

PAC teoria

Silviu Filote

November 7, 2024

Contents

1	Introduzione	6
1.1	Algoritmi	6
1.2	Ciclo di sviluppo di codice algoritmico	6
2	Modelli di calcolo e metodologie di analisi	7
2.1	Notazione asintotica	7
2.2	Proprietá	8
2.3	Regole di semplificazione	8
2.4	Notazione asintotica versus numeri naturali	8
2.5	Teorema dei limiti	8
2.6	Terminologia	8
2.7	Complessitá di un problema	8
2.8	Metodi di analisi	8
3	Sommatorie	9
4	Strutture dati elementari	10
4.1	Tecniche di rappresentazione dei dati	10
4.2	Il tipo di dato Dizionario	10
4.3	Il tipo di dato Pila	11
4.4	Il tipo di dato Coda	11
4.5	Alberi	12
4.6	Alberi k-ari completi	12
4.7	Rappresentazioni collegate di alberi	12
4.8	Alberi binari	13
4.9	Algoritmi di visita	13
5	Modelli di calcolo e metodologie di analisi	14
5.1	Analisi di algoritmi ricorsivi	14
5.2	Equazioni di ricorrenza	14
5.3	Motodologie	14

6 Binary search tree (BST)	15
6.1 search(chiave k) → elem	15
6.2 Ricerca del massimo e del minimo	16
6.3 Predecessore e successore	16
6.4 insert(elem e, chiave k)	17
6.5 delete(chiave k)	17
6.6 Visita simmetrica di un BST	18
6.7 Costo delle operazioni dipende da h	18
6.8 Alberi binari bilanciati	18
6.9 Classe AlberoAVL	19
7 B-alberi	19
7.1 Costi operazioni	19
7.2 Definizione di B-albero	19
7.3 Altezza di un B-albero	20
7.4 Operazioni elementari	20
7.5 Esempi inserimenti	22
8 Tabelle hash	23
8.1 Tabelle ad accesso diretto	23
8.2 Funzioni hash perfette	23
8.3 Uniformità delle funzioni hash	23
8.4 Sicurezza di funzioni hash crittografiche	24
8.5 Risoluzione delle collisioni	24
8.6 Sequenze di scansione	25
9 Ordinamento per confronto	26
9.1 Algoritmi e tempi tipici	26
9.2 Alberi di decisione	26
9.3 Ordinamenti quadratici	27
9.3.1 InsertionSort	27
9.3.2 SelectionSort	27
9.3.3 BubbleSort	28
9.4 Ordinamenti ottimi	28
9.4.1 MergeSort	28
9.4.2 QuickSort	29
9.5 Riepilogo	30
10 Ordinamento lineare	30
10.1 IntegerSort	30
10.2 BucketSort	31
10.3 RadixSort	31
11 Programmazione dinamica	32
11.1 Distanza fra due stringhe	32

12 Strategia Greedy	33
12.1 Elementi della strategia golosa	33
12.2 Un problema di sequenziamento	33
12.3 Un algoritmo goloso	33
12.4 Il problema del distributore automatico di resto	34
12.5 Riepilogo	34
13 Grafi e visite di grafi	34
13.1 Terminologia	34
13.2 Componenti connesse di un grafo non orientato	35
13.3 Componenti fortemente connesse di un grafo orientato	35
13.4 Grafi, alberi e DAG	35
13.5 Grafi non orientati	35
13.6 Prestazioni della lista di archi	35
13.7 Prestazioni delle liste di adiacenza e di incidenza	35
13.8 Prestazioni della matrice di adiacenza	36
13.9 Prestazioni della matrice di incidenza	36
13.10 Grafi orientati	36
13.11 Visite di grafi	36
13.12 Visita in ampiezza	36
13.13 Visita in profondità	37
14 Cammini minimi	39
14.1 Rilassamento continuo	39
14.2 Proprietà del rilassamento	40
14.3 Algoritmo di Dijkstra	40
14.4 In letteratura	41
15 Esercizio 1	43
16 Esercizio 2	48
16.1 VisitaDFS pre-order o ordine anticipato	50
16.2 VisitaDFS in-order o simmetrico	50
16.3 VisitaDFS post-order	51
16.4 Visita BFS	51
16.5 Search con chiave $O(h)$	51
16.6 Ricerca il massimo	51
16.7 Predecessore	52
16.8 Successore	52
16.9 Insert di un nodo	53
16.10 Delete di un nodo	53
16.11 Costi esecuzione	54
17 Esercizio 3	55
17.1 Algoritmi greedy	55
17.2 Ordinamenti	55
17.3 Programmazione dinamica	58
17.4 Esercizio treno	61
17.5 Grafi	62

18 Esercizio 4

63

18.1 Architecture validation	63
18.2 Tipi di architettura	64

Riassunto

Dizionario: implementazioni

Tempo richiesto dall'operazione piú costosa:

<i>Liste</i>	$O(n)$
<i>Alberi di ricerca non bilanciati</i>	$O(n)$
<i>Alberi di ricerca bilanciati</i>	$O(\log n)$
<i>Tabelle hash</i>	$O(1)$

1 Introduzione

1.1 Algoritmi

Gli algoritmi forniscono la strategia risolutiva per giungere alla soluzione di un dato problema computazionale (o di calcolo). Classificazione dei problemi:

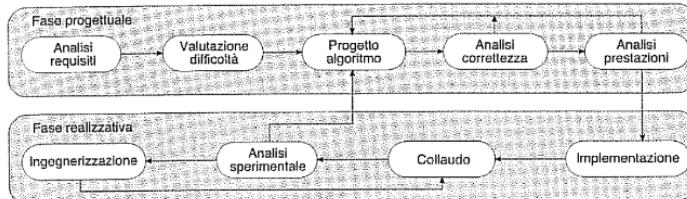
- **problemi decidibili**, è un problema per il quale esiste un algoritmo che può determinare in modo efficace (cioè, in un tempo finito) se la risposta a una domanda specifica è sì o no
 - **problemi trattabili (costo polinomiale)**
 - **problemi intrattabili (costo esponenziale)**
- **problemi non decidibili**, è un problema per il quale non esiste alcun algoritmo che possa determinare in modo generale e in un tempo finito se la risposta a una domanda specifica è sì o no

Def: una sequenza finita di operazioni non ambigue ed effettivamente calcolabili che, una volta eseguite, producono un risultato in una quantità finita di tempo

Proprietà fondamentali di un algoritmo:

- **Finitezza:** la sequenza di istruzioni deve essere finita
- **Non ambiguità:** le istruzioni devono essere espresse in modo non ambiguo
- **Realizzabilità:** le istruzioni devono essere eseguibili materialmente
- **L'efficacia** indica la capacità di raggiungere l'obiettivo prefissato, mentre **l'efficienza** valuta l'abilità di farlo impiegando le risorse minime indispensabili

1.2 Ciclo di sviluppo di codice algoritmico



Modello ciclico a due fasi: fase progettuale e fase realizzativa

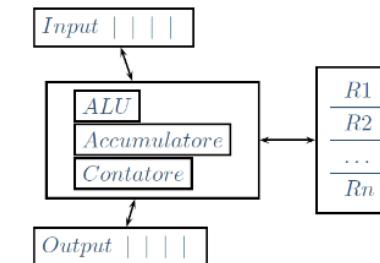
- **Analisi dei requisiti**, definire il problema di calcolo che si intende risolvere identificando: i **requisiti dei dati in ingresso** e i **requisiti dei dati in uscita** prodotti dall'algoritmo, identificare una metodologia di progettazione esempio **divide-et-impera** ⇒ un problema complesso può essere scomposto in un certo numero di sotto-problemi risolvibili separatamente
- **Valutazione delle difficoltà**, idealmente vorremmo algoritmi che minimizzino le risorse spazio/tempo utilizzate ⇒ **Complessità (spaziale o temporale) dell'algoritmo - SC e TC**, tuttavia per ogni problema computazionale esistono dei limiti inferiori alle quantità di risorse di calcolo necessarie alla sua risoluzione ⇒ **Complessità del problema**
- **Progetto di un algoritmo risolutivo**, richiede creatività ed esperienza, per mantenere il massimo grado di generalità descriveremo gli algoritmi in **pseudocodice**
- **Analisi di correttezza**, vogliamo algoritmi che producano correttamente il risultato desiderato e che siano efficienti in TC e SC, verifica formale della correttezza:

requisiti sull'output siano soddisfatti, dimostrare che vengano mantenute determinate proprietà deurante l'esecuzione del algoritmo; Ove l'analisi evidenzi discrepanze, è necessario ritornare alla fase di progettazione

- **Analisi delle prestazioni**, assumiamo di utilizzare Mono-Processore + RAM (Random Access Memory) assenza di concorrenza e parallelismo. **Analisi teorica** condotta fissando:

- **Modello di calcolo** e tipicamente la **macchina a registri RAM (Random Access Machine)**, si ispira all'architettura di Von Neumann e alla macchina di Turing, l'accesso diretto o casuale alla memoria è non sequenziale
- **Modello di costo teorico:** modello a costo **uniforme** (indipendente dalla dimensione degli operandi coinvolti) e modello a costo **logaritmico** (dipende dalla dimensione degli operandi coinvolti)

Macchine a registri sono caratterizzate da: nastri input e output, numero arbitrario di celle i di memoria detti **registri** contenenti un intero arbitrario $R[i]$ (intero o reale di dimensione arbitraria), ALU e registri speciali: **contatore** l'indirizzo istruzione successiva, **accumulatore R0** l'operando su cui agisce l'istruzione corrente; Passo di calcolo: istruzione di I/O su nastro, istruzione di I/O da/in memoria, istruzione aritmetica o logica ⇒ Il modello assume che **l'algoritmo sia sequenziale e deterministico**



Modello a costo uniforme: ogni istruzione elementare della macchina a registri costa una unità di tempo, ciascuna locazione di memoria compreso l'accumulatore richiede una unità di spazio, **pregio**: approccio molto semplice; **difetto**: analisi non realistica

Modello a costo logaritmico: la complessità del calcolo è proporzionale al numero di bit dell'operando e l'accesso è proporzionale al numero di bit dell'indirizzo, l'elaborazione di un intero n ha costo $O(\log n)$, l'accesso ad un registro i ha costo $O(\log i)$, si applica la medesima filosofia per l'occupazione di spazio, **pregio**: la valutazione dei costi è più realistica dal punto di vista asintotico (ovvero per operandi molto grandi, o per memoria molto grande); **difetto**: analisi più laboriosa

Il nostro modello di costo: limitiamo a $c \cdot \log n$ bit la dimensione dei numeri interi rappresentabili ove c è una **costante** e n è la dimensione dell'**input del problema**, non ammettiamo operazioni sui numeri in virgola mobile ⇒ dunque li algoritmi che presentiamo d'ora in poi sono implementabili su una macchina a registri con interi di dimensione $\leq n^c$

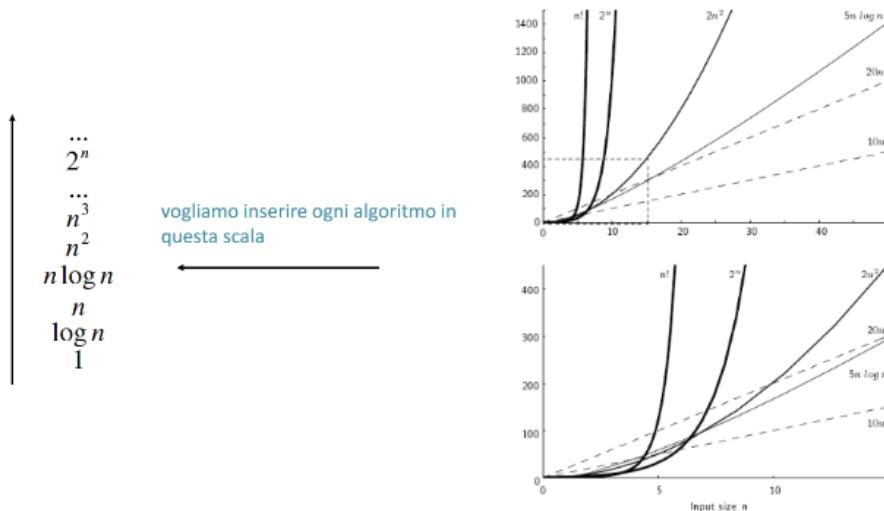
Tempo di esecuzione: calcoliamo il numero di linee di codice mandate in esecuzione, misura indipendente dalla piattaforma utilizzata ma approssimativa, meglio considerare il numero di **passi elementari** eseguiti dall'algoritmo al variare

della dimensione n dell'input: funzione $T(n) \Rightarrow$ ci interessa soprattutto il comportamento di $T(n)$ per valori grandi di n (comportamento asintotico)

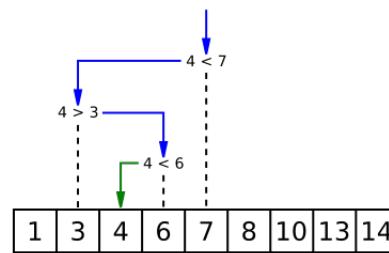
Nozioni teoriche:

- **Calcolabilità:** nozioni di algoritmo e di problema non decidibile
- **Complessità:** nozione di algoritmo efficiente e di problema intrattabile
- **Complessità di un algoritmo:** misura del numero di passi elementari che si devono eseguire per risolvere il problema in funzione della *taglia* (*dimensione dell'input*) del problema
- **Complessità di un problema:** complessità del migliore algoritmo che lo risolve in funzione della *taglia*
- **Taglia di un problema:** misura della dimensione dell'input, dipende dalla rappresentazione dei dati (struttura dati)

Paragonare tra loro algoritmi: *scala complessità e growth rate*



Notazione asintotica: ignorare le costanti moltiplicative \Rightarrow a questo scopo utilizzeremo la **notazione asintotica O** , diremo che $f(n) = O(g(n))$ se $f(n) \leq c \cdot g(n)$ per qualche costante c ed n abbastanza grande



- **Implementazione in un linguaggio di programmazione:** comporta molti dettagli implementativi non del tutto specificati nel pseudocodice
- **Collaudo e analisi sperimentale:** una data implementazione va testata su data set reali per identificare eventuali errori implementativi, una data implementazione

può risultare meno efficiente rispetto ai risultati dell'analisi teorica \Rightarrow occorre una messa a punto e re-ingegnerizzazione opportuna che tenga conto delle caratteristiche del linguaggio e delle piattaforme di esecuzione, tale analisi sperimentale ci aiuta anche ad identificare le **costanti nascoste**

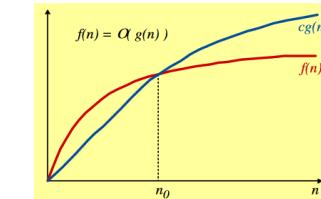
- **Ingengerizzazione:** per ottimizzare/fare refactoring a livello di codice

2 Modelli di calcolo e metodologie di analisi

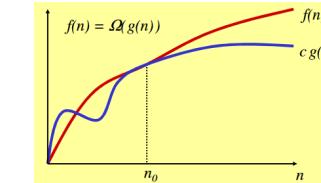
2.1 Notazione asintotica

- $f(n)$ è il tempo di esecuzione / occupazione di memoria di un algoritmo su input di dimensione n , la notazione asintotica è un'astrazione utile per descrivere l'ordine di grandezza di $f(n)$ ignorando i dettagli non influenti come costanti moltiplicative e termini di ordine inferiore
- O e Ω forniscono un limite lasso rispettivamente per i limiti superiore ed inferiore, mentre Θ fornisce un limite stretto, in alcuni contesti è difficile trovare un limite stretto per l'andamento delle f per cui ci si accontenta di un limite meno preciso

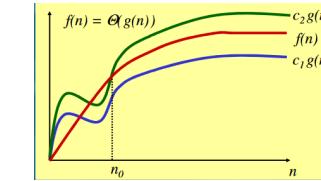
Notazione asintotica O : $f(n) = O(g(n))$ se \exists due costanti $c > 0$ e $n_0 \geq 0$ tali che $f(n) \leq c \cdot g(n)$ per ogni $n \geq n_0 \Rightarrow$ **estremo superiore**



Notazione asintotica Ω : $f(n) = \Omega(g(n))$ se \exists due costanti $c > 0$ e $n_0 \geq 0$ tali che $f(n) \geq c \cdot g(n)$ per ogni $n \geq n_0 \Rightarrow$ **estremo inferiore**



Notazione asintotica Θ : $f(n) = \Theta(g(n))$ se \exists tre costanti $c_1, c_2 > 0$ e $n_0 \geq 0$ tali che $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ per ogni $n \geq n_0 \Rightarrow$ **compresa tra**



- **Abuso di notazione:** $f(n) = O(g(n))$ invece di $f(n) \in O(g(n))$, si legge $f(n)$ è *o grande* di $g(n)$, $f(n) = O(g(n))$ è un limite superiore asintotico per la complessità tempo di *Alg* (algoritmo). **Teorema:** date due funzione $f(n)$ e $g(n)$, $f(n) \in \Theta(g(n))$ se e solo se $f(n) \in O(g(n))$ e $f(n) \in \Omega(g(n))$

2.2 Proprietà

Transitiva

$$f(n) = \Theta(g(n)) \text{ e } g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$

$$f(n) = O(g(n)) \text{ e } g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

$$f(n) = \Omega(g(n)) \text{ e } g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$

Riflessiva

$$f(n) = \Theta(f(n))$$

$$f(n) = O(f(n))$$

$$f(n) = \Omega(f(n))$$

Simmetria

$$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$$

Simmetria transposta

$$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$$

2.3 Regole di semplificazione

- Se $f(n) = O(k \cdot g(n))$ con $k > 0$ costante, allora $f(n) = O(g(n))$
- **Somma:** se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$, allora $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n))) \Rightarrow$ blocchi sequenziali di istruzioni, rami if-then-else, switch
- **Prodotto:** se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$, allora $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n)) \Rightarrow$ cicli for, while, do-while

Semplifica

$$f(n) = O(k \cdot g(n)) = O(g(n))$$

Somma

$$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$$

Prodotto

$$f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$

2.4 Notazione asintotica versus numeri naturali

$$f(n) = O(g(n)) \approx a \leq b$$

$$f(n) = \Omega(g(n)) \approx a \geq b$$

$$f(n) = \Theta(g(n)) \approx a = b$$

Per ogni coppia di numeri reali a e b , deve valere esattamente una delle seguenti espressioni:

$$a < b, \quad a = b, \quad a > b$$

Sebbene qualunque coppia di numeri possa essere confrontata, non tutte le funzioni sono asintoticamente confrontabili!

$$f(n) = n, \quad g(n) = n^{1+\sin n}$$

2.5 Teorema dei limiti

Se $f(n)$ e $g(n)$ sono asintoticamente positive, allora possibile dedurre che:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = O(g(n)), \text{ con } f(n) \neq \Omega(g(n)), f(n) \neq \Theta(g(n))$$

$$f(n) = O(g(n)) \Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \text{ (se esiste)} < \infty$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \Omega(g(n)), \text{ con } f(n) \neq O(g(n)), f(n) \neq \Theta(g(n))$$

$$f(n) = \Omega(g(n)) \Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \text{ (se esiste)} > 0$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow f(n) = \Theta(g(n)), g(n) = \Theta(f(n))$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \emptyset \Rightarrow f(n) \text{ e } g(n) \text{ non sono asintoticamente confrontabili}$$

2.6 Terminologia

$$O(1)$$

costante, non dipende dalla dimensione dei dati

$$O(\log n)$$

logaritmica, in generale $\log^K n$ con $K \geq 1$

$$O(n)$$

lineare

$$O(n \cdot \log n)$$

pseudolineare

$$O(n^2)$$

quadratica, n^k polinomiale con $k > 1$

$$O(2^n)$$

esponenziale, in generale k^n con $k > 1$

2.7 Complessità di un problema

Complessità intrinseca di un problema \neq Complessità computazionale di un algoritmo

Def: un problema computazionale ha complessità $O(f(n))$ **upper bound** se esiste un algoritmo per la sua risoluzione con delimitazione superiore $O(f(n))$

Def: un problema computazionale ha complessità $\Omega(f(n))$ **lower bound** se tutti gli algoritmi per la sua risoluzione hanno delimitazione inferiore $\Omega(f(n))$

Se dimostro che un problema ha delimitazione inferiore $\Omega(f(n))$ e trovo un algoritmo avente delimitazione superiore $O(f(n))$ allora, a meno di costanti, ho un algoritmo ottimale per risolvere il problema

2.8 Metodi di analisi

Misureremo le risorse di calcolo usate da un algoritmo (tempo di esecuzione/ occupazione di memoria) in funzione della dimensione n delle istanze di input, istanze diverse a parità di dimensione potrebbero però richiedere risorse diverse, distinguiamo quindi ulteriormente tra analisi caso peggiore migliore e medio. Sia $\text{tempo}(I)$ il tempo di esecuzione di un algoritmo sull'istanza I , sia $P(I)$ la probabilità di avere in ingresso un'istanza I

Caso peggiore

$$T_{\text{worst}}(n) = \max_{\text{istanze } I} \{\text{tempo}(I)\}$$

Caso migliore

$$T_{\text{best}}(n) = \min_{\text{istanze } I} \{\text{tempo}(I)\}$$

Caso medio

$$T_{\text{avg}}(n) = \sum_{\text{istanze } I} \{P(I) \cdot \text{tempo}(I)\}$$

3 Sommatorie

<i>ceiling:</i> [2, 3]	3
<i>floor:</i> [1, 7]	1

Quando un algoritmo contiene un costrutto di controllo iterativo come un *ciclo while o for*, il suo tempo di esecuzione può essere espresso come la somma dei tempi impiegati per ogni esecuzione del corpo del ciclo

Data una sequenza di numeri a_1, \dots, a_n la **somma finita** può essere scritta nel seguente modo:

$$\sum_{k=1}^n a_k$$

Data una sequenza di numeri a_k la **somma infinita** può essere scritta nel seguente modo:

$$\sum_{k=1}^{\infty} a_k = \lim_{n \rightarrow \infty} \sum_{k=1}^n a_k$$

Se il limite non esiste la serie diverge, altrimenti converge

4 Strutture dati elementari

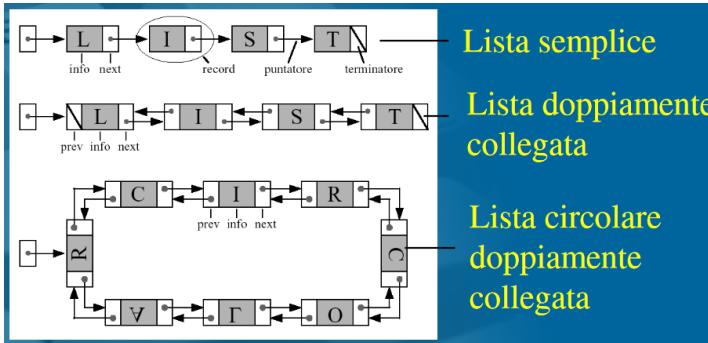
Tipo di dato (Abstract Data Type): cosa? Specifica delle operazioni di interesse su una collezione di oggetti (es. inserisci, cancella, cerca), in java un'interfaccia

Un **Abstract Data Type (ADT)** è un concetto in informatica che definisce un tipo di dato in termini delle operazioni che possono essere eseguite su di esso, senza specificare come queste operazioni vengono implementate. In altre parole, un ADT descrive cosa può fare un tipo di dato, ma non come lo fa.

Struttura dati: come? Organizzazione dei dati che permette di supportare le operazioni di un tipo di dato usando meno risorse di calcolo possibile

4.1 Tecniche di rappresentazione dei dati

- **Rappresentazioni indicizzate:** i dati sono contenuti in array, **pro:** accesso diretto ai dati mediante indici; **contro:** dimensione fissa (riallocazione array richiede tempo lineare)
- **Rappresentazione collegate:** i dati sono contenuti in record collegati fra loro mediante puntatori, **pro:** dimensione variabile (aggiunta e rimozione record in tempo costante); **contro:** accesso sequenziale ai dati



4.2 Il tipo di dato Dizionario

tipo Dizionario:

dati:

un insieme S di coppie $(elem, chiave)$.

operazioni:

$insert(elem e, chiave k)$

aggiunge a S una nuova coppia (e, k) .

$delete(chiave k)$

cancella da S la coppia con chiave k .

$search(chiave k) \rightarrow elem$

se la chiave k è presente in S restituisce l'elemento e ad essa associato, e null altrimenti.

PRIMO modo: Object + Comparable

```
public interface Dizionario {  
    public void insert(Object e, Comparable k);  
    public void delete(Comparable k);  
    public Object search(Comparable k);  
};
```

SECONDO modo: tipo generico + Comparable; T :

```
public interface Dizionario  
<E, K extends Comparable <? Super K >> {  
    public void insert(E e, K k);  
    public void delete(K k); // Null if none  
    public E search(K k); // Null if none  
};
```

Dizionario mediante array ordinato

classe ArrayOrdinato implementa Dizionario:

dati: $S(n) = \Theta(n)$
un array S di dimensione n contenente coppie $(elem, chiave)$.

operazioni:

$insert(elem e, chiave k)$ $T(n) = O(n)$
rialloca l'array S aumentandone la dimensione n di uno; cerca il più piccolo indice i tale che $k \leq S[i].chiave$ e pone $S[j] \leftarrow S[j - 1]$ per ogni j da $n - 1$ a $i + 1$; infine, pone $S[i] \leftarrow (e, k)$.

$delete(chiave k)$ $T(n) = O(n)$
trova l'indice i della coppia con chiave k in S e pone $S[j] \leftarrow S[j + 1]$ per ogni j da i a $n - 2$; infine, rialloca l'array S diminuendone la dimensione n di uno.

$search(chiave k) \rightarrow elem$ $T(n) = O(\log n)$
esegue l'algoritmo di ricerca binaria su S per verificare se S contiene la chiave k . Se la ricerca ha successo restituisce l'elemento e associato alla chiave, altrimenti restituisce null.

Dizionario mediante lista circolare

classe StrutturaCollegata implementa Dizionario:

dati: $S(n) = \Theta(n)$
una collezione di n record contenenti ciascuno una quadrupla $(elem, chiave, next, prev)$, dove $next$ e $prev$ sono puntatori al successivo e precedente record nella collezione, rispettivamente. Manteniamo inoltre un puntatore $list$ che contiene l'indirizzo di un record se la collezione non è vuota, e null altrimenti.

operazioni:

$insert(elem e, chiave k)$ $T(n) = O(1)$
viene creato un record p con elemento e e chiave k . Se $list = null$, si effettua $p.next \leftarrow p$, $p.prev \leftarrow p$ e $list \leftarrow p$. Altrimenti, si collega il record p tra $list$ e $list.next$ effettuando $p.next \leftarrow list.next$, $list.next.prev \leftarrow p$, $p.prev \leftarrow list$ e $list.next \leftarrow p$.

$delete(chiave k)$ $T(n) = O(n)$
si trova il record p con chiave k come nella $search$; poi, si effettua $p.prev.next \leftarrow p.next$ e $p.next.prev \leftarrow p.prev$; infine, viene distrutto il record p .

$search(chiave k) \rightarrow elem$ $T(n) = O(n)$
se $list = null$ si restituisce null. Altrimenti, si scandisce la struttura saltando di record in record con $p \leftarrow p.next$ fino a quando non diventa $p = list$, verificando se qualche p ha chiave k . In caso positivo si restituisce l'elemento trovato, e null altrimenti.

4.3 Il tipo di dato Pila

LIFO: Last In, First Out

tipo Pila:

dati:

una sequenza S di n elementi.

operazioni:

`isEmpty() → result`

restituisce `true` se S è vuota, e `false` altrimenti.

`push(elem e)`

aggiunge e come ultimo elemento di S .

`pop() → elem`

toglie da S l'ultimo elemento e lo restituisce.

`top() → elem`

restituisce l'ultimo elemento di S (senza toglierlo da S).

Implementazione di una Pila mediante array: quale estremo dell'array è il top? L'ultimo elemento; Quali sono i costi delle operazioni? $O(1)$

```
private int maxSize;//Max dimensione
private int top; //Indice elemento top
private Object [] listArray
```

Implementazione di una Pila mediante linked list: quali sono i costi delle operazioni? $O(1)$ inserendo e cancellando in testa; Quanto spazio richiede rispetto all'implementazione array-based? Numero effettivo di elementi

```
class LPila implements Pila {
    private Record top; //primo elemento della lista
    private int size;
    ...
}
```

4.4 Il tipo di dato Coda

FIFO: First in, First Out

tipo Coda:

dati:

una sequenza S di n elementi.

operazioni:

`isEmpty() → result`

restituisce `true` se S è vuota, e `false` altrimenti.

`enqueue(elem e)`

aggiunge e come ultimo elemento di S .

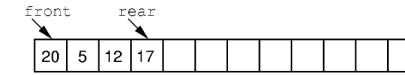
`dequeue() → elem`

toglie da S il primo elemento e lo restituisce.

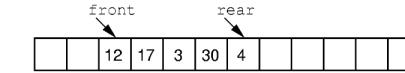
`first() → elem`

restituisce il primo elemento di S (senza toglierlo da S).

Implementazione di una coda mediante array(1)



(a)



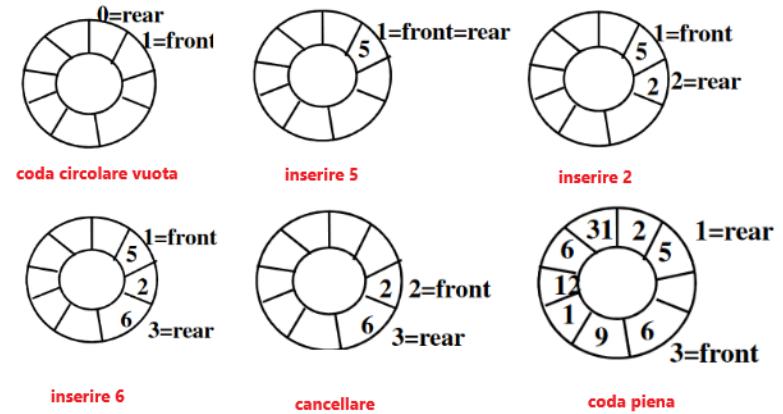
(b)

Implementazione di una coda mediante array circolare (2): le posizioni occupate dalla coda vanno da front a rear, front precede rear in senso orario



Front=2, rear=5 → coda=(A[2],A[3],A[4],A[5])

Front=5, rear=2 → coda=(A[5],...,A[8],A[0],A[1],A[2])



Osservazioni:

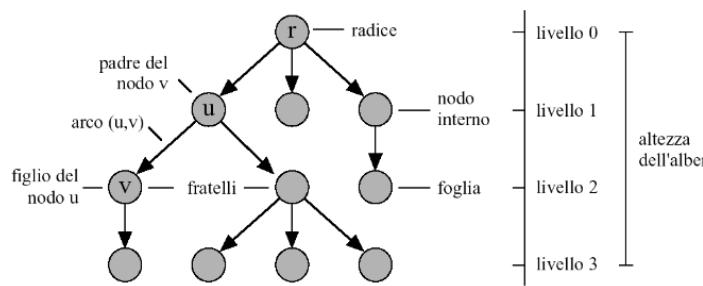
- Coda vuota:** front in posizione immediatamente successiva al rear
- Coda piena:** contiene $n - 1$ elementi (e non n , lascio una cella vuota) front a 2 posizioni successive al rear. Nota se coda contenesse n elementi allora front in posizione immediatamente successiva al rear ⇒ ma si confonderebbe con coda vuota

Metodi:

- clear():** poni $\text{front}=1$, $\text{rear}=0$ (quindi $\text{front}=\text{rear}+1 \% n$)
- isEmpty():** controlla se $\text{front}==(\text{rear}+1) \% n$
- isFull():** risultato del confronto ($\text{front}==(\text{rear}+2) \% n$)
- Enqueue(Q,x):** se piena restituisce FALSE, altrimenti $\text{rear}=(\text{rear}+1) \% n$; $Q[\text{rear}] = x$; return TRUE
- Dequeue(Q,x):** se coda vuota restituisce null, altrimenti $x = Q[\text{front}]$; $\text{front}=(\text{front}+1) \% n$; return x

4.5 Alberi

Un albero (radicato) è una coppia $T = (N, A)$ costituita da un insieme N di nodi e da un insieme A di coppie di nodi, dette archi.



Definizioni:

- **Grado di un nodo:** numero di figli del nodo
- **Cammino:** sequenza di nodi $< n_0, \dots, n_k >$ dove il nodo n_i è il padre del nodo n_{i+1} per $0 \leq i < k$, la lunghezza del cammino è k (numero di archi), dato un nodo esiste un unico cammino dalla radice dell'albero al nodo
- **Livello o profondità di un nodo:** lunghezza del cammino dalla radice al nodo; **definizione ricorsiva:** il livello della radice è 0, il livello di un nodo non radice è $1 +$ il livello del padre
- **Altezza dell'albero:** la lunghezza del più lungo cammino nell'albero (la max profondità a cui si trova una foglia), parte dalla radice e termina in una foglia

tipo Albero:

dati:

un insieme di nodi (di tipo *nodo*) e un insieme di archi.

operazioni:

numNodi() → intero
restituisce il numero di nodi presenti nell'albero.

grado(nodo v) → intero
restituisce il numero di figli del nodo v .

padre(nodo v) → nodo
restituisce il padre del nodo v nell'albero, o null se v è la radice.

figli(nodo v) → {nodo, nodo, ..., nodo}
restituisce, uno dopo l'altro, i figli del nodo v .

