

Complessità e formule associate

- $f(n)$  è  $O(g(n))$  se  $\exists n_0, c > 0 : \forall n \geq n_0 f(n) \leq c g(n)$
- $f(n)$  è  $\Omega(g(n))$  se  $\exists n_0, c > 0 : \forall n \geq n_0 f(n) \geq c g(n)$
- $f(n)$  è  $\Theta(g(n))$  quando  $f$  e  $g$  hanno lo stesso ordine di complessità ( $O + \Omega$ )
- Complessità: funzione sempre positiva che associa alla dimensione del problema il costo della sua risoluzione
- Algoritmo: Insieme finito di istruzioni tese a risolvere un problema. Ha Input ed Output. Ogni istruzione deve essere ben definita ed eseguibile in un tempo finito da un calcolatore. I risultati intermedi si possono salvare in memoria.

Prog Iterativi

$$- C[\text{if}(\epsilon) \text{c1} \text{else} \text{c2}] = C[\epsilon] + C[c_1] + C[c_2]$$

$$- C[\text{for}(E_1; E_2; E_3; ) \text{C}] = C[E_1] + C[E_2] + C[E_3] + C[C] + C[E_2] \Big\} \Theta(g(n))$$

$$- C[\text{while}(\epsilon) \text{C}] C[\epsilon] + (C[\epsilon] + C[C]) g(n)$$

Prog Ricorsivi

$$T(0) = d \quad \text{se } n \leq m$$

$$T(n) = b n^k + a T(n/b) \quad \begin{cases} T(n) \in O(n^k) \text{ se } a < b^k \\ T(n) \in O(n^k \log n) \text{ se } a = b^k \end{cases}$$

$$\quad \quad \quad \begin{cases} T(n) \in O(n^{\log_b a}) \text{ se } a > b^k \end{cases}$$

## Algoritmi di Teoria dei Numeri:

Sicoltà La complessità a partire dal numero di cifre del numero.

### Relazioni Lineari

$$T(0) = d$$

$$T(n) = bn^k + a_1 T(n-1) + a_2 T(n-2) + \dots + a_r T(n-r)$$

Questa risulta polinomiale se esiste al più uno  $a_i = 1$ , tutti gli altri  $= 0$   
Altimenti esponenziale

Esempio

$$T(n) = bn^k + T(n-1) \text{ è } O(n^{k+1})$$

$$T(n) = bn^k + 2T(n-1) \text{ è } O(2^n) \text{ [esponenziale!!!]}$$

~~Metodo Induttivo~~

### Induzione Naturale

Sia  $p$  una proprietà dei Naturali

Base:  $p$  vale per  $\emptyset$

Passo Induttivo: per ogni  $n \in \mathbb{N}$  se  $p$  vale per  $n \Rightarrow p$  vale per  $(n+1)$

### Induzione Completa

Sia  $p$  una proprietà sui naturali

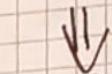
Base:  $p$  vale per  $\emptyset$

Passo Induttivo:  $\forall n \in \mathbb{N}$  se  $p$  vale per ogni  $m \leq n \Rightarrow p$  vale per  $(n+1)$

## Induzione Ben Fondata

Basis: Vale per i minimi di S

Passo Induttivo:  $\forall E \in S \Rightarrow E \text{ vale} \Rightarrow \forall E' \in S : E' < E \Rightarrow E' \text{ vale}$



Vale per S

## Selection Sort

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

$O(n^2)$

## QuickSort

```
void quickSort(int *A, int inf = 0, int sup = n - 1) {
    int perno = A[(inf + sup) / 2], s = inf, d = sup;
    while (s <= d) {
        while (A[s] < perno) s++;
        while (A[d] > perno) d--;
        if (s > d) break;
        swap(A[s], A[d]);
        s++; d--;
    }
    if (inf < d) quickSort(A, inf, d);
    if (s < sup) quickSort(A, s, sup);
}
```

$$T(1) = 3$$

$$T(n) = bn + T(k) + T(n-k)$$

$$\text{se } k=1 \Rightarrow O(n^2)$$

$$\text{se } k = n/2 \Rightarrow O(n \log n)$$



caso medio se le permut.  
Sono equiprobabili  
(equivale al caso migliore)

Ottimo nel caso medio

Linear Search  $O(n)$

bin Search  $O(\log n)$  // vettore ordinato!

