)

L'ereditarietà è forse il più importante dei costrutti che stiamo vedendo. Favorisce enormemente il riuso di codice già scritto con i vantaggi che già conosciamo (minor perdita di tempo e utilizzo di codice già sperimentato che sappiamo essere funzionante). La derivazione consente di trasmette un insieme di caratteristiche comuni da una classe *base* a una classe *derivata* senza che ciò comporti una duplicazione del codice, offrendo allo stesso tempo l'opportunità di *adattare* o *estendere* il comportamento a casi d'uso specifici.

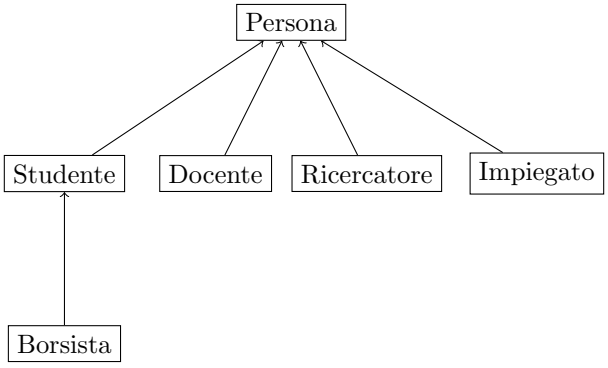
Gerarchia di classi Attraverso la derivazioni possiamo creare una vera e propria gerarchia.



- Andando dall'alto verso il basso si ha una *specializzazione* (ci si sposta da un livello di astrazione generico ad altri sempre più specifici).
- Andando dal basso verso l'alto si ha una *generalizzazione*.

La cosa può essere estremamente ramificata: posso avere classi derivate di primo livello, secondo livello, e così via...

Esempio Immaginiamo di avere due classi: *persona* e *studente*. La logica ci porta a immaginare l'ultima classe come "una sorta di espansione" della prima: tutti i campi dati presenti in *persona* potrebbero essere usati in *studente* per descriverlo. *persona* è la classe base, *studenti* quella derivata. Un oggetto di una classe derivata ha tutti i campi della classe base più quelli della classe derivata.
Creo altre classi derivate oltre a quella studente: *docente*, *ricercatore*, *impiegato*... Ne creo anche una derivata da un'altra classe derivata: *borsista* (colui che usufruisce di una borsa di studio) è uno studente (*studente* in questo caso sarà la classe base). Ogni oggetto *borsista* conterrà i campi dati di *persona* e di *studente*, oltre ai suoi.



Classe *persona* e *studente* Poniamo per comodità tutte le variabili membro pubbliche (dopo vedremo che non sarà così)

```
class persona {
public:
        char  nome [20];
        int eta;
};

// classe derivata studente, classe base persona

class studente : public persona{
public:
        int esami;
        int matricola;
};
```

Vediamo gli oggetti che caratterizzeranno la classe *studente*

Classe *studente* e *borsista* Andiamo oltre e creiamo anche una classe *borsista*

```
class borsista : public studente{
public:
        int borsa;
        int durata;
};
```

che avrà i seguenti oggetti

Classe *persona* e emphimpiegato Creiamo la classe *impiegato*: ovviamente l'impiegato non è uno *studente* (potrebbe esserlo ma non siamo in grado di concepirlo come un sottoinsieme degli studenti. Il borsista, per esempio, è per forza uno studente; l'impiegato non è detto).

```
class impiegato : public persona{
public:
        int livello;
        int stipendio;
};
```

Esempio di main Vediamo un main

```
void main() {
        persona p;
        studente s;
        impiegato i;
        borsista b;
}
```

15.4.1 Compatibilità fra tipi

15.4.1.1 Oggetti

Un oggetto di un tipo può essere convertito in un supertipo, ma non vale il contrario. Cioè possiamo compiere conversioni implicite da classi derivate a classi base, e non viceversa.

- **Esempi validi**: conversioni da *studente* a *persona*, da *docente* a *persona*, da *borsista* a *studente*, da *borsista* a *persona*
  - $p = s$, corretto (conversione implicita da studente a persona)
  - $s = p$, sbagliato (supertipo assegnato a sottotipo)
  - $s = i$, sbagliato (tipi diversi non collegati logicamente fra loro)
  - $p = b$, corretto (conversione implicita da borsista a persona)
  - $s = b$, corretto (conversione implicita da borsista a studente)
- **Esempi non validi**: conversioni da *persona* a *docente*, da *persona* a *studente*, da *studente* a *borsista*.

Esempio con $p = s$ La conversione comporta perdita di informazioni!

15.4.1.2 Puntatori

Un puntatore a oggetto di un tipo può essere convertito in un puntatore a supertipo, ma non vale il contrario. Vale quanto detto prima con una differenza fondamentale:

- se nell'operazione di assegnamento andiamo ad uguagliare un oggetto di tipo *persona* (*lvalue*) a un oggetto di tipo *studente* (*rvalue*) i campi dati della classe derivata scompaiono
- se nell'operazione di assegnamento andiamo ad assegnare a un puntatore a *persona* l'indirizzo di un oggetto di tipo *studente* i campi dati non saranno eliminati ma resi inaccessibili.

Aggiorniamo il main Includiamo nel main alcuni puntatori

```
void main() {
        persona p;
        studente s;
        borsista b;
        impiegato i;
        studente* ps;
        persona* pp;
}
```

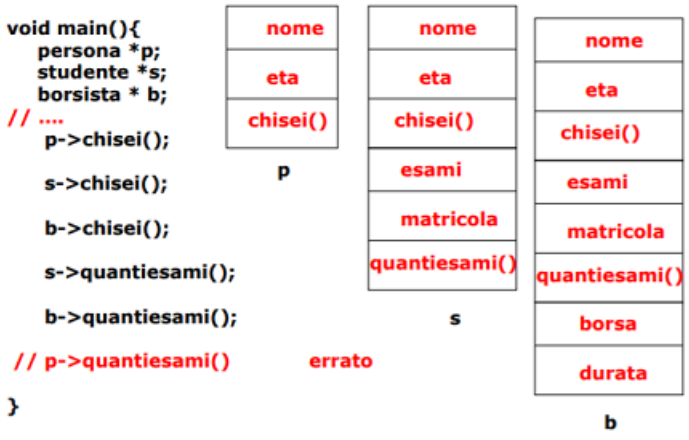
Esempi di operazioni di assegnamento con puntatori

- $pp = \&p$, corretto (assegno a un puntatore a persona l'indirizzo di un oggetto persona).
- $pp = \&s$, corretto (conversione implicita)
- $pp = \&b$, corretto (conversione implicita)
- $pp = new\ studente$, corretto (conversione implicita)
- $ps = \&p$, sbagliato (assegno a un puntatore a studnete l'indirizzo di un oggetto persona)

15.4.2 Funzioni membro

Quanto visto sui campi dati è valido anche con le funzioni membro: una funzione membro di *persona* potrà essere utilizzata nelle istanze delle classi derivate *studente*,*borsista*,*docente*. Funzioni membro delle classi derivate non potranno essere utilizzate nelle istanze della classe base *persona*

Esempio con *persona*, *studente* e *borsista* Rivediamo, con l'aggiunta di funzioni membro cosa può essere usato e non usato dalle varie classi.



15.4.3 Regole di visibilità

A questo punto le cose si complicano. Dobbiamo formulare delle regole di visibilità per gestire eventuali situazioni ambigue. Come vedremo valgono le stesse regole dei blocchi.

- **Oggetti con stesso nome** Se un campo dato della classe base ha lo stesso nome di un campo dato della classe derivata potrò recuperare il primo ricorrendo all'operatore di *risoluzione di visibilità* (valgono le stesse regole dei blocchi: una dichiarazione più interna sovrascrive la dichiarazione più esterna finchè si rimane all'interno di quel blocco).

```
void main(){
   studente * s=new studente;
   borsista * b=new borsista;

   b->esami=4;                 // = b.borsista::esami

   b->studente::esami=5;   // risolutore di visibilità

   cout << b->esami;          // 4

   s=b;                        // conversione

   cout << s->esami;          // 5
}
```

matricola	
esami	5
borsa	
durata	
esami	4

- **Oggetti con stesso nome distribuiti su più di due classi** Utilizzo sempre l'*operatore di risoluzione di visibilità* indicando prima del simbolo dell'operatore il nome della classe base con il dato che mi interessa. Esempio: ho la classe derivata *borsista*, la cui classe base è *studente*. *studente* è a sua volte classe derivata da *persona*. Tutte e tre hanno in comune un campo con nome *esami* (in persona sono gli esami clinici, in studenti gli esami svolti dall'immatricolazione, in borsista gli esami svolti durante la borsa di studio).
  - Se non utilizzo operatori di risoluzione prevale la dichiarazione più recente e interna: con *borsista.esami* ottengo gli esami svolti durante la borsa di studio
  - Utilizzando l'operatore di risoluzione richiamo gli esami di *studente* con l'identificatore *borsista.studente::esami* e gli esami di *persona* con *borsista.persona::esami*.

```
void main(){
   classe3 obj;
   obj.a=2;              // obj.classe3::a
   obj.classe1::a=7;
   obj.classe2::a=8;

   cout << obj.a;              // 2

   cout << obj.classe1::a;     // 7

   cout << obj.classe2::a;     // 8
}
```

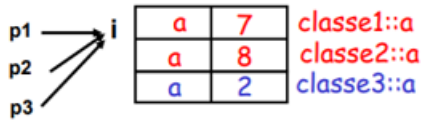
a	7	classe1::a
a	8	classe2::a
a	2	classe3::a

obj

- **Puntatori** Riprendiamo l'esempio di prima, ma utilizziamo dei puntatori: uno (*p1*) a un oggetto di tipo *persona*, uno (*p2*) a un oggetto di tipo *studente* e uno (*p3*) a un oggetto di tipo *borsista*. Inizializzo i puntatori ponendo come indirizzo quello di un oggetto di tipo *borsista*.
  Se utilizzo l'operatore di selezione membro con dereferenziazione otterrò il campo del tipo astratto associato al puntatore.
  - con *p1*− >*esami* otterrò gli esami clinici
  - con *p2*− >*esami* otterrò gli esami universitari svolti dall'immatricolazione
  - con *p3*− >*esami* otterrò gli esami universitari svolti durante la borsa di studio

```
classe1* p1=&obj;          // conversione
classe2* p2=&obj;          // conversione
classe3* p3=&obj;
cout << p1->a;        //   7
cout << p2->a;        //  8
cout << p3->a;        //   2
}
```

p1->a : i
p2->a : i+1
p3->a : i+2



- **Funzioni membro** Non esiste l'overloading per funzioni appartenenti a classi diverse: segue che dovremo utilizzare gli operatori di risoluzione quando necessario. Il meccanismo non è molto diverso da quello già visto per i campi dati.

```
#include<iostream.h>

class uno {
// ..
public:
        uno() { }
        void f(int) {
                cout << "uno";
         }
};

class due: public uno {
//..
public:
        due() {}
        void f() {
                cout << "due";
        }
};
```

```
void main (){
        due* p= new due;
//      p->f(6);    errore
        p->uno::f(6);  // uno
        p->f();        // due
}
```

no overloading per funzioni appartenenti a classi diverse

Morale della favola sui puntatori

pp (tipo *persona) e ps (tipo * studente) hanno lo stesso valore, ma possono accedere soltanto ai campi relativi al loro tipo:

Per pp: pp->nome, pp->eta
Per ps: ps->nome, ps->eta, ps->esami, ps->matricula
pp->esami ERRORE

La scelta del campo a cui si accede avviene a tempo di compilazione in base al tipo del puntatore

Capitolo 16

Venerdì 22/05/2020

Registro 14 (*Ven 22/05/2020 11:30-13:30 (2:0 h) lezione*)

complementi di c++: ereditarietà delle classi, funzioni virtuali e classi astratte.

16.1 Continuiamo sull'ereditarietà

16.1.1 Specificatori di accesso (*protected*)

Fino ad ora abbiamo considerato tutti i membri di una classe come pubblici. Un approccio del genere non può essere utilizzato a livello professionale: porre membri pubblici significa che questi possono essere modificati al di fuori della classe (in aperto contrasto col principio dell'*information hiding*). Segue un ulteriore problema.

Problema I campi dati di una classe base posti nella parte privata non sono accessibili alla classe derivata.

Soluzione Introduciamo per rimediare a questo limite la parte *protected*: i campi dati (e le funzioni) posti al suo interno possono essere usati dalle classi derivate ma non all'esterno (poichè non pubblici). Otteniamo una soluzione intermedia tra il *public* e il *private*.

```
class uno {
protected:
   int x;
};

class due : public uno{
   int y;
   void f() {x=5; y=6; }  // ok perchè x è protetto
};

due * s= new due;

s->x=2 ;  // no perchè x è protetto ma non pubblico
```

Derivazione pubblica/protetta/privata Invece di *public uno* si potrebbe scrivere anche *protected uno* o *private uno*. Si ha un tipo di derivazione diversa. Noi abbiamo visto solo la derivazione public e ci interessa solo questa.

16.1.2 Costruzione degli oggetti

Nella costruzione di un oggetto si costruisce prima la parte base e successivamente la parte derivata. Prima viene chiamato il costruttore della classe base, successivamente il costruttore della parte derivata. Una classe base può avere più costruttori:

- mediante *liste di inizializzazione* nel costruttore della classe derivata possiamo chiamare esplicitamente un costruttore in particolare (ponendo gli argomenti attuali)
- se non utilizziamo liste di inizializzazione viene chiamato il costruttore default (privo di argomenti o con argomenti tutti opzionali).
- il compilatore segnala errore se non si effettua una chiamata esplicita del costruttore della classe base e la stessa non ha costruttori default

Situazione semplice In questo caso abbiamo semplici chiamate con costruttori default.

```
class uno {
public:
   uno(){cout << "nuovo uno" << endl;}
};

class due: public uno {
public:
   due()  {cout << "nuovo due"<< endl;}
};

class tre: public due {
public:
   tre() {cout << "nuovo tre"<< endl;}
};

void main (){
   due obj2;    // nuovo uno
                // nuovo due
   tre obj3;    // nuovo uno
                // nuovo due
                // nuovo tre
}
```

Situazione più elaborata In questo caso si chiama prima il costruttore default (non abbiamo indicato di fare diversamente) e poi il costruttore di due (l'unico presente)

```
class uno {
protected:
   int a;
Public:
   uno() {a=5; cout << "nuovo uno" << a << endl;}
   uno(int x) {a=x; cout << "nuovo uno" << a << endl;}
};

class due: public uno {
   int b;
public:
   due(int x) {b=x; cout << "nuovo due" << x << endl;}
};

void main (){

   due obj2(8);       // nuovo uno  5
                      // nuovo due  8
}
```

Chiamata di un costruttore diverso da quello default In questo caso non chiamiamo il costruttore default di uno ma l'altro (che ha come argomento *int x*). Indico ciò nella lista di inizializzazione.

```
class uno {
protected:
   int a;
Public:
   uno() {a=5; cout << "nuovo uno" << a << endl;}
   uno(int x) {a=x; cout << "nuovo uno" << a << endl;}
};

class due: public uno {
   int b;
public:
   due(int x): uno(x+1) {b=x; cout << "nuovo due" << x << endl;}
};

void main (){

   due obj2(8);      // nuovo uno 9
                     // nuovo due  8
}
```

16.1.3 Ordine di chiamata dei costruttori per una gerarchia a due livelli

1. costruttori degli oggetti membri della classe base
2. costruttore della classe base
3. costruttori degli oggetti membri della classe derivata
4. costruttore della classe derivata

16.1.4 Distruzione degli oggetti

La distruzione avviene in senso inverso:

- prima distruggo la classe derivata
- poi distruggo la classe base

```
class uno {
public:
   uno();
   ~uno();
};

uno::uno(){cout << "nuovo uno" << endl;}
uno::~uno(){cout << "via uno" << endl;}

class due: public uno {
public:
   due();
   ~due();
};

due::due(){cout << "nuovo due" << endl;}
due::~due(){cout << "via due" << endl;}
```

```
void main (){
      due obj2;
      // nuovo uno
      // nuovo due

      // via due
      // via uno
}
```

16.1.5 Membri statici

Riprendiamo l'esempio di *persona*, *studente* e *borsista*: immaginiamo di avere in ciascuno una variabile statica per contare il numero di istanze (numero di persone, numero di studenti, numero di borsisti). I valori dei membri statici, aggiornati mediante i costruttori, si comportano nel seguente modo:

- se creo un'istanza di tipo *persona* incremento il membro statico di *persona*

- se creo un'istanza di tipo *studente* incremento il membro statico di *persona* e quello di *studente*
- se creo un'istanza di tipo *borsista* incremento il membro statico di *persona*, quello di *studente* e quello di *borsista*.

```
class A {
public:
    static int quantiA;
    A(){
     cout << "A =  "
    << ++quantiA << endl;}
};

int A::quantiA=0;

class B : public A{
public:
    static int quantiB;
    B(){
    cout << "B =  "
    << ++quantiB << endl;}
};

int B::quantiB=0;
```

```
void main(){
        A p1;
                // A = 1
        B s1;
                // A = 2
                // B = 1
        A p2;
                // A = 3
        B s2;
                // A = 4
                // B = 2
}
```

16.2 Funzioni virtuali

Ricordiamo In una gerarchia di classi il metodo (la funzione) da chiamare viene scelto dinamicamente a tempo di esecuzione.