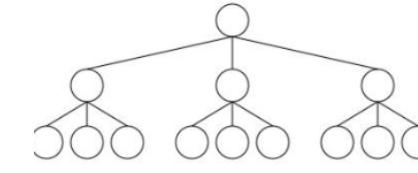
aggiungiNodo(nodo u) → nodo
inserisce un nuovo nodo v come figlio di u nell'albero e lo restituisce.
Se v è il primo nodo ad essere inserito nell'albero, esso diventa la radice
(e u viene ignorato).

aggiungiSottoalbero(Albero a, nodo u)
inserisce nell'albero il sottoalbero a in modo che la radice di a diventi
figlia di u .

rimuoviSottoalbero(nodo v) → Albero
stacca e restituisce l'intero sottoalbero radicato in v . L'operazione
cancella dall'albero il nodo v e tutti i suoi discendenti.

4.6 Alberi k -ari completi

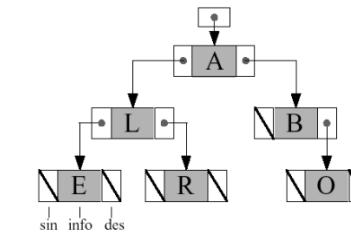
Un albero $k - ario$ completo è un albero $k - ario$ in cui tutte le foglie hanno la stessa profondità e tutti i nodi interni hanno grado k



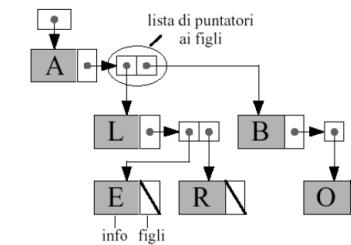
- **Il numero di foglie di un albero k -ario completo è:** la radice ha k figli a profondità 1, ognuno dei figli ha k figli a profondità 2 per un totale di $k \cdot k$ foglie, a profondità h (altezza dell'albero) si hanno k^h foglie
- **Il numero di nodi interni di un albero $k - ario$ completo di altezza h è:** $1 + k + k^2 + \dots + k^{h-1} = \sum_{i=0}^{h-1} k^i = (k^h - 1)/(k - 1)$
- Quindi un albero binario completo ha $2^h - 1$ nodi interni e 2^h foglie

4.7 Rappresentazioni collegate di alberi

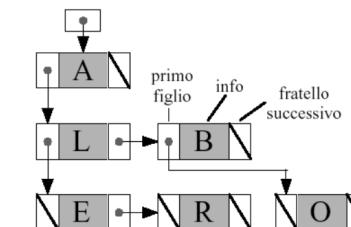
- **Rappresentazione con puntatori ai figli:** $S(n) = O(n)$



- **Rappresentazione con liste di puntatori ai figli:** $S(n) = O(n)$



- **Rappresentazione di tipo primo figlio- fratello successivo**

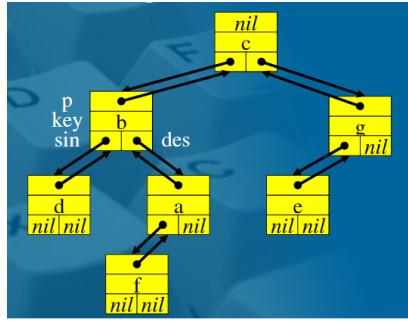


4.8 Alberi binari

Definizione ricorsiva di albero binario: l'insieme vuoto \emptyset è un albero binario, se T_s e T_d sono alberi binari ed r è un nodo allora la terna ordinata (r, T_s, T_d) è un albero binario

In memoria l'albero binario: si rappresenta con due puntatori ai sottoalberi, ed uno al padre:

$$T = \left(c, \left(b, (d, \emptyset, \emptyset), (a, (f, \emptyset, \emptyset), \emptyset) \right), \left(g, (e, \emptyset, \emptyset), \emptyset \right) \right)$$



4.9 Algoritmi di visita

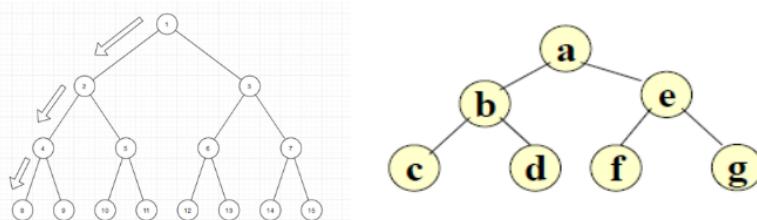
Algoritmi che consentono l'accesso sistematico ai nodi e agli archi di un albero. Gli algoritmi di visita si distinguono in base al particolare ordine di accesso ai nodi:

- **Algoritmi di visita generica:** `visitaGenerica` visita il nodo r e tutti i suoi discendenti in un albero, richiede tempo $O(n)$ per visitare un albero con n nodi a partire dalla radice

algoritmo `visitaGenerica(nodo r)`

1. $S \leftarrow \{r\}$
2. **while** ($S \neq \emptyset$) **do**
3. estrai un nodo u da S
4. visita il nodo u
5. $S \leftarrow S \cup \{ \text{figli di } u \}$

- **Algoritmo di visita in profondità (DFS):** parte da r e procede visitando nodi di figlio in figlio fino a raggiungere una foglia, retrocede poi al primo antenato che ha ancora figli non visitati (se esiste) e ripete il procedimento a partire da uno di quei figli.



Con un albero binario sono possibili 3 strategie:

- **preordine o ordine anticipato:** si visita prima il nodo e poi i sottoalberi sx e dx

Nodo \rightarrow sx \rightarrow dx

Ese: $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g$

algoritmo `visitaDFS(nodo r)`

```
Pila S
S.push(r)
while (not S.isEmpty()) do
    u ← S.pop()
    if ( $u \neq \text{null}$ ) then
        visita il nodo  $u$ 
        S.push(figlio destro di  $u$ )
        S.push(figlio sinistro di  $u$ )
```

algoritmo `visitaDFSRicorsiva(nodo r)`

1. **if** ($r = \text{null}$) **then return**
2. visita il nodo r
3. `visitaDFSRicorsiva(figlio sinistro di r)`
4. `visitaDFSRicorsiva(figlio destro di r)`

- **inordine o ordine simmetrico:** si visita prima il sottoalbero sx e poi il nodo e poi il sottoalbero dx

sx \rightarrow nodo \rightarrow dx

Ese: $c \rightarrow b \rightarrow d \rightarrow a \rightarrow f \rightarrow e \rightarrow g$

- **postordine o ordine posticipato:** si visita prima il sottoalbero sx, poi quello dx e poi il nodo

sx \rightarrow dx \rightarrow nodo

Ese: $c \rightarrow d \rightarrow b \rightarrow f \rightarrow g \rightarrow e \rightarrow a$

- **L'algoritmo di visita in ampiezza (BFS):** parte da r e procede visitando nodi per livelli successivi, un nodo sul livello i può essere visitato solo se tutti i nodi sul livello $i-1$ sono stati visitati.

a \rightarrow b \rightarrow e \rightarrow c \rightarrow d \rightarrow f \rightarrow g

algoritmo `visitaBFS(nodo r)`

```
Coda C
C.enqueue( $r$ )
while (not C.isEmpty()) do
    u ← C.dequeue()
    if ( $u \neq \text{null}$ ) then
        visita il nodo  $u$ 
        C.enqueue(figlio sinistro di  $u$ )
        C.enqueue(figlio destro di  $u$ )
```

5 Modelli di calcolo e metodologie di analisi

5.1 Analisi di algoritmi ricorsivi

La quantità di risorsa usata da algoritmi ricorsivi può essere espressa tramite **relazioni di ricorrenza**, risolvibili tramite vari metodi generali: iterazione, sostituzione, alberi di ricorsione, teorema Master, .. Una **relazione di ricorrenza** descrive come una funzione ricorsiva si ricorre su sé stessa.

algoritmo ricercaBinariaRic(*array L, elemento x*) \rightarrow booleano

1. $n \leftarrow$ lunghezza di *L*
2. **if** ($n = 0$) **then return** non trovato
3. $i \leftarrow \lceil n/2 \rceil$
4. **else if** ($L[i] = x$) **then return** trovato
5. **else if** ($L[i] > x$) **then return** ricercaBinariaRic(*x,L[1;i-1]*)
6. **else return** ricercaBinariaRic(*x,L[i+1;n]*)

5.2 Equazioni di ricorrenza

Il tempo di esecuzione della ricerca binaria (versione ricorsiva) può essere descritto tramite l'equazione di ricorrenza

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \text{ (passo base)} \\ c + T(\lceil \frac{n}{2} \rceil) & \text{se } n > 1 \end{cases}$$

O semplicemente (assumendo n potenza del 2):

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \text{ (passo base)} \\ c + T(\frac{n}{2}) & \text{se } n > 1 \end{cases}$$

5.3 Motodologie

• **Metodo dell'iterazione:** iterare la ricorsione ottenendo una sommatoria dipendente solo dalla dimensione n e dalle condizioni iniziali, le tecniche per limitare le sommatorie possono essere usate poi per fornire limiti alla soluzione, può richiedere l'uso di molta algebra e calcoli complessi, bisogna concentrarsi su due parametri: il **numero di volte che la ricorrenza deve essere iterata** per raggiungere le condizioni iniziali (a contorno), **la somma dei termini originati da ogni livello** del processo di iterazione; talvolta nel processo di iterazione si può indovinare la soluzione senza fare tutti i calcoli matematici in tal caso è possibile passare \Rightarrow al **metodo di sostituzione**

Esempio - ricerca binaria:

$$T(n) = c + T(n/2)$$

$$T(n/2) = c + T(n/4)$$

$$T(n/4) = c + T(n/8)$$

$$T(n) = c + T(n/2) = 2c + T(n/4) = 3c + T(n/8) = \dots$$

$$\begin{aligned} &= \left(\sum_{j=1}^i c \right) + T(n/2^i) \\ &= ic + T(n/2^i) \end{aligned}$$

Raggiungiamo il passo base per $n/2^i = 1$ e cioè per $i = \lg n$, da cui:

$$T(n) = c \lg n + T(1) = O(\lg n)$$

- **Metodo della sostituzione:** l'idea è indovinare una soluzione ed usare l'induzione matematica per provare che la soluzione dell'equazione di ricorrenza è effettivamente quella intuita

Esempio:

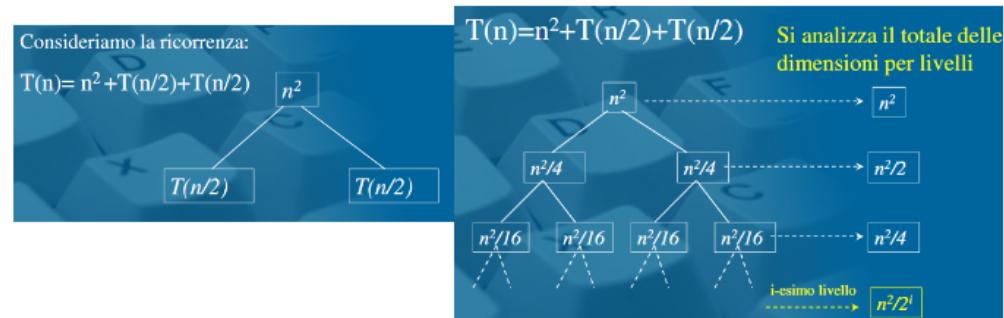
$$T(n) = n + T(n/2), \quad T(1) = 1$$

Assumiamo che la soluzione sia $T(n) \leq cn$ per una costante c opportuna

$$\text{Passo base:} \quad T(1) = 1 \leq c \cdot 1, \forall c \geq 1$$

$$\begin{aligned} \text{Passo induittivo:} \quad T(n) &= n + T(n/2) \leq n + c(n/2) = (1 + c/2)n \\ \text{da cui} \quad T(n) &\leq cn \text{ per } 1 + c/2 \leq c \Rightarrow c \geq 2 \end{aligned}$$

- **Alberi di ricorsione:** è un metodo molto comodo per visualizzare tutto ciò che accade quando si itera una ricorrenza, i nodi riportano le dimensioni dei sotto-problemi, terminiamo quando la dimensione del sotto-problema è 1: $n/2^i = 1$



$$T(n) = n^2 + T(n/2) + T(n/2)$$

$$T(n/2) = n^2/4 + T(n/4) + T(n/4)$$

$$T(n/2^i) \Rightarrow \frac{n}{2^i} = 1 \Rightarrow i = \lg n$$

Poiché il valore decresce geometricamente, il valore totale differisce, al più di un fattore costante dal termine più grande, quindi la $O(n^2)$

$$T(n) = \sum_{i=0}^{\lg n} \frac{n^2}{2^i}$$

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\lg n} \frac{n^2}{2^i} \\
&= n^2 \sum_{i=0}^{\lg n} \left(\frac{1}{2^i} \right) = n^2 \frac{1 - (1/2)^{\lg n + 1}}{1/2} \\
&= n^2 \left(2 - (1/2)^{\lg n} \right) = n^2 \left(2 - n^{\lg(1/2)} \right) \\
&= 2n^2 - n \\
&= \Theta(n^2)
\end{aligned}$$

- **Teorema master:** viene usato per risolvere relazioni di ricorrenza che emergono da algoritmi di tipo ***divide et impera***. La forma generale della ricorrenza è:

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

Divide et impera: dividi il problema di dimensione n in sottoproblemi di dimensione n/b con $a \geq 1$, $b > 1$, risolvi i sottoproblemi ricorsivamente e ricombina le soluzioni. Sia $f(n)$ il tempo impiegato per dividere e ricombinare i sottoproblemi in istanze di dimensioni n . Il teorema master fornisce una soluzione alla relazione di ricorrenza in tre casi, in base al confronto tra $f(n)$ e $n^{\log_b a}$:

- 1) $T(n) = \Theta(n^{\log_b a})$ se $f(n) = O(n^{\log_b a - \varepsilon})$ per $\varepsilon > 0$
- 2) $T(n) = \Theta(n^{\log_b a} \cdot \log n)$ se $f(n) = \Theta(n^{\log_b a})$
- 3) $T(n) = \Theta(f(n))$ se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ per $\varepsilon > 0$ e $a \cdot f(n/b) \leq c \cdot f(n)$ per $c < 1$ e n suff. grande

Esempi

- 1) $T(n) = n + 2T(n/2)$
 $a=2, b=2, f(n)=n=\Theta(n^{\log_2 2}) \Rightarrow T(n)=\Theta(n \log n)$
(caso 2 del teorema master)
- 2) $T(n) = c + 3T(n/9)$
 $a=3, b=9, f(n)=c=O(n^{\log_9 3 - \varepsilon}) \Rightarrow T(n)=\Theta(\sqrt{n})$
(caso 1 del teorema master) per $\varepsilon = 0,5$
- 3) $T(n) = n + 3T(n/9)$
 $a=3, b=9, f(n)=n=\Omega(n^{\log_9 3 + \varepsilon})$
 $3(n/9) \leq c n$ per $c=1/3$ e $\varepsilon = 0,5 \Rightarrow T(n)=\Theta(n)$
(caso 3 del teorema master)

6 Binary search tree (BST)

Gli alberi di ricerca sono usati per realizzare in modo efficiente il tipo di dato dizionario

tipo Dizionario:

dati:

un insieme S di coppie $(elem, chiave)$.

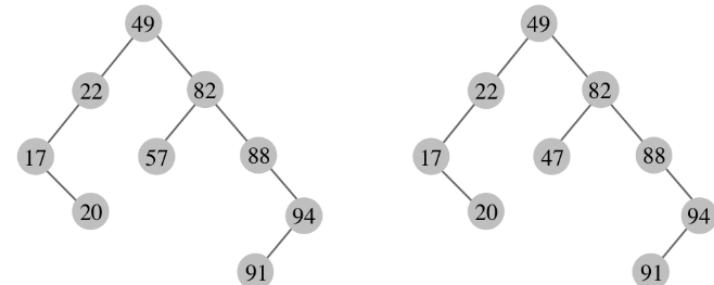
operazioni:

insert($elem e, chiave k$)
aggiunge a S una nuova coppia (e, k) .

delete($chiave k$)
cancella da S la coppia con chiave k .

search($chiave k \rightarrow elem$)
se la chiave k è presente in S restituisce l'elemento e ad essa associato,
e null altrimenti.

Albero binario che soddisfa le seguenti proprietà: ogni nodo v contiene un elemento $elem(v)$ cui è associata una chiave $chiave(v)$ presa da un dominio totalmente ordinato, le chiavi nel sottoalbero sinistro di v sono $\leq chiave(v)$, le chiavi nel sottoalbero destro di v sono $\geq chiave(v)$

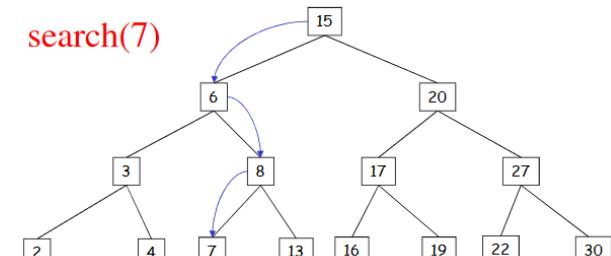


Albero binario di ricerca

Albero binario non di ricerca (47 a dx di 49)

6.1 search($chiave k \rightarrow elem$)

Sfruttando la proprietà di ricerca, partendo dalla radice, su ogni nodo decidiamo se proseguire nel sottoalbero sinistro o destro

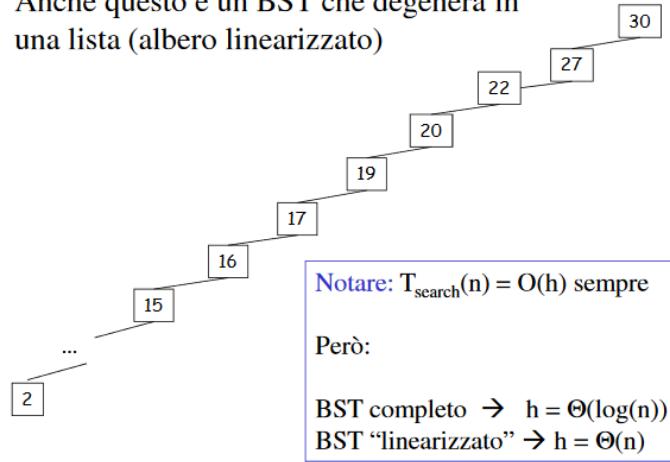


Traccia un cammino nell'albero partendo dalla radice: su ogni nodo, usa la proprietà di ricerca per decidere se proseguire nel sottoalbero sinistro o destro \Rightarrow costo $O(h)$

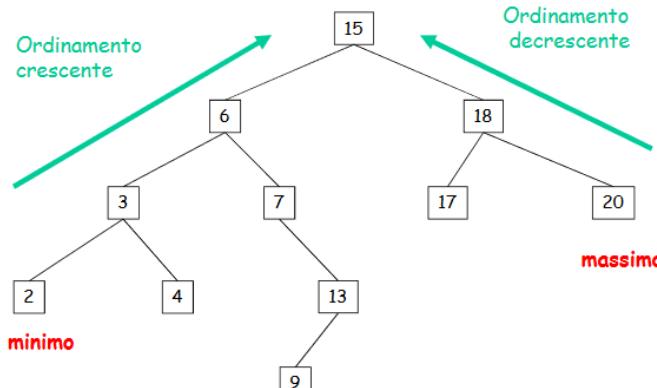
algoritmo search(chiave k) \rightarrow elem

1. $v \leftarrow$ radice di T
2. **while** ($v \neq \text{null}$) **do**
3. **if** ($k = \text{chiave}(v)$) **then return** $\text{elem}(v)$
4. **else if** ($k < \text{chiave}(v)$) **then** $v \leftarrow$ figlio sinistro di v
5. **else** $v \leftarrow$ figlio destro di v
6. **return** null

Anche questo è un BST che degenera in una lista (albero linearizzato)



6.2 Ricerca del massimo e del minimo

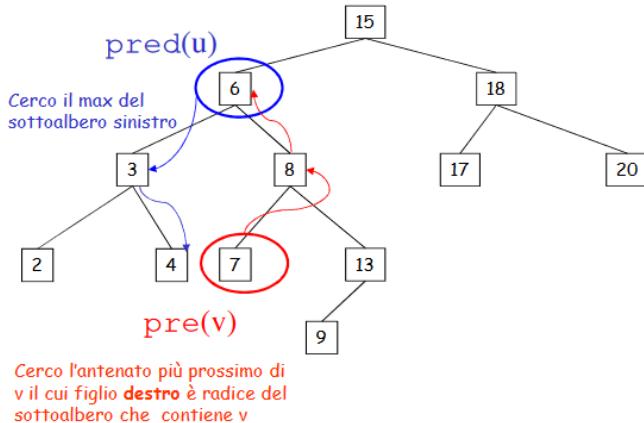


algoritmo max(nodo u) \rightarrow nodo

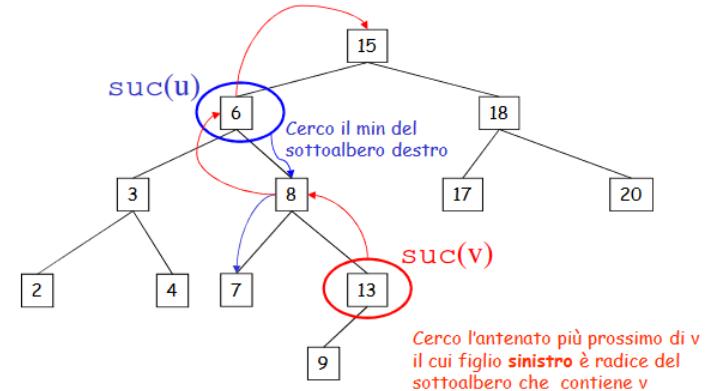
1. $v \leftarrow u$
2. **while** (figlio destro di $v \neq \text{null}$) **do**
3. $v \leftarrow$ figlio destro di v
4. **return** v

6.3 Predecessore e successore

- Il **precedessore** di un nodo u in un BST è il nodo v nell'albero di chiave massima $\leq \text{chiave}(u)$. In altre parole è il più grande nodo che precede u nell'ordinamento in-order dell'albero.
 - **Massimo del sottoalbero sinistro:** se u ha un sottoalbero sx, il predecessore di u è il nodo con il valore massimo in questo sottoalbero $\Rightarrow \text{pred}(8) = 7$
 - **Antenato più prossimo:** se u non ha un sottoalbero sx, il predecessore di u è il primo antenato v tale che u è nel sottoalbero dx di v $\Rightarrow \text{pred}(7) = 6$

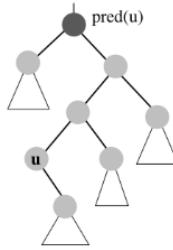
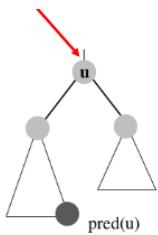


- Il **successore** di un nodo u in un BST è il nodo v nell'albero di chiave minima $\geq \text{chiave}(u)$. In altre parole, è il più piccolo nodo che segue u nell'ordinamento in-order dell'albero.
 - **Minimo del sottoalbero destro:** se u ha un sottoalbero dx, il successore di u è il nodo con il valore minimo in questo sottoalbero.
 - **Antenato più prossimo:** se u non ha un sottoalbero dx, il successore di u è il primo antenato v tale che u è nel sottoalbero sx di v .



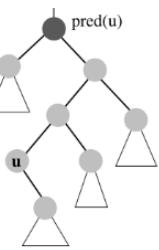
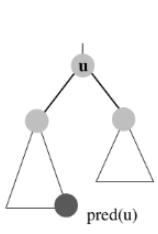
algoritmo pred(*nodo u*) \rightarrow *nodo*

1. **if** (*u* ha figlio sinistro *sin(u)*) **then**
2. **return** max(*sin(u)*)
3. **while** (*parent(u)* \neq null e *u* è figlio sinistro di suo padre) **do**
4. *u* \leftarrow *parent(u)*
5. **return** *parent(u)*



algoritmo pred(*nodo u*) \rightarrow *nodo*

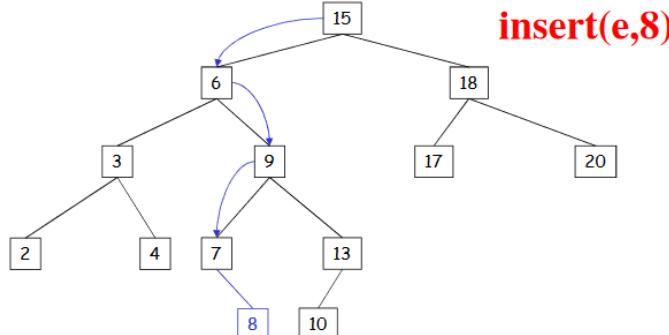
1. **if** (*u* ha figlio sinistro *sin(u)*) **then**
2. **return** max(*sin(u)*)
3. **while** (*parent(u)* \neq null e *u* è figlio sinistro di suo padre) **do**
4. *u* \leftarrow *parent(u)*
5. **return** *parent(u)*



6.4 insert(*elem e, chiave k*)

Idea: aggiunge la nuova chiave come nodo foglia simulando una ricerca con la chiave da inserire per individuare la corretta posizione

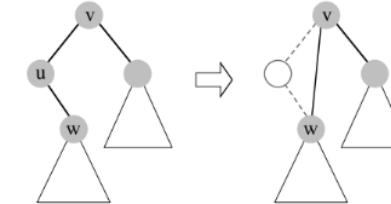
- Crea un nuovo nodo *u* con *elem = e* e *chiave = k*
- Cerca la chiave *k* nell'albero, identificando così il nodo *v* che diventerà padre di *u*
- Appendi *u* come figlio sx/dx di *v* in modo che sia mantenuta la proprietà di ricerca



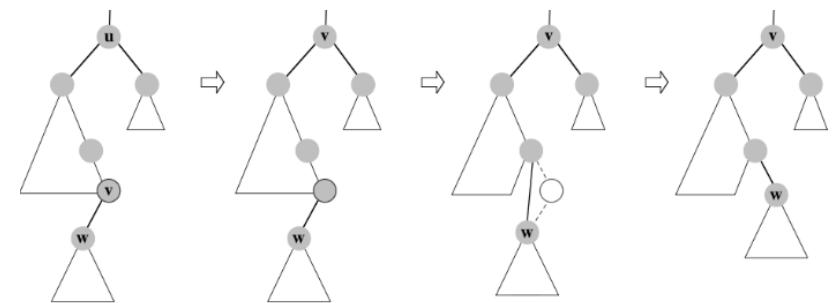
6.5 delete(*chiave k*)

Sia *u* il nodo contenente l'elemento da cancellare:

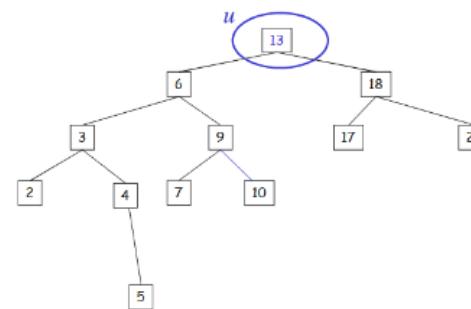
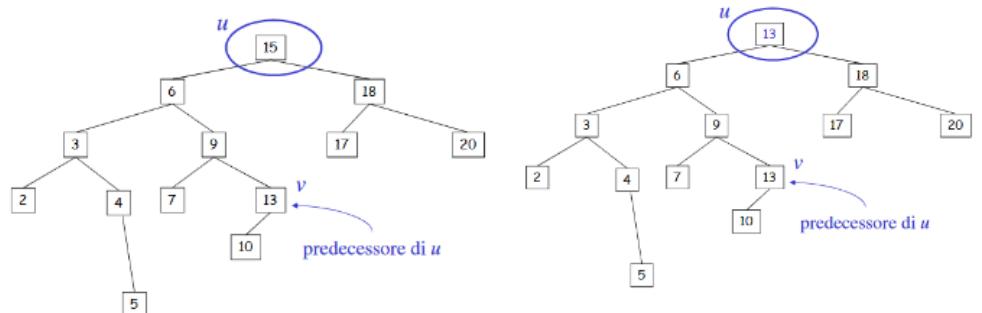
1. *u* è una foglia \Rightarrow rimuovila
2. *u* ha un solo figlio



3. *u* ha due figli: sostituisco con il predecessore(*v*) e rimuovi fisicamente il predecessore \Rightarrow il predecessore ha un solo figlio al piú casi 1 e 2), altrimenti si può usare il successore di *u*



Esempio: delete(15)

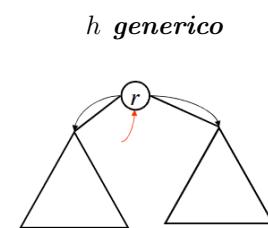
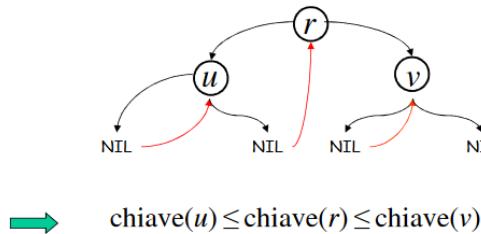


6.6 Visita simmetrica di un BST

Visita in ordine simmetrico: dato un nodo x elenco prima il sotto-albero sx di x (in ordine simmetrico) poi il nodo x poi il sotto-albero dx (in ordine simmetrico), una visita simmetrica di un BST equivale a visitare i nodi del BST in ordine crescente rispetto alla chiave

Verifica di correttezza: supponiamo per semplicità che l'albero sia completo e indichiamo con h l'altezza dell'albero, vogliamo mostrare che la visita in ordine simmetrico restituisce la sequenza ordinata

Per induzione sull'altezza $h = 1$ (caso base)



6.7 Costo delle operazioni dipende da h

- Tutte le operazioni principali su un BST (come inserimento, cancellazione e ricerca) hanno un costo nel caso peggiore pari a $O(h)$, dove h è l'altezza dell'albero. Questo perché, nel peggio dei casi, l'operazione richiederà di percorrere l'albero dalla radice fino a una foglia, coprendo un percorso lungo al massimo h .
- Caso peggiore (BST degenerato in una lista):** se il BST è molto sbilanciato e degenerato in una lista, l'altezza h dell'albero sarà uguale al numero di nodi meno uno, quindi $h = n - 1$. In questo caso, il costo delle operazioni diventa $O(n)$, poiché l'albero è estremamente sbilanciato e l'altezza è massima.
- Caso migliore (BST bilanciato):** se l'albero è bilanciato, cioè l'altezza h è la più piccola possibile rispetto al numero di nodi n , allora h sarà approssimativamente $O(\log n)$. In questo caso, le operazioni hanno un costo nel caso peggiore di $O(\log n)$.

Esempio: albero binario completo con n nodi, un albero è completo se ogni livello è completamente riempito tranne forse l'ultimo

- Nodi per livello:** in un albero binario completo, il numero totale di nodi n è dato dalla somma dei nodi a ciascun livello, che segue la formula:

$$n = \sum_{i=0}^h 2^i = 2^{h+1} - 1$$

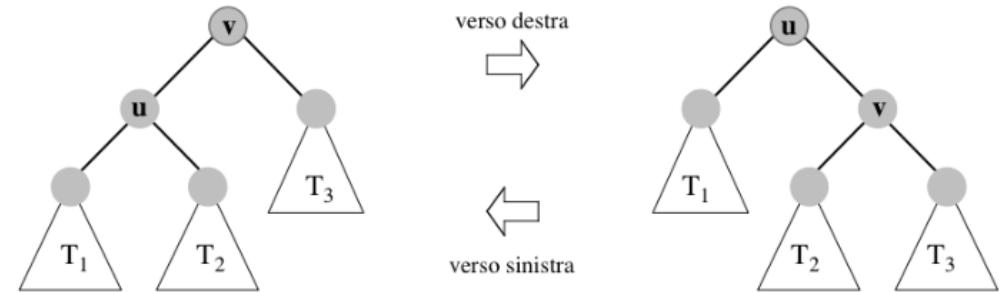
- Passando al logaritmo:** per trovare l'altezza h in funzione di n , si applica il logaritmo. Dato che il logaritmo di una quantità lineare come n è $O(\log n)$, possiamo concludere che l'altezza di un albero binario completo con n nodi è $O(\log n)$:

$$h = \log_2(n + 1) - 1 = O(\log n)$$

6.8 Alberi binari bilanciati

- Fattore di bilanciamento** di un nodo v : $\beta(v) = |h(\text{sinistro}(v)) - h(\text{destro}(v))|$
- Un albero binario si dice **bilanciato in altezza** se ogni nodo v ha: $\beta(v) \leq 1$
- Un albero binario completo ha $\beta(v) = 0$ su ogni nodo v
- Un albero binario che degenera in una lista ha $\beta(v) = h$
- Alberi AVL = alberi binari di ricerca bilanciati in altezza** dai noti ideatori, un albero AVL con n nodi ha altezza $h = O(\log n)$
- Mantenere il **bilanciamento** sembra cruciale per ottenere buone prestazioni, la search non crea problemi ma inserimento e cancellazione potrebbero sbilanciare l'albero, esistono vari approcci per mantenere il bilanciamento attraverso **trasformazioni dell'albero**: tramite **rotazioni** anche in cascata (alberi AVL), tramite **separazioni o fusioni** di nodi con grado variabile (alberi rosso-neri) \Rightarrow in tutti questi casi si ottengono tempi di esecuzione logaritmici nel caso peggiore

Rotazione di base: mantiene la proprietà di ricerca, richiede tempo $O(1)$



insert(elem e, chiave k):

- Crea un nuovo nodo u con $elem = e$ e $chiave = k$
- Inserisci u come in un BST
- Ricalcola i fattori di bilanciamento dei nodi nel cammino dalla radice a u : sia v il più profondo nodo con fattore di bilanciamento pari a ± 2 (nodo critico)
- Esegui una rotazione opportuna su v
- una sola rotazione è sufficiente, poiché l'altezza dell'albero coinvolto diminuisce di 1

delete(elem e):

- Cancella il nodo come in un BST
- Ricalcola i fattori di bilanciamento dei nodi nel cammino dalla radice al padre del nodo eliminato fisicamente (che potrebbe essere il pred del nodo contenente e)
- Ripercorrendo il cammino dal basso verso l'alto, esegui l'opportuna rotazione semplice o doppia sui nodi sbilanciati
- Potrebbero essere necessarie $O(\log n)$ rotazioni

6.9 Classe AlberoAVL

classe AlberoAVL estende AlberoBinarioDiRicerca:

dati: $S(n) = O(n)$
albero binario di ricerca T ereditato, più il fattore di bilanciamento di ogni nodo.

operazioni:

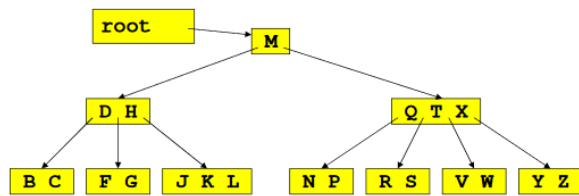
search(chiave k) → elem $T(n) = O(\log n)$
ereditata.

insert(elem e, chiave k) $T(n) = O(\log n)$
chiama **insert()** ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente ribilancia tramite $O(1)$ rotazioni.

delete(elem e) $T(n) = O(\log n)$
chiama **delete()** ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente ribilancia tramite $O(\log n)$ rotazioni.