### Merge Sort

$O(n \log n)$

void mergeSort (sequenza S) {

if ( $|S| \leq 1$ ) return;

else {

< dividi S in 2 sottosequenze  $S_1$  e  $S_2$  di

lunghezza uguale;  $\sim O(n)$

mergeSort( $S_1$ );

mergeSort( $S_2$ );

< fonda  $S_1$  ed  $S_2$  >;

}

}

### Esempio di esecuzione

mergeSort([2, 1, 3, 5])

split([2, 1, 3, 5])

mergeSort([2, 3])

split([2, 3])

mergeSort([2]) [2]

mergeSort([3]) [3]

merge([2], [3]) [2, 3]

mergeSort([1, 5])

split([1, 5])

mergeSort([5]) [5]

mergeSort([1]) [1]

merge([5], [1]) [1, 5]

merge([2, 3], [1, 5]) [1, 2, 3, 5]

# ALBERI

## Alberi Binari

- Null è un albero binario
- un nodo  $P$  più due alberi binari  $L_B, R_B$  forma un albero binario
  - $P$  è radice
  - $L_B$  sottoalbero sx
  - $R_B$  " " dx
  - alberi etichettati

## Visite su alberi binari

• PreOrder (Visita anticipata)	• PostOrder (Visita differita)
void preOrder(albero){ if (!albero) return; esamina la radice; preOrder(LB); preOrder(RB);}	Come preOrder, ma esaminano la radice dopo le due call ricorsive  • InOrder (Visita simmetrica) Come sopra, ma esaminano la radice tra la 1^a e la 2^a call ricorsiva

## Memorizzazione in Lista multipla

```
struct Node{  
    InfoType label;  
    Node *left;  
    Node *right;};
```

## Complessità Visite (numero nodi)

$$T(0) = a$$

$$T(n) = b + T(n_s) + T(n_d) \text{ con } n_s + n_d = n - 1$$

$$\text{nel caso } n_s = n_d = (n-1)/2 \Rightarrow T(n) = b + 2T((n-1)/2) \Rightarrow T(n) \in O(n)$$

## Albero binario bilanciato

I nodi di tutti i livelli eccetto l'ultimo hanno 2 figli

Un albero bilanciato con livello  $K$  ha: (si parte da 0!)

- $2^{K+1} - 1$  nodi
- $2^K$  foglie

## Albero binario quasi bilanciato

Un albero bilanciato fino al penultimo livello

## Albero pienamente binario

Tutti i nodi eccetto le foglie hanno 2 figli

Il numero di nodi interni è uguale al numero di foglie - 1

## Complessità visite (livelli)

Se l'albero è bilanciato

$$T(0) = a$$

$$T(K) = b + 2T(K-1)$$

$$O(2^K)$$

## Albero generico

- un nodo  $p$  è un albero
- un nodo più una sequenza di alberi  $A_1, A_2, \dots, A_n$  è un albero

Le visite disponibili sono

- preOrder
- postOrder

### Memorizzazione Figlio - Fratello:

- primo figlio a sinistra
- primo fratello a destra

Utilizzando questa memorizzazione otteniamo un "trasformato" sul quale:

- la visita PreOrder corrisponde alla Preorder dell'albero generico.
- la visita InOrder corrisponde alla PostOrder dell'albero generico.

### Albero binario di Ricerca

Esso è un albero binario tale che per ogni nodo p

- i nodi del LB hanno etichetta minore di p
- i nodi del RB hanno etichetta maggiore di p.