Esempio Utilizziamo un puntatore ad oggetto di tipo *persona* con indirizzo di un oggetto di tipo *studente*: possiamo osservare come le funzioni eseguita mediante chiamata siano quelle del tipo astratto associato al puntatore.
Presumiamo di avere una funzione membro *carta_identita* in entrambe le classi: se svolgo la chiamata di funzione mediante il puntatore sarà eseguita la funzione membro della classe *persona* (nonostante il puntatore abbia come valore l'indirizzo di un oggetto di tipo *studente*).

Soluzione Possiamo superare questo problema mediante le cosiddette *funzioni virtuali*: esse mi permettono di ridefinire in una classe derivata una funzione ereditata da una classe base!

Differenze tra funzioni virtual e non virtual

- senza l'attributo *virtual* la funzione viene scelta a tempo di compilazione in base al tipo del puntatore
- con l'attributo *virtual* la funzione viene scelta a tempo di compilazione in base all'oggetto effettivamente puntato

Differenza tra oggetti e puntatori Ciò risulta valido solo utilizzando puntatori: se la chiamata avviene dall'oggetto l'attributo virtual non ha effetto.

Gerarchia di funzioni virtuali Una funzione è virtuale nella classe in cui è definita con l'apposito attributo e in tutte le sue sottoclassi.

```
class uno {
   //..
public:
   uno() {}
   void f() {
      cout << 1 << endl; }
};

class due : public uno{
public:
   due () {}
   void virtual f() {
      cout << 2 << endl; }
};

class tre: public due {
public:
   tre () {}
   void f()  {
      cout << 3 << endl; }
};
```

```
void main(){
        due* p2= new tre;
        p2->f(); // 3   tre::f()
        uno* p1= new tre;
        p1->f();  // 1  uno::f()
}
```

f è virtuale in due e tre ma non in uno

Una funzione e' virtuale in tutte le classi che si trovano sotto quella che la definisce come virtuale

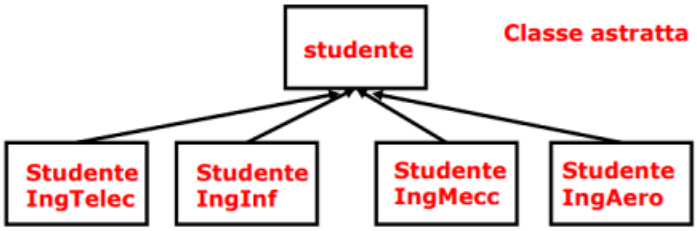
La classe *uno* non ha funzione f virtuale e pertanto non è possibile sovrascriverla mediante nuove definizioni nelle classi derivate (quindi se ho un puntatore a oggetti *uno* con indirizzo di un oggetto *due* eseguirò la funzione f membro di *uno*).

Distruttori virtuali Anche i distruttori possono essere definiti con attributo *virtual*. Prendiamo come esempio le solite classi *persona* e *studente*: ho il solito puntatore ad oggetti *persona* con indirizzo ad oggetto di tipo *studente*. L'attributo virtual mi permette di eseguire sia il distruttore degli oggetti *persona* che quello degli oggetti *studente* al termine di vita dell'istanza. Senza quell'attributo eseguirei soltanto il distruttore degli oggetti *persona*.

```
class uno {
public:
   uno() {};
   virtual ~uno() {cout << "via uno" << endl;}
};
class due: public uno {
public:
   due(){};
   ~due() {cout << "via due" << endl
};

void main (){
   uno* obj=new due;
   //...
   delete obj;}

 // via due              ~due()
 // via uno

senza virtual :

// via uno              ~uno()
```

16.3 Classi astratte e Polimorfismo

Situazione



Siamo in un ambiente universitario caratterizzato da studenti. Il mio interesse non è definire lo studente in senso generico ma lo studente in quanto parte di una certa facoltà: per ogni facoltà lo studente presenta delle peculiarità.

Classi astratte Se non è nostro interesse definire uno studente in senso generico allora possiamo parlare di classi astratte.

- Serve come classe base nelle derivazioni.
- Viene specializzata nelle classi derivate.
- Definisce una interfaccia unica verso le applicazioni.
- Non viene definite completamente: ha almeno una funzione virtuale pura.
- Una funzione virtuale pura è una funzione (ereditata o no) senza definizione: $F(\dots) = 0$.
- Non si possono istanziare oggetti di una classe astratta

Funzione virtuale pura Una funzione virtuale pura è dichiarata ma non implementata. Solitamente viene introdotta con la forma *type virtual function_name() = 0*.

Esempio di classe astratta con classi derivate

```
class studente {
        int matricola; int esami;
public:
        studente (int m){ esami=0; matricola=m; }
        // ...
        void virtual chisei() =0;

                        // funzione virtuale pura
};
```

```
class studenteIngInf : public studente {
        //...
public:
        studenteIngInf(int m) : studente(m) {}
        // ...
        void   chisei() {
        cout << "studente di ingegneria informatica" <<endl;
        }
};

class studenteIngMecc : public studente{
        // ..
public:
        studenteIngMecc(int m) : studente(m) {}
        // ...
        void   chisei() {
        cout << "studente di ingegneria meccanica" << endl;
        }
};
```

Capitolo 17

Martedì 26/05/2020

Registro 15 (*Mar 26/05/2020 10:30-13:30 (3:0 h) lezione*)

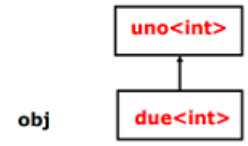
Complementi di c++: gestione delle eccezioni. Esercizi.

17.1 Derivazione e classi modello insieme

I due argomenti introdotti nelle scorse lezioni possono essere utilizzati insieme: possiamo creare una classe modello e adottare l'approccio dell'ereditarietà Vediamo tre esempi

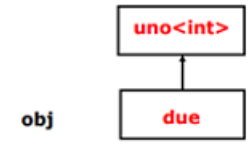
```
template <class T>
class uno {
    T a;
public:
    uno(T x) {
        a=x;
    cout << a << endl;
    }
};

template <class tipo>
class due: public uno<tipo> {
    tipo b;
public:
    due(tipo x, tipo y):
        uno<tipo>(x) {
    b=y; cout << b << endl;
    }
};
```

```
void main (){
    due<int>  obj(7,8);
}


7   uno<int>::uno<int>(7)
8   due<int>::due<int>(7,8)
```



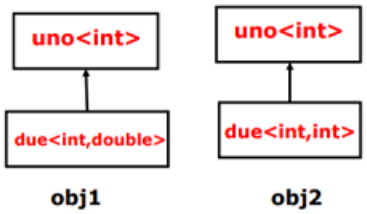

```
template <class T>
class uno {
    T a;
public:
    uno(T x) {
        a=x;
    cout << a << endl;
    }
};

class due: public uno<int> {
    int b;
public:
    due(int x, int y):
        uno<int> (x) {
    b=y; cout << b << endl;
    }

};
```

```
void main (){
    due  obj(7,8);
}

7   uno<int>::uno<int>(7)
8   due::due(7,8)
```



```
template <class T>
class uno {
    T a;
public:
    uno(T x) {
        a=x;
    cout << a << endl;
    }
};

template <class tipo1, class tipo2>
class due: public uno<tipo1> {
    tipo2 b;
public:
    due(tipo1 x, tipo2 y):
        uno<tipo1>(x) {
    b=y; cout << b << endl;
    }
};
```

```
void main (){
  due<int,double> obj1(7,8.5);
  due<int,int>  obj2(7,8.5);
}
 7      uno<int>::uno(7)
8.5 due<int,double>::due<int,double>(7,8.5)
 7      uno<int>::uno(7)
 8      due<int,int>::due<int,int>(7,8.5)
```



Nei primi due esempi la classe *due* non è assolutamente una classe modello, ma un qualcosa di definito a partire da una classe ben precisa (*uno<int>*). Nel terzo esempio la classe *due* è anche classe modello: come ogni classe modello è necessario indicare i due parametri in modo esplicito.

17.2 Eccezioni

Con eccezioni intendiamo tutte quelle situazioni che possono provocare il *refresh* dell'applicazione:

- Errori a runtime (divisione per 0, indice array fuori dall'intervallo)
- Situazioni anomale non rilevabili dal compilatore
- Situazioni che possono causare il crash dell'applicazione

Invece di farci dire dal sistema le motivazioni di arresto possiamo intervenire noi in anticipo gestendo queste situazioni che provocano arresto. Alcuni errori, inoltre, non è detto siano individuati dal compilatore (per esempio la divisione dello 0 o l'indice fuori dall'array).

Vantaggi

- Possibilità di individuare le eccezioni e gestirle da programma a tempo di esecuzione
- Metodo formale e ben definito
- Netta separazione tra il codice che rileva l'eccezione e il codice che lo gestisce

Costrutto Il costrutto sintattico alla base della gestione delle eccezioni si articola fondamentalmente in due parti:

- il blocco *try*, introdotto mediante l'omonima keyword. Al suo interno sono presenti le istruzioni normalmente eseguite e quelle *throw* che lanciano le eccezioni.
- i gestori *catch*, che gestiscono le varie eccezioni contenendo le istruzioni da eseguire.

```
try  {
..
throw espressione_1;          // lancio eccezione
.....
throw espressione_m;          // lancio eccezione
.....
}

   catch (tipo_1 e) { ..... }    // gestione eccezione
   ..
   catch (tipo_n e) { ..... }    // gestione eccezione

comandi_successivi ...
```

Le istruzioni *throw* sono sempre accompagnate da una *expression* che ci permette di identificare quale gestione è stata scelta. I *catch* presentano sempre un argomento.

- Se non vengono lanciate eccezioni l'esecuzione del try non viene interrotta e si passa immediatamente alle istruzioni successive
- Se l'eccezione lanciata non viene catturata ("catch") il programma termina con errore
- Se l'eccezione lanciata viene catturata l'esecuzione del blocco try viene interrotta e si eseguono le istruzioni contenute all'interno del blocco del gestore. Dopo ciò si passa alle istruzioni successive ignorando quelle rimanenti all'interno del blocco *try*.

Etichette delle eccezioni Le eccezioni possono essere associate ai gestori:

- ponendo la definizione di un oggetto, il cui valore potrà essere utilizzato all'interno delle istruzioni del blocco del gestore
- ponendo l'identificatore di un tipo
- ponendo i punti di sospensione . . . , in questo caso gestirò ogni tipo di eccezione

Dettagli

- non si hanno conversioni implicite (a parte la conversione da sottotipo a sopratipo vista in queste lezioni)

- l'eccezione viene gestita a tempo di esecuzione esaminando i gestori nell'ordine in cui compaiono a partire dal blocco più recente incontrato: viene scelto il primo con argomento corrispondente all'eccezione lanciata.

Posso coinvolgere negli argomenti dei gestori anche tipi riferimento e puntatore! Si osserva inoltre la possibilità di gestire gli errori creando classi dedite a tale scopo.

17.2.1 Esempi

Primo esempio Lancio di eccezione a partire da un intero

```
int main() {
    try {
        int a,b;

        cout<<"Divisione tra due numeri:"<<endl;
        cin>>a>>b;

        if(a == 0 && b == 0)
            throw 1;
        else if(b == 0)
            throw 2;

        double result = a/(double)b;
        cout<<"Risultato: "<<result<<endl;
    }
    catch(int n) {
        if(n == 1)
            cout<<"Operandi nulli, operazione indeterminata";
        if(n == 2)
            cout<<"Denominatore nullo, operazione impossibile";
    }
}
```

Secondo esempio Lancio di un eccezione con stringa

```
int main() {
    try {
        int a,b;

        cout<<"Divisione tra due numeri:"<<endl;
        cin>>a>>b;

        if(a == 0 && b == 0)
            throw "Operandi nulli, operazione indeterminata";
        else if(b == 0)
            throw "Denominatore nullo, operazione impossibile";

        double result = a/(double)b;

        cout<<"Risultato: "<<result<<endl;
    }
    catch(const char* frase) {
        cout<<frase<<endl;
    }
}
```

Terzo esempio Lancio di eccezione misto (sia con tipo intero che con punti di sospensione), con catch che cattura qualsiasi eccezione

```
int main() {
    try {
        int a,b;

        cout<<"Divisione tra due numeri:"<<endl;
        cin>>a>>b;

        if(a == 0 && b == 0)
            throw 0;
        else if(b == 0)
            throw 'c';

        double result = a/(double)b;
        cout<<"Risultato: "<<result<<endl;
    }
    catch(int) {
        cout<<"Operandi nulli, operazione indeterminata";
    }
    catch(...) {
        cout<<"Denominatore nullo, operazione impossibile";
    }
}
```

Quarto esempio *try* e *catch* all'interno di una funzione.

```
void div (int x, int y) {
        try {
             if (y==0) throw 0;
             cout << x/y << endl;
        }
        // nota
        catch (int) {
             cout << "divisione per 0" << endl;
        }

        cout << "fine div" << endl ;
}

void main(){
        int x,y;
        cin >> x >> y;
        div(x,y);
        cout << endl << "fine main";
}
```

Ponendo 10 e 0 otteniamo il seguente output

```
divisione per 0
fine div
fine main
```

Quinto esempio *try* e *catch* in funzioni diverse

```
void div (int x, int y) {
        // lancio eccezione
```

```
        if (y==0) throw "divisione per 0";
        cout << x/y << endl;
        cout << "fine div" << endl ;
}

void main() {
        try { // gestione eccezione nel main
             int x,y;
             cin >> x;
             cin >> y;
             div(x,y);
        }
        catch (char* p) {
             cout << p << endl;
        }

        cout << endl << "fine main";
}
```

Ponendo 10 e 0 otteniamo il seguente output

```
divisione per 0
fine main
```

Sesto esempio Ulteriore esempio di clausola catch generica

```
void main(){
        try {
             int x; cin >> x;
             if (x==0) throw x;
             if (x <0) throw 7.8;
        }
        catch(int) {
             cout << "eccezione da main" << endl;
        }
        catch(...) {
             cout << "eccezione non prevista da main" << endl;
        }
        cout << "fine main";
}
```

Con input -1 otteniamo

```
eccezione non prevista da main
fine main
```

Settimo esempio Rilancio delle eccerzioni, abbiamo una situazione estremamente gerarchica.

```
void f(int x) {
        if (x==0) throw x;
        cout << "fine f" << endl;
}

void g(int& x) {
        try {
             f(x);
        }
        catch(...) {
             throw;
```

```
        }
}

void main(){
        try {
             int x; cin >> x; g(x);
        }
        catch(int) {
             cout << "eccezione da main" << endl;
        }
        cout << "fine main";
}
```

Con input 0 otteniamo

```
eccezione da main
fine main
```

Ottavo esempio con classe Utilizziamo una classe per gestire in modo più formale un errore.

```
class Errore {
    const int ID;
    public:
        Errore(const int n) : ID(n) {

        }

        void print() {
            if(ID == 0)
                cout<<"Operandi nulli, operazione indeterminata";
            else if(ID == 1)
                cout<<"Denominatore nullo, operaz. impossibile";
        }
};
```

A pagina dopo due esempi di main: uno con oggetti e uno con puntatori

```
int main() {
        try {
                int a,b;

                cout<<"Divisione tra due numeri:"<<endl;
                cin>>a>>b;

                if(a == 0 && b == 0)
                        throw Errore(0);
                else if(b == 0)
                        throw Errore(1);

                double result = a/(double)b;
                cout<<"Risultato: "<<result<<endl;
        }
        catch(Errore e) {
                e.print();
        }
}

int main() {
        try {
```

```
                int a,b;

                cout<<"Divisione tra due numeri:"<<endl;
                cin>>a>>b;

                if(a == 0 && b == 0) {
                        Errore e = 0;
                        throw &e;
                }
                else if(b == 0) {
                        Errore e = 1;
                        throw &e;
                }

                double result = a/(double)b;
                cout<<"Risultato: "<<result<<endl;
        }
        catch(Errore* e) {
                e->print();
        }
}
```

Consiglio Rivedersi le diapositive della professoressa.