7 B-alberi

- I B-alberi sono alberi bilanciati particolarmente adatti per memorizzare grandi quantità di dati in memoria secondaria (disco), sono progettati in modo tale da **minimizzare il numero di accessi a disco**, le operazioni fondamentali sui B-alberi sono **Insert**, **Delete** e **Search** alle quali si aggiungono **Split** e **Join** per mantenere il bilanciamento
- A differenza dei nodi degli alberi binari di ricerca che contengono una sola chiave ed hanno due figli, i nodi dei B-alberi possono contenere un numero n di chiavi ed avere $n + 1$ figli con $n \geq 1$



7.1 Costi operazioni

Nel valutare la complessità delle operazioni, terremo separate le due componenti:

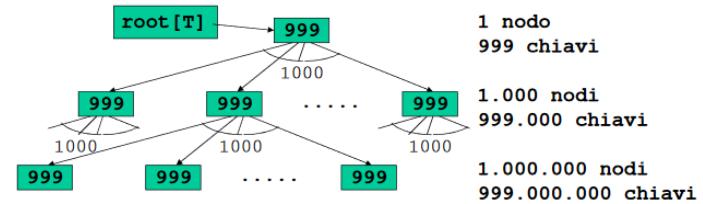
- **tempo di lettura-scrittura su disco** (che assumiamo proporzionale al numero di blocchi letti-scritti su disco, ordine dei millisecondi)
- **tempo di CPU** (tempo di calcolo in memoria centrale), ordine dei microsecondi

L'operazione di lettura da disco è: $DiskRead(x)$ che dato il riferimento x ad un oggetto legge l'oggetto da disco in memoria centrale, assumiamo che sia possibile accedere al valore di un oggetto soltanto dopo che esso sia stato letto in memoria centrale, assumiamo inoltre che se l'oggetto da leggere si trova già in memoria centrale l'operazione $DiskRead(x)$ non abbia alcun effetto.

L'operazione di scrittura su disco è: $DiskWrite(x)$ che dato il riferimento x ad un oggetto presente in memoria centrale scrive tale oggetto su disco, tipicamente l'elaborazione di un oggetto residente su disco segue lo schema:

<i>DiskRead(x)</i>	
.....	▷ elaborazioni che accedono/modificano x .
<i>DiskWrite(x)</i>	▷ non necessaria se x resta invariato
.....	▷ elaborazioni che non modificano x .

Un **elevato grado di diramazione** riduce in modo drastico sia l'altezza dell'albero che il numero di letture da disco necessarie per cercare una chiave, la figura seguente mostra che un B-albero di altezza 2 con un grado di diramazione 1000 può contenere $10^9 - 1$ chiavi (un miliardo)



7.2 Definizione di B-albero

Un B-albero T è un albero di radice $root[T]$ tale che:

- Ogni nodo x contiene i seguenti campi:

$n[x]$	il numero di chiavi presenti nel nodo
$n[x] + 1$	il numero di figli del nodo
$key_1[x] \leq \dots \leq key_{n[x]}[x]$	le $n[x]$ chiavi in ordine non decrescente
$leaf[x]$	true se il nodo x è foglia, false altrimenti

- se il nodo non è una foglia contiene anche: $c_1[x], \dots, c_{n[x]+1}[x]$, gli $n[x]+1$ puntatori ai figli
- le $n[x]$ chiavi $key_1[x] \leq \dots \leq key_{n[x]}[x]$ di un nodo interno separano gli intervalli **contenenti le chiavi dei sottoalberi**, in altre parole se $k_1, \dots, k_{n[x]+1}$ sono chiavi appartenenti rispettivamente ai sottoalberi di radici $c_1[x], \dots, c_{n[x]+1}[x]$ allora:

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \dots \leq key_{n[x]}[x] \leq k_{n[x]+1}$$

- Le foglie sono tutte allo stesso livello h detto altezza dell'albero.
- Vi è un *limite superiore* ed un *limite inferiore* al numero di chiavi contenuto in un nodo e tali limiti dipendono da una **costante t detta grado minimo del B-albero**. Precisamente:

- ogni nodo, eccetto la radice, ha almeno $t - 1$ chiavi e almeno t figli (se non è una foglia): $n[x] \geq t - 1$
- se l'albero non è vuoto, la radice contiene almeno una chiave e se la radice non è foglia ha almeno due figli.
- ogni nodo ha al più $2t - 1$ chiavi e al più $2t$ figli (se non è foglia): $n[x] \leq 2t - 1$, un nodo con $2t - 1$ chiavi si dice **pieno**.

<i>Costante fissa:</i>	t
<i>Chiavi</i>	$t - 1 \leq n[x] \leq 2t - 1$
<i>figli:</i>	$t \leq n[x] \leq 2t$

Ad ogni chiave sono generalmente associate delle informazioni ausiliarie. Assumeremo implicitamente che quando viene copiata una chiave vengano copiate anche tali informazioni. I B-alberi più semplici sono quelli con grado minimo $t = 2$. Ogni nodo interno ha 2, 3 o 4 figli. In questo caso si dicono anche **2-3-4-alberi**.

7.3 Altezza di un B-albero

Proprietà: ogni B-albero di grado minimo t contenente N chiavi ha altezza $h \leq \log_t \frac{N+1}{2}$ (assumendo per convenzione che l'albero vuoto abbia altezza -1)

$$h \leq \log_t \left(\frac{N+1}{2} \right)$$

Questo risultato stabilisce un limite superiore sull'altezza h dell'albero in termini di N , il numero di chiavi e t il grado minimo del B-albero.

Dimostrazione: vedere slide 9 \Rightarrow l'altezza di un B-albero è $O(\log_t N)$, dello stesso ordine $O(\log_2 N)$ degli alberi binari bilanciati, ma la costante nascosta nella O è inferiore di un fattore $\log_2 t$ che, per $50 \leq t \leq 2000$, è compreso tra 5 e 11.

7.4 Operazioni elementari

Adottiamo le seguenti convenzioni per le operazioni sui B-alberi: la radice del B-albero è sempre in memoria, i nodi passati come parametri alle funzioni sono stati preventivamente letti in memoria

Defineremo le seguenti operazioni:

- **BTree** costruttore di un albero vuoto;
- **Search** che cerca una chiave nell'albero;
- **Insert** che aggiunge una chiave;
- **Delete** che toglie una chiave.

ci serviranno anche tre procedure ausiliarie:

- **SearchSubtree**
- **SplitChild**
- **InsertNonfull**

Le operazioni:

- **BTree:** complessità $O(1)$

```
BTree(T)
root[T] ← nil
```

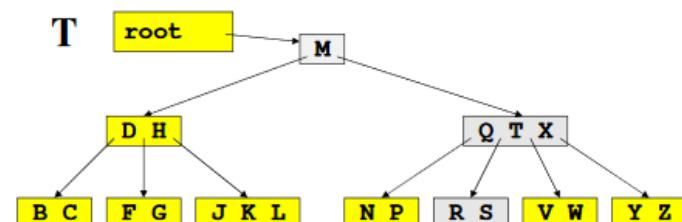
- **Search:** si limita a richiamare la funzione ausiliaria *SearchSubtree*

```
Search(T, k)
if root[T] = nil then return nil
else return SearchSubtree(root[T], k)
```

- **SearchSubtree:** restituisce (y, i) con y nodo ed i il più piccolo indice t.c. $y.key_i = k$ se k si trova nel B-albero, *nil* altrimenti.

```
SearchSubtree(x, k)
1. i ← 1
2. while i ≤ n[x] and k > key_i[x] do
3.   i ← i + 1 ▷ Ricerca dell'indice i t.c.: k ≤ key_i[x]
      ▷ Invariante: key_{1..i-1}[x] < k ≤ key_{i..n[x]}[x]
4. if i ≤ n[x] and k = key_i[x] return x, i ▷ Ric. successo
5. else if leaf[x] return nil ▷ Ric. senza successo
6. else ▷ Ric. ricorsiva nel sotto-albero c_i[x]
7.   DiskRead(c_i[x])
8.   return SearchSubtree(c_i[x], k)
```

Esempio: $search(T, R)$



Il **numero di DiskRead** è al più uguale all'altezza h dell'albero ed è quindi $O(h) = O(\log_t N)$ con N numero di chiavi nel B-albero. Il **tempo T di CPU** di *Search* è: $T = O(t \cdot h) = O(t \cdot \log_t N)$ poiché $n[x] \leq 2t - 1 \leq 2t$ e quindi il tempo impiegato per il ciclo while (righe 2-3) per ogni nodo è $O(t)$. Essendo le chiavi in un nodo ordinata, si può fare di meglio con una **ricerca binaria** nel nodo: $T = O(\log t \cdot h) = O(\log t \cdot \log_t N) = O(\log N)$. La procedura ausiliaria *SearchSubtree* con ricerca binaria nel nodo è:

```

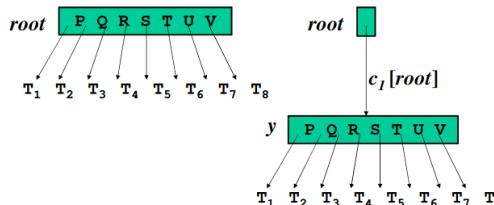
SearchSubtree(x, k)
   $i \leftarrow 1, j \leftarrow n[x]+1$ 
   $\triangleright \text{INVARIANTE: } key_{1..i-1}[x] < k \leq key_{j..n[x]}[x]$ 
  while  $i < j$  do  $\triangleright \text{Ricerca binaria}$ 
    if  $k \leq key_{\lfloor(i+j)/2\rfloor}[x]$  then  $j \leftarrow \lfloor(i+j)/2\rfloor$ 
    else  $i \leftarrow \lfloor(i+j)/2\rfloor + 1$ 
   $\triangleright key_{1..i-1}[x] < k \leq key_{i..n[x]}[x]$ 
  if  $i \leq n[x]$  and  $k = key_i[x]$  return  $x, i$ 
  if  $leaf[x]$  return nil
  DiskRead( $c_i[x]$ )
  return  $\text{SearchSubtree}(c_i[x], k)$ 

```

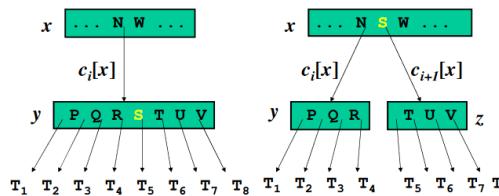
- **Insert:** non possiamo aggiungere una chiave ad un nodo interno perché dovremmo aggiungere anche un sottoalbero \Rightarrow l'aggiunta di una chiave in un B-albero può avvenire soltanto in una foglia, questo si può fare soltanto se *la foglia non è piena*.

Possiamo assicurarsi che la foglia a cui arriveremo non sia piena se durante la discesa dalla radice a tale foglia ci assicuriamo ad ogni passo che il figlio su cui scendiamo non sia pieno, nel caso in cui tale figlio sia pieno chiamiamo prima una particolare funzione **SplitChild** che lo divide in due parti, la stessa cosa dobbiamo fare all'inizio per la radice se essa è piena

Esempio: ecco come funziona Insert se la radice è piena in un B-albero di grado minimo $t = 4$, dopo di che richiama **SplitChild**



Come funziona **SplitChild** su un figlio pieno in un B-albero di grado minimo $t = 4$



Operazione taglia ed incolla: il nodo z adotta i figli maggiori di y e diventa nuovo figlio di x , subito dopo y , e la chiave mediana si sposta da y ad x per separare y e z

Il numero di **DiskRead** e **DiskWrite** è $O(h)$ perché sono eseguite $O(1)$ operazioni di accesso al disco tra due chiamate consecutive di **InsertNonfull**. Il tempo **T** di CPU di è $O(t \cdot h)$ perché **InsertNonfull** è ricorsiva in coda, in alternativa può essere iterativa con un ciclo while.

```

Insert(T,k)
  if  $root[T] = \text{nil}$  then  $\triangleright$  l'albero è vuoto
     $x \leftarrow \text{AllocateNode}()$ 
     $n[x] \leftarrow 1, key_1[x] \leftarrow k, leaf[x] \leftarrow \text{true}$ 
     $root[T] \leftarrow x$   $\triangleright$  Qui l'altezza cresce di 1
  else  $\triangleright$  l'albero è non vuoto
    if  $n[root[T]] = 2t - 1$  then  $\triangleright$  se la radice è piena
       $y \leftarrow root[T]$ 
       $x \leftarrow \text{AllocateNode}()$   $\triangleright$  x è la nuova radice
       $n[x] \leftarrow 0, c_1[x] \leftarrow y, leaf[x] \leftarrow \text{false}$ 
      SplitChild( $x, 1, y$ )  $\triangleright$  Split della radice (nodo y)
       $root[T] \leftarrow x$   $\triangleright$  Qui l'altezza cresce di 1
    InsertNonfull( $root[T], k$ )

```

- **SplitChild:**

```

SplitChild(x, i, y)
   $\triangleright$  PRECONDIZIONE:  $y$  è figlio  $i$ -esimo di  $x$  ed è pieno
   $z \leftarrow \text{AllocateNode}()$ 
   $leaf[z] \leftarrow leaf[y]$ 
   $\triangleright$  Sposta le ultime  $t - 1$  chiavi di  $y$  in  $z$ 
  for  $j \leftarrow 1$  to  $t - 1$  do
     $key_j[z] \leftarrow key_{j+t}[y]$ 
   $\triangleright$  Se  $y$  non è una foglia sposta gli ultimi  $t$  puntatori
   $\triangleright$  di  $y$  in  $z$ 
  if  $\text{not } leaf[y]$  then
    for  $j \leftarrow 1$  to  $t$  do
       $c_j[z] \leftarrow c_{j+t}[y]$ 
     $n[z] \leftarrow t - 1$ 

```

```

 $\triangleright$  Sposta la  $t$ -esima chiave di  $y$  in  $x$ 
for  $j \leftarrow n[x]$  downto  $i$  do
   $key_{j+1}[x] \leftarrow key_j[x]$   $\triangleright$  fa spazio in  $x$  alla nuova chiave
for  $j \leftarrow n[x]+1$  downto  $i+1$  do
   $c_{j+1}[x] \leftarrow c_j[x]$   $\triangleright$  fa spazio in  $x$  al nuovo figlio  $z$ 
   $key_i[x] \leftarrow key_i[y]$   $\triangleright$  copia della  $t$ -esima chiave di  $y$  in  $x$ 
   $c_{i+1}[x] \leftarrow z$   $\triangleright$  inserimento in  $x$  del nuovo figlio  $z$ 
   $n[x] \leftarrow n[x] + 1$   $\triangleright$  ora  $x$  ha una chiave in più
   $\triangleright$  Rimuove le ultime  $t$  chiavi da  $y$ 
   $n[y] \leftarrow t - 1$ 
   $\triangleright$  Scrive su disco i nodi modificati
  DiskWrite( $x$ ), DiskWrite( $y$ ), DiskWrite( $z$ )

```

- *InsertNonfull*: se x è una foglia inseriamo k direttamente in x nella posizione appropriata, se x non è foglia determiniamo il figlio di x su cui far scendere la ricorsione (con eventuale split del figlio)

InsertNonfull(x, k)

▷ Precondizione: x non è pieno

1. if $leaf[x]$ then ▷ Inserisce la chiave k nel nodo x
2. $i \leftarrow n[x]+1$
3. while $i > 1$ and $key_{i-1}[x] > k$ do
4. $key_i[x] \leftarrow key_{i-1}[x]$, $i \leftarrow i - 1$
5. $key_i[x] \leftarrow k$, $n[x] \leftarrow n[x]+1$
6. *DiskWrite(x)*

7. else ▷ Cerca il figlio in cui inserirla
8. $i \leftarrow 1$, $j \leftarrow n[x]+1$
9. ▷ INVARIANTE: $key_{1..i-1}[x] < k \leq key_{j..n[x]}[x]$
10. while $i < j$ do ▷ Ricerca binaria
11. if $k \leq key_{\lfloor(i+j)/2\rfloor}[x]$ then $j \leftarrow \lfloor(i+j)/2\rfloor$
12. else $i \leftarrow \lfloor(i+j)/2\rfloor + 1$
13. *DiskRead(c_i[x])*
14. if $n[c_i[x]] = 2t - 1$ then ▷ Split del figlio se è pieno
15. *SplitChild(x, i, c_i[x])*
16. if $k > key_i[x]$ then $i \leftarrow i + 1$ ▷ Scelta dei due (dopo lo split) figli di x su cui scendere
16. *InsertNonfull(c_i[x], k)*

- *Delete*: essa si limita a richiamare la funzione ausiliaria *DeleteNonmin* sulla radice dell'albero dopo essersi assicurata che l'albero non sia vuoto. Se al ritorno la radice non contiene alcuna chiave essa viene rimossa.

Delete(T,k)

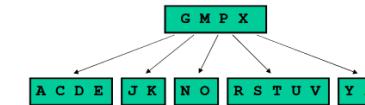
```
if root[T] ≠ nil then
  DeleteNonmin(root[T],k)
  if n[root[T]] = 0 then
    root[T] ← c_i[root[T]] ▷unico figlio
```

La procedura *DeleteNonmin* usa la procedura *AugmentChild* per assicurarsi di scendere sempre su di un nodo che non contiene il minimo numero di chiavi.

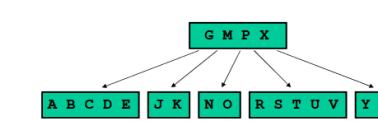
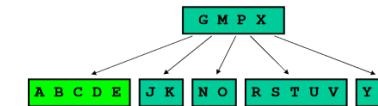
La procedura *AugmentChild* aumenta il numero di chiavi del figlio i -esimo prendendone una da uno dei fratelli adiacenti. Se i fratelli adiacenti hanno tutti il minimo numero di chiavi allora riunisce (fusione) il figlio i -esimo con uno dei fratelli adiacenti ⇒ *Tempi di Delete come Insert*

7.5 Esempi inserimenti

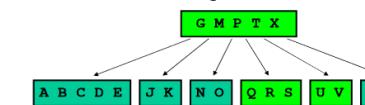
Grado $t = 3$ al più 5 chiavi e al più 6 figli



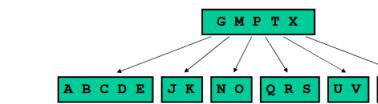
Insert(T,B)



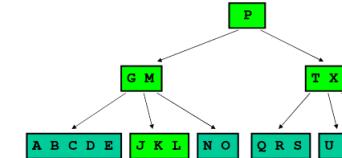
Split del figlio (R S T U V)
Inserimento di Q nella metà a sx



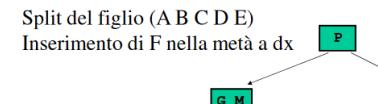
Insert(T,Q)



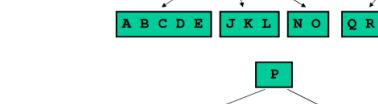
Split della radice che è piena
Inserimento di L nella foglia (J K)



Insert(T,L)



Split del figlio (A B C D E)
Inserimento di F nella metà a dx



Insert(T,F)

8 Tabella hash

8.1 Tabella ad accesso diretto

Dizionari basati sulla rappresentazione indicizzata mediante array. **Idea:** dizionario memorizzato in array V di m celle, a ciascun elemento è associata una chiave intera k in $[0, m - 1]$, elemento con chiave k contenuto in $V[k]$, al più $n \leq m$ elementi nel dizionario

Key	Value
1	New York
2	Boston
3	Mexico
4	Kansas
5	Detroit
6	California

classe TavolaAccessoDiretto implementa Dizionario:
dati: $S(m) = \Theta(m)$
 un array v di dimensione $m \geq n$ in cui $v[k] = elem$ se c'è un elemento $elem$ con chiave k nel dizionario, e $v[k] = null$ altrimenti. Le chiavi k devono essere interi nell'intervallo $[0, m - 1]$.

operazioni:

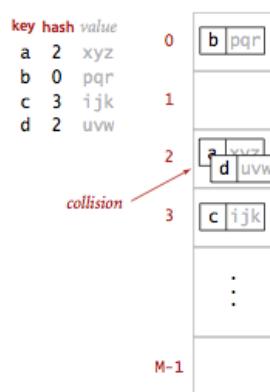
$insert(elem e, chiave k)$	$T(n) = O(1)$
$v[k] \leftarrow e$	
$delete(chiave k)$	$T(n) = O(1)$
$v[k] \leftarrow null$	
$search(chiave k) \rightarrow elem$	$T(n) = O(1)$
$return v[k]$	

Fattore di carico: misuriamo il grado di riempimento di una tabella usando il fattore di carico $\alpha = \frac{n}{m}$, grande spreco di memoria se $m \geq n$. **Esempio:** tabella con nomi di studenti indicizzati da numeri di matricola a 6 cifre

$$n = 100 \quad m = 10^6 \quad \alpha = 0.0001 = 0.01\%$$

Pregi: tutte le operazioni richiedono tempo $O(1)$. **Difetti:** le chiavi devono essere necessariamente interi in $[0, m - 1]$, lo spazio utilizzato è proporzionale ad m non al numero n di elementi: può esserci grande spreco di memoria!

Tabelle hash: per ovviare agli inconvenienti delle tabelle ad accesso diretto ne consideriamo un'estensione \Rightarrow le **tabelle hash**. **Idea:** chiavi prese da un universo totalmente ordinato U (possono non essere numeri), **funzione hash** $h : U \rightarrow [0, m - 1]$ elemento con chiave k in posizione $v[h(k)]$. Le tabelle hash presentano il fenomeno delle **collisioni**



Si ha una collisione quando: si deve inserire nella tabella hash un elemento con chiave u , e nella tabella esiste già un elemento con chiave v con $u \neq v$ tale che $h(u) = h(v)$ (stesso hash), il nuovo elemento andrebbe a sovrascrivere il vecchio.

V	<i>array</i>
m	<i>celle array</i> $[0, \dots, m - 1]$
k	<i>key dell'array</i> $V[k]$
$h : U \rightarrow [0, \dots, m - 1]$	<i>funzione hash</i>
$k \in U$	<i>universo delle chiavi, insieme totalmente ordinato</i>

8.2 Funzioni hash perfette

Un modo per evitare il fenomeno delle collisioni è usare **funzioni hash perfette**. Una funzione hash si dice **perfetta** se è iniettiva:

$$\forall u, v \in U : u \neq v \Rightarrow h(u) \neq h(v)$$

Deve essere $|U| \leq m$ necessaria per avere una funzione hash perfetta è però raramente conveniente, disperdere in modo uniforme le chiavi, **suriettiva** \Rightarrow **possibili collisioni**, se **biunivoca** quand si considerano solo le chiavi usate \Rightarrow allora é una funzione **hash perfetta**

classe TavolaHashPerfetta implementa Dizionario:
dati: $S(m) = \Theta(m)$
 un array v di dimensione $m \geq n$ in cui $v[h(k)] = e$ se c'è un elemento e con chiave $k \in U$ nel dizionario, e $v[h(k)] = null$ altrimenti. La funzione $h : U \rightarrow \{0, \dots, m - 1\}$ è una funzione hash perfetta calcolabile in tempo $O(1)$.

operazioni:

$insert(elem e, chiave k)$	$T(n) = O(1)$
$v[h(k)] \leftarrow e$	
$delete(chiave k)$	$T(n) = O(1)$
$v[h(k)] \leftarrow null$	
$search(chiave k) \rightarrow elem$	$T(n) = O(1)$
$return v[h(k)]$	

8.3 Uniformità delle funzioni hash

Per ridurre la probabilità di collisioni, una buona funzione hash dovrebbe essere in grado di distribuire in modo uniforme le chiavi nello spazio degli indici della tabella $[0, \dots, m - 1] \Rightarrow$ questo si ha se la funzione hash gode della **proprietà di uniformità semplice**

Sia $P(k)$ a probabilità che la chiave k sia presente nel dizionario e $Q(i)$ rappresenta la probabilità che la cella i sia occupata:

$$Q(i) = \sum_{k: h(k)=i} P(k)$$

una funzione hash h gode **dell'uniformità semplice** se per ogni indice scelto a caso i in $[0, \dots, m - 1]$

$$Q(i) = \frac{1}{m}$$

In altre parole, quando si sceglie un indice i casualmente nell'intervallo $[0, \dots, m - 1]$, la probabilità che la cella i sia occupata (cioè contenga almeno una chiave) è la stessa per tutte le celle della tabella. Questo è un desiderabile per minimizzare le collisioni e garantire che lo spazio della tabella sia utilizzato in modo efficiente.

Dunque:

- $P(k)$ è la probabilità che una chiave specifica k sia presente nel dizionario. Se il dizionario contiene una certa chiave k , questa probabilità è $P(k)$, o $P(k) = 0$
- $h(k) = i$ questo indica che la chiave k viene mappata dalla funzione hash h nella cella i della tabella.
- La sommatoria $\sum_{k:h(k)=i}$ scorre su tutte le chiavi k che quando mappate dalla funzione hash finisco nella cella i , include quindi tutte le chiavi per le quali $h(k) = i$
- $Q(i)$ è la probabilità totale che la cella i sia occupata da almeno una chiave. Si ottiene sommando le probabilità $P(k)$ di tutte le chiavi k che vengono mappate a quella cella i .

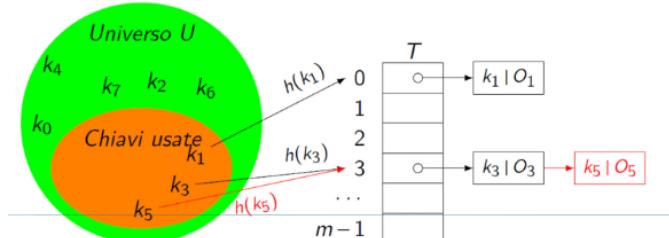
8.4 Sicurezza di funzioni hash crittografiche

- Resistenza alla collisione
- **Resistenza alla preimmagine:** dato un valore di hash x deve essere difficile risalire ad un messaggio m con $h(m) = x$, deriva dal concetto di unidirezionalità
- **Resistenza alla seconda preimmagine:** dato un input m_1 deve essere difficile trovare un secondo input m_2 tale che $h(m_1) = h(m_2)$, la resistenza alla collisione implica una resistenza alla seconda preimmagine

8.5 Risoluzione delle collisioni

nel caso in cui non si possano evitare le collisioni, dobbiamo trovare un modo per risolverle. Due metodi classici sono i seguenti:

- **Liste di collisione (chaining):** gli elementi sono contenuti in liste esterne alla tabella, $v[i]$ punta alla lista degli elementi tali che $h(k) = i$



- Tutti gli elementi con medesima chiave hash sono posti su una lista;
- **inserisci(k, o):** inserisci o nella lista $T[h(k)]$;
- **ricerca(k):** restituisci l'oggetto con chiave k nella lista $T[h(k)]$;
- **cancella(k):** cancella l'oggetto con chiave k nella lista $T[h(k)]$.

Implementazione:

classe **TavolaHashListeColl** implementa **Dizionario**:

$$S(m, n) = \Theta(m + n)$$

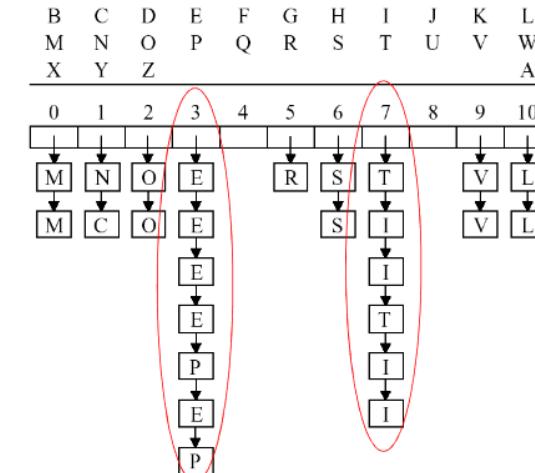
dati:
un array v di dimensione m in cui ogni cella contiene un puntatore a una lista di coppie $(elem, chiave)$. Un elemento e con chiave $k \in U$ è nel dizionario se e solo se (e, k) è nella lista puntata da $v[h(k)]$, con $h : U \rightarrow \{0, \dots, m-1\}$ funzione hash con uniformità semplice calcolabile in tempo $O(1)$.

operazioni:

insert($elem e, chiave k$)	$T(n) = O(1)$
aggiungi la coppia (e, k) alla lista puntata da $v[h(k)]$.	
delete($chiave k$)	$T_{avg}(n) = O(1 + n/m)$
rimuovi la coppia (e, k) nella lista puntata da $v[h(k)]$.	
search($chiave k \rightarrow elem$)	$T_{avg}(n) = O(1 + n/m)$
se (e, k) è nella lista puntata da $v[h(k)]$, allora restituisci e , altrimenti restituisci null.	

Esempio di tabella hash basata su liste di collisione contenente le lettere della parola: PRECIPITEVOLIS SIMEVOLMENTE

$$h(k) = \text{ascii}(k) \bmod 11, \quad \alpha = 26/11 = 2, 36$$



- **Indirizzamento aperto:** tutti gli elementi sono contenuti nella tabella, se una cella è occupata se ne cerca un'altra libera

Supponiamo di voler inserire un elemento con chiave k e la sua posizione naturale $h(k)$ sia già occupata (collisione). **Idea:** in caso di collisione si occupa un'altra cella della stessa tabella, la selezione della nuova cella avviene tramite una sequenza pre-stabilita di indici (**sequenza di scansione degli indici**) \Rightarrow dove tipicamente $c(k, 0) = h(k)$, dove $c(k, i)$ è detta **funzione di scansione** che determina la posizione associata a k , dopo che gli i tentativi precedenti hanno generato una collisione

$$c(k, 0), c(k, 1), \dots, c(k, m-1)$$

Implementazione:

classe TavolaHashApertaBis **implementa** Dizionario:
dati: un array v di dimensione m in cui ogni cella contiene una coppia $(elem, chiave)$.
 $S(m) = \Theta(m)$

operazioni:

```

insert(elem e, chiave k)
1.   for i = 0 to m - 1 do
2.     if (v[c(k, i)].elem = null or v[c(k, i)].elem = canc) then
3.       v[c(k, i)] ← (e, k)
4.     return
5.   errore tavola piena

delete(chiave k)
1.   for i = 0 to m - 1 do
2.     if (v[c(k, i)].elem = null) then
3.       errore chiave non in dizionario
4.     if (v[c(k, i)].chiave = k and v[c(k, i)].elem ≠ canc) then
5.       v[c(k, i)].elem ← canc
6.     errore chiave non in dizionario

search(chiave k) → elem
1.   for i = 0 to m - 1 do
2.     if (v[c(k, i)].elem = null) then
3.       return null
4.     if (v[c(k, i)].chiave = k and v[c(k, i)].elem ≠ canc) then
5.       return v[c(k, i)].elem
6.   return null

```

Indirizzamento aperto esempio: dove $c(k_5, 0) = 3$ indica che il primo tentativo di inserimento di k_5 avviene nella cella 3

$$h(k_1) = 0 \\ h(k_3) = h(k_5) = 3$$

0	$k_1 \dots$
1	$k_5 \dots$
2	
3	$k_3 \dots \quad k_5 + \dots$
...	
$m - 1$	

$$c(k_5, 0) = 3, \quad c(k_5, 1) = 1, \quad \dots, \quad c(k_5, m - 1) = m - 1$$

8.6 Sequenze di scansione

Il metodo di selezione delle celle di scansione determina il tipo di reindirizzamento:

- **Lineare:** in questo metodo se una chiave k viene mappata in una cella già occupata, si scansionano le celle successive una per una, fino a trovare una cella libera. Questa tecnica è definita dalla seguente funzione di scansione:

$$c(k, i) = (h(k) + i) \mod m, \quad 0 \leq i < m$$

$h(k)$ è l'indice iniziale determinato dalla funzione hash, i rappresenta il numero di tentativi di scansione, e m è la dimensione della tabella hash. Questo metodo è semplice da implementare, ma ha uno svantaggio: provoca effetti di **agglomerazione primaria** (primary clustering). L'agglomerazione si verifica quando si formano gruppi di celle consecutive occupate. Questi gruppi allungano la scansione, rallentando l'inserimento e la ricerca delle chiavi successive.