Un albero così costruito non contiene doppiani e fa visita simmetrica restituisc le etichette in ordine crescente

### Operazioni permesse

- Ricerca di un nodo
- Inserimento di un nodo
- Cancellazione di un nodo

#### Ricerca

$$T(0) = a$$

$$T(n) = b + T(k) \text{ con } k < n$$

$$\text{caso medio: } T(n) = b + T(n/2) \Rightarrow O(\log n)$$

$$\Rightarrow \text{caso peggiore: } T(n) = b + T(n-1) \Rightarrow O(n)$$

#### Inserimento

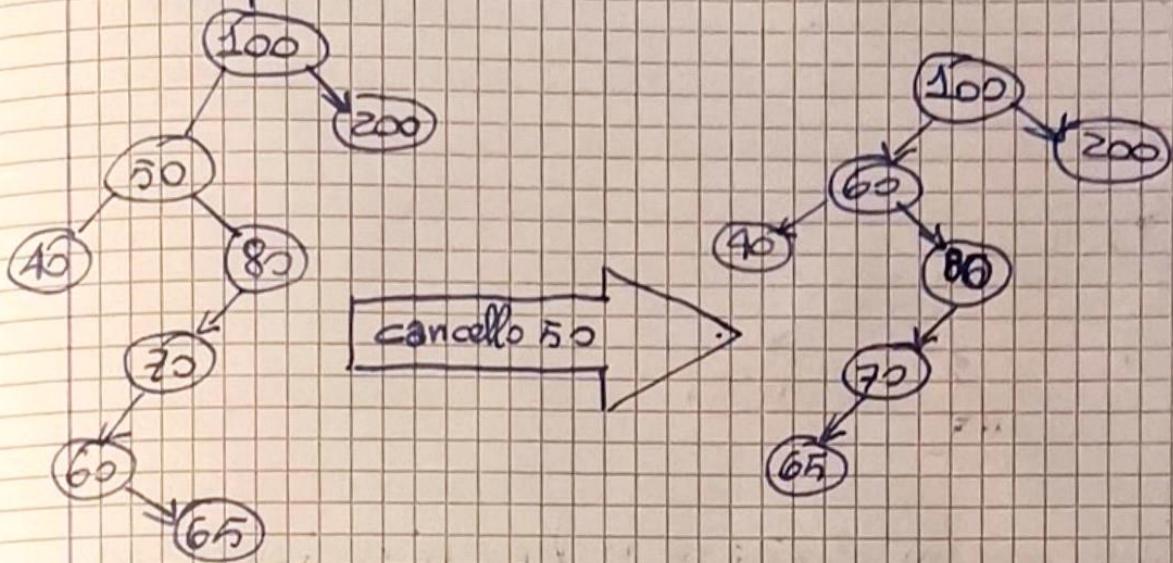
$$O(\log(n))$$

#### Cancellazione

Restituisce l'etichetta del Nodo più piccolo di un albero ed elimina il nodo che lo contiene

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

Esempio



### Limiti inferiori per i problemi

Un problema si dice di origine  $\Omega(f(n))$  se non è possibile trovare un algoritmo che lo risolva con complessità minore di  $f(n)$

[Tutti gli altri algoritmi hanno complessità  $\Omega(f(n))$ ]

### Alberi di decisione

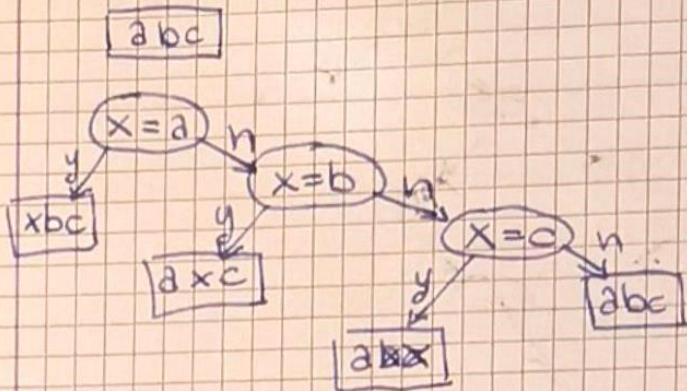
Si applicano soltanto agli algoritmi:

- basati su confronti;
- che hanno complessità proporzionale al numero di confronti effettuati durante l'esecuzione dell'algoritmo.