Parte II

Lezioni di Alfeo

Capitolo 18

Giovedì 19/03/2020

Registro 16 (*Gio 19/03/2020 13:30-15:30 (2:0 h) laboratorio*)

Presentazione prova pratica d'esame, strutture dati basilari e modalita' di debugging. Nello specifico: Debug Manuale Gestione Dinamica Input e Liste Debug Assistito Soluzioni della Standard Template Library Gestione Stringhe e Vectors (lezione tenuta in modalità telematica - secondo sottogruppo di studenti)

Prova pratica La prova pratica ha una valutazione "booleana": passato, non passato.

Prerequisiti Fondamenti di programmazione, utilizzo del compilatore e comandi base unix (in particolare quelli che permettono di svolgere la redirezione)

Avviso Imparare i codici per svolgere la redirezione: se chiesti al compito non saranno dati. Questi codici dovrebbero essere già conosciuti grazie al corso di Fondamenti di programmazione!

Argomenti di oggi Nella lezione di oggi analizzeremo il concetto di debug, vedremo alcune buone tecniche di programmazione e introdurremo alcune classi presenti nella *stl*: la *Standard Template library*. Si consiglia, almeno inizialmente, di continuare a scrivere i codici *from scratch*.

18.1 Debug

Il *debug* consiste in un processo che permette di individuare i bug del nostro programma. Partiamo da un principio: la macchina ha sempre ragione: essa si limita ad eseguire il codice da noi scritto. La necessità di debug può essere ridotta con un codice ordinato e con la presenza di commenti che spiegano i procedimenti adottati (utili soprattutto per chi modifica codici non propri): resta un qualcosa che non si può evitare, utile soprattutto per prepararsi al compito (non per svolgere il compito).

> If debugging is the process of removing software bugs, then programming must be the process of putting them in. *(E. Dijkstra)*

Conclusione Cadere nell'errore è inevitabile!

Tecniche di debug Vediamo le tecniche.

- **Debug visuale**, in cui altero il codice del programma per esplicitare le operazioni svolte dallo stesso.
  Esempio: ho un selectionSort che non funziona, mediante cout stampo gli indici degli elementi scambiati. In questo modo vedo in che punto avviene l'errore.

- Debug attraverso il **compilatore**: se il programma non compila vengono stampati gli errori
- Debug mediante ricorso a programmi, i cosiddetti **debugger**: essi permettono di gestire l'esecuzione del programma bloccandola quando necessario, individuare errori come il segmentation fault.
- Analisi della memoria con **Valgrind**, che stampa le allocazioni di memoria effettuate con indirizzi ed eventuali errori

Conoscenza delle strutture dati Il debug può essere evitato o svolto più velocemente con una conoscenza adeguata delle strutture dati. Dobbiamo saper scegliere le strutture dati adeguate per salvare i nostri dati e tutte le loro caratteristiche peculiarie. Prendiamo per esempio la *lista*: le operazioni fondamentali sono l'inserimento in testa, l'inserimento in coda, lo scorrimento, l'estrazione. Nella lista dobbiamo impostare in modo adeguato i valori dei puntatori, in particolare porre nulli i puntatori che non puntano a nessun indirizzo (il puntatore a lista vuota, il puntatore presente nell'ultimo elemento...). La conoscenza delle strutture dati permette la creazione di algoritmi con minore complessità: per esempio potrei utilizzare, oltre agli elementi base della lista, un puntatore a elemento finale o creare doppie liste.

18.2 STL - *Standard Template Library*

La *Standard Template Library* contiene classi immediatamente utilizzabili in grado di semplificarci l'esistenza. Come già scritto si ribadisce che conviene, almeno per ora, continuare a scrivere codici *from scratch*.

18.2.1 vector

La classe *vector* unisce la semplicità degli array con la dinamicità delle liste! Si introduce mediante il seguente costrutto (tipico delle classi modello, di cui parleremo più avanti nel corso):

```
vector<int> stlArray
```

Vediamo alcune funzioni a nostra disposizione:

- push_back(), che permette l'aggiunta di elementi in fondo all'array
- begin(), che restituisce l'indice del primo elemento dell'array
- end(), che restituisce l'indice dell'ultimo elemento dell'array
- size(), che restituisce la dimensione dell'array

L'allocazione è ovviamente dinamica. Risulta necessario essere cauti: è ovvio che non posso mettermi ad inserire un numero infinito di elementi! A un certo punto incontrerei una parte di memoria già occupata, creando conflitti.

18.2.2 string

Fino ad ora non abbiamo mai considerato le stringhe come un tipo di dato astratto: adesso possiamo farlo, grazie a una classe che permette di superare tutte le limitazioni tipiche dell'array (in particolare la non applicazione di certi operatori all'array nel suo complesso).

- Possiamo modificare il valore della stringa, dopo averla definita, mediante operatore di assegnamento (cosa normalmente non possibile)
- Possiamo concatenare più stringhe mediante l'operatore aritmetico di addizione.

Vediamo alcune funzioni:

- find(value), veifico se è presente una parola all'interno della stringa. Se è presente restituisco l'indice associato alla prima lettera della parola nella stringa. Se non è presente restituisco l'enumeratore string::npos.
- compare(parola), stesse funzionalità della strcmp vista a Fondamenti di programmazione.

18.3 Redirezione

Osservazione importante Prendere con le pinze quanto contenuto in questa sezione. Normalmente l'esame viene eseguito su macchine con Linux. I comandi qui presenti funzionano senza problemi su Windows, ma potrebbero non andare per Linux. Per i comandi da utilizzare vedere le diapositive di Alfeo. Non ho perso molto tempo su questa parte poichè le regole dell'esame telematico prevedono la scrittura di codice a mano.

La redirezione deve essere utilizzata obbligatoriamente durante l'esame (i comandi vanno imparati e non verranno forniti in caso di lacune durante l'esecuzione della prova). Di principio bisognerebbe già conoscere la redirezione grazie al corso di *Fondamenti di programmazione*.

Cartella di lavoro Il prompt dei comandi ha una sua cartella di lavoro, detta *directory*. Prima di eseguire i comandi di redirezione è necessario scegliere la cartella giusta, cioè quella dove si trovano i nostri file. Possiamo visualizzare le cartelle e i file presenti nella *directory* col comando

```
dir
```

e cambiare la cartella col comando

```
cd path
```

Potete porre direttamente il percorso eseguendo il comando una volta sola, o eseguire cd più volte spostandovi una cartella alla volta.

Operazioni di lettura Col comando di esecuzione del programma lo stream *cin* può essere rediretto su un file residente su memoria di massa. Utilizziamo il seguente comando

```
file.exe < input.in
```

Operazioni di scrittura Col comando di esecuzione del programma lo stream *cout* può essere rediretto su un file residente su memoria di massa. Utilizziamo il seguente comando

```
file.exe > output.out
```

Append Se poniamo >> al posto di > la redirezione non sovrascrive il contenuto del file (se esistente), ma aggiunge l'output in fondo al documento.

```
file.exe >> output.out
```

Redirezione in entrambi i sensi Possiamo compiere redirezione in entrambi i sensi scrivendo un comando del genere

```
file.exe < input.in > output.out
```

In questo modo avviamo l'eseguibile fornendo un input e ricevendo un output.

Capitolo 19

Giovedì 16/04/2020

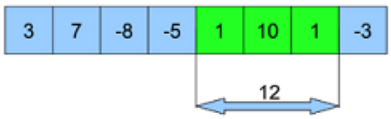
Registro 17 (*Gio 16/04/2020 13:30-15:30 (2:0 h) laboratorio*)

> *- Soluzione esercizi volta precedente e relativa complessità - Insertion Sort - Merge Sort - Altri Esercizi (lezione tenuta in modalità telematica - secondo e terzo gruppo di studenti)*

Oggi vediamo gli esercizi assegnati la scorsa volta cercando di ottenere la migliore complessità possibile. Analizzeremo, inoltre, due algoritmi di ordinamento: l'*insertionSort* e il *mergeSort*.

Complessità all'esame La complessità del programma, in sede di esame, riguarderà l'interno main, non soltanto la funzione. Considerare sempre la complessità massima nell'analisi dell'esercizio.

19.1 Somma massima



Input Array

Output Somma massima di elementi consecutivi

```
int somme1(int a[], int size) {
        int somma;
        int i, j, k;
        int max = a[0];
        for(int i = 0; i < size; i++) }
                for(j = i; j < size; j++) {
                        somma = 0;
                        for(k = i; k <= j; k++) {
                                somma += a[k];
                        }
                        if(somma > max) max = somma;
                }
        }
        return max;
}
```

Parto dalla prima locazione, scorro per individuare tutte le possibili combinazioni di sottoarray. Un sottoarray, ovviamente, è caratterizzato da una posizione iniziale e una finale. Utilizziamo dei for annidati, dove l'ultimo si occupa di effettuare la somma, salvandola se il valore è il massimo fino ad ora.

Complessità La complessità di questa funzione è $O(n^3)$, assolutamente insoddisfacente.

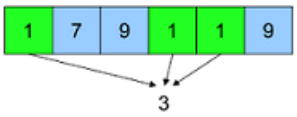
Versione più efficiente Vediamo quest'altra versione, dove eliminiamo un for.

```
int somme1(int a[], int size) {
        int somma;    int i, j, k;    int max = a[0];
        for(int i = 0; i < size; i++) }
                somma = 0;
                for(j = i; j < size; j++) {
                        somma += a[j];
                        if(somma > max) max = somma;
                }
        }
        return max;
}
```

La somma viene aggiornata man mano che si scorre l'array, individuando ogni volta se si ha un massimo o meno (abbiamo più operazioni di confronto, ma un for in meno).

Complessità La complessità è $O(n^2)$: abbiamo ridotto la complessità di un ordine di grandezza.

19.2 Occorrenza valore più frequente



Input Elementi array, intero k

Output Occorrenza valore più frequenti.

Tipicamente si pensa a una soluzione con complessità quadratica (for annidati, si scorre l'array verificando la frequenza di ogni elemento), ma la cosa può essere migliorata! Possiamo fare:

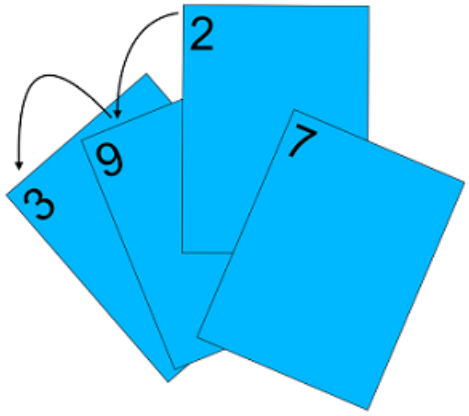
- un inserimento di complessità $O(n)$
- un ordinamento di complessità $O(n \log n)$
- calcolare le occorrenze su un array ordinato (complessità $O(n)$)
- stampare

La seguente soluzione ha complessità $O(n)$.

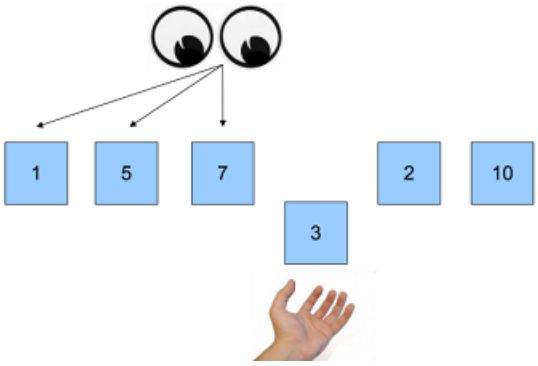
```
int maxOcc(int arr[], int size) {
        int countMax = 0;    int tmpCount = 1;
        for(i = 0; i < size-1; i++) {
            if(arr[i] == arr[i+1])
                tmpCount += 1;
            else {
                tmpCount > countMax ? countMax = tmpCount : countMax = countMax;
                tmpCount = 1;
            }
        }
        return countMax;
}
```

Tengo un unico contatore, che riporto a 1 ogni volta che trovo un valore diverso. Prima di ricominciare verifico se il conteggio è superiore al valore massimo, ed eventualmente salvo. Ho, in conclusione, una complessità $O(n \log n)$.

19.3 insertionSort



Possiamo rappresentare questo algoritmo attraverso un mazzo di carte che vogliamo ordinare: guardiamo il valore di una carte e lo raffrontiamo col valore delle carte adiacenti, trovando la posizione più idonea. Scambiamo la posizione delle carte finchè non individuiamo una situazione ordinata! Informaticamente parlando abbiamo un array (mano) e delle locazioni (carte), ma lo stesso meccanismo.