- **Hashing doppio:** in questo metodo, si utilizza una seconda funzione hash per determinare il passo di scansione *riducendo* significativamente *l'agglomerazione*. La funzione di scansione per l'hashing doppio è:

$$c(k, i) = \lfloor h_1(k) + i \cdot h_2(k) \rfloor \mod m, \quad 0 \leq i < m$$

Dove $h_1(k)$ è la funzione hash principale e $h_2(k)$ è la seconda funzione hash $h_2(k)$ deve essere scelta in modo che non produca mai zero per evitare cicli infiniti.

Questo metodo è molto efficiente in termini di riduzione delle collisioni e minimizzazione dell'agglomerazione sia primaria che secondaria $\Rightarrow m$ e $h_2(k)$ primi fra loro $MCD = 1$

Analisi del costo di scansione per la ricerca di una chiave: assumendo che le chiavi siano prese con probabilità uniforme da U

- **Tempo richiesto nel caso peggiore:** $O(n)$
- **Tempo richiesto nel caso medio:** dove $\alpha = \frac{n}{m} < 1$ (fattore di carico)

esito ricerca	sc. lineare	hashing doppio
chiave trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)}$	$-\frac{1}{\alpha} \log_e(1 - \alpha)$
chiave non trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$	$\frac{1}{1-\alpha}$

9 Ordinamento per confronto

Input: sequenza a_1, \dots, a_n di elementi su cui è definita una relazione d'ordine totale \leq . **Output:** a'_1, \dots, a'_n permutazione di a_1, \dots, a_n tale che $a'_1 \leq \dots \leq a'_n$

Come misurare la complessità temporale: numero confronti, numero operazioni (confronti + spostamenti), numero passi elementari (darà lo stesso risultato del precedente)

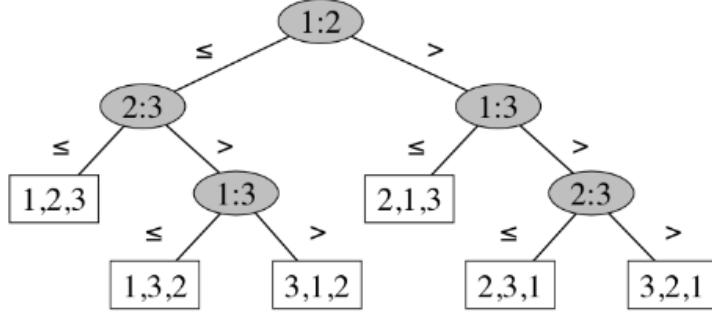
9.1 Algoritmi e tempi tipici

Tre tempi tipici: $O(n^2)$, $O(n \log n)$, $O(n)$, $O(n^2)$ è l'*upper bound* del problema perché corrisponde ad eseguire tutti i confronti

Lower bound: delimitazione inferiore alla quantità di una certa risorsa di calcolo necessaria per risolvere un problema, $\Omega(n \log n)$ è un lower bound al numero di confronti richiesti per ordinare n oggetti, considereremo un generico algoritmo \mathcal{A} che ordina eseguendo solo confronti: dimostreremo che \mathcal{A} esegue $\Omega(n \log n)$ confronti

<i>Istanza</i>	specifica sequenza di input, sequenza
\mathcal{A}	algoritmo di ordinamento

9.2 Alberi di decisione



Proprietà: per una particolare istanza i confronti eseguiti da \mathcal{A} su quella istanza rappresentano un cammino radice foglia \Rightarrow ossia per ogni specifica sequenza di input (istanza del problema) l'algoritmo di ordinamento \mathcal{A} segue un percorso specifico nell'albero di decisione, questo percorso o cammino inizia dalla radice dell'albero e termina in una foglia, il numero di confronti nel caso peggiore è pari all'**altezza dell'albero di decisione**, un albero di decisione per l'ordinamento di n elementi contiene almeno $n!$ foglie

Altezza in funzione delle foglie: un albero binario con k foglie ha altezza almeno $\log_2 k$, un albero binario completo ha 2^h foglie, dunque un albero binario ha al più 2^h foglie $\Rightarrow k \leq 2^h$ da cui $h \geq \log_2 k$ cioè $h = \Omega(\lg k)$

Teorema: $\Omega(n \log n)$ è un limite inferiore per la complessità del problema dell'ordinamento basato su confronti. **Dimostrazione:** siccome le permutazioni di $1, \dots, n$ sono $n!$ l'albero di decisione deve avere almeno $k = n!$ foglie, l'altezza dell'albero di

decisione è almeno $\lg k$ e cioè $h \geq \lg(n!)$ \Rightarrow dunque nel caso peggiore il generico algoritmo \mathcal{A} deve eseguire almeno $\lg(n!)$ confronti

Spiegazione: considerare il problema di ordinare n elementi distinti \Rightarrow esistono $n!$ possibili permutazioni (ordini) di questi n elementi, un algoritmo di ordinamento che si basa su confronti può essere rappresentato come un albero di decisione binario, in questo albero: ogni nodo interno rappresenta un confronto tra due elementi, ogni foglia rappresenta una delle possibili permutazioni ordinate dei n elementi. Poiché esistono $n!$ possibili permutazioni l'albero di decisione deve avere almeno $k = n!$ foglie perché deve poter rappresentare ogni possibile permutazione come un risultato finale di una sequenza di confronti, un albero completo con altezza h ha al più 2^h foglie e dunque per rappresentare tutte le $n!$ permutazioni deve valere $k \leq 2^h$, con $k = n!$ dunque otteniamo $n! \leq 2^h$, prendendo il logaritmo da entrambi le parti otteniamo $\lg(n!) \leq h$

$$k \leq 2^h$$

$$n! \leq 2^h$$

$$k = n!$$

\log_2 entrambe le parti

$$\lg(n!) \leq h$$

Dimostrazione (senza formula di stirling):

$$\begin{aligned} \log_2(n!) &\geq \log_2 \left(\left(\frac{n}{2} \right)^{n/2} \right) \\ &= \left(\frac{n}{2} \right) \log_2 \left(\frac{n}{2} \right) \\ &= \Omega(n \log n) \end{aligned}$$

Per ogni algoritmo generale di ordinamento \mathcal{A} possiamo concludere che $\Omega(n \log n)$ è un limite inferiore per la complessità del problema dell'ordinamento

MergeSort: algoritmo di ordinamento MergeSort risolve il problema dell'ordinamento con complessità

$$T_{\text{worst}}^{\text{MergeSort}}(n) = O(n \log n)$$

Dunque $O(n \log n)$ è anche un limite superiore per la complessità del problema di ordinamento. Siccome limite superiore e inferiore coincidono $O(n \log n)$ è un limite stretto per il problema dell'ordinamento

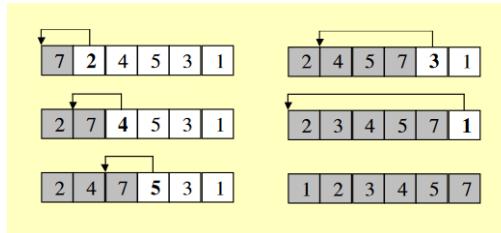
9.3 Ordinamenti quadratici

$O(n^2)$ confronti nel caso peggiore

- **InsertionSort:** approccio incrementale
- **SelectionSort:** approccio incrementale
- **BubbleSort**

9.3.1 InsertionSort

Approccio incrementale: estende l'ordinamento da $j-1$ a j elementi, posizionando l'elemento j -esimo nella posizione corretta rispetto ai primi $j-1$ elementi



InsertionSort (A)

1. **for** $j=2$ **to** n **do**
2. $x = A[j]$ //elemento da inserire nella sotto-seguenza ordinata $A[1..j-1]$
3. $i=j-1$ //indice di scansione da destra verso sinistra (da $j-1$ ad 1)
4. **while** $i>0$ e $A[i] > x$ **do**
5. $A[i+1]=A[i]$ //sposta di una posizione tutti gli elementi $A[i] > x$
6. $i=i-1$
7. $A[i+1]=x$ //Inserisce x nella locazione che gli compete

- **Analisi caso peggiore:** array ordinato in ordine decrescente, bisogna confrontare ogni elemento $A[j]$ per $j = 2, \dots, n$ con ogni elemento del sotto-array $A[1, \dots, j-1]$, ovvero le linee 4- 6 richiedono $j-1$ confronti nel caso peggiore

$$T_{worst}(n) = \sum_{j=2}^n j - 1 = O(n^2)$$

- **Analisi caso migliore:** array ordinato in ordine crescente, per ogni elemento $A[j]$ per $j = 2, \dots, n$ si trova che $A[i] \leq x$ quando $i = j-1$ (cioè al primo confronto), ovvero le linee 4-6 richiedono 1 solo confronto per un totale di $n-1$

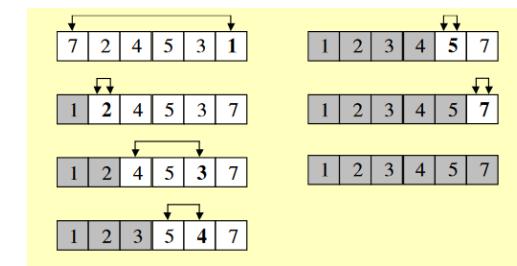
$$T_{best}(n) = \sum_{j=2}^n 1 = n - 1 = O(n)$$

- **Analisi del caso medio:** array in ordine sparso, per ogni elemento $A[j]$ per $j = 2, \dots, n$ mediamente metà elementi in $A[1, \dots, j-1]$ sono più piccoli di $A[j]$ e quindi si effettuano $(j-1)/2$ confronti

$$T_{avg}(n) = \sum_{j=2}^n \frac{j-1}{2} = O(n^2)$$

9.3.2 SelectionSort

Approccio incrementale: estende l'ordinamento da k a $k+1$ elementi, scegliendo il minimo degli $n-k$ elementi non ancora ordinati (caselle bianche) e mettendolo in posizione $k+1$



SelectionSort (A)

1. **for** $k=1$ **to** $n-1$ **do**
2. $m = k$ //indice del minimo elemento (inizializzato a k)
3. **for** $j=k+1$ **to** n **do** //ricerca del minimo in $A[k..n]$
4. **if** $A[j] < A[m]$ **then** $m = j$
5. **scambia** $A[m]$ con $A[k]$ se m è diverso da k

Analisi in tutti i casi: la k -esima estrazione di minimo richiede $n-k$ confronti, il ciclo più esterno è eseguito $n-1$ volte, cambiamento di variabile $i = n-k$, ossia $n = i+k$

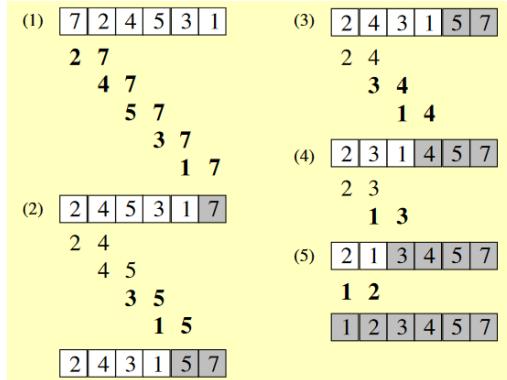
$$\sum_{k=1}^{n-1} n - k = \sum_{i=1}^{n-1} i = O(n^2)$$

Analisi spostamenti:

- **Caso peggiore:** l'elemento max è sempre nella prima posizione, ed i rimanenti sono in ordine, se in ogni k -esima iterazione il max è sempre nella k -esima posizione allora occorrono $n-1$ scambi (spostamenti) e quindi il numero di spostamenti è lineare: $O(n)$
- **Caso migliore:** array ordinato, nessun spostamento $O(1)$
- **Caso medio:** lineare $O(n)$

9.3.3 BubbleSort

Esegue una serie di scansioni dell'array: in ogni scansione confronta coppie di elementi adiacenti scambiandoli se non sono nell'ordine corretto, dopo una scansione in cui non viene effettuato nessuno scambio l'array è ordinato, dopo la k -esima scansione i k elementi più grandi sono correttamente ordinati ed occupano le k posizioni più a destra



BubbleSort (A)

```

1. for i=1 to n-1 do //i conta le scansioni
2.   scambiAvvenuti=false
3.   for j=2 to n-i+1 do //il max di A[1..n-i+1] è portato in n-i+1
4.     if A[j-1] > A[j]
5.       then scambia A[j] con A[j-1]; scambiAvvenuti=true;
6.   if not scambiAvvenuti then break

```

- **Analisi nel caso peggiore e medio:** array ordinato in ordine decrescente, cambiamento di variabile $j = n - i$, nella i -esima vengono eseguiti $n - i$ confronti

$$T_{avg}(n) = T_{worst}(n) = \sum_{i=1}^{n-1} n - i = \sum_{j=1}^{n-1} j = O(n^2)$$

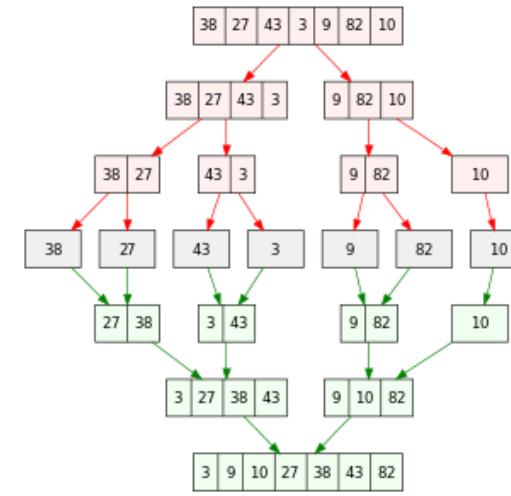
- **Analisi del caso migliore:** array ordinato in ordine crescente, il ciclo più esterno viene eseguito una volta, numero di spostamenti = 0 = $O(1)$

$$T_{best}(n) = n - 1 = O(n)$$

9.4 Ordinamenti ottimi

9.4.1 MergeSort

Usa la tecnica del divide et impera: divide l'array a metà (divide), risovi il sottoproblema ricorsivamente, fondi le due sottosequenze ordinate (impera). **Tempo esecuzione del MergeSort:** $O(n)$, dove n è il numero totale di elementi ($n = r - p + 1$)



MERGE-SORT(A, p, r)

```

1  if p < r
2    q = ⌊(p+r)/2⌋
3    MERGE-SORT(A, p, q)
4    MERGE-SORT(A, q + 1, r)
5    MERGE(A, p, q, r)

```

MERGE(A, p, q, r)

```

1  n1 = q - p + 1
2  n2 = r - q
3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4  for i = 1 to n1
5    L[i] = A[p + i - 1]
6  for j = 1 to n2
7    R[j] = A[q + j]
8  L[n1 + 1] = ∞
9  R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13   if L[i] ≤ R[j]
14     A[k] = L[i]
15     i = i + 1
16   else A[k] = R[j]
17     j = j + 1

```

Tempo di esecuzione del MergeSort: il numero di confronti del MergeSort è descritto dalla seguente relazione di ricorrenza per $n > 1$ ($T(1) = 1$):

$$T(n) = d(n) + 2 \cdot T(n/2) + c(n)$$

- $d(n)$: tempo necessario a dividere, $O(1)$
- $c(n)$: tempo necessario per combinare 2 sequenze ordinate di $n/2$ elementi, $O(n)$

Si ha $T(n) = 2 \cdot T(n/2) + f(n)$, con $f(n) = d(n) + c(n) = O(n)$, usando il teorema master caso 2 (a=b=2) si ottiene:

$$T(n) = O(n \log n)$$

9.4.2 QuickSort

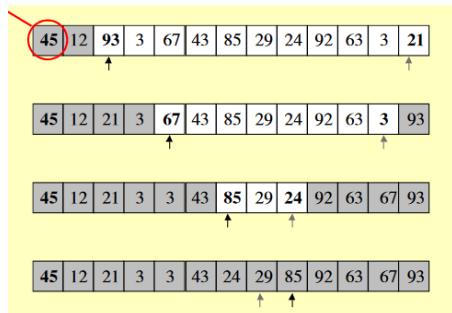
Usa la tecnica del divide et impera: scegli un elemento x della sequenza (pivot o perno) e partiziona la sequenza in elementi $\leq x$ ed elementi $> x$ (divide), risovi due sottoproblemi ricorsivamente, restituisce la concatenazione delle due sottosequenze ordinate (impera)

- QuickSort (non in loco)

QuickSort (A)

1. scegli un elemento x (il *pivot*) in A
2. partiziona A rispetto a x calcolando:
3. $A_1 = \{y \in A : y \leq x\}$
4. $A_2 = \{y \in A : y > x\}$
5. **if** ($|A_1| > 1$) **then** QuickSort(A_1)
6. **if** ($|A_2| > 1$) **then** QuickSort(A_2)
7. copia la concatenazione di A_1 e A_2 in A

Partizione in loco: scorri l'array in parallelo da sx verso dx e da dx verso sx, da sx verso dx ci si ferma su un elemento maggiore del perno, da dx verso sx ci si ferma su un elemento minore del perno, scambia li elementi e riprendi la scansione fino a quando gli indici non si incrociano. **Tempo di esecuzione (num. confronti):** $O(n)$, ogni elemento è confrontato con il pivot una sola volta quindi si hanno $n - 1$ confronti



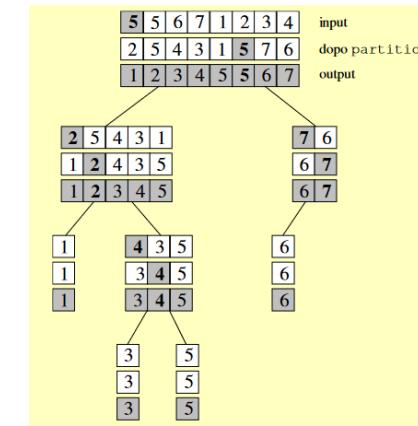
- Quicksort (in loco): tempo di esecuzione $O(n)$, restituisce l'indice del pivot (partition), m = indice del pivot

QuickSort (A, i, f)

1. **if** ($i \geq f$) **then return**
2. $m = Partition(A, i, f)$
3. QuickSort($A, i, m-1$)
4. QuickSort($A, m+1, f$)

Partition (A, i, f)

1. $x = A[i]$ //Partiziona $A[i..f]$ intorno al pivot $A[i]$
2. $inf = i$ $sup = f + 1$
3. **while** (true) **do**
4. **do** ($inf = inf + 1$) **while** ($inf \leq f$ and $A[inf] \leq x$)
5. **do** ($sup = sup - 1$) **while** ($A[sup] > x$)
6. **if** ($inf < sup$) **then** scambia $A[inf]$ e $A[sup]$
7. **else break**
8. scambia $A[i]$ e $A[sup]$
9. **return** sup



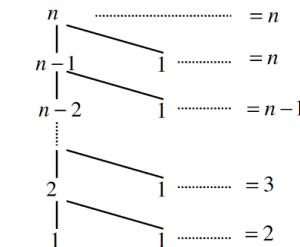
- **Analisi nel caso peggiore:** il perno scelto ad ogni passo è il minimo o il massimo degli elementi nell'array (ovvero quando l'array di partenza è ordinato o ordinato in senso inverso, il numero di confronti è pertanto:

$$T(n) = O(n) + T(1) + T(n-1) = O(n) + T(n-1)$$

svolgendo per iterazione si ottiene:

$$T(n) = O(n^2)$$

Albero di ricorsione dei costi (confronti)



$$T(n) = \sum_{i=2}^n i + n = O(n^2)$$

Randomizzazione: possiamo vitare il caso peggiore scegliendo come perno un elemento a caso

- **Analisi nel caso migliore:** il partizionamento produce un albero di ricorsione perfettamente bilanciato, due sottoalberi di dimensione non maggiore di $n/2$ ciascuno, si ha la ricorrenza del mergeSort che per il caso 2 del teorema master:

$$T(n) \leq 2 \cdot T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$$

- **Analisi nel caso medio:** la relazione di ricorrenza

$$C(n) = n - 1 + \sum_{a=0}^{n-1} \frac{2}{n} C(a)$$

ha soluzione $C(n) \leq 2n \log n$ quindi: $T(n) = O(n \log n)$

9.5 Riepilogo

- **Algoritmi elementari:** quadratici nel caso peggiore, ordinato in loco
 - InsertionSort
 - Selectionsort
 - BubbleSort
- **Algoritmi ottimi:** $O(n \log n)$ confronti
 - MergeSort (ma non ordina in loco), HeapSort
 - QuickSort (ordina in loco): $O(n \log n)$ solo nel caso medio, quadratico nel caso peggiore; altamente configurabile e per questo il più veloce e il più usato nella pratica

Nome	Migliore	Medio	Peggiori	Stabile	In loco	Caratteristiche dell'ordinamento
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	Sì	Sì	Algoritmo per confronto tramite scambio di elementi
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	Sì	Sì	A. per confronto tramite inserimento
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Sì	No	A. per confronto tramite unione di componenti, ottimale e facile da parallelizzare
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	No	Sì	A. per confronto tramite partizionamento. Le sue varianti possono: essere stabili
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Sì	A. per confronto tramite selezione di elementi

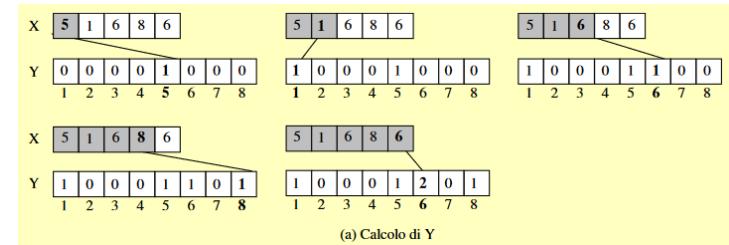
10 Ordinamento lineare

Per dati in input con proprietà particolari

- IntegerSort (o CountingSort)
- BucketSort
- RadixSort

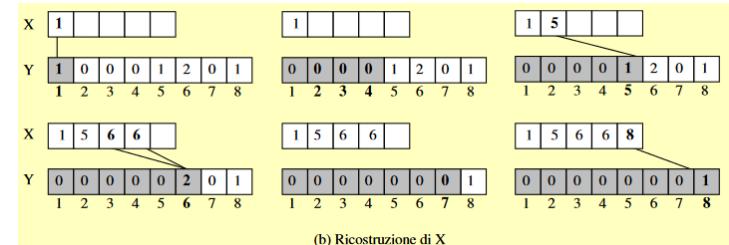
10.1 IntegerSort

Fase 1 (calcolo di Y): ordina un array X di n interi con valori in $[1, k]$ per k costante intera non troppo grande, mantiene un array Y di k contatori tale che $Y[i] =$ numero di volte che il valore i compare nell'array di input X



(a) Calcolo di Y

Fase 2 (ricostruzione di X): l'indice i di Y è il valore da ricopiare in X mentre l'elemento $y[i]$ è il numero di copie di i , scorre Y da sx a dx e se $Y[i] = c$ scrive in X il valore i per c volte



(b) Ricostruzione di X

Analisi: tempo $O(k)$ per inizializzare Y a 0, tempo $O(n)$ per calcolare i valori dei contatori, tempo $O(n + k)$ per ricostruire $X \Rightarrow O(n + k)$ tempo lineare se $k = O(n)$, contraddice il lower bound di $\Omega(n \log n)$? no perché l'integerSort non è un algoritmo basato su confronti

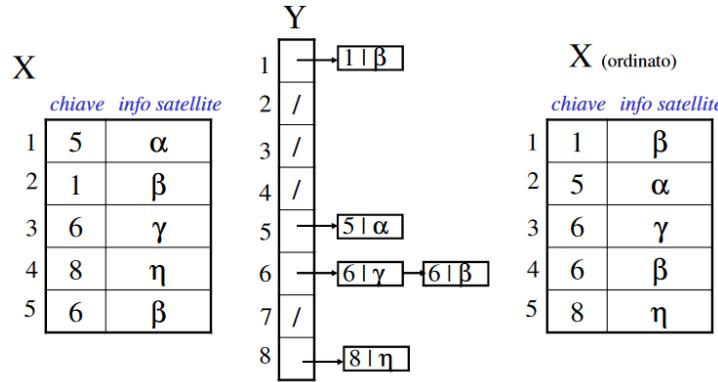
IntegerSort (X, k)

1. Sia Y un array di dimensione k //Fase 1: calcolo di Y
2. **for** $i=1$ **to** n **do** $Y[i]=0$ //inizializzazione di Y a 0
3. **for** $j=1$ **to** n **do** $Y[X[j]]++$
// $Y[i]$ ora contiene il num. di elementi in X uguali a i , $i=1..k$
4. $j=1$ //indice per scandire X
5. **for** $i=1$ **to** k **do** //Fase 2: ricostruzione di X a partire da Y
6. **while** ($Y[i] > 0$) **do** //considera solo i valori $Y[i]$ non nulli
7. $X[j]=i$
8. $j++$
9. $Y[i] = Y[i] - 1$

10.2 BucketSort

Per ordinare n record con chiavi intere in $[1, k]$, si potrebbe voler ordinare per matrizza o per anno di nascita

Input problema: n record mantenuti in un array, ogni elemento dell'array è un record con **campo chiave** (rispetto al quale ordinare), altri campi associati alla chiave (**informazione satellite**), per ordinare n record con chiavi intere in $[1, k]$ basta mantenere un array di liste anziché di contatori ed operare come per IntegerSort, la lista $Y[x]$ conterrà gli elementi con chiave uguale a x e concatenare poi le liste



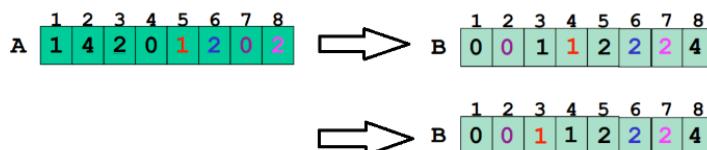
BucketSort: pseudocodice, in media tempo $O(n + k)$ ovvero $O(n)$ se $k = O(n)$

Caso peggiore: $O(n^2)$

BucketSort (X, k)

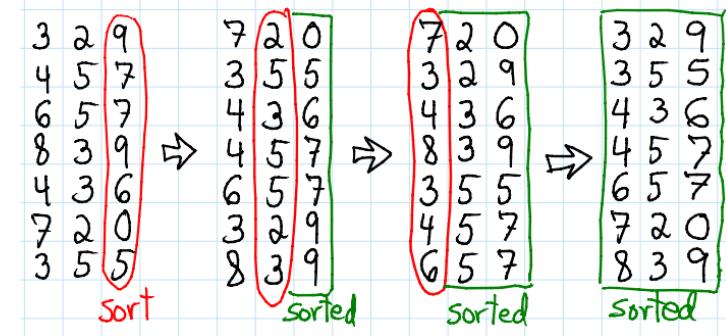
1. Sia Y un array di dimensione k
2. **for** $i=1$ to k **do** $Y[i] = \text{lista vuota}$ //Inizializzazione
3. **for** $j=1$ to n **do**
4. **if** ($\text{key}(X[j]) \notin [1, k]$) **then errore**
5. **else** appendi il record $X[j]$ alla lista $Y[\text{key}(X[j])]$
6. **for** $i=1$ to k **do** //Ordinamento classico a livello di bucket!
7. copia "ordinatamente" in X gli elementi della lista $Y[i]$

Stabilità: un algoritmo è stabile se preserva l'ordine iniziale tra elementi con la stessa chiave \Rightarrow il BucketSort è reso stabile appendendo gli elementi di X in coda alla opportuna lista $Y[i]$



10.3 RadixSort

Rappresentiamo i valori in una certa base b , ed eseguiamo una serie di BucketSort sulle cifre in modo **bottom-up**: dalla cifra meno significativa verso quella più significativa



Correttezza: se x e y hanno una diversa i -esima cifra la i -esima passata di BucketSort li ordina, se x e y hanno la stessa t -esima cifra la proprietà di stabilità del BucketSort li mantiene ordinati correttamente

Altre applicazioni: ordinamento tra stringhe, indicizzazione di testi, controllo copiature e plagio

Pseudocodice

```

procedura bucketSort(array A di n interi, interi b e t)
1. sia  $Y$  un array di dimensione  $b$ 
2. for  $i = 1$  to  $b$  do  $Y[i] \leftarrow \text{lista vuota}$ 
3. for  $i = 1$  to  $n$  do
4.      $c \leftarrow t\text{-esima cifra di } A[i] \text{ nella rappresentazione in base } b$ 
5.     appendi  $A[i]$  alla lista  $Y[c+1]$ 
6. for  $i = 1$  to  $b$  do
7.     copia ordinatamente in  $A$  gli elementi della lista  $Y[i]$ 

algoritmo radixSort(array A di n interi)
8.      $t \leftarrow 0$ 
9.     while (esiste un numero la cui  $t$ -esima cifra è  $\neq 0$ )
10.      bucketSort( $A, 10, t$ )
11.       $t \leftarrow t + 1$ 

```

Tempo di esecuzione: $O(\log_b k)$ passate di bucketsort per n interi in $[1, k]$, ciascuna passata richiede tempo $O(n + b)$

$$O((n+b) \log_b k)$$

Se $b = O(n)$ si ha $O(n \log_n k) = O\left(n \frac{\lg k}{\lg n}\right)$ (sapendo che $\log_2 k = \log_n k \cdot \log_2 n$) \Rightarrow tempo lineare se $k = O(n^c)$, con c costante

11 Programmazione dinamica

Divide et impera (tecnica top down): dividi l'istanza del problema in due o più sottoistanze, risovi ricorsivamente il problema sulle sottoistanze, ricombina la soluzione dei sottoproblemi allo scopo di ottenere la soluzione globale. Applicabile solo se: il numero di sotto-problemi da risolvere ricorsivamente è polinomiale nella dimensione dell'input, i sotto-problemi *sono indipendenti* \Rightarrow se ciò *non* è garantito si usano altre strategie come: **programmazione dinamica, golosa (greedy)**

Programmazione dinamica (tecnica bottom up): identificazione dei sottoproblemi del problema originario procedendo logicamente dai problemi più piccoli verso quelli più grandi, utilizza una **tabella delle soluzioni parziali** per memorizzare le soluzioni dei sottoproblemi, si usa quando i sottoproblemi *non sono indipendenti* (quando si presenta uno stesso sottoproblema è sufficiente recuperare la sua soluzione dalla tabella), può risultare onerosa perché risolve tutti i sottoproblemi in modo esaustivo

Esempi: numeri di fibonacci, distanza fra due stringhe

11.1 Distanza fra due stringhe

Come definire una distanza: *gaber e haber* sono stringhe molto simili (vicine), *zolle e abracadabra* sono molto diverse (lontane). **Edit distance:** numero minimo di modifiche elementari per trasformare una stringa in un'altra, usata nei correttori ortografici (spell checker), se una parola digitata non è nel dizionario sostituisce con la più vicina presente nel dizionario

Siano X e Y due stringhe di lunghezza m ed n , calcoliamo la distanza tra X e Y come minimo numero di operazioni elementari che trasformano X in Y scelte tra:

$$X = x_1 \cdot x_2 \cdots x_m \quad Y = y_1 \cdot y_2 \cdots y_n$$

- inserisci(a):** Inserisci il carattere a nella posizione corrente della stringa.
- cancella(a):** Cancelli il carattere a dalla posizione corrente della stringa.
- sostituisci(a, b):** Sostituisce il carattere a con il carattere b nella posizione corrente della stringa.