Utilizzeremo una variabile di appoggio per spostare un certo elemento in modo ordinato. Valutiamo di spostare il 3: lo metteremo tra 1 e 5. A quel punto posso considerare ordinate le prime quattro locazioni.

```
void sortArray(int arr[], int len) {
        int mano = 0;   int occhio = 0;
        for(int iter = 1; iter < len; iter++)  {
                mano = arr[iter];
                occhio = iter-1;

                while(occhio >= 0 && arr[occhio] > mano) {
                        arr[occhio+1] = arr[occhio];
                        occhio--;
                }

                arr[occhio+1] = mano;
        }
}
```

Facciamo attraverso i cicli ciò che facciamo attraverso gli occhi. All'interno del for:

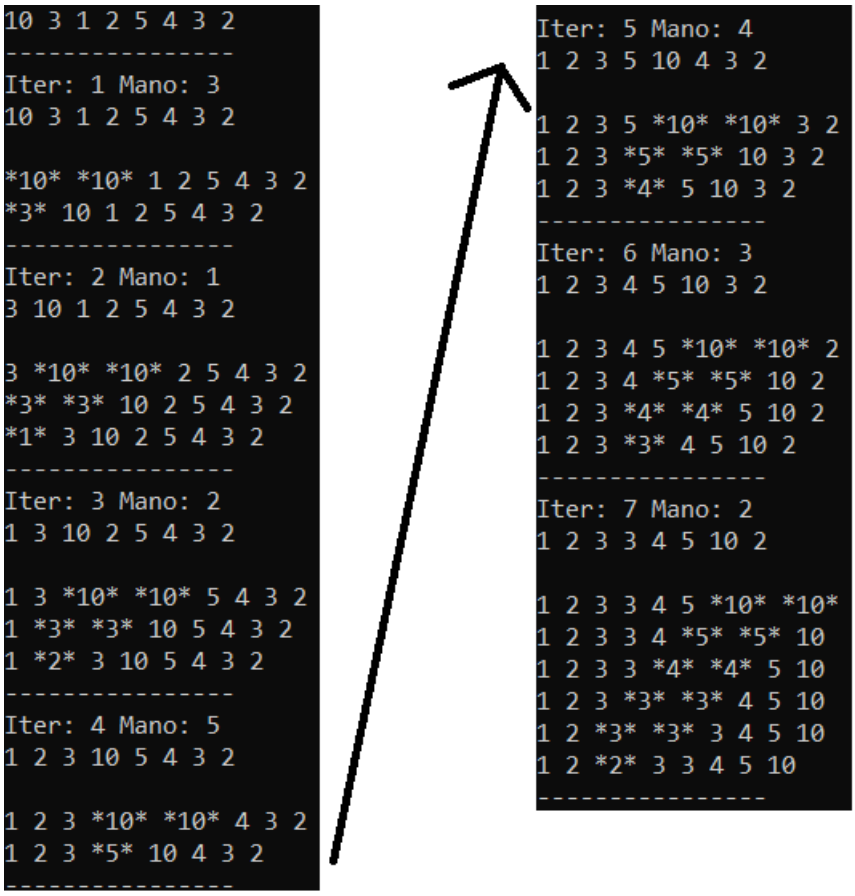
1. Inizializzo mano e occhio. L'occhio guarda l'elemento con indice antecedente a quello da cui abbiamo preso l'elemento messo nella mano.

2. Trovo posizione corretta scorrendo col while: man mano che mi muovo assegno all'elemento successivo a quello visto dall'occhio il valore di quello visto dall'occhio. Se osserviamo i valori nel corso dell'esecuzione dell'algoritmo individuiamo che avremo sempre due celle consecutive con lo stesso valore. Continuo finchè il valore dell'elemento visto con l'occhio sarà maggiore di quello presente nella mano[1].

3. Dopo aver fatto quanto detto prima avrò due elementi con lo stesso valore: prendo quello con indice minore e pongo come suo valore quello nella mano.

Osserviamo che si ha un operatore pre-incremento invece di quello post: questo perchè la mano deve essere uno step successivo rispetto all'occhio.

Esempio Ordiniamo l'array $[10, 3, 1, 2, 5, 4, 3, 2]$



Complessità Teniamo conto che più l'array è ordinato, minori saranno gli scambi effettuati. Nel caso peggiore ho tutti gli elementi disordinati: in quel caso ho complessità $O(n^2)$.

[1] Porre pre-decremento (pre-incremento) o pre-incremento (post-incremento) non fa differenza.

19.4 Funzione *sort* della STL

La STL fornisce una funzione *sort* che permette di ordinare gli elementi di un array (sia array tradizionale che vector).

`sort(stlArray.begin(), stlArray.end(), nome_funzione);`

Come argomenti abbiamo due iteratori: il primo consiste in un puntatore al primo elemento dell'array, il secondo in un puntatore all'ultimo elemento dell'array.

- Negli array possiamo ottenere un puntatore all'ultimo elemento sfruttando l'aritmetica dei puntatori vista a *Fondamenti di programmazione*. Dobbiamo conoscere il numero di elementi dell'array.
- La vector fornisce direttamente gli iteratori: begin() ed end()

Terzo argomento Il terzo argomento, opzionale, consiste nel nome di una funzione (*puntatori a funzione*, anche questa cosa vista a FdP) con cui stabilisco il metodo di ordinamento. Teniamo conto che di default gli elementi vengono ordinati in modo crescente.

- La funzione ha due argomenti, ciascuno del tipo degli elementi da ordinare (avrò int se sort dovrà ordinare, per esempio, un array o un vector di interi).
- La funzione restituisce un valore booleano: true se il primo elemento (il primo argomento) deve stare prima del secondo elemento (il secondo argomento), false in caso contrario.

Confronto con l'*insertionSort* La funzione *sort*() della STL è molto più efficiente: ha complessità $O(n \log n)$.

19.4.1 Esempio con due ordinamenti: crescente e decrescente

```
// In questo caso: il numero n1 va prima di n2 se n1 > n2
bool confronto(int n1, int n2) {
        return (n1 > n2);
}

int main() {

        int array[] = {5,1,7,2,9,3,8,4,5,10};
        int size = sizeof(array)/sizeof(array[0]);

        for(int i = 0; i < size; i++)
                cout<<array[i]<<endl;

        sort(array, array+size);

        cout<<endl;

        for(int i = 0; i < size; i++)
                cout<<array[i]<<endl;

        cout<<endl;

        sort(array, array + size, confronto);

        for(int i = 0; i < size; i++)
                cout<<array[i]<<endl;
}
```

19.4.2 Esempio un po' improprio: Articoli

Prendiamo ispirazione per un ulteriore esempio da alcuni CMS per siti web. Abbiamo una serie di articoli: ogni articolo è identificato da un ID univoco. Gli articoli più recenti avranno un'ID maggiore. Voglio caricare gli articoli dal più recente al più vecchio (quindi dall'ID più grande all'ID più piccolo). Voglio avere la possibilità di porre degli articoli in rilievo (cioè prima di tutti gli altri articoli) indipendentemente dalla data. Vediamo un ordinamento basato sulla *sort*:

```
struct articolo {
        int ID;
        bool rilievo;

        /* ... Altre info sull articolo ... */

        articolo(int i) : ID(i), rilievo(false) { }
        articolo(int i, bool r) : ID(i), rilievo(r) { }
};

bool confronto(articolo a1, articolo a2) {
        if(a1.rilievo > a2.rilievo)
                return true;
        else if(a1.rilievo == a2.rilievo)
                return (a1.ID > a2.ID);
        else
                return false;
}


int main() {
        vector<articolo> lista;

        articolo prova1(1, 1);
        lista.push_back(prova1);
        articolo prova2(2);
        lista.push_back(prova2);
        articolo prova3(3);
        lista.push_back(prova3);
        articolo prova4(4, 1);
        lista.push_back(prova4);
        articolo prova5(5);
        lista.push_back(prova5);

        for(int i = 0; i < lista.size(); i++)
                cout<<lista[i].ID<<endl;
        cout<<endl;

        sort(lista.begin(), lista.end(), confronto);

        for(int i = 0; i < lista.size(); i++)
                cout<<lista[i].ID<<endl;
        cout<<endl;

        return 0;
}
```

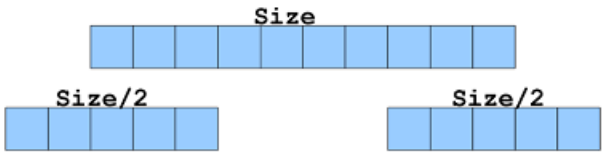
19.5 Confronto dei tempi di esecuzione

Possiamo fare un ulteriore confronto con la funzione time

```
time PERCORSO ESEGUIBILE
```

La funzione ci restituisce una sorta di feedBack in un confronto tra eseguibili: è ovvio che quello che va più veloce potrebbe avere minore complessità. Generalmente non si notano differenze sostanziali in esercizi semplici, la cosa cambia se lavoriamo con una grande quantità di dati!

19.6 mergeSort



Il mergeSort deriva dalla strategia del divide et impera: dividere un problema difficile in sottoproblemi. Ciò risulta vantaggioso con l'esecuzione in multiprocessore: la presenza di più processore permette di avere un certo parallelismo, quindi di svolgere calcoli in contemporanea. Se dobbiamo lavorare, per esempio, su 3GB di big data, risulterà necessario pensare il nostro codice in modo da renderlo *parallelizzabile su più processori*.
L'esempio più spinto si ha con le schede video: processori specializzati per svolgere calcoli in parallelo di complessità non esagerata.

Implementazione [2] La filosofia è la stessa già vista con la De Francesco. Di seguito vedremo un'implementazione per array. Ricordiamo che l'ordinamento di array può essere reso più efficiente con altri algoritmi.

```
void merge(int arr[], int start, int mid, int end) {
        int s = start;
        int d = mid;
        vector<int> buffer;

        while(true) {
                if(arr[s] < arr[d]  && s < mid)
                        buffer.push_back(arr[s++]);
                else if(d <= end)
                        buffer.push_back(arr[d++]);

                if(s == mid || d == end + 1) break;
        }

        if(s == mid && d <= end) {
                while(d < end + 1) buffer.push_back(arr[d++]);
        }
        else if(s < mid && d == end + 1) {
                while(s < mid) buffer.push_back(arr[s++]);
        }

        for(int i = start, k = 0; i <= end; i++, k++)
                arr[i] = buffer[k];
}
```

[2] Ringrazio l'amico Alessandro Corsi per la funzione *merge*

```
void mergeSort(int* arr, int start, int end) {
        int mid;
        if(start < end) {
                mid = (start+end)/2;
                mergeSort(arr, start, mid);
                mergeSort(arr, mid+1, end);
                merge(arr, start, mid+1, end);
        }
}
```

Filosofia del *merge*

- Utilizzeremo una vector di supporto (buffer).
- Iniziamo con un while che sarà interrotto solo con un break all'interno del body:
  - Immaginiamoci due scorrimenti in parallelo:
    - Uno che va dal primo elemento dell'array (start) fino a metà array (mid-1)
    - Uno che va dall'elemento medio (mid) fino all'ultimo elemento (end)
  - Metto a confronto un valore del primo scorrimento con un valore del secondo scorrimento:
    - Se il valore del primo scorrimento è minore del valore del secondo inserisco il primo nella vector buffer
    - Altrimenti inserisco il secondo nella vector buffer.
  - Ogni volta che confronto due elementi mi sposto di una posizione più avanti solo nello scorrimento dell'elemento appena inserito nella vector buffer (osservare gli operatori di post-incremento).
  - Questi inserimenti possono essere fatti solo se gli indici non sforano i confini stabiliti negli argomenti della funzione (cioè l'array $[start, end]$ e i due sottointervalli $[start, mid - 1]$ e $[mid, end]$). Appena si sfora uno di questi insiemi si esce dal while (vedere le condizioni nell'ultimo if del while).
- A questo punto può succedere che il buffer, dove stiamo costruendo l'array finale, rimanga incompleto: abbiamo inserito tutti gli elementi riguardanti uno scorrimento ma non abbiamo messo tutti gli elementi dell'altro.
- Ripensiamo alle condizioni con cui siamo usciti dal while: attraverso gli if stabiliamo quale tra i due scorrimenti non è stato inserito del tutto nella vector buffer.
- Dopo che abbiamo finito di riempire il buffer sovrascriviamo il contenuto dell'array.

Complessità Pensiamo al numero di passaggi da compiere. Divido l'array originale n volte (ottengo liste atomiche). Ho $\log(n) + 1$ livelli, quindi

$$n(\log(n) + 1) \longrightarrow n\log(n) + n$$

Quindi abbiamo complessità $O(n \log n)$. Non è "impattato" dall'input dato, ho sempre la complessità $O(n \log n)$ (caso migliore, peggiore e medio) poichè è obbligatorio "arrivare al fondo", cioè ottenere liste atomiche dividendo più volte la lista o l'array.

Capitolo 20

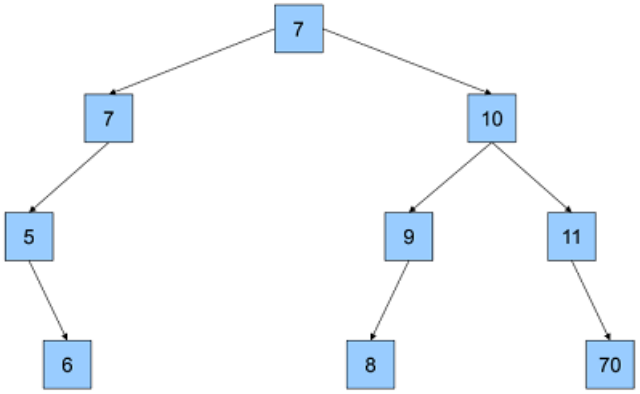
Venerdì 30/04/2020

Registro 18 (*Gio 30/04/2020 13:30-15:30 (2:0 h) laboratorio*)

- Alberi Binari di Ricerca - Progettazione e Inizializzazione - Esempi Funzioni con Alberi Binari di Ricerca - Alberi binari con etichette complesse - Esercizi

Oggi ci occupiamo di alberi binari di ricerca: impostazione di un albero, inizializzazione. Vedremo esempi di alberi sia con pseudocodice che con codice C++. Concluderemo, come le altre volte, con esercizi.

20.1 Alberi binari di ricerca



Un albero binario può essere immaginato come un qualcosa di rovesciato: la radice, negli alberi binari, consiste nel primo elemento, quello più in alto nella struttura. Ogni elemento può contenere informazioni, per esempio interi. Le foglie consistono in nodi privi di figli. Gli alberi binari di ricerca sono importanti poichè permettono una ricerca più efficiente di quella lineare.

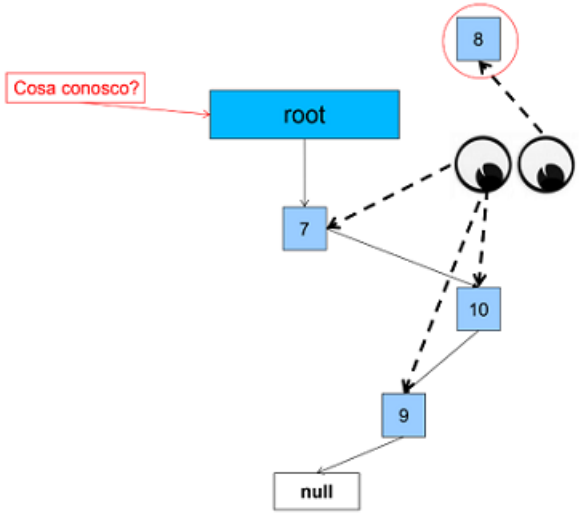
```
struct Node {
        int value;
        Node* left;
        Node* right;

        Node(int val):
                value(val), left(NULL), right(NULL) { }
};
```

Dobbiamo prevedere una struttura con un intero, due puntatori e, novità, inizializzazione dei valori: attraverso il codice indicato possiamo porre immediatamente i puntatori a NULL.

All'interno di una presunta classe *binTree* dovremo inserire un puntatore ad albero, precisamente un puntatore a radice. Anche questo deve essere posto a NULL se l'albero è vuoto (ricordare la definizione induttiva).