La distanza: è il numero minore di operazioni che trasforma la prima stringa nella seconda (e viceversa) \Rightarrow da *risotto* in *presto* la distanza è 4

Azione	Costo	Stringa ottenuta
Inserisco P	1	P RISOTTO
Mantengo R	0	PR ISOTTO
Sostituisco I con E	1	PRE SOTTO
Mantengo S	0	PRES OTTO
Cancello O	1	PRES TTO
Mantengo T	0	PREST TO
Cancello T	1	PREST O
Mantengo O	0	PRESTO

Approccio: denotiamo con $\delta(X, Y)$ la distanza tra X e Y , definiamo X_i il prefisso di X (i -esimo carattere incluso) per $0 \leq i \leq m$:

$$\begin{array}{ll} X_0 & \text{è la stringa vuota} \\ X_i = & x_1 \cdot x_2 \cdots x_i \text{ se } i \geq 1 \end{array}$$

Riduciamo il problema di calcolare $\delta(X, Y)$ al calcolo di $\delta(X_i, Y_j)$ per ogni i, j tali che $0 \leq i \leq m$ e $0 \leq j \leq n$, manteniamo le soluzioni parziali in una tabella D di dimensione $(m+1) \times (n+1)$

Inizializzazione della tabella: alcuni sottoproblemi sono molto semplici, $\delta(X_0, Y_j) = j$ partendo dalla stringa vuota X_0 basta inserire uno ad uno i j caratteri di Y_j , $\delta(X_i, Y_0) = i$ partendo da X_i basta rimuovere uno ad uno gli i caratteri per ottenere Y_0 , queste soluzioni sono memorizzate rispettivamente nella prima riga e prima colonna della tabella D

	P	R	E	S	T	O
0	1	2	3	4	5	6
R	1					
I	2					
S	3					
O	4					
T	5					
T	6					
O	7					

Avanzamento nella tabella $\delta(X_i, Y_j)$ per $i, j \geq 1$: se $x_i = y_j$ il minimo costo per trasformare X_i in Y_j è uguale al minimo costo per trasformare X_{i-1} in Y_{j-1}

$$D[i, j] = D[i - 1, j - 1]$$

se $x_i \neq y_j$ distinguiamo in base all'ultima operazione usata per trasformare X_i in Y_j in una sequenza ottima di operazioni:

- **inserisci(y_i):** il minimo costo per trasformare X_i in Y_j è uguale al minimo costo per trasformare X_i in $Y_{j-1} + 1$ per inserire il carattere y_j

$$D[i, j] = 1 + D[i, j - 1]$$

- **cancella(x_i):** il minimo costo per trasformare X_i in Y_j è uguale al minimo costo per trasformare X_{i-1} in $Y_j + 1$ per la cancellazione del carattere x_i

$$D[i, j] = 1 + D[i - 1, j]$$

- **sostituisci(x_i, y_i):** il minimo costo per trasformare X_i in Y_j è uguale al minimo costo per trasformare X_{i-1} in $Y_{j-1} + 1$ per sostituire il carattere x_i per y_j

$$D[i, j] = 1 + D[i - 1, j - 1]$$

In conclusione per $i, j \geq 1$:

$$D[i, j] = \begin{cases} D[i - 1, j - 1] & \text{se } x_i = y_j \\ 1 + \min\{D[i, j - 1], D[i - 1, j], D[i - 1, j - 1]\} & \text{se } x_i \neq y_j \end{cases}$$

Pseudocodice: tempo di esecuzione ed occupazione di memoria: $O(m \cdot n)$, se tengo traccia della cella della tabella che utilizzo per decidere il valore di $D[i, j]$ ho un metodo costruttivo per ricostruire la sequenza di operazioni di editing ottima.

```

algoritmo distanzaStringhe(stringa X, stringa Y) → intero
    matrice D di  $(m + 1) \times (n + 1)$  interi
    for i = 0 to m do  $D[i, 0] \leftarrow i$ 
    for j = 1 to n do  $D[0, j] \leftarrow j$ 
    for i = 1 to m do
        for j = 1 to n do
            if ( $x_i \neq y_j$ ) then
                 $D[i, j] \leftarrow 1 + \min\{D[i, j - 1], D[i - 1, j], D[i - 1, j - 1]\}$ 
            else  $D[i, j] \leftarrow D[i - 1, j - 1]$ 
    return  $D[m, n]$ 
```

		P	R	E	S	T	O
	0	1	2	3	4	5	6
R	1	1	1	2	3	4	5
I	2	2	2	2	3	4	5
S	3	3	3	3	2	3	4
O	4	4	4	4	3	3	3
T	5	5	5	5	4	3	4
T	6	6	6	6	5	4	4
O	7	7	7	7	6	5	4

NB: sono riportati altri esercizi in questa sezione e l'esempio dei numeri di fibonacci

12 Strategia Greedy

Tecnica golosa o greedy: per alcuni problemi di ottimizzazione la strategia dinamica può risultare onerosa perché risolve tutti i sottoproblemi in modo esaustivo, può in alcuni casi essere conveniente optare per una strategia greedy, strategia euristica *top-down* a problemi di ottimizzazione. **Idea:** per trovare una soluzione globalmente ottima scegli ripetutamente soluzioni ottime localmente, in questo modo per alcuni problemi si ottiene una soluzione globalmente ottima

Proprietà della strategia golosa:

- **Sottostruttura ottima**, ogni soluzione ottima non elementare si compone di soluzioni ottime di sottoproblemi
- **Proprietà della scelta golosa**, la scelta ottima localmente (golosa) non pregiudica la possibilità di arrivare ad una soluzione globalmente ottima

12.1 Elementi della strategia golosa

Vogliamo trovare la migliore soluzione a un problema, gli ingredienti sono: insieme di candidati, insieme dei candidati già esaminati, funzione obiettivo da minimizzare o massimizzare

- **Funzione ammissibile:** verifica se un insieme di candidati rappresenta una soluzione anche non ottima
- **Funzione ottimo:** verifica se un insieme di candidati rappresenta una soluzione ottima
- **Funzione seleziona:** indica quale dei candidati non ancora esaminati è al momento il più promettente (scelta greedy)

Pseudo codice generico per la tecnica golosa: goloso perché sceglie sempre il candidato più promettente

```

algoritmo paradigmaGreedy(insieme di candidati C) → soluzione
    S ← ∅
    while ((not ottimo(S)) and (C ≠ ∅)) do
        x ← seleziona(C)
        C ← C - {x}
        if (ammissibile(S ∪ {x})) then S ← S ∪ {x}
        if (ottimo(S)) then return S
        else errore non ho trovato soluzioni
```

12.2 Un problema di sequenziamento

Un server deve servire n clienti (n è fissato), il servizio richiesto dall' i -esimo cliente richiede t_i secondi per essere eseguito (**tempo di servizio**), chiamiamo $T(i)$ il tempo di attesa del cliente $i \Rightarrow$ vogliamo minimizzare il tempo di attesa medio

$$T_{avg} = \frac{T}{n} = \frac{1}{n} \sum_{i=1}^n T(i) \quad \Rightarrow \quad T = \sum_{i=1}^n T(i)$$

Avendo: $t_1 = 50$, $t_2 = 100$, $t_3 = 3$ [msec]. Dei sei possibili ordinamenti, il migliore è evidenziato in arancione

Ordine	T		
1 2 3	$50 + (50 + 100) + (50 + 100 + 3)$	msec	= 353 msec
1 3 2	$50 + (50 + 3) + (50 + 3 + 100)$	msec	= 256 msec
2 1 3	$100 + (100 + 50) + (100 + 50 + 3)$	msec	= 403 msec
2 3 1	$100 + (100 + 3) + (100 + 3 + 50)$	msec	= 356 msec
3 1 2	$3 + (3 + 50) + (3 + 50 + 100)$	msec	= 209 msec
3 2 1	$3 + (3 + 100) + (3 + 100 + 50)$	msec	= 259 msec

12.3 Un algoritmo goloso

Il seguente algoritmo genera l'ordine di servizio in maniera incrementale secondo una strategia greedy, supponiamo di aver deciso di sequenziare i clienti i_1, \dots, i_m , se adesso decidiamo di servire il cliente j il tempo totale di servizio diventa:

$$t_{i_1} + t_{i_2} + \dots + t_{i_m} + t_j$$

Scelta greedy: al generico passo j l'algoritmo goloso serve la richiesta più breve tra quelle rimanenti

```

algoritmo sequenziamento(array C)->soluzione
    S={}
    C = tempi (durata)  $t_j$  per  $j=1..n$  dei servizi richiesti
    Ordina C in modo non decrescente
    for  $j=1$  to n do
        S = S U {j}
    return S

```

Tempo di esecuzione: $O(n \log n)$, per ordinare le richieste

12.4 Il problema del distributore automatico di resto

NB: una strategia greedy non sempre garantisce l'ottimalità della soluzione prodotta

Come esempio: consideriamo il problema dei distributori automatici di restituire un certo resto R usando il minor numero di monete, supponiamo di avere un certo numero di monete da 1,5,10,20 e 50 centesimi di euro.

Formulazione del problema con strategia greedy:

- insieme C dei candidati è dato dall'insieme finito di monete
- **Funzione obiettivo:** numero di monete della soluzione
- **Funzione ammissibile:** true se il valore delle monete scelte non è superiore al resto R da restituire
- **Funzione seleziona:** la moneta di valore più grande tra quelle rimaste nel serbatoio C
- **Funzione ottimo:** true se il valore delle monete scelte è esattamente uguale al resto

```

algoritmo distribuisciResto(resto R) -> soluzione
    1.   C ← monete contenute nel serbatoio del distributore
    2.   S ← ∅
    3.   while ((valore(S) ≠ R) and (C ≠ ∅)) do
    4.       x ← moneta di valore più elevato in C
    5.       C ← C - {x}
    6.       if (valore(S ∪ {x}) ≤ R) then S ← S ∪ {x}
    7.       if (valore(S) = R) then return S come resto esatto
    8.       else return S come resto parziale

```

NB: non sempre l'algoritmo distribuisciResto è in grado di restituire il resto esatto

- **Es positivo:** $C = \{50, 50, 20\}$ ed $R = 70 : S = \{50, 20\}$, $\text{valore}(S) = 70 = R$
- **Es negativo:** $C = \{50, 20, 20, 20, 5\}$ ed $R = 65 : S = \{50, 5\}$, $\text{valore}(S) = 55 \neq R$

L'errore sta al primo passo: viene fatta la scelta sbagliata $x = 50$ che non potrà mai essere disfatta; non utilizzando invece la moneta da 50, potremmo restituire il resto esatto

12.5 Riepilogo

- **Divide et impera:** altre applicazioni nella ricerca binaria, mergesort, quicksort, moltiplicazione di matrici
- **Programmazione dinamica:** altre applicazioni nel calcolo dei numeri di Fibonacci, associatività del prodotto tra matrici, cammini minimi tra tutte le coppie di nodi (algoritmo di Floyd e Warshall)
- **Tecnica greedy:** altre applicazioni nel distributore automatico di resto, nel calcolo del minimo albero ricoprente (algoritmi di Prim, Kruskal, Boruvka) e dei cammini minimi a sorgente singola (algoritmo di Dijkstra)

13 Grafi e visite di grafi

Definizione: struttura dati non lineare e non di tipo gerarchico, un grafo $G = (V, E)$ consiste in un insieme di **vertici** o nodi V , un insieme $E \subseteq V \times V$ di coppie di vertici (relazione) detti **archi** o **spigoli**, ogni arco connette due vertici

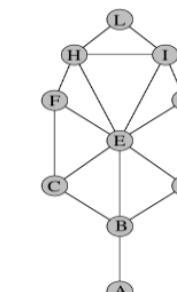
Grafo orientato: gli archi hanno una freccia che indica il verso, mentre un **grafo non orientato** non presenta alcun verso e l'arco prende il nome di **lato**

Definizioni: un **cappio** è un arco i cui estremi coincidono, un grafo non orientato è **semplice** se non ha cappi e non ci sono due archi con gli stessi estremi in caso contrario si parla di **multigrafo** ⇒ salvo indicazione contraria noi assumeremo sempre che un grafo sia semplice

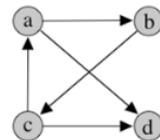
13.1 Terminologia

- L e I sono **adiacenti** (o vicini)
- Il lato (L, I) è **incidente** su L e su I
- I ha **grado** δ (numero archi incidenti): $4, \delta(I) = 4, n = |V|$ numero vertici
- $m = |E|$ numero di archi, varia da 0 a $|V|^2 - |V|$
- $< L, H, I, C, B, A >$ è un **cammino semplice** (tutti i vertici sono distinti) nel grafo di **lunghezza** (num. archi) 5, non è il più corto cammino tra L e A
- La lunghezza del più corto cammino tra due vertici si dice **distanza**: L ed A hanno distanza 4
- $< L, H, I, L >$ è un **ciclo semplice** di lunghezza 3
- Un cappio (u, u) è un ciclo di lunghezza 1
- Sommando i gradi di tutti i vertici contiamo ogni arco esattamente due volte:

$$\sum_{v \in V} \delta(v) = 2m$$



- b è **adiacente** (o vicino) ad a
- (a, b) **esce** dal vertice a ed **entra** in b
- a ha **grado** (num. archi entranti ed uscenti) 3: $\delta(a) = \delta_{in}(a) + \delta_{out}(a) = 3$
- Ogni arco (u, v) è entrante per v ed uscente per u : $\sum_{v \in V} \delta(v) = 2m$



13.2 Componenti connesse di un grafo non orientato

Una **componente连通** di G è un insieme massimale di vertici $U \subseteq V$ t.c. per ogni coppia di vertici in U esiste in G un cammino che li collega

Un grafo $G = (V, E)$ non orientato è **connesso** se esiste almeno un cammino tra ogni coppia di vertici o anche se esiste un'unica componente连通. Questo è equivalente a dire che il grafo ha una sola componente连通, ossia l'insieme di tutti i vertici del grafo.

La relazione di raggiungibilità tra vertici in un grafo non orientato ha le seguenti proprietà. **Riflessiva**: ogni vertice è raggiungibile da sé stesso. **Simmetrica**: se un vertice u è raggiungibile da un vertice v , allora anche v è raggiungibile da u . **Transitiva**: se un vertice u è raggiungibile da v e v è raggiungibile da w , allora anche u è raggiungibile da w .

E' facile verificare che la relazione tra vertici **essere raggiungibile da** è di equivalenza (gode della proprietà riflessiva, simmetrica e transitiva)

Ne segue che: le componenti connesse di un grafo non orientato sono le classi di equivalenza dei suoi vertici rispetto alla relazione di raggiungibilità. Ciascuna classe di equivalenza rappresenta una componente连通, ovvero un insieme di vertici tra cui esiste un cammino che li collega.

13.3 Componenti fortemente connesse di un grafo orientato

Una **componente fortemente connessa** di G è un insieme massimale di vertici $U \subseteq V$ t.c. per ogni coppia di vertici u e v in U esiste in G un cammino orientato da u a v e da v ad u che li collega

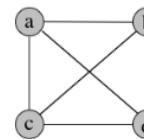
Un grafo $G = (V, E)$ orientato è **fortemente connesso** se esiste almeno un cammino (orientato) tra ogni coppia di vertici

Le componenti fortemente connesse sono le classi di equivalenza dei vertici rispetto alla relazione di equivalenza di connettività forte

13.4 Grafi, alberi e DAG

Osservazioni: gli alberi sono casi particolari di grafi, un albero è un **grafo non orientato connesso e aciclico** (senza cicli), un **DAG** (direct acyclic graph) è un grafo orientato ed aciclico

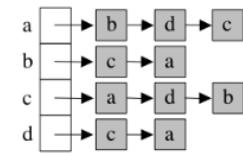
13.5 Grafi non orientati



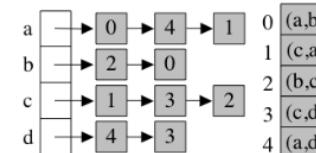
(a) Grafo non orientato G

(a,b)
(c,a)
(b,c)
(c,d)
(a,d)

$O(m+n)$
(b) Lista di archi di G



$O(m+n)$
(c) Liste di adiacenza di G



$O(m+n)$
(d) Liste di incidenza di G

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

$O(n^2)$
(e) Matrice di adiacenza di G

(a,b)	(c,a)
(b,c)	(c,d)
(c,d)	(a,d)
(a,d)	

$O(n m) = O(n^3)$
(f) Matrice di incidenza di G

13.6 Prestazioni della lista di archi

Operazione	Tempo di esecuzione
<code>grado(v)</code>	$O(m)$
<code>archiIncidenti(v)</code>	$O(m)$
<code>sonoAdiacenti(x, y)</code>	$O(m)$
<code>aggiungiVertice(v)</code>	$O(1)$
<code>aggiungiArco(x, y)</code>	$O(1)$
<code>rimuoviVertice(v)</code>	$O(m)$
<code>rimuoviArco(e)</code>	$O(m)$

13.7 Prestazioni delle liste di adiacenza e di incidenza

Operazione	Tempo di esecuzione
<code>grado(v)</code>	$O(\delta(v))$
<code>archiIncidenti(v)</code>	$O(\delta(v))$
<code>sonoAdiacenti(x, y)</code>	$O(\min\{\delta(x), \delta(y)\})$
<code>aggiungiVertice(v)</code>	$O(1)$
<code>aggiungiArco(x, y)</code>	$O(1)$
<code>rimuoviVertice(v)</code>	$O(m)$
<code>rimuoviArco(e = (x, y))</code>	$O(\delta(x) + \delta(y))$

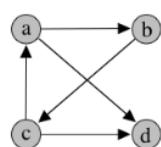
13.8 Prestazioni della matrice di adiacenza

Operazione	Tempo di esecuzione
<code>grado(v)</code>	$O(n)$
<code>archiIncidenti(v)</code>	$O(n)$
<code>sonoAdiacenti(x,y)</code>	$O(1)$
<code>aggiungiVertice(v)</code>	$O(n^2)$
<code>aggiungiArco(x,y)</code>	$O(1)$
<code>rimuoviVertice(v)</code>	$O(n^2)$
<code>rimuoviArco(e)</code>	$O(1)$

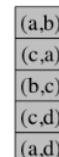
13.9 Prestazioni della matrice di incidenza

Operazione	Tempo di esecuzione
<code>grado(v)</code>	$O(m)$
<code>archiIncidenti(v)</code>	$O(m)$
<code>sonoAdiacenti(x,y)</code>	$O(m)$
<code>aggiungiVertice(v)</code>	$O(nm)$
<code>aggiungiArco(x,y)</code>	$O(nm)$
<code>rimuoviVertice(v)</code>	$O(nm)$
<code>rimuoviArco(e)</code>	$O(n)$

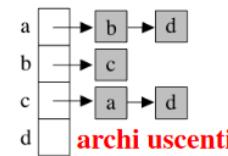
13.10 Grafi orientati



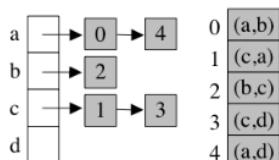
(a) Grafo orientato G



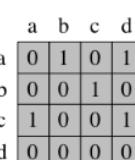
(b) Lista di archi di G



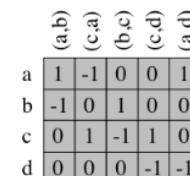
(c) Liste di adiacenza di G



(d) Liste di incidenza di G



(e) Matrice di adiacenza di G



(f) Matrice di incidenza di G

13.11 Visite di grafi

Una visita (o attraversamento) di un grafo G permette di esaminare i nodi e gli archi di G in modo sistematico a partire da un *vertice sorgente* s , esistono vari tipi di visite con diverse proprietà: *visita in ampiezza* ($BFS=breadth first search$) e visita in *profondità* ($DFS=depth first search$)

Osservazioni: un vertice viene *marcato* quando viene incontrato per la prima volta, la marcatura può essere mantenuta tramite un vettore di bit di marcatura, la visita genera un *albero di copertura T* del grafo radicato in s

Un insieme di vertici $F \subseteq T$ mantiene la *frangia* di T : $v \in F : v$ è *aperto* esistono archi incidenti su v non ancora esaminati, $v \in T - F : v$ è *chiuso* tutti gli archi incidenti su v sono stati esaminati:

- se la frangia F è implementata come *coda* si ha la *visita in ampiezza (BFS)*
- Se la frangia F è implementata come *pila* si ha la *visita in profondità (DFS)*

Costo della visita: il tempo di esecuzione dipende dalla struttura dati usata

- *Lista di archi*: $O(m \cdot n)$, invochiamo l'operazione `archiIncidenti(v)`, che richiede l'esame di tutti gli archi del grafo, per ogni vertice v
- *Liste di adiacenza o di incidenza*: $O(m + n)$, la somma su tutti i vertici delle lunghezze delle liste di adiacenza è $2m$
- *Matrice di adiacenza*: $O(n^2)$, l'operazione `archiIncidenti(v)` richiede la scansione di una intera riga e quindi un tempo $O(n)$ per ogni vertice v
- *Matrice di incidenza*: $O(m \cdot n)$, l'operazione `archiIncidenti(v)` richiede $O(m)$ per ogni vertice v

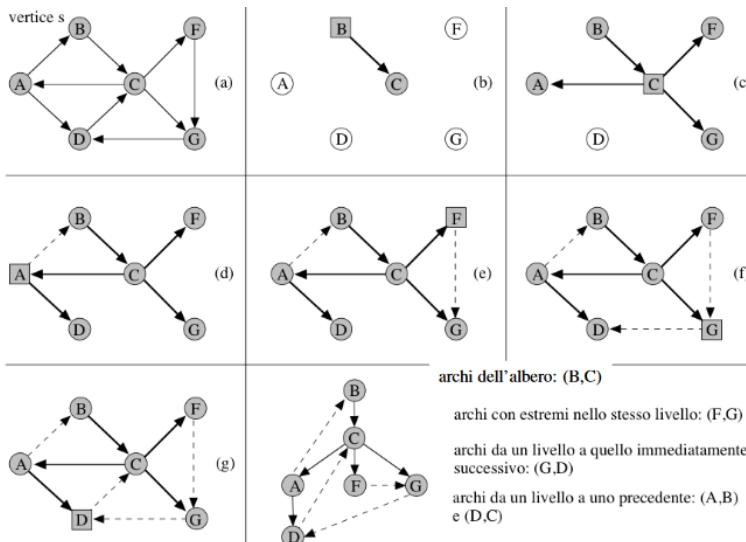
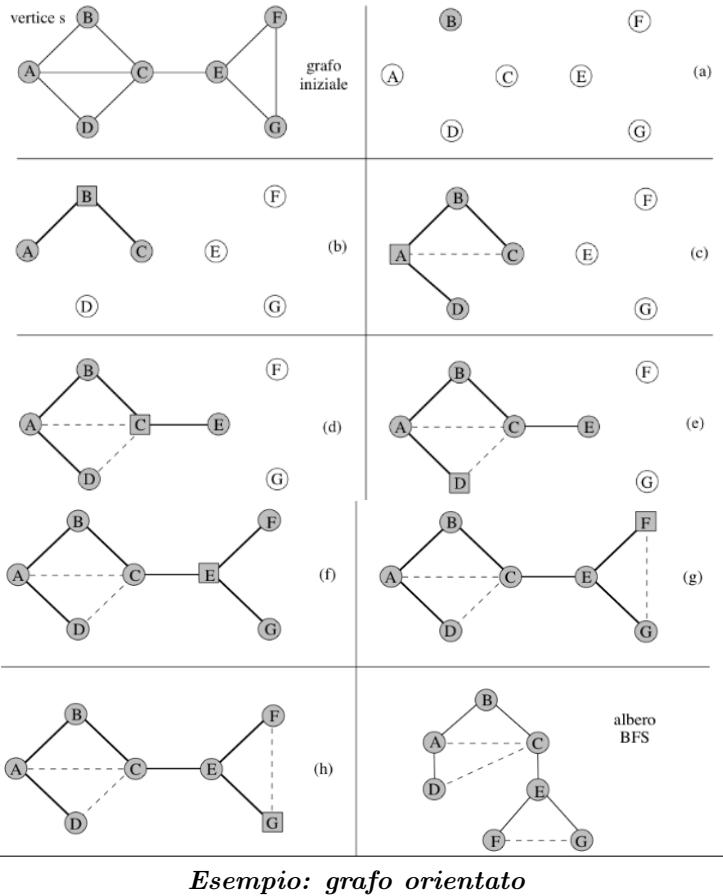
<i>Lista di archi</i>	$O(m \cdot n)$
<i>Lista di adiacenza o di incidenza</i>	$O(m + n)$
<i>Matrice di adiacenza</i>	$O(n^2)$
<i>Matrice di incidenza</i>	$O(m \cdot n)$

13.12 Visita in ampiezza

algoritmo `visitaBFS(vertice s) → albero`

1. rendi tutti i vertici non marcati
2. $T \leftarrow$ albero formato da un solo nodo s
3. Coda F
4. marca il vertice s
5. $F.\text{enqueue}(s)$
6. **while** (*not* $F.\text{isempty}()$) **do**
7. $u \leftarrow F.\text{dequeue}()$
8. **for each** (arco (u,v) in G) **do**
9. *if* (v non è ancora marcato) *then*
10. $F.\text{enqueue}(v)$
11. marca il vertice v
12. rendi u padre di v in T
13. **return** T

Osservazioni: gli archi incidenti in v possono essere esaminati in qualsiasi ordine, nell'albero BFS ogni vertice si trova il più vicino possibile alla radice s



Esempio 1: archi dell'albero (B, A), archi tra vertici dello stesso livello (A, C), archi tra livelli consecutivi (C, D)

Livello di un nodo nell'albero BFS: quando eseguiamo una BFS a partire da una sorgente s , il livello di un nodo v nell'albero BFS è definito come il numero minimo di archi che bisogna attraversare per andare da s a v . Questo numero rappresenta anche la distanza di v dalla sorgente s . **Sono allo stesso livello:** questo accade quando sia u che v sono alla stessa distanza dalla sorgente s . **Sono a livelli consecutivi:** questo accade quando v si trova ad una distanza dalla sorgente s maggiore di 1 rispetto a u . In altre parole, v è esattamente un passo in più distante da s rispetto a u , quindi si trovano in livelli consecutivi nell'albero BFS.

Proprietà: per ogni nodo v il livello di v nell'albero BFS è pari alla distanza di v dalla sorgente s , per ogni arco (u, v) di un grafo non orientato gli estremi u e v appartengono allo stesso livello o a livelli consecutivi dell'albero BFS, se il grafo è orientato possono esistere archi (u, v) che attraversano all'indietro più di un livello

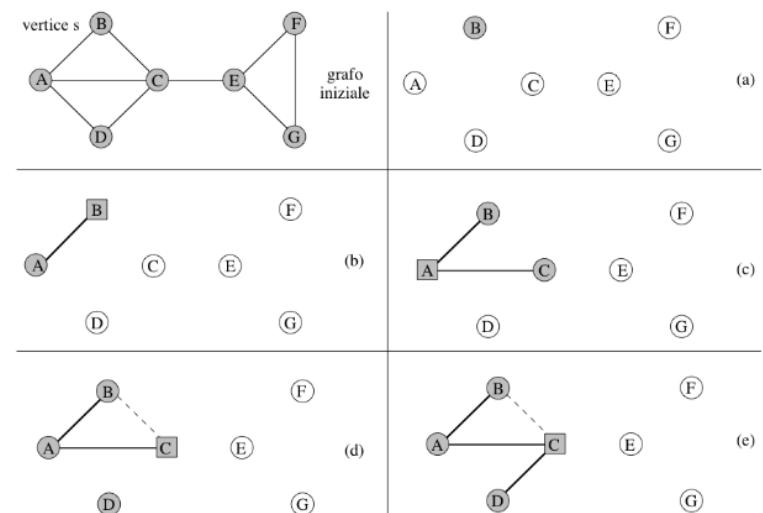
13.13 Visita in profondità

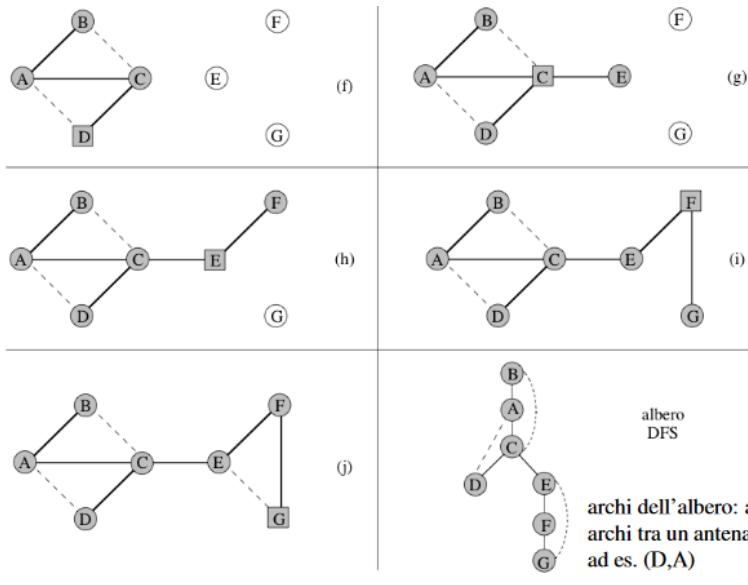
procedura visitaDFSRicorsiva(vertice v , albero T)

1. marca e visita il vertice v
2. **for each** (arco (v, w)) **do**
3. **if** (w non è marcato) **then**
4. aggiungi l'arco (v, w) all'albero T
5. visitaDFSRicorsiva(w, T)

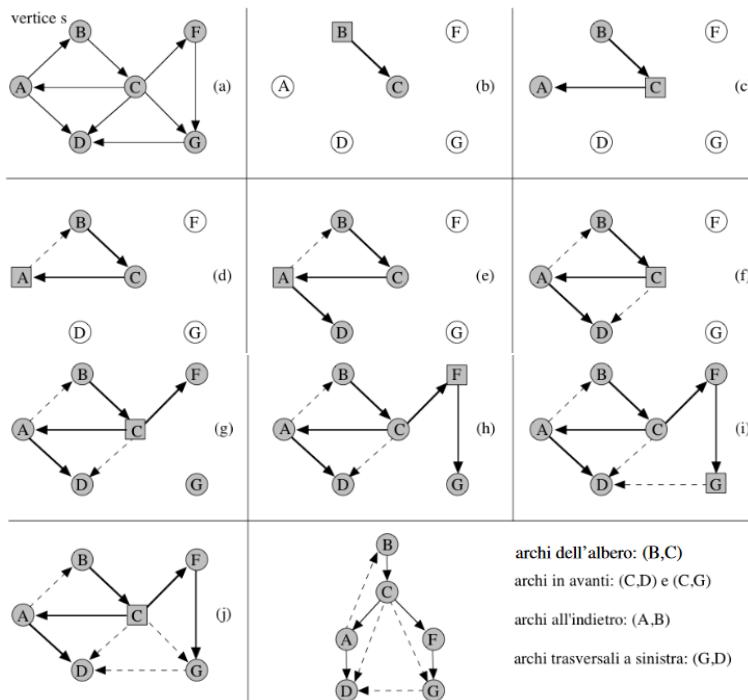
algoritmo visitaDFS(vertice s) → albero

6. $T \leftarrow$ albero vuoto
7. visitaDFSRicorsiva(s, T)
8. **return** T





Esempio: grafo orientato



Proprietà

- Sia (u, v) un arco di un grafo **non orientato**, allora: (u, v) è un arco dell'albero DFS, oppure i nodi uv sono l'uno discendente/antenato dell'altro, oppure archi in avanti e archi all'indietro
- Sia (u, v) un arco di un grafo **orientato**, allora: (u, v) è un arco dell'albero DFS, oppure i nodi u e v sono l'uno discendente/antenato dell'altro, oppure archi in avanti e archi all'indietro, (u, v) è un arco **trasversale a sx** ovvero il vertice v è in un sottoalbero visitato precedentemente ad u

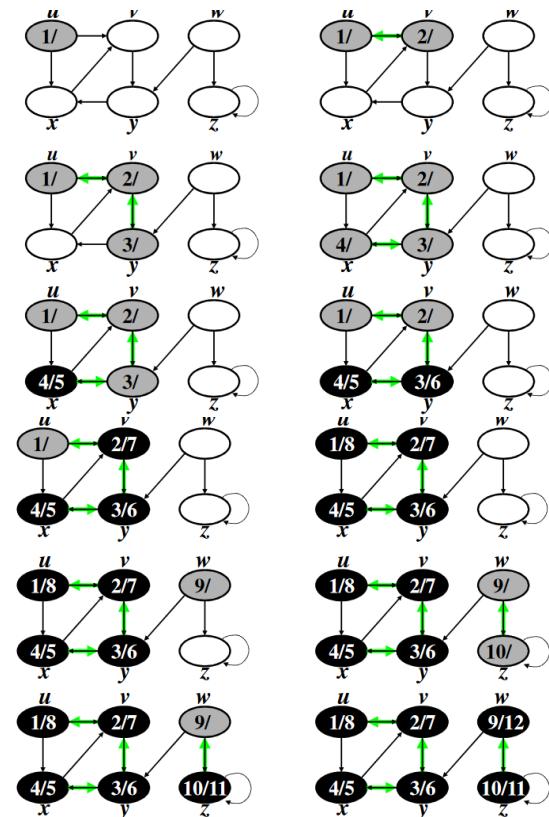
La variabile t utile per segnare i marcatempi inizio e fine visita

procedura visitaDFSRicorsiva(vertice v , albero T)

1. *marca e visita il vertice v* $inizio[v] \leftarrow t++$
2. **for each** (arco (v, w)) **do**
3. **if** (w non è marcato) **then**
4. aggiungi l'arco (v, w) all'albero T
5. visitaDFSRicorsiva(w, T)
6. **fine**[v] $\leftarrow t++$

algoritmo visitaDFS(vertice s) \rightarrow albero

6. $T \leftarrow$ albero vuoto $t \leftarrow 1$
7. visitaDFSRicorsiva(s, T)
8. **return** T



14 Cammini minimi

Sia $G = (V, E)$ un **grafo orientato** a cui archi (u, v) è associato un **costo** o **peso** $W(u, v)$ (numero reale). Il **costo di un cammino** $p = (v_0, \dots, v_k)$ è la somma dei costi degli archi che lo costituiscono:

$$W(p) = \sum_{i=1}^k W(v_{i-1}, v_i)$$

Il **costo di cammino minimo** da un vertice u ad un vertice v è definito nel seguente modo:

$$\delta(u, v) = \begin{cases} \min\{W(p)\} & \text{se esistono cammini } p \text{ da } u \text{ a } v \\ \infty & \text{altrimenti} \end{cases}$$

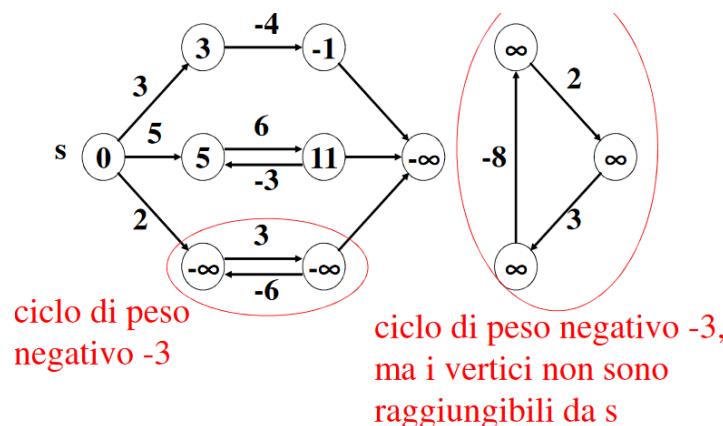
Un **cammino minimo** da u a v è un cammino p da u a v di costo $W(p) = \delta(u, v)$. Nel problema dei cammini minimi viene appunto richiesto di calcolare i cammini minimi. Vi sono quattro versioni del problema:

- Cammini minimi da un'unica sorgente a tutti gli altri vertici
- Cammini minimi da ogni vertice ad un'unica destinazione
- Cammini minimi da un'unica sorgente ad un'unica destinazione
- Cammini minimi tra tutte le coppie : da ogni vertice ad ogni altro vertice

Noi risolveremo la prima variante, la seconda variante si risolve simmetricamente, la terza si può risolvere usando la soluzione della prima, la quarta si può risolvere usando la soluzione della prima per ogni vertice del grafo ma in genere si può fare di meglio.

Archi di costo negativo: in alcuni casi il costo degli archi può essere negativo, questo non crea problemi nella ricerca dei cammini minimi da una sorgente s a meno che vi siano cicli di costo negativo raggiungibili da s . Se u è un vertice raggiungibile da s con un cammino p passante per un vertice v di un ciclo negativo allora esistono cammini da s a u di costi sempre minori e il costo di cammino minimo $\delta(s, u)$ non è definito \Rightarrow in questo caso poniamo $\delta(s, u) = -\infty$.

Esempio: i costi minimi da s sono riportati all'interno dei vertici:



Rappresentazione dei cammini minimi: in generale ci interessa calcolare non solo i costi dei cammini minimi dalla sorgente s ad ogni vertice del grafo ma anche i cammini minimi stessi, siccome i cammini minimi hanno sottostruttura ottima possiamo rappresentarli aumentando ogni vertice con un puntatore $p[v]$ che punta al vertice precedente in un cammino minimo da s a v .

Teorema (sottostruttura ottima dei cammini minimi): se il cammino $p = (v_0, \dots, v_k)$ è minimo allora sono minimi anche tutti i sottocammini $p_{ij} = (v_i, \dots, v_j)$ per $0 \leq i \leq j \leq k$.

Dimostrazione: per assurdo, se esistesse un cammino q da v_i a v_j di costo minore di p_{ij} allora sostituendo nel cammino p il sottocammino p_{ij} con il cammino q si otterebbe un cammino da v_0 a v_k di costo minore di p . Impossibile se p è minimo

Corollario (scomposizione dei costi di cammino minimo): se p è un cammino minimo da s ad un vertice v diverso da s ed u è il vertice che precede v nel cammino allora $\delta(s, v) = \delta(s, u) + W(u, v)$

Dimostrazione: conseguenza della sottostruttura ottima

$$\delta(s, v) = W(p) = \delta(s, u) + W(u, v)$$

Lemma (limite superiore per i costi di cammino minimo): per ogni arco (u, v) vale la disegualanza

$$\delta(s, v) \leq \delta(s, u) + W(u, v)$$

Dim: se u non è raggiungibile da s allora $\delta(s, u) = \infty$ e $\delta(s, v) \leq \infty + W(u, v)$, se u è raggiungibile da s allora $\delta(s, u) + W(u, v)$ è il costo di un cammino da s a v ed è quindi maggiore o uguale di $\delta(s, v)$

14.1 Rilassamento continuo

Tecnica del rilassamento: gli algoritmi che studieremo per il problema dei cammini minimi usano la tecnica del rilassamento. Aggiungiamo ad ogni vertice v del grafo un campo $d[v]$ che rappresenta una **stima di cammino minimo**: durante tutta l'esecuzione dell'algoritmo è un limite superiore per $\delta(s, v)$ mentre alla fine è proprio uguale a $\delta(s, v)$

L'inizializzazione dei campi $p[v]$ e $d[v]$ è la stessa per tutti gli algoritmi

Inizializza(G, s, d, p) for ogni $v \in V[G]$ do $p[v] \leftarrow \text{nil}$ $d[v] \leftarrow \infty$ $d[s] \leftarrow 0$	G grafo pesato sugli archi
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------

Il rilassamento di un arco (u, v) consiste nel controllare se è possibile migliorare il cammino finora trovato per v (e quindi la stima $d[v]$) allungando il cammino trovato per u con l'arco (u, v) .

```

Rilassa( $G, u, v, d, p, W$ )
  if  $d[v] > d[u] + W(u,v)$  then
     $d[v] \leftarrow d[u] + W(u,v)$ 
     $p[v] \leftarrow u$ 

```

14.2 Proprietà del rilassamento

- **Lemma 1 (effetto del rilassamento):** dopo aver eseguito $Rilassa(G, u, v)$ vale la diseguaglianza

$$d[v] \leq d[u] + W(u, v)$$

Ovvvero le stime $d[v]$ sono monotone non crescenti.

Dimostrazione: se $d[v] > d[u] + W(u, v)$ prima del rilassamento viene posto $d[v] = d[u] + W(u, v)$, se $d[v] \leq d[u] + W(u, v)$ prima del rilassamento non viene fatto nulla e quindi è vero anche dopo

- **Lemma 2 (invariante del rilassamento):** Dopo l'inizializzazione per ogni vertice v vale la diseguaglianza $d[v] \geq \delta(s, v)$ che rimane vera anche dopo un numero qualsiasi di rilassamenti. Inoltre se a un certo punto $d[v] = \delta(s, v)$ il suo valore non può più cambiare.

Dimostrazione: dopo l'inizializzazione $d[s] = 0 \geq \delta(s, s)$ e per ogni altro vertice $d[v] = \infty \geq \delta(s, v)$. Se $d[v]$ non viene modificata durante l'esecuzione di $Rilassa(G, u, v)$ la diseguaglianza resta ancora vera, se $d[v]$ viene invece modificata allora $d[v] = d[u] + W(u, v)$. Siccome $d[u]$ non è stata modificata vale la diseguaglianza $d[u] \geq \delta(s, u)$ e quindi per il limite superiore dei costi di cammino minimo:

$$d[v] \geq \delta(s, u) + W(u, v) \geq \delta(s, v)$$

Infine siccome il valore di $d[v]$ può soltanto diminuire e $d[v] \geq \delta(s, v)$ se $d[v] = \delta(s, v)$ allora il suo valore non può più cambiare.

- **Lemma 3 (correttezza di $d[v]$ per vertici non raggiungibili):** dopo l'inizializzazione per ogni vertice v non raggiungibile da s vale $d[v] = \delta(s, v)$ e tale uguaglianza rimane vera anche dopo un numero qualsiasi di rilassamenti.

Dimostrazione: dopo l'inizializzazione $d[v] = \infty$ per ogni vertice diverso da s . Se v non è raggiungibile da s allora $\delta(s, v) = \infty = d[v]$ e per l'invariante del rilassamento $d[v]$ non può più cambiare.

- **Lemma 4 (estensione della correttezza di $d[v]$ per vertici raggiungibili):** se (u, v) è l'ultimo arco di un cammino minimo da s a v e $d[u] = \delta(s, u)$ prima di eseguire il rilassamento dell'arco (u, v) , allora dopo il rilassamento $d[v] = \delta(s, v)$

Dimostrazione: dopo il rilassamento $d[v] \leq \delta(s, u) + W(u, v)$. Siccome (u, v) è l'ultimo arco di un cammino minimo: $\delta(s, v) = \delta(s, u) + W(u, v)$ e quindi $d[v] \leq \delta(s, v)$. Per l'invariante del rilassamento $\delta(s, v)$ è anche un limite inferiore di $d[v]$, per cui vale necessariamente l'uguaglianza: $d[v] = \delta(s, v)$

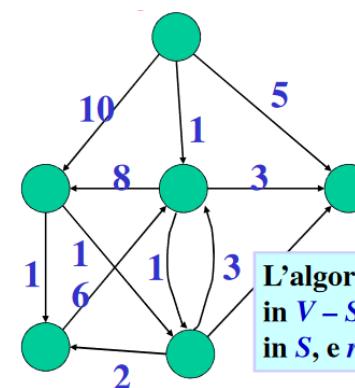
Possiamo concludere che qualsiasi algoritmo che esegua l'inizializzazione ed una sequenza di rilassamenti per cui alla fine $d[v] = \delta(s, v)$ per ogni vertice v calcola correttamente i cammini minimi. Vi sono due algoritmi classici di questo tipo, uno dovuto a **Dijkstra** ed uno dovuto a **Bellman** e **Ford**.

L'algoritmo di **Dijkstra** richiede che i **pesi degli archi non siano negativi** mentre quello di **Bellman-Ford** funziona anche nel caso generale.

$d[u]$	Stima del costo minimo del cammino dalla sorgente s al vertice u
$\delta(s, u)$	Costo del cammino minimo dalla sorgente s al vertice u
$p[u]$	Vertice precedente a u nel cammino minimo da s a u
$W(u, v)$	Costo o peso dell'arco (u, v) nel grafo orientato G
$G = (V, E)$	Grafo orientato con insiemi V (vertici) ed E (archi)
$Rilassa(G, u, v)$	Operazione di rilassamento dell'arco (u, v) nel grafo G

14.3 Algoritmo di Dijkstra

Algoritmo greedy che risolve il problema dei cammini minimi da singola sorgente per pesi non negativi



Utilizza un insieme S di vertici i cui pesi dei percorsi minimi sono già stati determinati.

L'algoritmo seleziona a turno il vertice u in $V - S$ col minimo valore $d[u]$, inserisce u in S , e rilassa tutti gli archi uscenti da u .

```

Dijkstra( $G, s, d, p, W$ )
  Inizializza( $G, s, d, p$ )
   $S = \emptyset$ 
   $Q = V(G)$  Coda (di priorità)
  while ( $Q \neq \emptyset$ )
     $u = \text{extract\_Min}(Q)$  // scelta greedy
     $S = S \cup \{u\}$ 
    for each vertice  $v$  adiacente a  $u$ 
      relax( $u, v, d, p, W$ )

```



```

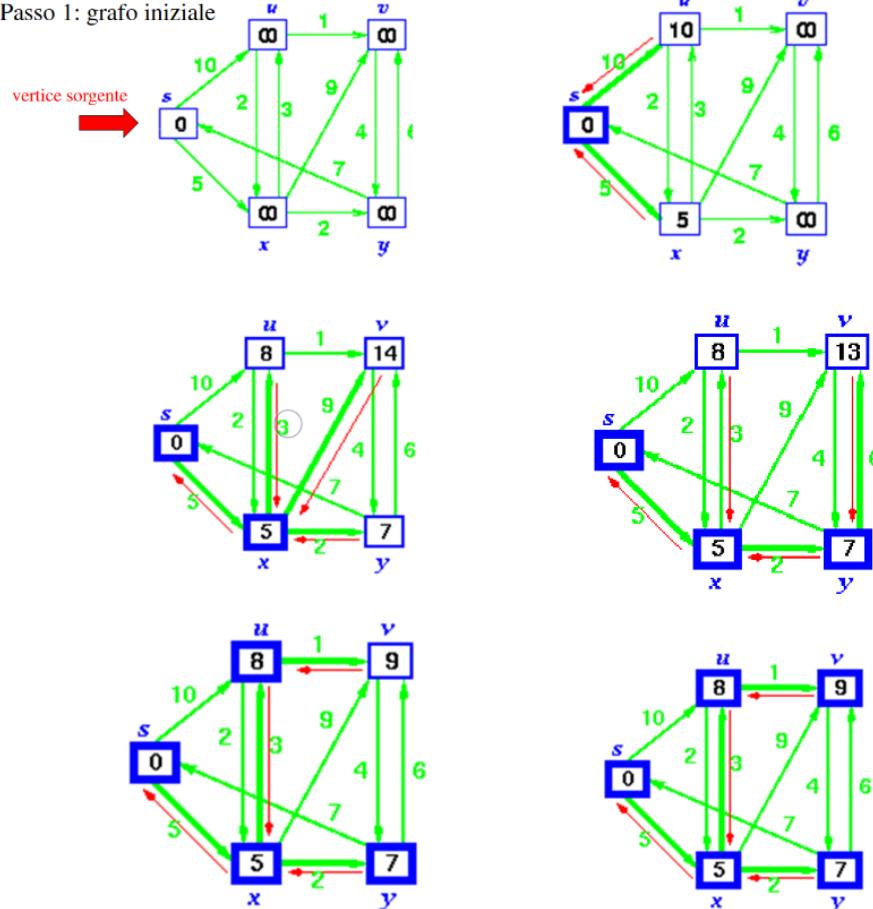
relax( $u, v, d, p, W$ )
  if  $d[v] > d[u] + w(u, v)$ 
    then decrease_key( $d[v], d[u] + w(u, v)$ )
     $p[v] = u$ 

```

Esempio di esecuzione: Dijkstra

1. grafo iniziale
2. s viene estratto da Q ed i vertici adiacenti x ed u vengono rilassati (le frecce rosse indicano i predecessori nel cammino minimo)
3. x viene estratto da Q e i vertici adiacenti u, v e y vengono rilassati
4. y viene estratto da Q ed il vertice adiacente v viene rilassato
5. u viene estratto da Q ed il vertice adiacente v viene rilassato
6. infine v viene estratto da Q . La lista dei predecessori ora definisce il cammino minimo da s per ogni nodo

Passo 1: grafo iniziale



Tempo di esecuzione: dipende dall'implementazione della coda di priorità, differenti implementazioni danno differenti costi per le operazioni sulla coda

- `extract_Min` viene eseguita $O(|V|)$ volte
- `relax (decrease_key)` viene eseguita $O(|E|)$ volte

$$\text{Tempo totale} = |V| \cdot T_{\text{extract-Min}} + |E| \cdot T_{\text{decreasekey}}$$

Se G è denso è preferibile l'array

Coda a priorità	$T_{\text{extract-Min}}$	$T_{\text{decrease_key}}$	Tempo totale
• Array	$O(V)$	$O(1)$	$O(V ^2)$
• Heap binario	$O(\lg V)$	$O(\lg V)$	$O((V + E) \lg V)$

Teorema: Se eseguiamo l'*algoritmo di Dijkstra* su un grafo orientato e pesato $G = (V, E)$, con funzione peso $w: E \rightarrow \mathbb{R}$ a valori reali *non-negativi* e un vertice sorgente s , allora, al termine $d[u] = \delta(s, u)$ per ogni vertice u in V .

14.4 In letteratura

Algoritmi classici per il calcolo di distanze (e quindi di cammini minimi), basati sulla tecnica del rilassamento (dove $n = |V|$ e $m = |E|$):

- **Dijkstra:** cammini minimi a sorgente singola, grafi diretti senza pesi negativi, tempo $O((m + n) \log n)$
- **Bellman e Ford:** cammini minimi a sorgente singola, grafi diretti senza cicli negativi, tempo $O(n \cdot m)$
- **Grafi diretti aciclici:** cammini minimi a sorgente singola in tempo $O(n + m)$
- **Floyd e Warshall:** cammini minimi tra tutte le coppie in tempo $O(n^3)$

15 Esercizio 1

Il nostro modello di costo: limitiamo a $c \cdot \log n$ bit la dimensione dei numeri interi rappresentabili ove c è una **costante** e n è la dimensione dell'**input del problema**, non ammettiamo operazioni sui numeri in virgola mobile \Rightarrow dunque li algoritmi che presentiamo d'ora in poi sono implementabili su una macchina a registri con interi di dimensione $\leq n^c$

Tempo di esecuzione: di un algoritmo misura il numero di operazioni elementari eseguite rispetto alla dimensione dell'input n . Questo si esprime con la funzione $T(n)$, che indica come il numero di passi varia al crescere di n . Il comportamento asintotico di $T(n)$ per valori grandi di n ci permette di valutare l'efficienza dell'algoritmo, indipendentemente dalla piattaforma utilizzata.

Notazione asintotica: abuso di notazione $f(n) = O(g(n))$, in realtà sarebbe $f(n) \in O(g(n))$.

Estremo sup $f(n) = O(g(n)) \Rightarrow \exists c, n_0 > 0 : f(n) \leq c \cdot g(n), \forall n \geq n_0$

Estremo inf $f(n) = \Omega(g(n)) \Rightarrow \exists c, n_0 > 0 : f(n) \geq c \cdot g(n), \forall n \geq n_0$

Between $f(n) = \Theta(g(n)) \Rightarrow \exists c_1, c_2, n_0 > 0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0$

Regole di semplificazione

Costante k $f(n) = O(k \cdot g(n))$ con $k > 0 \Rightarrow f(n) = O(g(n))$

Somma $f_1(n) = O(g_1(n)) \wedge f_2(n) = O(g_2(n)) \Rightarrow$

$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$ - if-then-else, switch

Prodotto $f_1(n) = O(g_1(n)) \wedge f_2(n) = O(g_2(n)) \Rightarrow$
 $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$ - cicli

Terminologia

$O(1)$ costante, non dipende dalla dimensione dei dati

$O(\log n)$ logaritmica, in generale $\log^K n$ con $k \geq 1$

$O(n)$ lineare

$O(n \cdot \log n)$ pseudolineare

$O(n^2)$ quadratica, n^k polinomiale con $k > 1$

$O(2^n)$ esponenziale, in generale k^n con $k > 1$

Sommatorie definizione

Serie aritmetica

$$\sum_{i=1}^n i = \frac{1}{2}n(n+1) = O(n^2)$$

$$\sum_{i=1}^{n-1} i = \frac{1}{2}(n-1)n = O(n^2)$$

$$\sum_{k=0}^n x^k = \frac{x^{n+1} - 1}{x - 1} = \frac{1 - x^{n+1}}{1 - x}, \quad \text{per } x \neq 1$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{x - 1}, \text{ con } |x| < 1$$

$$H_n = \sum_{k=1}^n \frac{1}{k} = \ln(n) + O(1)$$

$$\sum_{k=1}^n a_k - a_{k-1} = a_n - a_1$$

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

$$\prod_{k=1}^n a_k$$

$$\lg \left(\prod_{k=1}^n a_k \right) = \sum_{k=1}^n \lg a_k$$

Sommatorie:

$$\sum_{i=0}^{\log_2 n} 4n = \underbrace{4n \cdots 4n}_{\log_2 n+1} = 4n (\log_2 n + 1)$$

$$\begin{aligned} \sum_{k=1}^n \sum_{i=k+1}^n O(1) &= \sum_{k=1}^n n - k \\ &= \sum_{k=1}^n n - \sum_{k=1}^n k = \frac{1}{2}n^2 + \frac{1}{2}n \Rightarrow O(n^2) \end{aligned}$$

Finale - iniziale - compresi

$$n - (k+1) + 1 = n - k$$

Equazioni di ricorrenza: è un'espressione che definisce una funzione o una sequenza in termini di se stessa, utilizzando valori precedenti della stessa funzione o sequenza. In altre parole, un'equazione di ricorrenza specifica come calcolare il valore della funzione per un dato n (dimensione dell'input, numero di iterazioni, ecc.) in base ai suoi valori per dimensioni più piccole.

Determinare una equazione di ricorrenza:

- **Identifica il caso base:** individua il caso base dell'algoritmo, ossia il momento in cui la ricorsione si ferma. Il tempo di esecuzione per il caso base deve essere costante o facilmente calcolabile, come $O(1)$ o c
- **Descrivi il passo ricorsivo:** identifica come l'algoritmo divide il problema in sottoproblemi più piccoli, determina la dimensione dei sottoproblemi e quante volte vengono chiamati ricorsivamente. Se un algoritmo divide un problema di dimensione n in a sottoproblemi di dimensione n/b e ogni sottoproblema richiede lo stesso tempo $T(n/b)$, allora la parte ricorsiva sarà:

$$a \cdot T\left(\frac{n}{b}\right), \text{ dove } a \text{ sono } n \text{ chiamate ricorsive}$$

- **Calcola il lavoro extra non ricorsivo:** determina il lavoro svolto dall'algoritmo fuori dalla ricorsione ossia il tempo impiegato per dividere il problema e combinare le soluzioni dei sottoproblemi, questo lavoro è spesso descritto da una funzione $f(n)$ che rappresenta il costo di divisione fusione o altre operazioni svolte all'interno dell'algoritmo ma non all'interno delle chiamate ricorsive.

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \text{ (passo base)} \\ f(n) + a \cdot T\left(\lceil \frac{n}{b} \rceil\right) & \text{se } n > 1 \end{cases}$$

- **Note:** ogni ciclo può essere identificato tramite una sommatoria dove gli indici devono rispettare il passo del ciclo impiegato e il flusso di esecuzione → **deve rispondere alla domanda:** quante iterazioni totali effettua il ciclo? $a \cdot T(n)$ la a non è la costante che viene moltiplicata per $T(n)$ ma il numero di volte che viene richiamata la funzione

```

algoritmo FUN(int n) → int
  if (n ≤ 1) then return n;
  else
    int a := 1;
    for k := 1 to n do
      a := a * 2;
    endfor
    return a + 2 * FUN(n/2);
  endif

```

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \text{ (passo base)} \\ O(n) + T\left(\frac{n}{2}\right) & \text{se } n > 1 \end{cases}$$

Esempi: 04/04/2024

```

foo( n ){
  if (n < 10) {
    return 1;
  } else if (n < 1234){
    tmp = n;
    for (i = 1; i <= n; i = i+1)
      for (j = 1; j <= n; j = j+1)
        for (k = 1; k <= n; k = k+1)
          tmp = tmp + i * j * k;
  } else {
    tmp = n;
    for (i = 1; i <= n; i = i+1)
      for (j = 1; j <= n; j = j+1)
        tmp = tmp + i * j;
  }
  return tmp + foo( n/2 ) * foo( n/2 ) + foo( n/2 );
}

```

- **Caso base (se $n < 10$):** il codice restituisce semplicemente il valore 1. Questo rappresenta una computazione costante c , quindi la complessità in questo caso è $T(n) = 1$ oppure c .
- **Secondo caso (se $10 < n < 1234$):** l'algoritmo esegue tre cicli annidati (i, j, k) che eseguono un numero di operazioni proporzionale a n^4 . Dopo aver eseguito questo lavoro, la funzione viene chiamata ricorsivamente tre volte con una dimensione ridotta a $n/2$. Questo dà la ricorrenza:

$$f(n) = \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n O(1) = O(n^3)$$

$$T(n) = O(n^3) + 3T(n/2)$$

- **Terzo caso (se $n > 1234$):** il codice esegue due cicli annidati (i, j) , con un numero di operazioni pari a $O(n^2)$, seguito da tre chiamate ricorsive con dimensione $n/2$. Questo dà la ricorrenza:

$$T(n) = O(n^2) + 3T(n/2)$$

Asintoticamente ossia per $n \rightarrow \infty$ il ramo che contribuisce al comportamento dominante o maggiormente eseguito risulta essere il terzo.

$$T(n) = \begin{cases} 1 & \text{se } n < 10 \\ O(n^3) + 3T\left(\frac{n}{2}\right) & \text{se } 10 \leq n < 1234 \\ O(n^2) + 3T\left(\frac{n}{2}\right) & \text{se } n \geq 1234 \end{cases}$$

```
algoritmo Alg(A, n) :
    key <- 1
    while key ≤ n   do
        for i=1 to key do
            A[i] ++
        key <- key * 2
```

- **Ciclo esterno (while loop):** la variabile *key* inizia da 1 e viene moltiplicata per 2 ad ogni iterazione. Quindi, il ciclo while itera fino a quando $key \leq n$, il che significa che il numero di iterazioni sarà approssimativamente $\log_2 n$ (poiché *key* cresce esponenzialmente).

Alla *k-esima iterazione* → contatore

$$2^k \leq n \Rightarrow k \leq \log_2 n$$

- **Ciclo interno (for loop):** in ogni iterazione del ciclo while, il ciclo for itera da 1 fino al valore corrente di *key*. Quindi, se *key* è 2^k alla *k-esima* iterazione del ciclo while, il ciclo for esegue $O(2^k)$ operazioni.

$$T(n) = \sum_{k=1}^{\log_2 n} \sum_{i=1}^{2^k} O(1) = \sum_{k=1}^{\log_2 n} 2^k$$

Come risolvere le equazioni di ricorrenza

- **Metodo iterativo:** si itera la relazione ricorsiva fino a raggiungere il caso base. Questo metodo ci aiuta a esprimere la soluzione della ricorrenza in una forma chiusa, dipendente solo dalla dimensione *n* e dalle condizioni iniziali.

1. **Espandi la ricorrenza:** inizia iterando la ricorrenza più volte per esprimere $T(n)$ in funzione di $T(n/2)$, $T(n/4)$, e così via.
2. **Crea un pattern:** cerca un modello o schema che descriva come la somma cresce a ogni livello di iterazione
3. **Trova il numero di iterazioni necessarie:** identifica quante volte devi iterare la ricorrenza per raggiungere il caso base
4. **Sostituisci il caso base:** quando raggiungi il caso base (solitamente quando l'input è ridotto a una costante, come $n = 1$), sostituiscilo nella ricorrenza iterata
5. **Somma i termini:** somma i contributi di ciascun livello di iterazione per trovare la soluzione finale.

$$\begin{aligned} T(n) &= c + T(n/2) \\ T(n/2) &= c + T(n/4) \\ T(n/4) &= c + T(n/8) \\ &\vdots \\ T(n) &= c + T(n/2) \\ &= c + (c + T(n/4)) = 2c + T(n/4) \\ &= 2c + (c + T(n/8)) = 3c + T(n/8) \\ &= \sum_{j=1}^i c + T\left(\frac{n}{2^i}\right) = ic + T\left(\frac{n}{2^i}\right) \end{aligned}$$

Iteriamo per un *i* generico fino al caso base ossia quando:

$$\begin{aligned} \frac{n}{2^i} &= 1 \Rightarrow n = 2^i \rightarrow \lg n = \lg 2^i \Rightarrow \lg n = i \\ \frac{n}{2^i} &= 1 \Rightarrow \frac{n}{2^{\log_2 n}} = 1 \Rightarrow \frac{n}{n} = 1 \end{aligned}$$

Quindi, il numero di iterazioni necessarie per raggiungere il caso base è $i = \log n$.

Passo 4: nel passo base, abbiamo che $T(1)$ è una costante (generalmente considerata uguale a $T(1) = O(1)$).

Passo 5: poiché $i = \log n$, sostituendo:

$$\begin{aligned} T(n) &= ic + T(n/2^i) \\ T(n) &= c \log n + T(1) \end{aligned}$$

Poiché $T(1)$ è una costante, la complessità asintotica finale è:

$$T(n) = O(\log n)$$

- **Metodo della sostituzione:** consiste nell'indovinare una forma della soluzione e poi usare l'induzione matematica per dimostrare che la soluzione proposta è effettivamente corretta

1. **Indovina la soluzione:** proponi una forma per la soluzione dell'equazione di ricorrenza.
2. **Base dell'induzione:** verifica che la soluzione proposta funzioni per il caso base.
3. **Passo induttivo:** dimostra che se la soluzione è valida per un certo *n*, allora è valida anche per *n+1* (o per *n* in generale).

Esempio: $T(n) = n + T(n/2)$, con $T(1) = 1$

Passo 1 $T(n) \leq c \cdot n$

Passo 2 $T(1) = 1 \leq c \cdot 1$, caso base $n = 1$, valida $\forall c \geq 1$

Passo 3:

$$\begin{aligned} T(n) &= n + T(n/2) \\ &\leq n + c \left(\frac{n}{2}\right) \quad \text{ipotesi induttiva} \\ &\leq n + \frac{cn}{2} \\ &\leq \left(1 + \frac{c}{2}\right) n \leq cn \end{aligned}$$

Abbiamo quindi dimostrato per induzione che la nostra assunzione iniziale $T(n) \leq cn$ è vera per $c \geq 2$. Quindi, la soluzione finale della ricorrenza è $T(n) = O(n)$

$$\begin{aligned} 1 + \frac{c}{2} &\leq c \\ 1 &\leq c - \frac{c}{2} \\ 1 &\leq \frac{c}{2} \Rightarrow c \geq 2 \end{aligned}$$

- **Alberi di ricorsione:** sono uno strumento utile per visualizzare il comportamento di un algoritmo ricorsivo e per analizzarne le sue prestazioni. In particolare, vengono utilizzati per rappresentare come un problema di dimensione n viene suddiviso in sottoproblemi più piccoli fino a raggiungere una dimensione base (di solito 1). Ecco come funzionano:

1. Costruisci le recursive call fino a raggiungere il livello base $T(n = 1)$
2. Profondità dell'albero
3. Calcolare il costo per livello e somma i costi per livello
4. Analizza la complessità totale

Esempio: ogni livello dell'albero ricorsivo ha un costo aggiuntivo di n per il lavoro svolto fuori dalle chiamate ricorsive $T\left(\frac{n}{4}\right)$ e $T\left(\frac{3n}{4}\right)$.