20.1.1 Inserimento

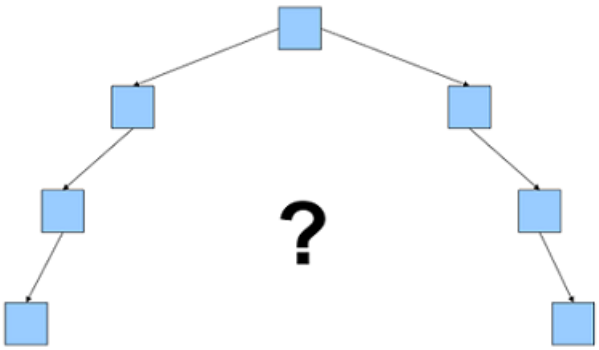
Voglio inserire un certo valore, effettuo una serie di confronti per trovare una posizione rispettosa delle caratteristiche dell'albero binario di ricerco.



Quindi:

- **Inizializzo**:
  - Inizializzo un nuovo elemento
  - Inizializzo le variabili di appoggio
- **Individuo posizione**:
  - finchè non raggiungo una foglia:
    - aggiorno variabili, se succede una certa cosa vado a sinistra, altrimenti a destra
- **Inserimento**:
  - se l'albero è vuoto aggiorno la radice
  - valuto se diventare figlio left o right

20.1.2 Troviamo min e max



Osserviamo che:

- Il minimo consiste nella foglia più a sinistra dell'albero
- Il massimo consiste nella foglia più a destra dell'albero

Possiamo farlo attraverso le seguenti funzioni:

```
Node* min() {
        Node* temp = root_;
        while(temp->left != NULL)
                temp = temp->left;
        return temp;
}

Node* max() {
        Node* temp = root_;
        while(temp->right != NULL)
                temp = temp->right;
        return temp;
}
```

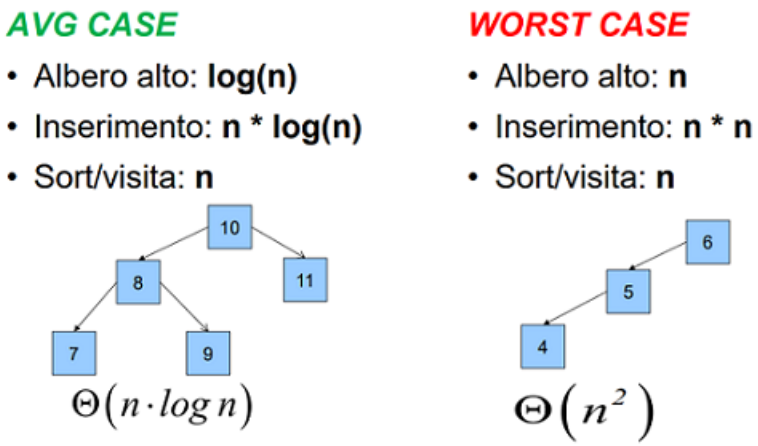
20.1.3 Visita dell'albero

Le visite risultano importanti per molte operazioni. Per esempio abbiamo la *inOrder*, che mi permette di porre gli elementi in ordine crescente (in un albero binario di ricerca). Le ricordiamo:

- *preOrder* (visita anticipata)
- *inOrder* (visita simmetrica)
- *postOrder* (visita differita)

20.1.4 Complessità per l'inserimento

Ho una complessità $O(n \log n)$ per l'inserimento. Ogni inserimento ha complessità $O(\log n)$: lo faccio per inserire n elementi! Il caso peggiore è quello in cui creo un albero degenere di fatto lineare: in questo caso avrò complessità $O(n^2)$. Se la complessità è quadratica l'albero binario di ricerca non risulta una soluzione migliore rispetto ad altre alternative.



20.1.5 Calcolare altezza albero

Con altezza albero intendiamo la distanza massima tra radice e foglie. Ogni volta calcoliamo l'altezza a sinistra e a destra. Dopo aver compiuto le ricerche restituisco il massimo tra i due risultati incrementato di 1.

```
int height(Node* tree) {
        int hLeft;
        int hRight;

        if(tree == NULL)
                return 0;

        hLeft = height(tree->left);
        hRight = height(tree->right);

        return 1 + max(hLeft, hRight);
}
```

20.1.6 Trova un elemento

La ricerca sarà orientata in base al valore che stiamo cercando: sfrutto le proprietà dell'albero binario di ricerca per orientarmi e capire che strada prendere per raggiungere il nodo.

```
bool search(Node* tree, int val) {
        if(tree == NULL)
                return false;

        bool found;

        if(tree->value == val)
                return true;
        else if(val <= tree->value)
                found = search(tree->left, val);
        else
                found = search(tree->right, val);

        return found;
}
```

20.1.7 Altezza di un elemento

Possiamo combinare le due funzionalità precedenti per individuare l'altezza di uno specifico elemento. Ricordiamo che normalmente, negli esercizi dati, non dovranno essere considerati duplicati: la posizione di un duplicato, se considerati, viene decisa da noi stabilendo una convenzione nel codice.

```
int search(Node* tree, int val) {
        if(tree == NULL)
                return 0;

        int cont = 0;
        if(tree->value == val)
                return 1;
        else if(val <= tree->value)
                cont = search(tree->left, val);
        else
                cont = search(tree->right, val);

        if(cont != 0)
                return cont+1;
        else
                return 0;
}
```

Quindi:

- Effettuo una ricerca per individuare un nodo avente una certa etichetta
- Quando l'ho trovato effettuo un conteggio dal nodo corrispondente alla radice.

20.2 Esercizi

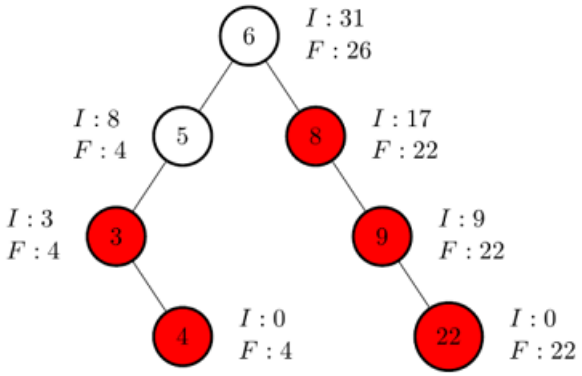
20.2.1 Somma Nodi

- **Input**: un intero N ed N interi
- **Operazioni**:
  - Inserire gli N interi in un albero binario di ricerca
  - Per ogni nodo u, calcolare $I(u)$ e $F(u)$
- **Output**: stampare le etichette dei nodi tale che $I(u) \leq F(u)$.

Spiegazioni somme

- $I(u)$: somma delle chiavi dei nodi interni del sottoalbero radicato in u, compresa la radice se e solo se questa non è anche foglia
- $F(u)$: somma delle chiavi delle foglie del sottoalbero radicato in u

Esempio di albero



Gli elementi rossi soddisfano la condizione $I(u) \leq F(u)$.

Soluzione personale

Osservazione importante: i valori di $I(u)$ e $F(u)$ di un nodo padre dipendono dagli stessi valori calcolati per i nodi figli.

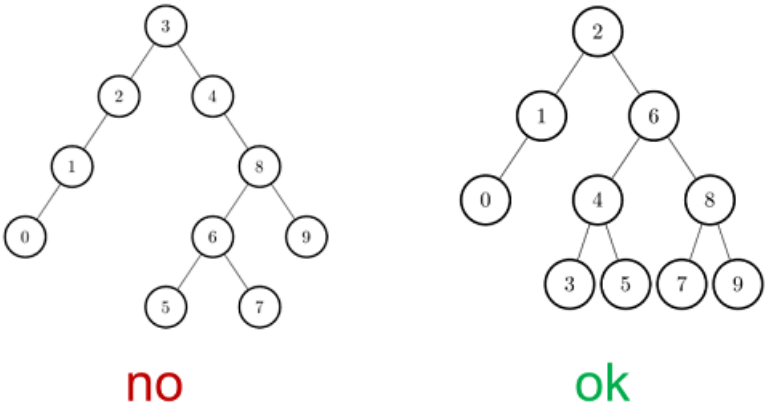
```
#include <iostream>
using namespace std;

struct nodo {
        int num;
        int I;
        int F;

        nodo* left;
        nodo* right;

        nodo(int val):
                num(val), left(NULL), right(NULL), I(0), F(0) { }
};

void inserisci(nodo*& albero, int num) {
        if(albero == NULL) {
```

```
                albero = new nodo(num);
                return;
        }
        if(num < albero->num)
                inserisci(albero->left, num);
        else if(num > albero->num)
                inserisci(albero->right, num);
}

void determina_I_F(nodo* albero, int& I, int& F) {
        if(!albero) {
                I = 0; F = 0;
                return;
        }

        int IL, IR, FL, FR;
        determina_I_F(albero->left, IL, FL);
        determina_I_F(albero->right, IR, FR);

        if(!albero->left && !albero->right)
                albero->F = (F = FL + FR + albero->num);
        else
                albero->F = (F = FL + FR);

        if(!(!albero->left && !albero->right))
                albero->I = (I = IL + IR + albero->num);
        else
                albero->I = (I = IL + IR);
}

void inOrder(nodo* alb) {
        if(alb) {
                inOrder(alb->left);
                if(alb->I <= alb->F)
                        cout<<alb->num<<endl;
                inOrder(alb->right);
        }
}

int main() {
        int N;
        cin>>N;

        nodo* albero = NULL;
        for(int i = 0; i < N; i++) {
                int n;
                cin>>n;

                inserisci(albero, n);
        }

        int I, F;
        determina_I_F(albero, I, F);
        inOrder(albero);
}
```

20.2.2 Altezza figli

- **Input**: N interi da inserire in un albero binario di ricerca
- **Verificare che**: per ciascun nodo, l'altezza dei suoi sottoalberi sinistro e destro deve differire al massimo di uno

Esempi di alberi



Soluzione personale

Attenzione al return false quando si ha un albero vuoto. Non posso non ignorare questo caso, altrimenti restituirrei sempre false con qualunque albero. Quindi considero i booleani falsi soltanto se derivano da alberi non vuoti (vedere la condizione immediatamente sotto le chiamate ricorsive)

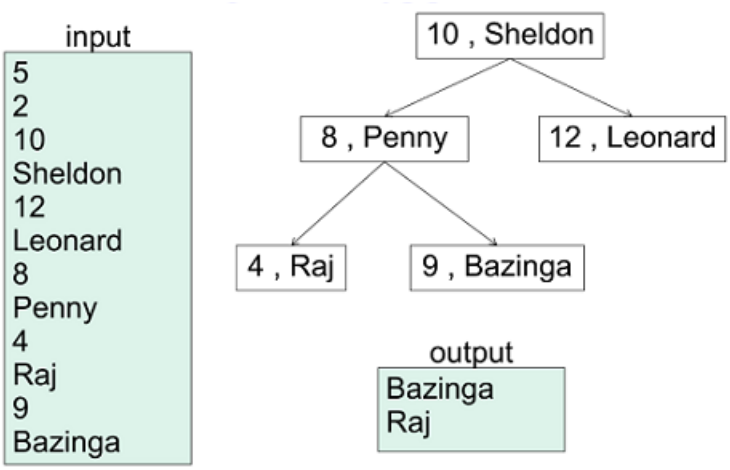
```
#include <iostream>
#include <stdlib.h>
using namespace std;

int N;

struct nodo {
        int num;

        nodo* left;
        nodo* right;

        nodo(int n) : num(n), left(NULL), right(NULL) { }
};

void inserisci(nodo*& albero, int num) {
        if(albero == NULL) {
                albero = new nodo(num);
                return;
        }

        if(num <= albero->num)
                inserisci(albero->left, num);
        else if(num > albero->num)
                inserisci(albero->right, num);
}
```

```
bool verifica_altezza(nodo* albero, int& H) {
        if(albero == NULL) {
                H = 0;
                return false;
        }

        int HL, HR;
        bool vl = verifica_altezza(albero->left, HL);
        bool vr = verifica_altezza(albero->right, HR);

        if((vl == false && albero->left != NULL || vr == false && albero->right
            != NULL))
                return false;

        H = max(HL, HR) + 1;

        if(abs(HL-HR) > 1)
                return false;
        else
                return true;
}

int main() {
        cin>>N;
        nodo* albero = NULL;

        for(int i = 0; i < N; i++) {
                int n;
                cin>>n;

                inserisci(albero, n);
        }

        int H;
        cout<<verifica_altezza(albero, H)<<endl;
}
```

20.2.3 Albero binario a etichette complesse

- **Input**: un intero N, un intero H, N coppie $[intero, stringa]$
- **Operazioni**: inserire le N coppie in un albero binario binario di ricerca (usando il valore intero come chiave)
- **Output**: stringhe che si trovano in nodi ad altezza H, stampate in ordine lessicografico.

Esempio



Procedimento

- Implemento struttura dati che supporti albero binario ed etichette multi valore
- Inserisco gli elementi
- Trovo i nodi ad altezza H
- Riordino gli elementi basandomi sulle stringhe

Soluzione personale

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int N, H;
vector<string> nomi;

struct nodo {
        int num;
        int H;
        string nome;

        nodo* left;
        nodo* right;

        nodo(int nu, string no, int h) : num(nu), nome(no), H(h), left(NULL),
            right(NULL) {}
};
```

```
void inserisci(nodo*& albero, int num, string nome, int h = 0) {
        if(albero == NULL) {
                albero = new nodo(num, nome, h);
                return;
        }

        if(num <= albero->num)
                inserisci(albero->left, num, nome, h + 1);
        else if(num > albero->num)
                inserisci(albero->right, num, nome, h + 1);
}

void inOrder(nodo* albero, int h) {
        if(albero) {
                inOrder(albero->left, h);

                if(albero->H == h)
                        nomi.push_back(albero->nome);

                inOrder(albero->right, h);
        }
}

int main() {
        cin>>N>>H;
        nodo* albero = NULL;

        for(int i = 0; i < N; i++) {
                int nu;
                string no;
                cin>>nu>>no;

                inserisci(albero, nu, no);
        }

        inOrder(albero, H);

        sort(nomi.begin(), nomi.end());
        for(int i = 0; i < nomi.size(); i++)
                cout<<nomi[i]<<endl;
}
```

20.2.4 Nodi *concordi* e *discordi*

Si consideri un sistema di memorizzazione che legga una sequenza di N interi unici e non negativi e li inserisca dentro un albero binario di ricerca (ABR). Si definiscono:

- Concordi i nodi c caratterizzati da altezza del nodo pari (dispari) e etichetta pari (dispari).
- Discordi i nodi d con altezza pari e etichetta dispari, o viceversa.

L' altezza del nodo x corrisponde al numero di nodi compresi tra x e la radice dell'albero, x escluso. La radice ha altezza 0. Scrivere un programma che:

- legga da tastiera N, il numero di interi da memorizzare nell'albero;
- legga da tastiera N interi, ovvero le etichette dei nodi da inserire nell'ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti (le etichette $\leq$ vanno inserite a sinistra);
- stampi la differenza tra la sommatoria S delle etichette dei nodi discordi e la stessa dei nodi concordi

$$S = \sum label(d) - \sum label(c)$$

Esempio di albero

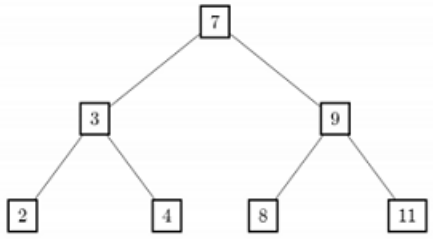
Esempio

Input