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + n$$

$$T\left(\frac{n}{4}\right) = T\left(\frac{n}{16}\right) + T\left(\frac{3n}{16}\right) + \frac{n}{4}$$

$$T\left(\frac{3n}{4}\right) = T\left(\frac{3n}{16}\right) + T\left(\frac{9n}{16}\right) + \frac{3n}{4}$$

$$T\left(\frac{9n}{16}\right) = T\left(\frac{9n}{64}\right) + T\left(\frac{27n}{64}\right) + \frac{9n}{16}$$

$$T\left(\frac{n}{16}\right) = T\left(\frac{n}{64}\right) + T\left(\frac{3n}{64}\right) + \frac{n}{16}$$

$$T\left(\frac{3n}{16}\right) = T\left(\frac{3n}{64}\right) + T\left(\frac{9n}{64}\right) + \frac{3n}{16}$$

$$T\left(\frac{9n}{16}\right) = T\left(\frac{9n}{64}\right) + T\left(\frac{27n}{64}\right) + \frac{9n}{16}$$

Ogni chiamata si espande ulteriormente, con lo stesso schema. Ad ogni livello, la somma dei costi delle chiamate rimane uguale a n , fino a quando non si raggiunge il livello di base ossia $T(1)$

Recursive call	Livello	Tree	Sum
$T(n)$	$i = 0$	n	n
$T\left(\frac{n}{4}\right), T\left(\frac{3n}{4}\right)$	$i = 1$	$\frac{n}{4}, \frac{3n}{4}$	n
$T\left(\frac{n}{16}\right), T\left(\frac{3n}{16}\right), T\left(\frac{9n}{16}\right), T\left(\frac{27n}{64}\right)$	$i = 2$	$\frac{n}{16}, \frac{3n}{16}, \frac{9n}{16}, \frac{27n}{64}$	n

L'albero continuerà a dividersi in sottoproblemi sempre più piccoli e ci saranno almeno $\log_4 n$ livelli, poiché il problema si riduce di un fattore di 4 in ogni ricorsione per $T(n/4)$. Ogni volta, la dimensione dei sottoproblemi si riduce fino a quando il numero di elementi non scende sotto 1, cioè quando $n/4 \approx 1$, che accade dopo circa $\log_4 n$ livelli.

$$T\left(\frac{n}{4^i}\right) \rightarrow T(1) \Rightarrow \frac{n}{4^i} = 1 \Rightarrow \log_2 n = i$$

Dopo questi livelli iniziali, il problema si riduce ulteriormente nel sottoproblema più grande $T(3n/4)$, **che si riduce a una velocità più lenta rispetto al sottoproblema più piccolo**. Tuttavia, il costo per ciascun livello rimane limitato da n fino a quando il sottoproblema non scende sotto 1, che avviene al livello $\log_{\frac{4}{3}} n$.

$$T\left(\left(\frac{3}{4}\right)^i n\right) \rightarrow T(1) \Rightarrow \left(\frac{3}{4}\right)^i n = 1 \Rightarrow \log_{\frac{4}{3}} n = i$$

In totale, avremo $\log_4 n \leq T(n) \leq \log_{\frac{4}{3}} n$ livelli, ciascuno dei quali contribuisce con un costo di n . Dato che ciascun livello dell'albero ha un costo di n e ci sono al massimo $\log_{\frac{4}{3}} n$ livelli, possiamo concludere che il tempo di esecuzione complessivo è:

$$\sum_{i=0}^{\log_{\frac{4}{3}} n} n = n \sum_{i=0}^{\log_{\frac{4}{3}} n} 1 = n \cdot (\log_{\frac{4}{3}} n + 1)$$

$$T(n) = \Theta(n \log n)$$

$$\begin{aligned} T(n) &= n^2 + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) \\ &= n^2 + 2 \cdot T\left(\frac{n}{2}\right) \end{aligned}$$

$$\begin{aligned} T\left(\frac{n}{2}\right) &= \left(\frac{n}{2}\right)^2 + T\left(\frac{n}{2} \cdot \frac{1}{2}\right) + T\left(\frac{n}{2} \cdot \frac{1}{2}\right) \\ &= \left(\frac{n}{2}\right)^2 + T\left(\frac{n}{4}\right) + T\left(\frac{n}{4}\right) \\ &= \left(\frac{n}{2}\right)^2 + 2 \cdot T\left(\frac{n}{4}\right) \end{aligned}$$

$$T\left(\frac{n}{4}\right) = \left(\frac{n}{4}\right)^2 + 2 \cdot T\left(\frac{n}{8}\right)$$

$$\begin{aligned} T(n) &= n^2 + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) \\ &= n^2 + 2 \cdot T\left(\frac{n}{2}\right) \\ &= n^2 + 2 \left(\left(\frac{n}{2}\right)^2 + 2 \cdot T\left(\frac{n}{4}\right) \right) = n^2 + \frac{n^2}{2} + 4 \cdot T\left(\frac{n}{4}\right) \\ &= n^2 + \frac{n^2}{2} + 4 \left(\left(\frac{n}{4}\right)^2 + 2 \cdot T\left(\frac{n}{8}\right) \right) \\ &= n^2 + \frac{n^2}{2} + \frac{n^2}{4} + 8 \cdot T\left(\frac{n}{8}\right) \end{aligned}$$

$$\frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n$$

Serie geometrica:

$$\begin{aligned} \sum_{i=0}^{\lg n} \frac{n^2}{2^i} &= n^2 \sum_{i=0}^{\lg n} \frac{1}{2^i} \\ &= n^2 \frac{1 - 1/2^{\lg n+1}}{1/2} \equiv n^2 \frac{1/2^{\lg n+1} - 1}{-1/2} \\ &= n^2 (2 - 1/2^{\lg n}) \\ &= n^2 (2 - n^{\lg 1/2}) \\ &= 2n^2 - n \\ &= \Theta(n^2) \end{aligned}$$

- **Teorema master:** viene usato per risolvere relazioni di ricorrenza che emergono da algoritmi di tipo **divide et impera**. La forma generale della ricorrenza è:

$$T(n) = \begin{cases} O(1) \equiv c & \text{se } n = 1 \\ aT(n/b) + f(n) & \text{altrimenti} \end{cases}$$

Divide et impera: dividi il problema di dimensione n in sottoproblemi di dimensione n/b con $a \geq 1$, $b > 1$, risovi i sottoproblemi ricorsivamente e ricombina le soluzioni. Sia $f(n)$ il tempo impiegato per dividere e ricombinare i sottoproblemi in istanze di dimensioni n . Il teorema master fornisce una soluzione alla relazione di ricorrenza in tre casi, in base al confronto tra $f(n)$ e $n^{\log_b a}$:

- 1) $T(n) = \Theta(n^{\log_b a}) \quad \text{se } f(n) = O(n^{\log_b a - \varepsilon}) \text{ per } \varepsilon > 0$
- 2) $T(n) = \Theta(n^{\log_b a} \cdot \log n) \quad \text{se } f(n) = \Theta(n^{\log_b a})$
- 3) $T(n) = \Theta(f(n)) \quad \text{se } f(n) = \Omega(n^{\log_b a + \varepsilon}) \text{ per } \varepsilon > 0 \text{ e}$
 $a \cdot f(n/b) \leq c \cdot f(n) \text{ per } c < 1 \text{ e } n \text{ suff. grande}$

1. $T(n) = n + 2T\left(\frac{n}{2}\right)$. **Passi:** $a = 2$, $b = 2$, e $f(n) = n$. Calcoliamo $n^{\log_b a}$:

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

Ora confrontiamo $f(n)$ con $n^{\log_b a}$: $f(n) = n$ e $n^{\log_b a} = n$, quindi $f(n) = \Theta(n^{\log_b a})$. Applichiamo il **Caso 2** del teorema master, perché $f(n)$ è uguale a $n^{\log_b a}$. Quindi la soluzione della ricorrenza è:

$$T(n) = \Theta(n \log n)$$

2. $T(n) = c + 3T\left(\frac{n}{9}\right)$. **Passi:** $a = 3$, $b = 9$, e $f(n) = c$, dove c è una costante. Calcoliamo $n^{\log_b a}$:

$$n^{\log_b a} = n^{\log_9 3} = n^{1/2}$$

Confrontiamo $f(n)$ con $n^{\log_b a}$: $f(n) = c = O(n^0)$, quindi abbiamo $f(n) = O(n^{1/2-\epsilon})$ con $\epsilon = 1/2$. Applichiamo il **Caso 1** del teorema master, perché $f(n) = O(n^{\log_b a - \epsilon})$ con $\epsilon > 0$. La soluzione della ricorrenza è:

$$T(n) = \Theta(n^{\log_9 3}) = \Theta(n^{1/2})$$

3. $T(n) = n + 3T\left(\frac{n}{9}\right)$. **Passi:** $a = 3$, $b = 9$, e $f(n) = n$. Calcoliamo $n^{\log_b a}$:

$$n^{\log_b a} = n^{\log_9 3} = n^{1/2}$$

$f(n) = n$ è maggiore di $n^{\log_b a} = n^{1/2}$, quindi $f(n) = \Omega(n^{1/2+\epsilon})$ con $\epsilon = 1/2$. Inoltre, dobbiamo verificare che esista una costante $c < 1$ tale che $af(n/b) \leq cf(n)$. Dato che:

$$af\left(\frac{n}{b}\right) = 3 \cdot \frac{n}{9} = \frac{n}{3} \leq c \cdot n$$

$$\frac{n}{3} \leq c \cdot n \Rightarrow \frac{1}{3} \leq c$$

Quindi, una possibile scelta di c è $c = \frac{1}{3}$, che soddisfa la condizione $c < 1$. Applichiamo il **Caso 3** del teorema master, perché $f(n) = \Omega(n^{\log_b a + \epsilon})$. Quindi la soluzione della ricorrenza è:

$$T(n) = \Theta(n)$$

Riassunto delle soluzioni:

1. $T(n) = n + 2T(n/2) \Rightarrow T(n) = \Theta(n \log n)$
2. $T(n) = c + 3T(n/9) \Rightarrow T(n) = \Theta(n^{1/2})$
3. $T(n) = n + 3T(n/9) \Rightarrow T(n) = \Theta(n)$

Teorema dei limiti: se $f(n)$ e $g(n)$ sono asintoticamente positive, allora possibile dedurre che:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = O(g(n)), \text{ con } f(n) \neq \Omega(g(n)), f(n) \neq \Theta(g(n))$$

$$f(n) = O(g(n)) \Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \text{ (se esiste)} < \infty$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \Omega(g(n)), \text{ con } f(n) \neq O(g(n)), f(n) \neq \Theta(g(n))$$

$$f(n) = \Omega(g(n)) \Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \text{ (se esiste)} > 0$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow f(n) = \Theta(g(n)), g(n) = \Theta(f(n))$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \emptyset \Rightarrow f(n) \text{ e } g(n) \text{ non sono asintoticamente confrontabili}$$

Note:

- Posso scrivere in questo modo:

$$T(n) = \begin{cases} O(1) = c_1 & \text{se } n = 1 \\ aT(n/b) + O(n^2) = c_2 \cdot n^2 & \text{altrimenti} \end{cases}$$

- Dunque qualsiasi sia $O(n^2) = c \cdot n^2$ è a meno di una costante c
- le sommatoria interessa molto il passo che ha il contatore del ciclo e le iterazione effettive totali

16 Esercizio 2

Alberi

Un algoritmo **divide-et-impera** è una strategia per risolvere problemi complessi suddividendoli in sottoproblemi più semplici e ricombinando i risultati. È un paradigma ricorsivo molto comune in informatica, usato per progettare algoritmi efficienti.

tipo Dizionario:

dati:
un insieme S di coppie $(\text{elem}, \text{chiave})$.

operazioni:

insert($\text{elem } e, \text{chiave } k$)
aggiunge a S una nuova coppia (e, k) .

delete($\text{chiave } k$)
cancella da S la coppia con chiave k .

search($\text{chiave } k \rightarrow \text{elem}$)
se la chiave k è presente in S restituisce l'elemento e ad essa associato,
e null altrimenti.

tipo Pila:

dati:
una sequenza S di n elementi.

operazioni:

isEmpty() $\rightarrow \text{result}$
restituisce `true` se S è vuota, e `false` altrimenti.

push($\text{elem } e$)
aggiunge e come ultimo elemento di S .

pop() $\rightarrow \text{elem}$
toglie da S l'ultimo elemento e lo restituisce.

top() $\rightarrow \text{elem}$
restituisce l'ultimo elemento di S (senza toglierlo da S).

tipo Coda:

dati:
una sequenza S di n elementi.

operazioni:

isEmpty() $\rightarrow \text{result}$
restituisce `true` se S è vuota, e `false` altrimenti.

enqueue($\text{elem } e$)
aggiunge e come ultimo elemento di S .

dequeue() $\rightarrow \text{elem}$
toglie da S il primo elemento e lo restituisce.

first() $\rightarrow \text{elem}$
restituisce il primo elemento di S (senza toglierlo da S).

tipo Albero:

dati:

un insieme di nodi (di tipo *nodo*) e un insieme di archi.

operazioni:

numNodi() → intero
restituisce il numero di nodi presenti nell'albero.

grado(nodo v) → intero
restituisce il numero di figli del nodo *v*.

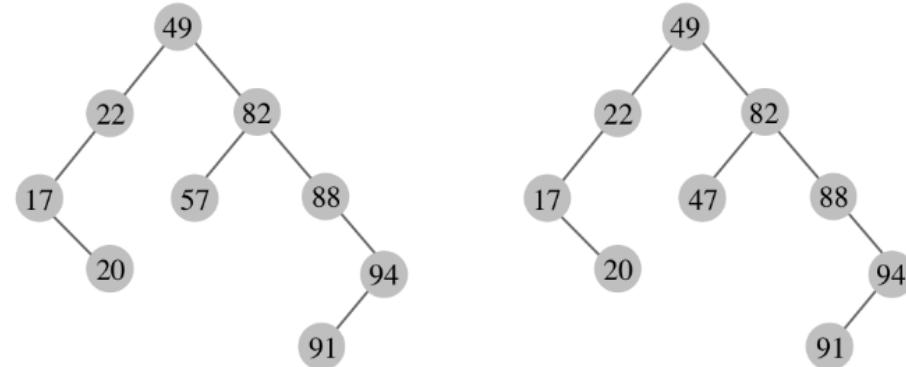
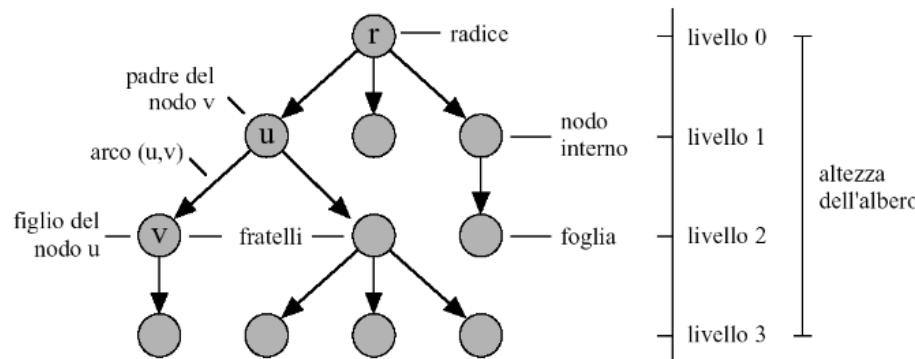
padre(nodo v) → nodo
restituisce il padre del nodo *v* nell'albero, o null se *v* è la radice.

figli(nodo v) → ⟨nodo, nodo, …, nodo⟩
restituisce, uno dopo l'altro, i figli del nodo *v*.

aggiungiNodo(nodo u) → nodo
inserisce un nuovo nodo *v* come figlio di *u* nell'albero e lo restituisce.
Se *v* è il primo nodo ad essere inserito nell'albero, esso diventa la radice
(e *u* viene ignorato).

aggiungiSottoalbero(Albero a, nodo u)
inserisce nell'albero il sottoalbero *a* in modo che la radice di *a* diventi
figlia di *u*.

rimuoviSottoalbero(nodo v) → Albero
stacca e restituisce l'intero sottoalbero radicato in *v*. L'operazione
cancella dall'albero il nodo *v* e tutti i suoi discendenti.



Albero binario di ricerca

Albero binario non di ricerca (47 a dx di 49)

Notazione:

Pila (LIFO)

S

Coda (FIFO)

C

Albero

T

Altezza albero

$\max(\text{livello}) = h$

Albero completo

lvl intermedi tutti ripiti tranne ultimo

Nodo

nodo v

Radice

nodo r

Radice

$v \leftarrow \text{radice di } T$

Regola albero

$\text{chiave}(v.\text{sx}) < \text{chiave}(v.\text{dx})$

Figlio sx di nodo v

v.sinistro

Figlio dx di nodo v

v.destro

Padre del nodo v

parent(v)

Chiave nodo v

chiave(v)

Elemento nodo v

elem(v)

Nodo v è foglia

leaf(v)

Alberi k-ari completi

Figli per lvl

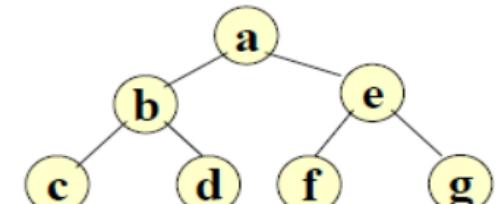
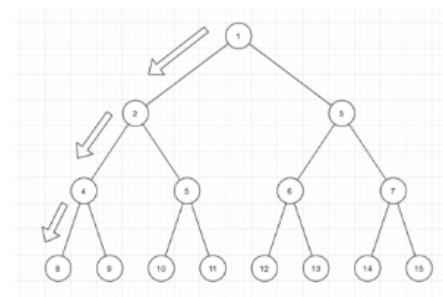
$k \text{ figli}$

Nodi foglia

k^h

Nodi interni

$$\sum_{i=0}^{h-1} k^i = \frac{k^h - 1}{k - 1}$$



16.1 VisitaDFS pre-order o ordine anticipato

Prima si visita il nodo e poi i suoi sottoalberi.

nodo corrente - sotto sx - sotto dx
 $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g$

Algorithm 1 visitaDFSRicorsiva(nodo r)

```
1: if ( $r = \text{null}$ ) then
2:   return
3: end if
4: visita il nodo  $r$ 
5: VisitaPreOrdineRicorsiva(figlio sinistro di r)
6: VisitaPreOrdineRicorsiva(figlio destro di r)
```

Algorithm 2 visitaDFS(nodo r)

```
1: Pila  $S$ 
2:  $S.\text{push}(r)$ 
3: while (not  $S.\text{isEmpty}()$ ) do
4:    $n \leftarrow S.\text{pop}()$ 
5:   if ( $n \neq \text{null}$ ) then
6:     visita il nodo  $n$ 
7:      $S.\text{push}(\text{figlio destro di } n)$ 
8:      $S.\text{push}(\text{figlio sinistro di } n)$ 
9:   end if
10: end while
```

16.2 VisitaDFS in-order o simmetrico

Si esplora prima il sottoalbero sx poi si visita il nodo corrente ed infine si passa al sottoalbero dx

sotto sx - nodo corrente - sotto dx
 $c \rightarrow b \rightarrow d \rightarrow a \rightarrow f \rightarrow e \rightarrow g$

Algorithm 3 visitaInOrdineRicorsiva(nodo r)

```
1: if  $r = \text{null}$  then
2:   return
3: end if
4: visitaInOrdineRicorsiva(figlio sinistro di r)
5: visita il nodo  $r$ 
6: visitaInOrdineRicorsiva(figlio destro di r)
```

Algorithm 4 visitaInOrdine(nodo r)

```
1: Pila  $S$ 
2:  $n \leftarrow r$ 
3: while  $n \neq \text{null}$  or not  $S.\text{isEmpty}()$  do
4:   while  $n \neq \text{null}$  do
5:      $S.\text{push}(n)$ 
6:      $n \leftarrow \text{figlio sinistro di } n$ 
7:   end while
8:    $n \leftarrow S.\text{pop}()$ 
9:   visita il nodo  $n$ 
10:   $n \leftarrow \text{figlio destro di } n$ 
11: end while
```

16.3 VisitaDFS post-order

Prima si visitano i sottoalberi, poi il nodo

sotto sx - sotto dx - nodo corrente
 $c \rightarrow d \rightarrow b \rightarrow f \rightarrow g \rightarrow e \rightarrow a$

Algorithm 5 visitaPostOrdineRicorsiva(nodo r)

```
1: if  $r = \text{null}$  then
2:   return
3: end if
4: visitaPostOrdineRicorsiva(figlio sinistro di r)
5: visitaPostOrdineRicorsiva(figlio destro di r)
6: visita il nodo  $r$ 
```

Algorithm 6 VisitaPostOrdine(nodo r)

```
1: Pila  $S$ 
2:  $n \leftarrow r$ 
3:  $\text{ultimoVisitato} \leftarrow \text{null}$ 
4: while  $n \neq \text{null}$  or not  $S.\text{isEmpty}()$  do
5:   while  $n \neq \text{null}$  do
6:      $S.\text{push}(n)$ 
7:      $n \leftarrow n.\text{sinistro}$ 
8:   end while
9:    $n \leftarrow S.\text{top}()$ 
10:  if  $n.\text{destro} = \text{null}$  or  $n.\text{destro} = \text{ultimoVisitato}$  then
11:    visita il nodo  $n$ 
12:     $\text{ultimoVisitato} \leftarrow S.\text{pop}()$ 
13:     $n \leftarrow \text{null}$ 
14:  else
15:     $n \leftarrow n.\text{destro}$ 
16:  end if
17: end while
```

16.4 Visita BFS

$a \rightarrow b \rightarrow e \rightarrow c \rightarrow d \rightarrow f \rightarrow g$

Algorithm 7 VisitaBFS(nodo r)

```
1: Coda  $Q$ 
2:  $Q.\text{enqueue}(r)$ 
3: while not  $Q.\text{isEmpty}()$  do
4:    $n \leftarrow Q.\text{dequeue}()$ 
5:   visita il nodo  $n$ 
6:   if  $n.\text{sinistro} \neq \text{null}$  then
7:      $Q.\text{enqueue}(n.\text{sinistro})$ 
8:   end if
9:   if  $n.\text{destro} \neq \text{null}$  then
10:     $Q.\text{enqueue}(n.\text{destro})$ 
11:  end if
12: end while
```

16.5 Search con chiave $O(h)$

Algorithm 8 Search(key k) \rightarrow elem

```
1:  $v \leftarrow \text{root of } T$ 
2: while  $v \neq \text{null}$  do
3:   if  $k = \text{key}(v)$  then
4:     return  $\text{elem}(v)$ 
5:   else if  $k < \text{key}(v)$  then
6:      $v \leftarrow \text{left child of } v$ 
7:   else
8:      $v \leftarrow \text{right child of } v$ 
9:   end if
10: end while
11: return null
```

16.6 Ricerca il massimo

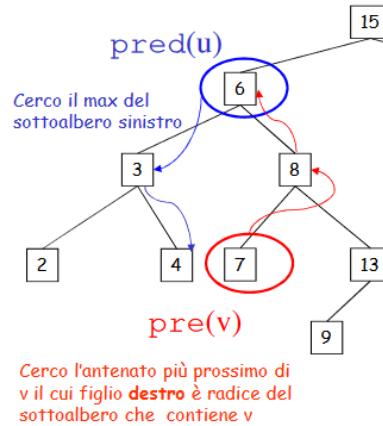
Algorithm 9 max(nodo u) \rightarrow nodo

```
1:  $v \leftarrow u$ 
2: while figlio destro di  $v \neq \text{null}$  do
3:    $v \leftarrow \text{figlio destro di } v$ 
4: end while
5: return  $v$ 
```

16.7 Predecessore

È il più grande nodo che precede u nell'ordinamento in-order dell'albero.

- **Massimo del sottoalbero sinistro:** se u ha un sottoalbero sx, il predecessore di u è il nodo con il valore massimo in questo sottoalbero $\Rightarrow \text{pred}(8) = 7$
- **Antenato più prossimo:** se u non ha un sottoalbero sx, il predecessore di u è il primo antenato v tale che u è nel sottoalbero dx di $v \Rightarrow \text{pred}(7) = 6$



Algorithm 10 $\text{pred}(\text{nodo } u) \rightarrow \text{nodo}$

```

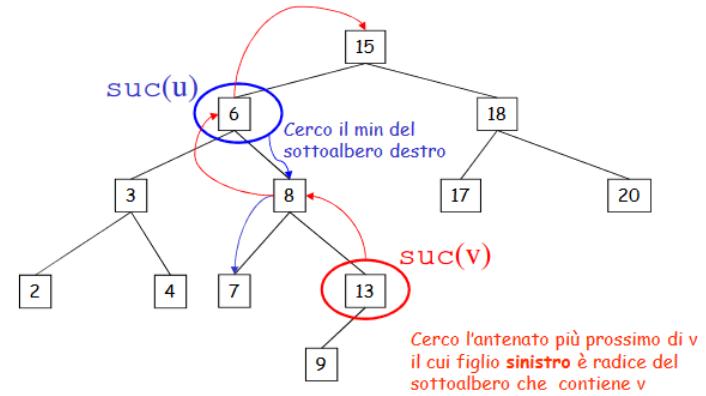
1: if  $u$  ha figlio sinistro  $\text{sin}(u)$  then
2:   return  $\max(\text{sin}(u))$ 
3: end if
4: while  $\text{parent}(u) \neq \text{null}$  e  $u$  è figlio dx di suo padre do
5:    $u \leftarrow \text{parent}(u)$ 
6: end while
7: return  $u$ 

```

16.8 Successore

È il più piccolo nodo che segue u nell'ordinamento in-order dell'albero.

- **Minimo del sottoalbero destro:** se u ha un sottoalbero dx, il successore di u è il nodo con il valore minimo in questo sottoalbero.
- **Antenato più prossimo:** se u non ha un sottoalbero dx, il successore di u è il primo antenato v tale che u è nel sottoalbero sx di v .



Algorithm 11 $\text{succ}(\text{nodo } u) \rightarrow \text{nodo}$

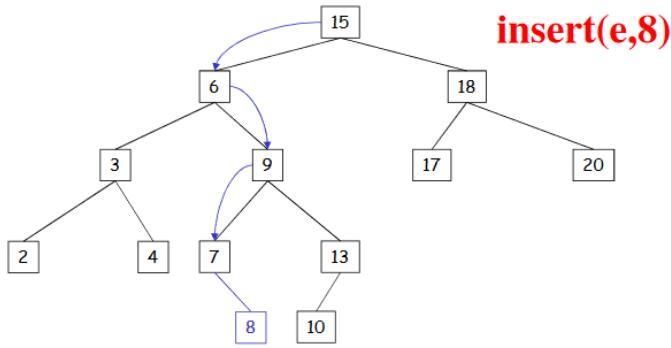
```

1: if  $u$  ha figlio destro  $\text{dest}(u)$  then
2:   return  $\min(\text{dest}(u))$ 
3: end if
4: while  $\text{parent}(u) \neq \text{null}$  e  $u$  è figlio sx di suo padre do
5:    $u \leftarrow \text{parent}(u)$ 
6: end while
7: return  $u$ 

```

16.9 Insert di un nodo

- Inserirò sempre una foglia
- Appendi u come figlio sx/dx di v in modo che sia mantenuta la proprietà di ricerca



insert(e,8)

Algorithm 12 insert(elem e, chiave k)

1: Crea un nuovo nodo u con *elem* = *e* e *chiave* = *k*

2: *v* \leftarrow null

3: *n* \leftarrow radice

4: **while** *n* \neq null **do**

5: *v* \leftarrow *n*

6: **if** *k* < chiave(*n*) **then**

7: *n* \leftarrow *n.sinistro*

8: **else**

9: *n* \leftarrow *n.destro*

10: **end if**

11: **end while**

12: **if** *v* = null **then**

13: l'albero è vuoto, imposta *u* come radice

14: **else**

15: **if** *k* < chiave(*v*) **then**

16: *v.sinistro* \leftarrow *u*

17: **else**

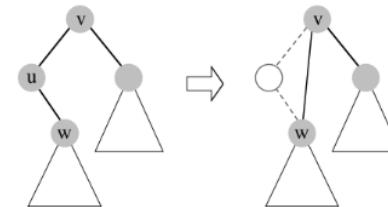
18: *v.destro* \leftarrow *u*

19: **end if**

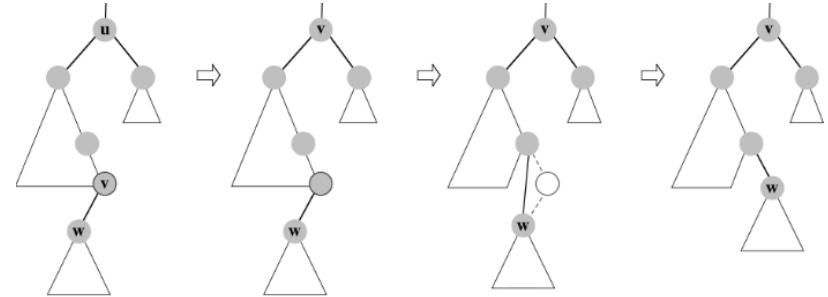
20: **end if**

16.10 Delete di un nodo

Ho un solo figlio: II if



Ho 2 figli: III if



Algorithm 13 delete(chiave k)

1: *u* \leftarrow cerca(*k*)

2: **if** *u* = null **then**

3: l'elemento con chiave *k* non esiste

4: **return**

5: **end if**

6: **if** *u* è una foglia **then**

7: rimuovi *u*

8: **else if** *u* ha un solo figlio **then**

9: **if** *u* è il figlio sinistro di *v* **then**

10: *v.sinistro* \leftarrow figlio di *u*

11: **else**

12: *v.destro* \leftarrow figlio di *u*

13: **end if**

14: **else**

15: *v* \leftarrow predecessore(*u*)

16: *k* \leftarrow chiave(*v*)

17: cancella *v*

18: copia il valore di *k* in *u*

19: **end if**

Algorithm 14 delete(chiave k)

```

1:  $u \leftarrow \text{cerca}(k)$ 
2: if  $u = \text{null}$  then
3:   l'elemento con chiave  $k$  non esiste
4:   return
5: end if
6:  $v \leftarrow \text{parent}(u)$ 
7: if  $u$  è una foglia then
8:   rimuovi  $u$ 
9:   if  $v$  è null then
10:    l'albero è ora vuoto
11:   else
12:     if  $u$  è il figlio sinistro di  $v$  then
13:        $v.sinistro \leftarrow \text{null}$ 
14:     else
15:        $v.destro \leftarrow \text{null}$ 
16:     end if
17:   end if
18: else if  $u$  ha un solo figlio then
19:   if  $u$  è il figlio sinistro di  $v$  then
20:      $v.sinistro \leftarrow \text{figlio di } u$ 
21:   else
22:      $v.destro \leftarrow \text{figlio di } u$ 
23:   end if
24: else
25:    $v \leftarrow \text{predecessore}(u)$ 
26:    $k \leftarrow \text{chiave}(v)$ 
27:    $p \leftarrow \text{parent}(v)$ 
28:   if  $v$  è una foglia then
29:     rimuovi  $v$ 
30:     if  $v$  è il figlio sinistro di  $p$  then
31:        $p.sinistro \leftarrow \text{null}$ 
32:     else
33:        $p.destro \leftarrow \text{null}$ 
34:     end if
35:   else if  $v$  ha un solo figlio then
36:     if  $v$  è il figlio sinistro di  $p$  then
37:        $p.sinistro \leftarrow \text{figlio di } v$ 
38:     else
39:        $p.destro \leftarrow \text{figlio di } v$ 
40:     end if
41:   end if
42:   copia il valore di  $k$  in  $u$ 
43: end if
```

16.11 Costi esecuzione

- Tutte le operazioni principali su un BST (come inserimento, cancellazione e ricerca) hanno un costo nel caso peggiore pari a $O(h)$, dove h è l'altezza dell'albero. Questo perché, nel peggiore dei casi, l'operazione richiederà di percorrere l'albero dalla radice fino a una foglia, coprendo un percorso lungo al massimo h .
- Caso peggiore (BST degenerato in una lista):** se il BST è molto sbilanciato e degenerato in una lista, l'altezza h dell'albero sarà uguale al numero di nodi meno uno, quindi $h = n - 1$. In questo caso, il costo delle operazioni diventa $O(n)$, poiché l'albero è estremamente sbilanciato e l'altezza è massima.
- Caso migliore (BST bilanciato):** se l'albero è bilanciato, cioè l'altezza h è la più piccola possibile rispetto al numero di nodi n , allora h sarà approssimativamente $O(\log n)$. In questo caso, le operazioni hanno un costo nel caso peggiore di $O(\log n)$.

Albero binario completo:

$$n = \sum_{i=0}^h 2^i = 2^{h+1} - 1$$

$$h = \log_2(n + 1) - 1 = O(\log n)$$

Algorithm	tempo esecuzione
pre-order(nodo r)	$O(n)$
in-order(nodo r)	$O(n)$
post-order(nodo r)	$O(n)$
visitaBFS(nodo r)	$O(n)$, n nodi
search(key k) \rightarrow elem	$O(h)$
max(nodo u) \rightarrow nodo	$O(h)$
pred(nodo u) \rightarrow nodo	$O(h)$
succ(nodo u) \rightarrow nodo	$O(h)$
insert(elem e , chiave k)	$O(h)$
delete(chiave k)	$O(h)$

17 Esercizio 3

17.1 Algoritmi greedy

- **seleziona(C):** prendere una decisione locale migliore in quel preciso istante e che offra il massimo beneficio immediato aggiornando l'insieme delle candidate
- **ammissibile($S \cup \{x\}$):** se la scelta migliore selezionata risulta essere coerente con il tipo di soluzione richiesta
- **ottimo(S):** verificare se la soluzione converge verso una min/max globale oppure converge verso una soluzione locale
- **NB:** una strategia greedy non sempre garantisce l'ottimalità della soluzione finale

```
algoritmo paradigmaGreedy(insieme di candidati C) → soluzione
    S ← ∅
    while ((not ottimo(S)) and (C ≠ ∅)) do
        x ← seleziona(C)
        C ← C - {x}
        if (ammissibile(S ∪ {x})) then S ← S ∪ {x}
        if (ottimo(S)) then return S
        else errore non ho trovato soluzioni
```

17.2 Ordinamenti

Fare il return se l'array è composto da un solo elemento

Algorithm 15 SelectionSort(A)

```
1: for k = 0 to n - 1 do
2:   m ← k {Indice del minimo elemento}
3:   for j = k + 1 to n do
4:     if A[j] < A[m] then
5:       m ← j
6:     end if
7:   end for
8:   if m ≠ k then
9:     Scambia A[m] con A[k]
10:  end if
11: end for
```

Algorithm 16 InsertionSort(A)

```
1: for j = 1 to n do
2:   x ← A[j] {Elemento da inserire}
3:   i ← j - 1
4:   while i >= 0 and A[i] > x do
5:     A[i + 1] ← A[i]
6:     i ← i - 1
7:   end while
8:   A[i + 1] ← x
9: end for
```

Algorithm 17 BubbleSort(A)

```
1: for i ← 1 to n - 1 do
2:   scambiAvvenuti ← false
3:   for j ← 2 to n - i + 1 do
4:     if A[j - 1] > A[j] then
5:       Scambia A[j - 1] con A[j]
6:       scambiAvvenuti ← true
7:     end if
8:   end for
9:   if ¬scambiAvvenuti then
10:    break
11: end if
12: end for
```

Algorithm 18 MergeSort(A, p, r)

```
1: if p < r then
2:   q ← ⌊(p + r)/2⌋
3:   MergeSort(A, p, q)
4:   MergeSort(A, q + 1, r)
5:   Merge(A, p, q, r)
6: end if
```

Algorithm 19 Merge(A, p, q, r)

```
1: n1 ← q - p + 1
2: n2 ← r - q
3: Let L[1...n1 + 1] and R[1...n2 + 1] be new arrays
4: for i ← 1 to n1 do
5:   L[i] ← A[p + i - 1]
6: end for
7: for j ← 1 to n2 do
8:   R[j] ← A[q + j]
9: end for
10: L[n1 + 1] ← ∞ // valore sentinella
11: R[n2 + 1] ← ∞ // valore sentinella
12: i ← 1
13: j ← 1
14: for k ← p to r do
15:   if L[i] ≤ R[j] then
16:     A[k] ← L[i]
17:     i ← i + 1
18:   else
19:     A[k] ← R[j]
20:     j ← j + 1
21:   end if
22: end for
```

Algorithm 20 QuickSort(A) **

```

1: if |A| > 1 then
2:   Scegli un elemento  $x$  (pivot) in  $A$ 
3:   Partiziona  $A$  in due sottoarray:
4:    $A_1 = \{y \in A : y \leq x\}$ 
5:    $A_2 = \{y \in A : y > x\}$ 
6:   QuickSort( $A_1$ )
7:   QuickSort( $A_2$ )
8:   Copia  $A_1$  seguito da  $A_2$  in  $A$ 
9: end if

```

QuickSort (A, i, f)

1. **if** ($i \geq f$) **then return**
2. $m = Partition(A, i, f)$
3. QuickSort($A, i, m - 1$)
4. QuickSort($A, m + 1, f$)

Partition (A, i, f)

1. $x = A[i]$ //Partiziona $A[i..f]$ intorno al pivot $A[i]$
2. $inf = i$ e $sup = f + 1$
3. **while** (true) **do**
4. **do** ($inf = inf + 1$) **while** ($inf \leq f$ and $A[inf] \leq x$)
5. **do** ($sup = sup - 1$) **while** ($A[sup] > x$)
6. **if** ($inf < sup$) **then** scambia $A[inf]$ e $A[sup]$
7. **else break**
8. scambia $A[i]$ e $A[sup]$
9. **return sup**

Algorithm 21 RECURSIVE_SELECTION_SORT(A, k, n)

```

1: if  $k \geq n - 1$  then
2:   return
3: end if
4:  $m \leftarrow k$  {Indice del minimo elemento}
5: for  $j = k + 1$  to  $n$  do
6:   if  $A[j] < A[m]$  then
7:      $m \leftarrow j$ 
8:   end if
9: end for
10: if  $m \neq k$  then
11:   SCAMBIA  $A[m]$  CON  $A[k]$ 
12: end if
13: RECURSIVE_SELECTION_SORT(A,  $k + 1, n$ )

```

Remark:

- **QuickSort in loco (in-place)** * è una versione dell'algoritmo che utilizza un numero costante di spazio aggiuntivo, modificando direttamente l'array di input. Non richiede spazio aggiuntivo significativo oltre a quello per le variabili temporanee.
- **QuickSort non in loco (out-of-place)** ** richiede spazio aggiuntivo per memorizzare i sottoarray creati durante la partizione. Questo può includere nuovi array per gli elementi minori e maggiori rispetto al pivot.