```
7
7
3
9
2
4
8
11
```

Output

```
-8
```



20.2.5 Nodi completi

Si consideri un sistema di memorizzazione che legga una sequenza di N interi unici e non negativi e li inserisca dentro un albero binario di ricerca (ABR). Si definiscono completi i nodi c aventi due nodi figli, incompleti i nodi i restanti. Scrivere un programma che:

- legga da tastiera N, il numero di interi da memorizzare nell'albero;
- legga da tastiera N interi, ovvero le etichette dei nodi da inserire nell'ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti (le etichette $\leq$ vanno inserite a sinistra);
- stampi la differenza tra la sommatoria S delle etichette dei nodi completi e la stessa dei nodi incompleti

$$S = \sum label(c) - \sum label(i)$$

Esempio di albero

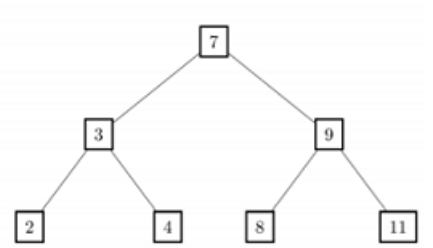
Esempio

Input

```
7
7
3
9
2
4
8
11
```

Output

```
-6
```



Possibile Soluzione: nodi “concordi” e “discordi”

```
#include <iostream>      // std::cout
#include <algorithm>     // std::sort
#include <vector>        // std::vector
#include <fstream>
#include <math.h>        /* floor */
#include <stdlib.h>
#include <cmath>         /* pow  */

using namespace std;


struct Node
{
    int value;
    Node * left;
    Node * right;

    Node( int i ): value(i) , left(NULL) , right(NULL) {}
};
```



```
class BinTree
{
    Node * root_;
public:
    BinTree() { root_ = NULL ; }

    Node * getRoot() { return root_; cout << "getRoot" << endl;}

    void insert( int i )
    {
        Node * node = new Node(i);

        Node * pre = NULL;
        Node * post = root_;
        while( post != NULL)
        {
            pre = post;
            if( i <= post->value )
            {
                post = post->left;
            }
            else
            {
                post = post->right;
            }
        }

        if( pre == NULL )
            root_ = node;
        else if( i <= pre->value )
        {
            pre->left = node;
        }
        else
        {
            pre->right = node;
        }
        return;
    }
};
```



```
int layerTree( Node * tree, int altezza)
{
    // Nodo non trovato
    if( tree == NULL)  {
       return 0;
    }

    if (altezza++%2 != tree->value%2)  {
          //cout<<tree->value<<" : "<<altezza-1<<" discorde"<<endl;
          return layerTree( tree->left , altezza) + layerTree( tree->right , altezza) + tree->value;
    }
    else   {
          //cout<<tree->value<<" : "<<altezza-1<<" concorde"<<endl;
          return layerTree( tree->left , altezza) + layerTree( tree->right , altezza) - tree->value;
    }
}


int main()
{
    int N ;
    int x ;
    BinTree albero ;

    cin >> N ;

    // riempio l' albero
    for(int i=0 ; i<N ; ++i )   {
        cin >> x;
        albero.insert(x);
    }

    cout<<layerTree(albero.getRoot(), 0)<<endl;
}
```


Possibile soluzione: nodi “completi”

```
#include <iostream>      // std::cout
#include <algorithm>     // std::sort
#include <vector>        // std::vector
#include <fstream>
#include <math.h>        /* floor */
#include <stdlib.h>
#include <cmath>         /* pow  */

using namespace std;


struct Node
{
    int value;
    Node * left;
    Node * right;

    Node( int i ): value(i) , left(NULL) , right(NULL) {}
};
```



```
class BinTree
{
    Node * root_;
public:
    BinTree() { root_ = NULL ; }

    Node * getRoot() { return root_; cout << "getRoot" << endl;}

    void insert( int i )
    {
        Node * node = new Node(i);

        Node * pre = NULL;
        Node * post = root_;
        while( post != NULL)
        {
            pre = post;
            if( i <= post->value )
            {
                post = post->left;
            }
            else
            {
                post = post->right;
            }
        }

        if( pre == NULL )
            root_ = node;
        else if( i <= pre->value )
        {
            pre->left = node;
        }
        else
        {
            pre->right = node;
        }
        return;
    }
};
```



```
int complete( Node * tree)
{
    // Nodo non trovato
    if( tree == NULL)  {
       return 0;
    }

    if (tree->left != NULL && tree->right != NULL)  {
          return complete( tree->left ) + complete( tree->right ) + tree->value;
    }
    else   {
          return complete( tree->left ) + complete( tree->right ) - tree->value;
    }
}


int main()
{
    int N ;
    int x ;
    BinTree albero ;

    cin >> N ;

    // riempio l' albero
    for(int i=0 ; i<N ; ++i )   {
        cin >> x;
        albero.insert(x);
    }

    cout<<complete(albero.getRoot())<<endl;
}
```

Capitolo 21

Giovedì 21/05/2020

Registro 19 (*Gio 21/05/2020 13:30-15:30 (2:0 h) laboratorio*)

- Heap - Ordinamento tramite Heap - Hashing - Hashing e tipi di input - Esercizi

21.1 Heap

Gli heap sono uno specifico tipo di albero binario completo. Contrariamente a un albero normale abbiamo una struttura differente senza perdita di informazione. Ovviamente seguono delle proprietà diverse e degli usi diversi. Il tutto ci permette di svolgere certe operazioni con complessità logaritmica.

L'inserimento avviene per livelli: si riempe un livello e poi si passa a quello successivo. L'unica forma di non completezza ammessa può manifestarsi solo nell'ultimo livello dell'albero, a destra. Non possiamo porre spazi vuoti tra elementi dello heap.

Rappresentazione dello heap Lo heap è rappresentato mediante array. La radice ha indice 0. Per semplificarci la vita possiamo creare classe.

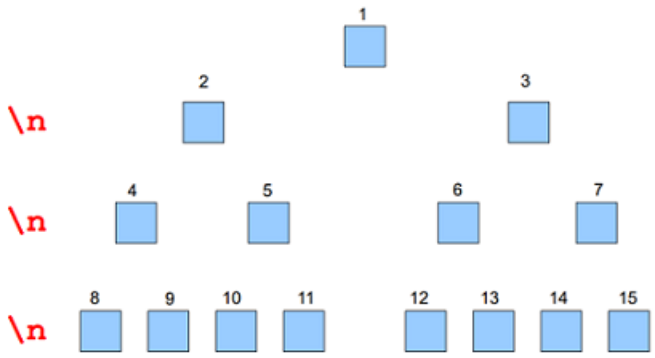
```
class Heap
{
    std::vector<int> data_;

    int length_;     // lunghezza Array
    int size_;   // dimensione Heap

public:
    Heap() {};

    void fill( int l );
    void printVector();

   ...
}
```

Abbiamo due interi: uno con la lunghezza dell'array, uno con la dimensione dello heap. Avremo tra le funzioni riempimento, estrazione e stampa degli elementi (e non solo).

Navigazione Attraverso le seguenti funzioni possiamo individuare il padre di un nodo, il figlio sinistro o il figlio destro. La cosa avviene praticamente a costo zero. In riferimento al padre abbiamo un arrotondamento per difetto (funzione *floor*).

```
int parent(int i) {
        return floor((i-1)/2);
}

int getLeft(int i) {
        return (i*2)+1;
}

int getRight(int i)  {
        return (i*2)+2;
}
```

Quando andare a capo? Nella stampa è possibile individuare quando andare a capo, cioè quando finisce il livello di un albero.



Sfruttiamo le nozioni imparate nella *rappresentazione dell'informazione*. Sappiamo benissimo che ogni numero viene rappresentato attraverso una sequenza binaria. Consideriamo le posizioni degli elementi come nella foto (primo elemento è in posizione 1).
Prendiamo, per esempio, l'elemento in posizione 7 (ultimo elemento del livello 2) e quello in posizione 8 (primo elemento del livello 3). Applicando l'operatore AND bit a bit si ottiene il seguente risultato

$$1000\&0111 = 0000 = 0$$

Questa cosa vale sempre e ci permette di capire quando un certo elemento è il primo o l'ultimo di un certo livello. Possiamo scrivere la seguente funzione

```
bool isFirstChild(int i) {
        if( (i!=0) && ((i&(i-1)) == 0))
                return true;
        else
                return false;
}
```

Restituisco *true* se l'elemento in questione è il primo di un certo livello. Segue

```
void print() {
        for( int i=0 ; i < length_ ; ++i ) {
                if( isFirstChild(i+1) )
                        cout << endl;
                cout << data_[i] << "\t";
        }
}
```

Ogni volta mi chiedo se l'elemento successivo a quello che sto analizzando risulta essere il primo elemento di un nuovo livello.

Ulteriore osservazione Dato un livello k attraverso l'operazione 2^k posso individuare il numero di elementi presenti in un certo livello. Vediamo che

- $2^0 = 1$. Il livello 0 ha un solo elemento (la radice)
- $2^1 = 2$. Il livello 1 ha due elementi (i figli della radice)
- $2^2 = 4$. Il livello 2 ha quattro elementi
- $2^3 = 8$. Il livello 3 ha otto elementi

Proprietà dei valori Si distinguono i *max-heap* dai *min-heap*. Dato un qualunque nodo con etichetta

- nel primo heap i figli avranno etichetta minore;
- nel secondo heap i figli avranno etichetta maggiore.

Trovare il massimo? La cosa ha complessità $O(1)$. Grazie alle proprietà dello heap so che l'elemento con valore più grande si trova in prima posizione (ovviamente parliamo di un *max-heap*.

21.1.1 Costruire uno heap

Abbiamo un array e vogliamo trasformarlo in una rappresentazione di uno *heap*. Creiamo una funzione ricorsiva *buildMaxHeap*

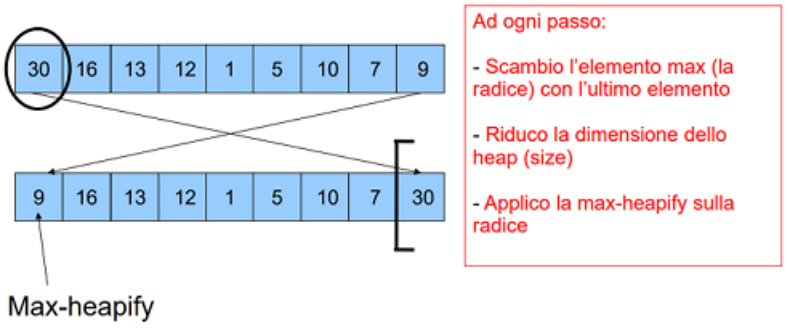
```
void buildMaxHeap() {
        size_ = lenght_;
        int i = floor(lenght_/2)-1;
        for(; i >= 0; --i) {
                maxHeapify(i);
                print();
        }
}

void maxHeapify(int i) {
        int left = getLeft(i);
        int right = getRight(i);
        int largest;

        if((left < size_) && (data_[left] > data_[i]))
                largest = left;
        else
                largest = i;

        if((right < size_) && (data_[right] > data_[largest]))
                largest = right;

        if(largest != i) {
                scambia(i, largest);
                maxHeapify(largest);
        }
}
```

21.1.2 heapSort

Abbiamo già visto con la De Francesco come certi meccanismi tipici dello *heap* possano essere utilizzati per ottenere l'ordinamento di un array. Creiamo la seguente funzione:

```
void heapSort() {
        int i = lenght_ -1;
        for(;i > 0; --i) {
                scambia(0,i);
                --size_;
                maxHeapify(0);
        }
}
```

Fondamentalmente:



21.1.3 Funzioni di libreria riguardanti lo heap

Nella STL abbiamo le seguenti funzioni, entrambe con due iteratori come argomenti:

```
#include <algorithm>
make_heap(inizio, fine);
pop_heap(inizio, fine);
```

- Con *make_heap* trasformo l'array in uno heap
- Con *pop_heap* sposto l'elemento di valore più grande (nel *max-heap* quello con indice 0) nell'ultima posizione disponibile dell'array. Se dopo questa funzione chiamo anche

L'ultima funzione, assieme alla *pop_back()* della vector (con cui rimuovo l'elemento in ultima posizione dell'array), ci permette di completare l'estrazione di un elemento dallo heap.

Esempio Nel seguente esempio abbiamo la trasformazione di un array in uno heap e il suo svuotamento (le estrazioni di elementi dallo heap continuano finchè il vettore non sarà vuoto).

```
#include <vector>
#include <algorithm>
vector<int> vect;

for( int i = 0 ; i < quanti ; ++i ) {
        cin >> val;
        vect.push_back(val);
}

make_heap(vect.begin(),vect.end());
```

```
while(!vect.empty()) {
        cout << "top " << *vect.begin() << endl;
        pop_heap(vect.begin(),vect.end());
        vect.pop_back();
}
```

21.1.4 Funzioni di libreria riguardanti la coda

Possiamo gestire una coda con priorità (già vista a *Fondamenti di programmazione*) attraverso la seguente classe

```
#include <queue>
priority_queue<int> prioQ;
prioQ.push(val);
prioQ.top();
prioQ.pop();
```

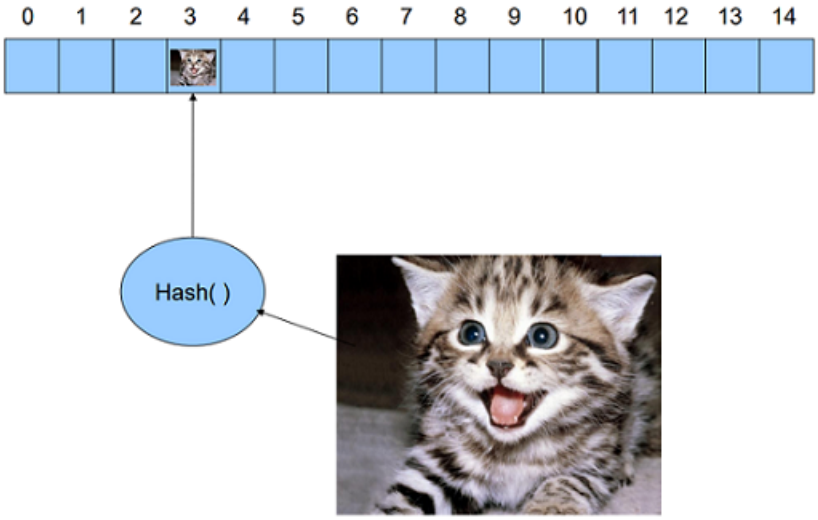
- La *push* mi permette di aggiungere un certo elemento *val* in coda.
- Con la *top* individuo l'elemento in cima alla coda
- Con la *pop* elimino l'elemento in cima alla coda facendo scorrere il tutto.

Esempio Vediamo un esempio in cui aggiungiamo una serie di elementi alla coda e poi li rimuoviamo (rimuovo elementi dalla testa della coda finchè questa non risulterà vuota).

```
#include <queue>
priority_queue<int> prioQ;

for( int i = 0 ; i<quanti ; ++i ) {
        cin >> val;
        prioQ.push(val);
}

while(!prioQ.empty()) {
        cout << "top " << prioQ.top() << endl;
        prioQ.pop();
}
```

21.2 Hash

Gli array sono caratterizzati da un indirizzamento diretto. Con l'hashing superiamo il limite del dover sapere cosa è presente all'interno dell'insieme. L'hashing fornisce un indirizzamento non basato sulla memoria ma su un valore da noi fornito. Abbiamo una funzione che restituisce un indice della memoria dato come argomento ciò che vogliamo trovare.



Struttura dati Come struttura dati possiamo usare l'array o il vector. Risulta importante porre una dimensione massima all'array (discorso in riferimento al vector).

21.2.1 Simple hash table

Prendiamo l'esempio più semplice

- Abbiamo come input interi positivi
- Tutte le chiavi delle locazioni possono essere ottenute mediante la funzione hash
- La funzione hash è la funzione modulo
- Si pone una marca (per esempio lo 0) come valore delle locazioni di elementi non presenti.

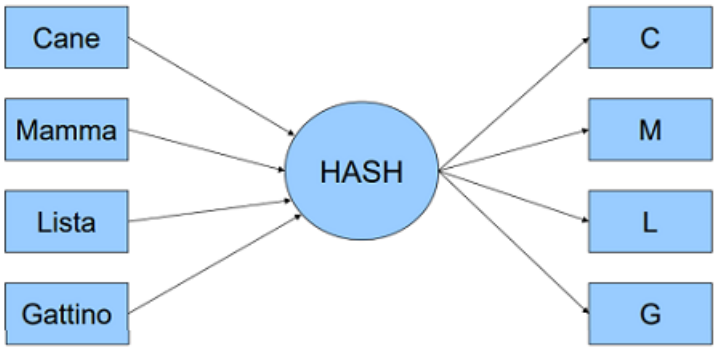
21.2.1.1 Esempio di inserimento

Consideriamo le seguenti funzioni appartenenti a una pseudoclasse *HashTable*

```
int HashTable::hash( int key ) {
        return key % size_;
}

bool HashTable::insert( int key ) {
        int index = hash(key);

        if( table_[index] != 0 )
                return false;

        table_[index] = key;
        return true;
}
```

La prima consiste nella funzione hash adottata, la seconda permette di inserire un elemento. Se la casella è già utilizzata restituisco *false* e non effettuo modifiche, negli altri casi modifico il valore della locazione.

21.2.2 Caso particolare: hashing di stringhe



Potrebbe capitare la necessità di porre stringhe all'interno del mio array. Quale funzione hash potrei adottare? Potrebbe essere cosa interessante utilizzare la prima lettera della stringa (considerando quanto detto a *Fondamenti di programmazione* sugli operatori matematici applicati al tipo char).

```
int HashTable::hash(string key ) {
        return key[0]%size_;
}
```

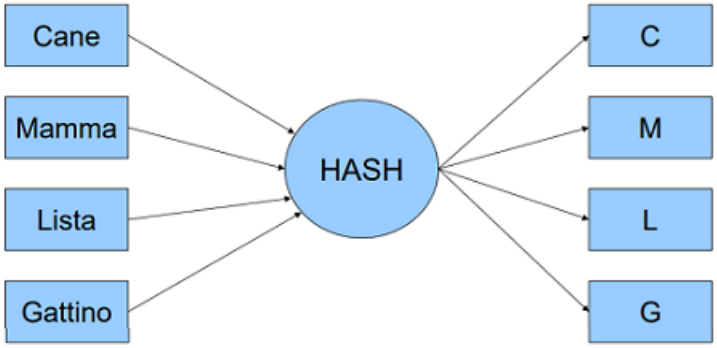
Altra alternativa potrebbe essere considerare tutti i caratteri attraverso un for

```
int HashTable::hash(string key ) {
        for(int i = 0; i < key.lenght(); i++)
                index = (index + key[i])%size_;

        return index;
}
```

21.2.3 Good HASH

- Dipende dal tipo di applicazione
- Applicazioni che coinvolgono la crittografia richiedono di essere difficilmente invertibili.
- Per applicazioni di indexing è fondamentale l'uniformità
- Lavorano su rappresentazione binaria.

21.2.4 Gestione delle collisioni



Possiamo adottare degli ulteriori accorgimenti e inserire un qualunque elemento anche in presenza di collisione (cioè di cella già utilizzata). Consideriamo una grande varietà di elementi (precisamente n elementi) e un array con m celle. Convenzione vuole che $m < n$. Possiamo utilizzare il metodo visto a lezione con la De Francesco (*legge di scansione lineare*, scorrimento dell'array) oppure le cosiddette *liste di trabocco*

- Abbiamo un array di puntatori
- Ogni cella contiene un puntatore a lista
- Ogni elemento della lista punta sia all'elemento precedente che a quello successivo. Considero che uno dei puntatori è NULL se ci troviamo nel primo o nell'ultimo elemento della lista che stiamo scorrendo.

```
struct Elem {
        int key;
        Elem * next;
        Elem * prev;

        Elem(): next(NULL) , prev(NULL) {}
};
```

Le funzioni che ci potrebbero interessare sono:

- l'inserimento
- la stampa
- la ricerca

21.3 Classe *map* della STL

Possiamo utilizzare la seguente classe nella gestione degli hash

```
std::map<key_T,obj_t> table;
```

con Key_T che consiste nel tipo della chiave e obj_t nel tipo di dati. Ponendo $Key\_t = obj\_t = string$ possiamo fare

```
table[‘uno’] = ’’valore uno’’;
table.find(‘uno’);
```

Esercizio

Si consideri un sistema di memorizzazione dei dati relativi a dei veicoli. Ogni veicolo è rappresentato da un valore *targa* univoco, intero e positivo, e da un valore intero *categoria* compreso tra 0 e $C-1$. Il sistema legge i dati relativi a N veicoli e li inserisce dentro una *tabella hash*, utilizzando la *targa* come etichetta. La tabella hash è realizzata con il metodo di concatenazione.

Per ogni indice i della tabella hash si definisce $M(i)$ come la categoria con più veicoli in i, e $V(i)$ come il numero di veicoli in i appartenenti a $M(i)$. A parità di numero di veicoli, si considerino le *categorie* in ordine crescente. Scrivere un programma che:

- legga da tastiera una sequenza di N coppie di interi {*targa*,*categoria*} e le inserisca nella tabella hash all'indirizzo dato dalla seguente funzione hash:

  $$h(x) = \{[(\ a \times x\ ) +\ b\ ]\ \%\ p\ \}\ \%\ (C)$$

  dove p=999149 , a=1000 e b=2000;

- Stampi a video i primi K indirizzi i della tabella hash in ordine di $V(i)$ decrescente. A parità di $V(i)$, si considerino gli indirizzi in ordine crescente.

L'**input** è formattato nel seguente modo: la prima riga contiene gli interi N, K e C separati da uno spazio. Seguono N righe contenenti una coppia di interi ciascuna.

L'**output** contiene gli elementi della soluzione, uno per riga.

Esempio

Input	Output
11 6 7 170 1 96 2 577 0 692 0 873 6 446 6 732 1 394 1 845 1 715 1 456 3	0 3 1 2 4 6

Possibile soluzione

```cpp
#include <iostream>     // std::cout
#include <algorithm>    // std::sort
#include <vector>       // std::vector
#include <fstream>
#include <math.h>       /* floor */
#include <stdlib.h>
#include <cmath>        /* pow  */

using namespace std;


struct Veicolo
{
    int targa;
    int categoria;

    Veicolo(): targa(-1) , categoria(-1) {};
    Veicolo( int t , int c ): targa(t) , categoria(c) {}
};

// conterrà le info relative a ciascuna entrata della tabella hash
struct Info
{
    int indirizzo;
    int veicoli;
};
```



```cpp
class Concatenazione
{
    vector<int> categorie_;
    vector<Veicolo> veicoli_;

public:
    Concatenazione( int C )
    {
        categorie_.resize(C, 0);
    }

    // inserisci e aggiorna l'utente max
    void inserisci( Veicolo v )
    {
        veicoli_.push_back(v);

        ++categorie_[v.categoria];
    }

    int getMaxV()
    {
        int vMax = -1;
        int c;

        for( c = 0 ; c < categorie_.size(); ++c )
        {
//            cout << "\t" << c << "," << categorie_[c] << endl;
            if( categorie_[c] > vMax )
            {
                vMax = categorie_[c];
            }
        }
        return vMax;
    }
};
```



```cpp
bool compare( Info a , Info b )
{
    if( a.veicoli > b.veicoli )
        return true;
    else if( a.veicoli == b.veicoli)
        return a.indirizzo < b.indirizzo;
    else
        return false;
}

int main()
{
    unsigned int N , K , C;
    int x , c ;

    cin >> N >> K >> C;
    Concatenazione cBase(C);
    vector<Concatenazione> hashTable(C, cBase);

    int pos;

    Veicolo tempVeicolo;
    // riempio le liste di concatenazione
    for(unsigned int i=0 ; i<N ; ++i )
    {
        // leggo e creo un nuovo utente
        cin >> x >> c;
        tempVeicolo.targa = x;
        tempVeicolo.categoria = c;

        pos = hashFun( x , C );

//          cout << x << "," << c << " -> " << pos << endl;

        hashTable[pos].inserisci(tempVeicolo);
    }
```



```cpp
    // calcolo insieme I
    vector <Info> info;
    Info tempInfo;
    for( unsigned int i=0 ; i<C ; ++i )
    {
        tempInfo.indirizzo = i;

//          cout << endl << "============ " << i << " ============" << endl;
        tempInfo.veicoli = hashTable[i].getMaxV();
        info.push_back( tempInfo );
    }
    sort( info.begin() , info.end() , compare );

    for( unsigned int i=0 ; i < K && i<info.size() ; ++i )
    {
        cout << info[i].indirizzo << endl;
    }
}
```

Parte III

Materiale aggiuntivo

Appendice A

Relazioni di ricorrenza

A.1 Regole (riprese dalla lezione dedicata)

Divide et impera

$$\begin{cases} T(n) = d & n \leq m \\ T(n) = hn^k + aT(n/b) & n > m \end{cases}$$

con $h > 0$

Teorema

- Se $a < b^k, T(n) \in O(n^k)$
- Se $a = b^k, T(n) \in O(n^k \log n)$
- Se $a > b^k, T(n) \in O(n^{\log_b a})$

Relazioni lineari

$$\begin{cases} T(0) = d \\ T(n) = bn^k + a_1 T(n-1) + a_2 T(n-2) + \cdots + a_r T(n-r) \end{cases}$$

Due chiamate ricorsive, complessità esponenziale:

$$a_1 = 1, a_i = 0 \; i > 1$$

Caso elementare

$$\begin{cases} T(0) = d \\ T(n) = bn^k + T(n-1) \end{cases}$$

$T(n) \in O(n^{k+1})$

A.2 Raccolta

Fattoriale ($O(n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(n-1) \end{cases}$$

selectionSort ($O(n^2)$)

$$\begin{cases} T(1) = a \\ T(n) = bn + T(n-1) \end{cases}$$

quickSort ($O(n \log n)$)

$$\begin{cases} T(1) = a \\ T(n) = bn + T(k) + T(n-k) \end{cases}$$

Caso peggiore ($O(n)$)

$$\begin{cases} T(1) = a \\ T(n) = bn + a + T(n-1) \end{cases}$$

Caso migliore ($O(n \log n)$

$$\begin{cases} T(1) = a \\ T(n) = bn + 2T(n/2) \end{cases}$$

Ricerca lineare ($O(n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(n-1) \end{cases}$$

Ricerca binaria ($O(\log n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(n/2) \end{cases}$$

Ricerca pseudo-binaria ($O(n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + 2T(n/2) \end{cases}$$

Torre di Hanoi ($O(2^n)$)

$$\begin{cases} T(1) = a \\ T(n) = b + 2T(n-1) \end{cases}$$

Serie di Fibonacci

Fibonacci cattivo ($O(2^n)$)

$$\begin{cases} T(0) = T(1) = a \\ T(n) = b + T(n-1) + T(n-2) \end{cases}$$

Fibonacci buono ($O(n)$)

$$\begin{cases} T(0) = d \\ T(n) = b + T(n-1) \end{cases}$$

mergeSort ($O(n \log n)$)

Funzione principale ($O(n \log n)$)

$$\begin{cases} T(0) = T(1) = d \\ T(n) = bn + T(n/2) \end{cases}$$

split ($O(n)$)

$$\begin{cases} T(0) = T(1) = d \\ T(n) = b + T(n-2) \end{cases}$$

merge ($O(n)$)

$$\begin{cases} T(0) = d \\ T(n) = b + T(n-1) \end{cases}$$

Alberi

Visita di un albero ($O(n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(n_s) + T(n_d) \;\; con\ n_s + n_d = n - 1\ con\ n > 0 \end{cases}$$

Forma da noi calcolabile valida per un albero bilanciato

$$\begin{cases} T(0) = a \\ T(n) = b + 2T((n-1)/2) \end{cases}$$

Visite in funzione dei livelli ($O(2^k)$)

$$\begin{cases} T(0) = a \\ T(k) = b + 2T(k-1) \end{cases}$$

Ricerca in un ABR $O(\log n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(k) \quad k < n \end{cases}$$

Caso peggiore ($O(n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(n-1) \end{cases}$$

Caso migliore ($O(\log n)$)

$$\begin{cases} T(0) = a \\ T(n) = b + T(n/2) \end{cases}$$

Moltiplicazione veloce ($O(n^{\log_2 3})$)

$$\begin{cases} T(1) = d \\ T(n) = bn + 3T(n/2) \end{cases}$$

A.3 [Osservazione per esercizi] Somma dei primi n numeri

Negli esercizi possono capitare cicli del seguente tipo:

```
int a = 0;

for(int i = 0; i <= n; i++)
        a += i;
```

In questo caso abbiamo la somma dei primi n numeri. La cosa è rilevante se dobbiamo calcolare il risultato di una funzione. Troviamo che

$$O\left(\frac{n(n+1)}{2}\right) = O\left(\frac{n^2+n}{2}\right) = O(n^2)$$

A.4 [Esempio 1] Esercizio sulla complessità

Calcolare la complessità dell'istruzione

```
for (int i=0; i<=f(t); i++) cout << g(t->left) + g(t->right);
```

in funzione del numero di nodi di t. Indicare per esteso le relazioni di ricorrenza.
Supporre che t sia un albero binario bilanciato e le funzioni **f** e **g** siano definite come segue:

```
int f(Node* t) {
   if (! t)
     return 1;
  return g(t->left) + f(t->left) +
f(t->right);
}
```

```
int g(Node * t) {
  if (!t) return 0;
  int a = g(t->left);
  int b = g(t->right);
  return 1 +2a + 2b;
}
```

Spiegazione

- Troviamo le relazioni di ricorrenza di g (necessaria per trovare la relazione di f):
  - Complessità:
    - Il caso base (0 elementi, albero vuoto) ha complessità costante $O(1)$
    - Ogni volta ho due chiamate di funzione: una per il sottoalbero sinistro e una per il sottoalbero destro. Considerando che l'albero è bilanciato ogni chiamata coinvolge metà degli n elementi.
    - Oltre a casi base e chiamate ricorsive non ho altre istruzioni rilevanti che potrebbero aumentare la complessità
    - Ottengo

      $$\begin{cases} T_g(0) = d \\ T_g(n) = c + 2T_g(n/2) \end{cases}$$

      Segue $T_g \in O(n)$ con le regole che conosciamo.

- Risultato:
  - Con il caso base (0 elementi, albero vuoto) ho risultato 0.
  - Analizzo il return: ogni volta restituisco una somma dove sono coinvolti i risultati di due chiamate ricorsive. Entrambi i risultati, nella somma restituita, sono moltiplicati per due. Entrambi i risultati sono ottenuti coinvolgendo metà degli n elementi.
  - Ottengo
$$\begin{cases} R_g(0) = 0 \\ R_g(n) = c + 4R_g(n/2) \end{cases}$$
Segue $R_g \in O(n^2)$ con le regole che conosciamo.

- Troviamo le relazioni di ricorrenza di f (adesso abbiamo i dati di g che ci servono):
  - Complessità:
    - Il caso base (0 elementi, albero vuoto) ha complessità costante $O(1)$
    - Ho due chiamate ricorsive, ciascuna per un sottoalbero. Considerando che l'albero è bilanciato ciascuna delle chiamate coinvolge metà degli n elementi.
    - Oltre alle due richiamate ricorsive ho una chiamata della funzione f. Sappiamo che $T_g \in O(n)$.
    - Ottengo
$$\begin{cases} T_f(0) = d \\ T_f(n) = cn + 2T_f(n/2) \end{cases}$$
Segue $T_f \in O(n \log n)$ con le regole che conosciamo.
  - Risultato:
    - Con il caso base (0 elementi, albero vuoto) restituisco 1
    - Analizzo il return: ho due risultati ottenuti da chiamate ricorsive. Entrambi i risultati, nella somma restituita, sono ottenuti coinvolgendo metà degli n elementi.
    - Nella somma ho un ulteriore elemento: un risultato ottenuto dalla funzione g. Abbiamo visto che $R_g \in O(n^2)$.
    - Ottengo:
$$\begin{cases} R_f(0) = 1 \\ R_f(n) = cn^2 + 2R_f(n/2) \end{cases}$$
Segue $R_f(n) \in O(n^2)$ con le regole che conosciamo.

- Riprendiamo il frammento iniziale:
  - Il numero di iterazioni del for è $O(n^2)$
  - La complessità di una singola iterazione è:
$$O(n) + O(n) + O(n \log n) = O(n \log n)$$
Ho considerato anche la complessità di f (ogni volta che verifico se la condizione del for è true eseguo la funzione f)
  - Complessità del for:
$$O(n \log n) * O(n^2) = O(n^3 \log n)$$

A.5 [Esempio 2] Esercizio sulla complessità

Calcolare la complessità del blocco (indicando le relazioni di ricorrenza di tempo e risultato per ogni funzione) in funzione di n:

```
{
  int a = 0;
  for (int i=0; i <= g(n)/n; i++)
     a += f(n);
}
```

con le funzioni **f** e **g** definite come segue:

```
int f(int x) {
   if (x<=1)
     return 1;
   cout << f(x/3) + f(x/3);
   return f(x/3) + 1;
}
```

```
int g(int x) {
  int a=0;
  for (int i=0; i <= f(x); i++)
     a++;
  for (i=0; i <= 2*f(x); i++)
     a+=i;
  return a;
}
```

Indicare per esteso le relazioni di ricorrenza e, per ogni comando ripetitivo, il numero di iterazioni e la complessità dell'iterazione singola.

Spiegazione

- Troviamo le relazioni di ricorrenza di f (necessario dopo per g):
  - Complessità:
    - Il caso base si ha con $n \leq 1$: tenendo conto che i valori possibili sono maggiori o uguali a 0 individuo che il caso base si ha con $0, 1$.
    - Il cout ha complessità $O(1)$
    - Ho tre chiamate ricorsive, ciascuna con argomento $x/3$
    - Ottengo
      $$\begin{cases} T_f(0) = T_f(1) = d \\ T_f(n) = c + 3T_f(n/3) \end{cases}$$
      Segue $T_f(n) \in O(n)$ con le regole che conosciamo.
  - Risultato:
    - Il caso base si ha con $n \leq 1$: il valore restituito è 1. Ciò avviene coi valori $0, 1$
    - Analizzo il return: ogni volta restituisco la somma tra 1 e il risultato di una chiamata ricorsiva (con argomento $n/3$)
    - Ottengo
      $$\begin{cases} R_f(0) = R_f(1) = 1 \\ R_f(n) = 1 + R_f(n/3) \end{cases}$$
- Troviamo complessità e risultato di g (funzione non ricorsiva):
  - Complessità:
    - Gli unici elementi rilevanti sono i due for.
    - Il post-incremento ha complessità $O(1)$, ma devo tenere conto che ogni volta viene chiamata la funzione f, con complessità $O(n)$. Il numero di iterazioni è il risultato di f, cioè $O(\log n)$. Segue che la prima parte avrà complessità $O(n \log n)$.
    - Il secondo for consiste nella somma dei primi $2 \log n$ numeri[1]. La somma ha complessità $O(1)$, ma anche qua f viene eseguita ogni volta con complessità $O(n)$. Il numero di iterazioni è $O(2 \log n) = O(\log n)$. Segue che il secondo for avrà complessità $O(n \log n)$.

[1] Non sono d'accordo che il for consiste in 2 volte la somma dei primi $\log n$ come scritto dalla prof.

$$2\frac{\log n(\log n + 1)}{2} = \frac{2 \log n(2 \log n + 1)}{2}$$

I membri non diventano uguali semplificando. Il risultato di R_g, comunque sia, non cambia.

* Ottengo

$$T_g \in O(n \log n)$$

– Risultato:

* La funzione restituisce l'intero a, inizialmente uguale a 0.
* Nel primo for incremento a $\log n$ volte.
* Nel secondo for sommo a quanto già trovato i primi $2 \log n$ numeri.
* Ottengo

$$O(\log n)+\left(\frac{2\log n(2\log n+1)}{2}\right) = O(\log n)+O\left(\frac{4(\log n)^2+2\log n}{2}\right) = O(\log n)+O(2(\log n)^2+\log n) = O((\log n)^2)$$

Quindi $R_g \in O((\log n)^2)$

- Concludiamo col for del blocco:
  – Il numero di iterazioni consiste nel risultato di g fratto n, cioè $\frac{g(n)}{n} = \frac{(\log n)^2}{n}$
  – La singola iterazione ha la seguente complessità

$$O(n) + O(n \log n) = O(n \log n)$$

considero sia la somma ad a che l'esecuzione, ogni volta della funzione g

  – La complessità totale sarà

$$O(n \log n) * O\left(\frac{(\log n)^2}{n}\right) = O((\log n)^3)$$

A.6 [Esempio 3] Esercizio sulla complessità

Calcolare la complessità del blocco (indicando le relazioni di ricorrenza di tempo e risultato per ogni funzione) in funzione del numero di nodi dell'albero t,

a) supponendo che t sia quasi bilanciato.
b) supponendo che t sia completamente sbilanciato

Indicare per esteso numero di iterazioni e complessità di ogni iterazione per i comandi ripetitivi.

```
{
  int a = 0;
  for (int i=0; i <= g(t)*f(t); i++)
     a += Nodes(t);
}
```

Le funzioni **f** e **g** sono definite come segue:

```
int f(Node* tree) {
  if (!tree) return 1;
  int x=0;
  for (int i=1;i<= Nodes(tree);i++)
  x+=i;
  int b = 4*f(tree->left);
  return x+b;
}
```

```
int g(Node * tree) {
  if (!tree) return 1;
  int a=0;
  for (int i=1;i<=f(tree)/Nodes(tree);i++)
    { a++;
      cout << a;
    }
  return 3;
  }
```

Spiegazione

- Consideriamo, in entrambi i casi, una funzione *Nodes* con complessità lineare. Tutte le volte scorro l'albero per individuare il numero dei suoi nodi.
- Albero t quasi bilanciato:
  – Un albero quasi bilanciato presenta le proprietà di un albero bilanciato fino al penultimo livello: abbiamo una situazione di sostanziale equilibrio che ci permette di gestire questi alberi come se fossero bilanciati.
  – Troviamo le relazioni di ricorrenza per f:

    * Complessità:
        · Il caso base ha complessità $O(1)$
        · L'iterazione del for ha complessità $O(n)$: l'istruzione di somma è $O(1)$ ma devo tener conto della chiamata della funzione *Nodes*. Il numero di iterazioni è il risultato di *Nodes*, cioè $O(n)$. Segue che il for avrà complessità $O(n^2)$.
        · Ho una chiamata ricorsiva di f, che coinvolge la metà degli n elementi.
        · Ottengo
        $$\begin{cases} T_f(0) = d \\ T_f(n) = dn^2 + T_f(n/2) \end{cases}$$
        Segue $T_f(n) \in O(n^2)$ con le regole che conosciamo.
    * Risultato:
        · Il caso base restituisce 1
        · Il for aggiorna il valore di x, che inizialmente è uguale a 0. Sommo i primi n numeri (nella condizione del for abbiamo *Nodes*, che restituisce n).
        $$O\left(\frac{n(n+1)}{2}\right) = O\left(\frac{n^2+n}{2}\right) = O(n^2)$$
        · Nel return ho anche b, che consiste nel risultato di una chiamata ricorsiva di f moltiplicato per quattro. Coinvolgo, nel calcolo del risultato, la metà di n elementi.
        · Ottengo
        $$\begin{cases} R_f(0) = 1 \\ R_f(n) = dn^2 + 4R_f(n/2) \end{cases}$$
        Segue $R_f(n) \in O(n^2 \log n)$ con le regole che conosciamo.
- Troviamo complessità e risultato di g (funzione non ricorsiva):
    * Complessità:
        · Se l'albero è vuoto ho complessità $O(1)$
        · L'unico elemento che determina la complessità è il for.
        · Le istruzioni nel blocco del for hanno complessità $O(1)$. Nel calcolo della complessità dell'iterazione devo tenere conto anche delle chiamate di f e *Nodes* nella condizione del for. Ottengo
        $$O(1) + O(n) + O(n^2) = O(n^2)$$
        · Il numero di iterazioni del for si ottiene dividendo il risultato di f per il risultato di *Nodes*, cioè
        $$O\left(\frac{n^2 \log n}{n}\right) = O(n \log n)$$
        · Segue la complessità del for
        $$O(n \log n) * O(n^2) = O(n^3 \log n)$$
        quindi $T_g(n) \in O(n^3 \log n)$
    * Risultato:
        · Il risultato del caso base è 1
        · Il for aggiorna il valore di a, tuttavia è assolutamente ininfluente nel valore restituito dalla funzione.
        · La funzione restituisce ogni volta 3.
        · Ottengo $R_g \in O(1)$
- Concludo col for del blocco:
    * Il numero di iterazioni del blocco si ottiene moltiplicando il risultato di f per il risultato di g
    $$O(1) * O(n^2 \log n) = O(n^2 \log n)$$

    - La complessità dell'iterazione è data dalla chiamata di *Nodes* nel blocco e dalle chiamate di g ed f nella condizione
    $$O(n) + O(n^2) + O(n^3 \log n) = O(n^3 \log n)$$
    - La complessità del for si ottiene moltiplicando quanto detto prima
    $$O(n^2 \log n) * O(n^3 \log n) = O(n^5 (\log n)^2)$$

- Albero t completamente sbilanciato:
  - L'albero assume i tratti di una lista e la complessità della visita viene degradata diventando lineare. I ragionamenti sono simili a quelli fatti prima: cambiano gli argomenti delle chiamate ricorsive e i risultati delle funzioni.
  - In riferimento a f: abbiamo le stesse relazioni di ricorrenza, cambiano solo gli argomenti ($n-1$ invece di $n/2$)
    - Complessità:
    $$\begin{cases} T_f(0) = d \\ T_f(n) = dn^2 + T_f(n-1) \end{cases}$$
    Segue $T_f(n) \in O(n^3)$ con le regole che conosciamo.
    - Risultato:
    $$\begin{cases} R_f(0) = 1 \\ R_f(n) = dn^2 + 4R_f(n-1) \end{cases}$$
    Qui addirittura abbiamo $R_f(n) \in 4^n$.
  - In riferimento a g:
    - Complessità:
      - Cambia il numero di iterazioni del for (è cambiato il risultato della funzione f): $O(4^n/n)$.
      - La complessità dell'iterazione sarà la seguente
      $$O(1) + O(n^3) + O(n) = O(n^3)$$
      - Segue la complessità del for, che è R_g
      $$O(n^3) * O(4^n/n) = O(n^2 4^n)$$
    - Risultato: $O(1)$ come prima.
  - Concludiamo col for:
    - Il numero di iterazioni del for è il prodotto tra i risultati di g ed f
    $$O(4^n) * O(1) = O(4^n)$$
    - La complessità dell'iterazione sarà
    $$O(n) + O(n^3) + O(n^2 4^n) = O(n^2 4^n)$$
    - Segue la complessità del for del blocco
    $$O(4^n) * O(n^2 4^n) = O(n^2 (4^n)^2)$$

A.7 [Esempio 4] Esercizio sulla complessità

Calcolare la complessità in funzione di **n**>0 dell'istruzione

```
y=g(f(n));
```

con le funzioni **f** e **g** definite come segue:

```
int f(int x) {
   if (x<=1) return 1;
   int b=0, i, j, c;
   for (i=1; i<=x; i++) b+=i;
   c = b*b;
   for (j=1; j<=c; j++) b+=j;
   return b + f(x-1);
}
```

```
int g(int x) {
  if (x<=1) return 10;
  int a=0;
  for (int i=0; i<f(x); i++)
      a++;
   return a+2*g(x/2);
}
```

Indicare le eventuali relazioni di ricorrenza e spiegare brevemente il calcolo della complessità dei cicli.

Spiegazione

- Dobbiamo trovare la complessità di una funzione composta, precisamente $y = g(f(n))$. Dovremo considerare la complessità di f e di g, ma anche il risultato di f, che costituirà argomento della funzione g!
- Troviamo le relazioni di ricorrenza per f:
  - Complessità:
    - Abbiamo due for che provocano un incremento della complessità
    - Il primo somma a b, inizialmente uguale a 0, i primi n numeri. Avrò complessità $O(n)$ (numero di iterazioni) e risultato $O\left(\frac{n(n+1)}{2}\right) = O(n^2)$.
    - SuccessivameNTE moltiplico il risultato con se stesso, ottengo
    $$O(n^2) * O(n^2) = O(n^4)$$
    - Il secondo for somma a b, già uguale alla somma dei primi n numeri, la somma dei primi n^4 numeri. La complessità del for è $O(n^4)$.
    - Ho una chiamata ricorsiva lineare $(n-1)$
    - Ottengo la seguente relazione
    $$\begin{cases} T_f(0) = 0 \\ T_f(n) = n^4 + T_f(n-1) \end{cases}$$
    con le regole che conosciamo otteniamo $O(n^{4+1}) = O(n^5)$
  - Risultato:
    - Considero che restituiamo la somma dei risultati dei due for più il risultato restituito dalla funzione con argomento $n-1$
    - Nel primo for sommiamo i primi n numeri. Come già detto il risultato di questo for è
    $$O\left(\frac{n(n+1)}{2}\right) = O(n^2)$$
    - Nel secondo for sommiamo i primi n^4 numeri, quindi otteniamo come risultato
    $$O\left(\frac{n^4(n^4+1)}{2}\right) = O(n^8)$$
    - Il risultato dei for sarà $O(n^2) + O(n^8) = O(n^8)$
    - Ottengo la seguente relazione
    $$\begin{cases} R_f(0) = 1 \\ R_f(n) = n^8 + R_f(n-1) \end{cases}$$
    con le regole che conosciamo otteniamo $O(n^{8+1}) = O(n^9)$
- Troviamo le relazioni di ricorrenza per g:

  - Complessità:
    * Abbiamo un for che incrementa la complessità.
    * Il body del for ha complessità costante $O(1)$, tuttavia nella condizione troviamo una chiamata alla funzione f.
    * Sappiamo che la funzione f ha complessità $O(n^5)$ (viene eseguita ogni volta che si verifica la condizione) e restituisce risultato $O(n^9)$.
    * Segue che avrò $O(n^9)$ iterazioni, quindi il for avrà complessità
    $$O(n^9) * O(n^5) = O(n^{14})$$
    * Ottengo la seguente relazione
    $$\begin{cases} T_g(0) = a \\ T_g(n) = n^{14} + T_g(n/2) \end{cases}$$
    con le regole che conosciamo otteniamo $O(n^{14})$
  - Il risultato non ci interessa

- Concludiamo:
  - Sommo la complessità di f e quella di g, ponendo come argomento di g n^9, cioè il risultato di f.
  $$T_f(n) + T_g(n^9) = O(n^5) + O(n^{9^{14}}) = O(n^{126})$$