Nome	Migliore	Medio	Peggiore	Stabile	In loco	Caratteristiche dell'ordinamento
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	Sì	Sì	Algoritmo per confronto tramite scambio di elementi
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	Sì	Sì	A. per confronto tramite inserimento
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Sì	No	A. per confronto tramite unione di componenti, ottimale e facile da parallelizzare
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	No	Sì	A. per confronto tramite partizionamento. Le sue varianti possono: essere stabili
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Sì	A. per confronto tramite selezione di elementi

L'algoritmo cerca un elemento all'interno di un array che deve **necessariamente essere ordinato** in ordine crescente, effettuando mediamente meno confronti rispetto a una ricerca sequenziale, e quindi più rapidamente rispetto a essa perché, sfruttando l'ordinamento, dimezza l'intervallo di ricerca a ogni passaggio.

Dove A è un array di interi, p indica la posizione del primo elemento dell'array, r indica la posizione dell'ultimo elemento dell'array e v è l'elemento che sto cercando.

Entrambe le funzioni ritornano l'indice in cui si trova il valore (se trovato) oppure -1 per indicare che non è stato trovato.

Algorithm 22 binarySearchIterative(A, p, r, v)

```

1: if  $v < A[p]$  OR  $v > A[r]$  then
2:   return -1
3: end if
4: while  $p \leq r$  do
5:    $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
6:   if  $A[q] = v$  then
7:     return  $q$ 
8:   else if  $A[q] > v$  then
9:      $r \leftarrow q - 1$ 
10:  else
11:     $p \leftarrow q + 1$ 
12:  end if
13: end while
14: return -1

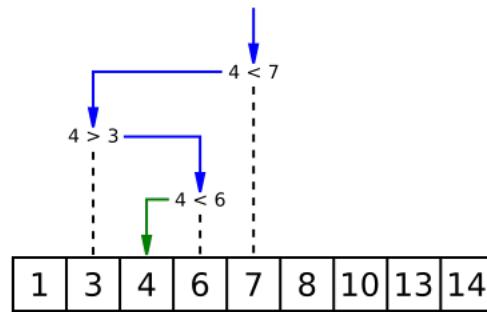
```

Algorithm 23 binarySearchRicorsivo(A, p, r, v)

```

1: if  $p > r$  then
2:   return -1
3: else if  $v < A[p]$  OR  $v > A[r]$  then
4:   return -1
5: end if
6:  $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
7: if  $A[q] = v$  then
8:   return  $q$ 
9: else if  $A[q] > v$  then
10:  return binarySearchRicorsivo( $A, p, q - 1, v$ )
11: else
12:  return binarySearchRicorsivo( $A, q + 1, r, v$ )
13: end if

```



Caso peggiore

$O(\log n)$

Caso medio

$O(\log n)$

Caso ottimo

$O(1)$

Note

A ordinato, confronto indici

17.3 Programmazione dinamica

Divide et impera (tecnica top down): dividi l'istanza del problema in due o più sottoistanze, risovi ricorsivamente il problema sulle sottoistanze, riconcilia la soluzione dei sottoproblemi allo scopo di ottenere la soluzione globale \Rightarrow solo se i sottoproblemi sono *indipendenti*

Programmazione dinamica: suddividere il problema in sottoproblemi più semplici, costruisci una formula ricorsiva tra i sottoproblemi, indentificare i casi base, memorizza i risultati intermedi in una tabella/array per evitare calcoli ripetuti e ridurre il tempo di calcolo, riempire la tabella partendo dai casi base e risolvendo progressivamente sottoproblemi più grandi, ricostruire la soluzione \Rightarrow *sottoproblemi dipendenti*

Distanza stringhe

Siano X e Y due stringhe di lunghezza m ed n , calcoliamo la distanza tra X e Y come minimo numero di operazioni elementari che trasformano X in Y scelte tra:

$$X = x_1 \cdot x_2 \cdots x_m \quad Y = y_1 \cdot y_2 \cdots y_n$$

Problema più semplice: riduciamo il problema di calcolare $\delta(X, Y)$ al calcolo di $\delta(X_i, Y_j)$ per ogni i, j tali che $0 \leq i \leq m$ e $0 \leq j \leq n$, manteniamo le soluzioni parziali in una tabella D di dimensione $(m + 1) \times (n + 1)$

Inizializzazione della tabella (caso base): alcuni sottoproblemi sono molto semplici, $\delta(X_0, Y_j) = j$ partendo dalla stringa vuota X_0 basta inserire uno ad uno i j caratteri di Y_j , $\delta(X_i, Y_0) = i$ partendo da X_i basta rimuovere uno ad uno gli i caratteri per ottenere Y_0 , queste soluzioni sono memorizzate rispettivamente nella prima riga e prima colonna della tabella D

Avanzamento nella tabella:

- **inserisci(y_i):** il minimo costo per trasformare X_i in Y_j è uguale al minimo costo per trasformare X_i in $Y_{j-1} + 1$ per inserire il carattere y_j

$$D[i, j] = 1 + D[i, j - 1]$$

- **cancella(x_i):** il minimo costo per trasformare X_i in Y_j è uguale al minimo costo per trasformare X_{i-1} in $Y_j + 1$ per la cancellazione del carattere x_i

$$D[i, j] = 1 + D[i - 1, j]$$

- **sostituisci(x_i, y_i):** il minimo costo per trasformare X_i in Y_j è uguale al minimo costo per trasformare X_{i-1} in $Y_{j-1} + 1$ per sostituire il carattere x_i per y_j

$$D[i, j] = 1 + D[i - 1, j - 1]$$

In conclusione per $i, j \geq 1$:

$$D[i, j] = \begin{cases} D[i - 1, j - 1] & \text{se } x_i = y_j \\ 1 + \min\{D[i, j - 1], D[i - 1, j], D[i - 1, j - 1]\} & \text{se } x_i \neq y_j \end{cases}$$

		P	R	E	S	T	O
0	1	2	3	4	5	6	
R	1						
I	2						
S	3						
O	4						
T	5						
T	6						
O	7						

Distanza tra X e Y

$$\delta(X, Y)$$

Stringhe vuote

$$X_0, Y_0$$

Ottenerne X_0 inserimento j caratteri

$$\delta(X_0, Y_j) = j$$

Ottenerne Y_0 cancellazione i caratteri

$$\delta(X_i, Y_0) = i$$

	P	R	E	S	T	O	
0	1	2	3	4	5	6	
R	1	1	1	2	3	4	5
I	2	2	2	2	3	4	5
S	3	3	3	3	2	3	4
O	4	4	4	4	3	3	3
T	5	5	5	5	4	3	4
T	6	6	6	6	5	4	4
O	7	7	7	7	6	5	4

Costo per trasformare X_i in Y_j :

Inserimento

$$D[i, j] = 1 + D[i, j - 1]$$

Cancellazione

$$D[i, j] = 1 + D[i - 1, j]$$

sostituzione

$$D[i, j] = 1 + D[i - 1, j - 1]$$

$$D[i, j] = \begin{cases} D[i - 1, j - 1] & \text{se } x_i = y_j \\ 1 + \min\{D[i, j - 1], D[i - 1, j], D[i - 1, j - 1]\} & \text{se } x_i \neq y_j \end{cases}$$

Pseudocodice: tempo di esecuzione ed occupazione di memoria: $O(m \cdot n)$, se tengo traccia della cella della tabella che utilizzo per decidere il valore di $D[i, j]$ ho un metodo costruttivo per ricostruire la sequenza di operazioni di editing ottima.

```

algoritmo distanzaStringhe(stringa X, stringa Y) → intero
    matrice D di  $(m + 1) \times (n + 1)$  interi
    for  $i = 0$  to m do  $D[i, 0] \leftarrow i$ 
    for  $j = 1$  to n do  $D[0, j] \leftarrow j$ 
    for  $i = 1$  to m do
        for  $j = 1$  to n do
            if  $(x_i \neq y_j)$  then
                 $D[i, j] \leftarrow 1 + \min\{D[i, j - 1], D[i - 1, j], D[i - 1, j - 1]\}$ 
            else  $D[i, j] \leftarrow D[i - 1, j - 1]$ 
    return  $D[m, n]$ 
```

Metodo alternativo: utilizzare due array al posto della matrice

Algorithm 24 distanza_stringhe(s1, s2)

```

1:  $n \leftarrow$  lunghezza di s1
2:  $m \leftarrow$  lunghezza di s2
3: previous_row ← [0, 1, 2, ..., m]
4: current_row ← [0, 0, ..., m]
5: for  $i = 1$  to n do
6:     current_row[0] ← i
7:     for  $j = 1$  to m do
8:         if  $s1[i - 1] = s2[j - 1]$  then
9:             difference ← 0
10:            else
11:                difference ← 1
12:            end if
13:            current_row[j] ← min(
14:                previous_row[j] + 1,
15:                current_row[j - 1] + 1,
16:                previous_row[j - 1] + difference)
17:        end for
18:        Scambia previous_row con current_row
19:    end for
20:    return previous_row[m]
```

Knapsack problem:

$values = [60, 100, 120]$

$weights = [10, 20, 30]$

$capacity = 50$

Items	0	10	20	30	40	50
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60	100	160	180	220

Si utilizza una matrice di due row che alterniamo come soluzione corrente e precedente. Si scorre la matrice fissando la riga e andando fino a capacity. Dove togliere $i - 1$ per considerare il peso e il valore attuale che sto considerando perché i cicli partono da 0.

Algorithm 25 knapsack(values, weights, capacity)

```

1: kns ← matrice [0...1][0...capacity] {Matrice per risparmiare memoria}
2: items ← lunghezza(values)
3: for  $i = 0$  to items do
4:     for  $c = 0$  to capacity do
5:         if  $i = 0$  or  $c = 0$  then
6:             kns[i mod 2][c] ← 0
7:         else if  $c \geq weights[i - 1]$  then
8:             kns[i mod 2][c] ← max(
9:                 values[i - 1] + kns[(i - 1) mod 2][c - weights[i - 1]],
10:                 kns[(i - 1) mod 2][c])
11:         else
12:             kns[i mod 2][c] ← kns[(i - 1) mod 2][c]
13:         end if
14:     end for
15: end for
16: return kns[items mod 2][capacity]
```

Notazione:

- estremo di for è incluso, ossia $i \leq items$
- per far capire che si esclude l'estremo usare nella matrice $[0 \dots 2)$

Problema costo min macchina

Esame 4 aprile 2024: Costo Minimo per Raggiungere il km N

Algorithm 26 costo_minimo (N, r, n, D, C) \rightarrow intero

```

1:  $D[0] \leftarrow 0$ 
2:  $C[0] \leftarrow 0$ 
3:  $D[n + 1] \leftarrow N$ 
4:  $C[n + 1] \leftarrow 0$ 
5:  $DP[0 \dots n + 1] \leftarrow$  inizializzalo con  $\infty$ 
6:  $DP[n + 1] \leftarrow 0$ 
7: for  $i \leftarrow n$  downto 0 do
8:    $costo\_min \leftarrow \infty$ 
9:   for  $j \leftarrow i + 1$  to  $n + 1$  do
10:    if  $D[j] \leq D[i] + r$  then
11:       $costo\_min \leftarrow \min(costo\_min, C[i] + DP[j])$ 
12:    end if
13:   end for
14:    $DP[i] \leftarrow costo\_min$ 
15: end for
16: return  $DP[0]$ 
```

Remark:

- $D[0] \leftarrow 0$ km 0 non percorro alcun kilometro
- $C[0] \leftarrow 0$ parto con il serbatoio pieno alcun costo
- $D[n + 1] \leftarrow N$ destinazione finale al N km
- $C[n + 1] \leftarrow 0$ stazione a destinazione costo 0
- $DP[n + 1] \leftarrow 0$ costo totale 0
- Vado a scorrirete tutte le stazioni i-esima fissata fino alla stazione più lontana j (che scorre) se la stazione i-esima raggiunge la j-esima data la distanza. Se così fosse vado a prendere il costo minimo da stazione a stazione fino alla j-esima e lo piazzo poi in $DP[i]$. Il costo minimo è dato dalla somma di $C[i]$ alla stazione raggiunta j + il valore presente in $DP[j]$.
- Dunque il costo minimo $i - esimo$ è un costo minimo si va ad aggiornare fino alla $j - esima$ stazione e si aggiorna come un **accumulatore rispetto agli altri costi** di conseguenza ottenendo il minore possibile in funzione di tutte le altre fino all'ultima stazione.
- Il valore finale si trova in $DP[0]$
- Costo cumulato è $DP[j]$

Esempio:

$D = [0, 3, 6, 10, 15]$ distanze delle stazioni

$C = [0, 5, 2, 8, 0]$ costi delle batterie in ciascuna stazione

$n = 4$ numero di stazioni di servizio

$r = 6$ autonomia della batteria

$N = 15$ lunghezza dell'autostrada

Iterazioni

1. Stazione $i = 3$ (Distanza 10, Costo 8)

- Cerchiamo tutte le stazioni $j > 3$ raggiungibili da 10 con autonomia $r = 6$.
- $D[4] = 15$ è raggiungibile ($10 + 6 \geq 15$).
- Aggiorniamo $DP[3]$:

$$DP[3] = \min(\infty, C[3] + DP[4]) = \min(\infty, 8 + 0) = 8$$

Risultato aggiornato: $DP = [\infty, \infty, \infty, 8, 0]$

2. Stazione $i = 2$ (Distanza 6, Costo 2)

- Cerchiamo tutte le stazioni $j > 2$ raggiungibili da 6 con autonomia $r = 6$.
- $D[3] = 10$ è raggiungibile ($6 + 6 \geq 10$).
- $D[4] = 15$ è raggiungibile ($6 + 6 \geq 15$).
- Aggiorniamo $DP[2]$:

$$DP[2] = \min(\infty, C[2] + DP[3]) = \min(\infty, 2 + 8) = 10$$

$$DP[2] = \min(10, C[2] + DP[4]) = \min(10, 2 + 0) = 2$$

Risultato aggiornato: $DP = [\infty, \infty, 2, 8, 0]$

3. Stazione $i = 1$ (Distanza 3, Costo 5)

- Cerchiamo tutte le stazioni $j > 1$ raggiungibili da 3 con autonomia $r = 6$.
- $D[2] = 6$ è raggiungibile ($3 + 6 \geq 6$).
- $D[3] = 10$ è raggiungibile ($3 + 6 \geq 10$).
- $D[4] = 15$ è raggiungibile ($3 + 6 \geq 15$).
- Aggiorniamo $DP[1]$:

$$DP[1] = \min(\infty, C[1] + DP[2]) = \min(\infty, 5 + 2) = 7$$

$$DP[1] = \min(7, C[1] + DP[3]) = \min(7, 5 + 8) = 7$$

$$DP[1] = \min(7, C[1] + DP[4]) = \min(7, 5 + 0) = 5$$

Risultato aggiornato: $DP = [\infty, 5, 2, 8, 0]$

4. Stazione $i = 0$ (Distanza 0, Costo 0)

- Cerchiamo tutte le stazioni $j > 0$ raggiungibili da 0 con autonomia $r = 6$.
- $D[1] = 3$ è raggiungibile ($0 + 6 \geq 3$).
- $D[2] = 6$ è raggiungibile ($0 + 6 \geq 6$).
- $D[3] = 10$ è raggiungibile ($0 + 6 \geq 10$).
- Aggiorniamo $DP[0]$:

$$DP[0] = \min(\infty, C[0] + DP[1]) = \min(\infty, 0 + 5) = 5$$

$$DP[0] = \min(5, C[0] + DP[2]) = \min(5, 0 + 2) = 2$$

$$DP[0] = \min(2, C[0] + DP[3]) = \min(2, 0 + 8) = 2$$

Risultato finale: $DP = [2, 5, 2, 8, 0]$

Cardinalità massima taglio pezzi

Algorithm 27 tubo_metallico(L, S) → intero

```
1:  $n \leftarrow$  lunghezza di  $S$ 
2: Inizializza la tabella  $DP[0 \dots L] \leftarrow 0$ 
3: for  $i = 1$  to  $L$  do
4:   for  $j = 0$  to  $n - 1$  do
5:     if  $S[j] \leq i$  then
6:        $DP[i] \leftarrow \max(DP[i], DP[i - S[j]] + 1)$ 
7:     end if
8:   end for
9: end for
10: return  $DP[L]$ 
```

- $DP[i]$ rappresenta il numero massimo di segmenti che si possono ottenere con un tubo di lunghezza i .
- Per ogni lunghezza i , controlliamo se possiamo utilizzare il segmento $S[j]$ e, in caso affermativo, aggiorniamo il valore di $DP[i]$.
- La complessità computazionale di questo algoritmo è $O(n \times L)$, dove n è il numero di segmenti e L è la lunghezza del tubo.
- **NB:** voglio massimizzare il numero effettivo di segmenti che posso tagliare con L . Dunque i sono le varie lunghezze del tubo che arrivano fino ad L mentre j sono i segmenti che posso tagliare.

17.4 Esercizio treno

Costo minimo totale, giorni per acquistare l'abbonamento $\rightarrow O(n)$

Algorithm 28 MinCost(n, D, F, X)

```
1:  $dp \leftarrow$  array di dimensione  $n + 1$  inizializzato a  $\infty$ 
2:  $dp[0] \leftarrow 0$ 
3:  $buy\_pass\_days \leftarrow$  lista vuota
4: for  $i = 0$  to  $n - 1$  do
5:    $dp[i + 1] \leftarrow dp[i] + F[i]$ 
6:    $j \leftarrow i + 1$ 
7:   while  $j > 0$  and  $D[i] - D[j - 1] < 30$  do
8:      $j \leftarrow j - 1$ 
9:   end while
10:  if  $dp[j] + X < dp[i + 1]$  then
11:     $dp[i + 1] \leftarrow dp[j] + X$ 
12:    ADD  $D[j]$  TO  $buy\_pass\_days$ 
13:  end if
14: end for
15: return  $dp[n], buy\_pass\_days$ 
```

n: il numero totale di viaggi

D: un array che contiene i giorni in cui si svolgono i viaggi

F: un array che contiene i costi per ciascun viaggio

X: il costo dell'abbonamento mensile

- **Riga 1:** viene creato un array dp di dimensione $n + 1$ (per includere un elemento per il caso senza abbonamento) e inizializzato a ∞ per rappresentare un costo massimo iniziale.
- **Riga 2:** imposto $dp[0]$ a 0 perché non ci sono costi associati se non ci sono viaggi programmati.
- **Riga 3:** Crea una lista vuota per memorizzare i giorni in cui conviene acquistare un abbonamento.
- **Riga 5:** calcolo del costo biglietto singolo, aggiunge il costo del viaggio attuale $f[i]$ al costo minimo precedente $dp[i]$.
- **Riga 6:** viene utilizzato per cercare i giorni in cui conviene acquistare un abbonamento. Parto da $j = i + 1$ in maniera tale da non arrivare a considerare $D[i] - D[-1]$.
- **Riga 7-8-9:** quest ciclo decrementa j finché $j > 0$ cioè ci sono giorni da controllare. La differenza tra il giorno del viaggio attuale $D[i]$ e il giorno $D[j-1]$ è inferiore a 30. Questo serve per identificare i viaggi che rientrano nella validità di un abbonamento di 30 giorni.
- **Riga 10:** controllo di convenienza dell'abbonamento, verifica se il costo totale ottenuto utilizzando l'abbonamento, cioè $dp[j] + X$ (costo fino al giorno j più il costo dell'abbonamento), è inferiore al costo attuale per i biglietti singoli $dp[i + 1]$.
- **Riga 11-12:** se l'abbonamento è più conveniente, aggiorna il costo per il viaggio $i+1$ in dp al costo ottenuto con l'abbonamento e aggiungi $D[i]$ alla lista buy_pass_days poiché questo è un giorno in cui conviene acquistare un abbonamento.

17.5 Grafi

Un grafo

Vertici/spigoli

Lati

Lato (A,B) è incidente

Numero vertici

Numero archi

Cammino semplice

Lunghezza cammino

Grafo connesso

Lati incidenti al nodo I, grado

Albero

DAG

Visita da un grafo

Costo di visita

$$G = (V, E)$$

$$V = \{A, B, C\}$$

$$E = \{(A, B), (B, C)\}$$

su i vertici A e B

$$n = |V|$$

$$m = |E|$$

$$k = < L, I, E, C, B, A >$$

$$|k| = 5$$

$$\exists k : \forall A, B \in V$$

$$\delta(I)$$

non orientato connesso e aciclico

orientato connesso e aciclico

si parte da un nodo sorgente s

genera un albero di copertura

$$O(n + m)$$

Algorithm 29 visitaBFS(vertice s) → albero

```

1: rendi tutti i vertici non marcati
2:  $T \leftarrow$  albero formato da un solo nodo  $s$ 
3: Coda  $F$ 
4: marca il vertice  $s$ 
5:  $F.enqueue(s)$ 
6: while not  $F.isEmpty()$  do
7:    $u \leftarrow F.dequeue()$ 
8:   for each arco  $(u, v)$  in  $G$  do
9:     if  $v$  non è ancora marcato then
10:       $F.enqueue(v)$ 
11:      marca il vertice  $v$ 
12:      rendi  $u$  padre di  $v$  in  $T$ 
13:    end if
14:   end for
15: end while
16: return  $T$ 

```

Algorithm 30 visitaDFS(vertice s) → albero

```

1: rendi tutti i vertici non marcati
2:  $T \leftarrow$  albero formato da un solo nodo  $s$ 
3: dfs( $s, T$ )
4: return  $T$ 

```

Algorithm 31 dfs(vertice u, albero T)

```

1: marca il vertice  $u$ 
2: for each arco  $(u, v)$  in  $G$  do
3:   if  $v$  non è ancora marcato then
4:     aggiungi l'arco  $(u, v)$  a  $T$ 
5:     dfs( $v, T$ )
6:   end if
7: end for

```

Algorithm 32 visitaDFS(vertice s) → albero

```

1: rendi tutti i vertici non marcati
2:  $T \leftarrow$  albero formato da un solo nodo  $s$ 
3: Pila  $S$ 
4: marca il vertice  $s$ 
5:  $S.push(s)$ 
6: while not  $S.isEmpty()$  do
7:    $u \leftarrow S.pop()$ 
8:   for each arco  $(u, v)$  in  $G$  do
9:     if  $v$  non è ancora marcato then
10:       $S.push(v)$ 
11:      marca il vertice  $v$ 
12:      rendi  $u$  padre di  $v$  in  $T$ 
13:    end if
14:   end for
15: end while
16: return  $T$ 

```

18 Esercizio 4

18.1 Architecture validation

MTTF (Mean Time To Failure): the average time that a non-repairable component or system is expected to operate before it fails, is often used for products or components that cannot be repaired, a higher MTTF means that the product is more reliable as it is expected to operate for a longer period before a failure occurs.

MTTF Expected time before next failure, defined as $MTTF = E[X]$

X Random variable representing the time of failure of a component

F(t) Cumulative distribution function (CDF) of X, representing $P(X \leq t)$

R(t) Reliability function, the P that the component is still working at time t

A availability, the component is ready to be used?

$$F(t) = P(X \leq t) = 1 - e^{-\frac{t}{MTTF}}$$

$$\begin{aligned} R(t) &= P(X > t) = 1 - F(t) \\ &= e^{-\frac{t}{MTTF}} \end{aligned}$$

How to increase reliability: use multiple redundant components, use element internally high reliable (high MTTF), the working state of the whole system depends on how many and which components are operative, its behaviour can be represented using reliability diagrams **RBD (Reliability Block Diagram)**

Components in serial

non redundant

Components in parallel

redundant

Reliability in serial

$$R = \prod R_i$$

Reliability in parallel

$$R = 1 - \prod(1 - R_i)$$

MTTR (Mean Time To Repair): the average time required to repair a failed component or system and return it to operational status, a lower MTTR means that repairs are faster reducing downtime and increasing the availability of the system.

Availability: represents the probability (stationary mean) that a given instant of time the system is working:

$$A = \frac{MTTF}{MTTF + MTTR}$$

Availability in serial

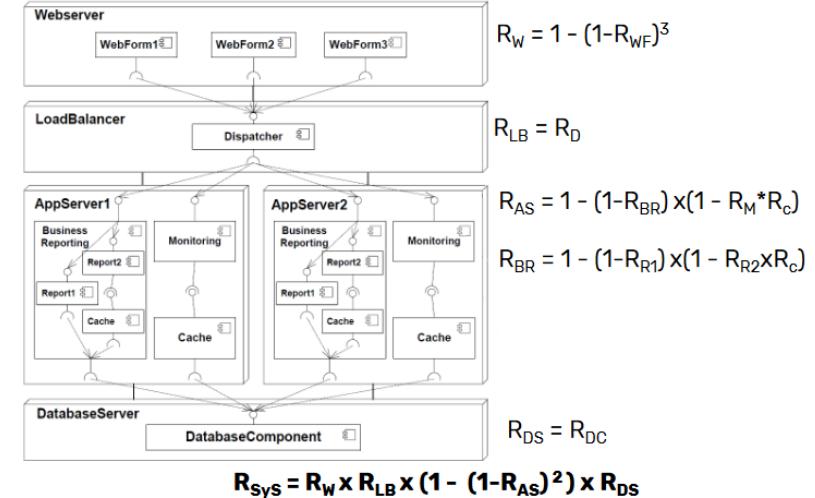
$$A = \prod A_i$$

Availability in parallel

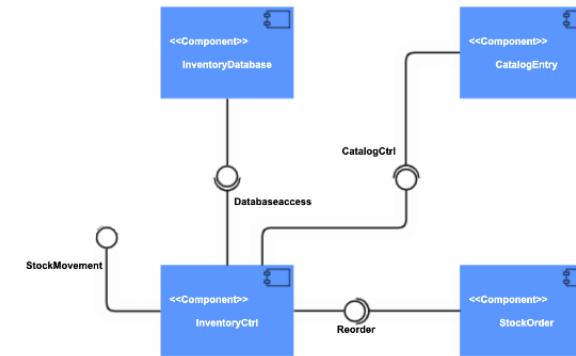
$$A = 1 - \prod(1 - A_i)$$

Reliability provides insight into the expected lifespan of a system/component (how long until it fails), while **Availability** gives a broader picture of how often the system is ready for use, considering both its operational and repair times.

$$\begin{aligned} R &= R_{lb} \cdot R_{web} \\ &= R_{lb} \cdot [1 - (1 - R_{web}) \cdot (1 - R_{web}) \cdot (1 - R_{web})] \\ &= R_{lb} \cdot [1 - (1 - R_{web})^3] \end{aligned}$$



Creazione RBD: gennaio 2024



- Ogni interfaccia identifica un componente, se non dovesse essere presente il componente risulta essere anonimo
- I componenti sono in parallelo esterni se usufruiscono di una interfaccia sono in parallelo rispetto a quello di riferimento
- Provided interface:** ball
- Required interface:** socket

- Quando abbiamo componenti tutti diversi si vengono a delineare a partire dal parallelo tre diversi flussi serie di interazione con calcolo dell'affidabilità seguente:

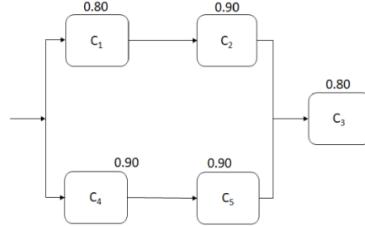
$$S = (SM \parallel CE \parallel SO) + IC + ID$$

$$R_1 = R_{SM} + R_{IC} + R_{ID}$$

$$R_2 = R_{CE} + R_{IC} + R_{ID}$$

$$R_3 = R_{SO} + R_{IC} + R_{ID}$$

Esercizio:



$$R_{12} = R_{c1} \cdot R_{c2}$$

$$R_{45} = R_{c4} \cdot R_{c5}$$

$$R_{s1} = 1 - (1 - R_{12})(1 - R_{45})$$

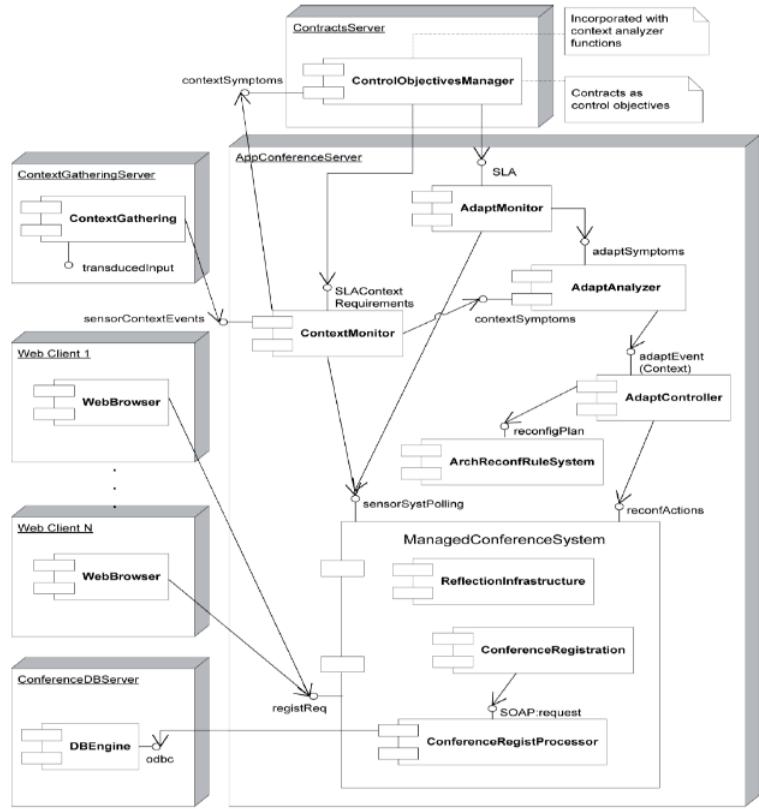
$$(C_3 \text{ guasto}) R_S = R_{s1} \cdot R_{c3} = 0$$

18.2 Tipi di architettura

- Pipe and Filter
- Big Data Pipeline on Cloud
- Multilayer
- Client-server
- Two-tier con three layers
- Model-View-Controller (MVC)
- Three-tier MVC (piu usato)
- Broker
- Message-Oriented Architecture
- Service-Oriented Architecture - SOA
- Architettura esagonale
- Architettura a microservizi

- Cloud computing
- Edge computing
- MAPE-K
- API-led architecture

Ese 1:



Architettura software di un sistema auto-adattativo di gestione delle conferenze. Sono presenti due cicli di controllo nella forma **MAPE-K** per via delle due componenti M (Monitor), una componente di Analisi (A) e componenti P+E fuse in un'unico componente C (Controller) che è AdaptController. I due componenti M , ovvero ContextMonitor e AdaptMonitor, comunicano entrambi con il componente A che è AdaptAnalysis. Un altro semplice pattern applicato più volte è il **client-server** nella forma three-tier (a tre strati).

MAPE-K: è composto da monitor, analyzer, planner, executor e k sta per knowledge.