Essi sono alberi binari che corrispondono all'algoritmo:

- Ogni foglia rappresenta una soluzione per un particolare assetto dei dati iniziali.
- Ogni cammino dalla radice ad una foglia rappresenta una esecuzione dell'algoritmo (sequenza di confronti) per giungere alla soluzione relativa alla foglia.

## Esempio, Ricerca Lineare



Ogni algoritmo con  $s(n)$  soluzioni ha un albero di decisione con almeno  $S(n)$  foglie

Un algoritmo ottimo nel caso peggiore (medio) ha il più corto cammino max (medio) dalla radice alle foglie

### Extra sugli alberi binari:

- Un albero binario con  $K$  livelli ha un massimo di  $2^K$  foglie (quando è bilanciato)
- Un albero binario con 5 foglie ha almeno  $\log_2 5$  livelli;
- Gli alberi binari bilanciati minimizzano sia il caso peggiore che quello medio; hanno  $\log s(n)$  livelli

La ricerca è in generale  $O(n \log n)$ , salvo casi particolari quali:

- Counting Sort
- radix sort.

## Counting Sort

- Ordina una sequenza di interi
- Si può usare se si conoscono min e max dell'insieme da ordinare
- Per ogni valore nell'array, si contano gli elementi con quel valore utilizzando un array ausiliario avente come dimensione l'intervalle dei valori
- Si ordinano i valori, tenendo conto dell'array ausiliario

Non è basato sui confronti ed ha complessità  $O(n + \max)$ .

E' conveniente quando max è  $O(n)$ , e necessita di memoria ausiliaria.

## Radix Sort

- Ordina una sequenza di interi
- Si può usare quando si conosce la lunghezza max (n.cifre)<sup>d</sup> dei numeri da ordinare
- Si eseguono d passate ripartendo, in base alla d-esima cifra, i numeri in K contenitori, dove K sono i possibili valori di una cifra, e rileggendo il risultato con un determinato ordine.

### Esempio

$s (d=3, K=10) : 190 | 051 | 207 | 010 |$

fattura	1) 190   051	2) 207   010	3) 051   190	risultato
1)	190   051	207   010	051   190	$\Rightarrow 190, 051, 207$
2)	051   190	207   010	190   051	$\Rightarrow 207, 010, 051, 190$
3)	190   051	010   207	051   190	$\Rightarrow 010, 051, 190, 207$

Questo ordinamento:

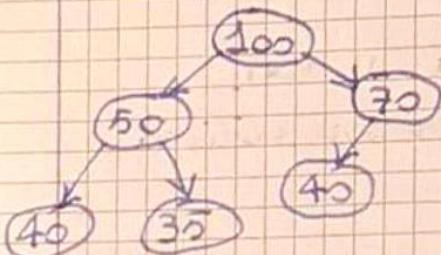
- Non è basato sui confronti
- È fondamentale partire dalla cifra meno significativa
- è  $O(d(n+K))$
- Conveniente per  $d \ll n$
- Necessaria memoria ausiliaria

## Heap

Lo Heap è un albero binario quasi bilanciato con le proprietà:

- I nodi dell'ultimo livello sono addossati a sinistra
- In ogni sottoalbero l'etichetta della radice è maggiore o uguale a quella di tutti i discendenti.

## Memorizzazione in array



0	1	2	3	4	5	6	7
100	50	70	40	35	40		

- figlio sinistro di  $i = 2i+1$
- figlio destro di  $i = 2i+2$
- padre di  $i = (i-1)/2$

## Operazioni

- Inserimento di un nodo
- Estrazione dell'elemento maggiore

## Inserimento ( $O(\log n)$ )

L'elemento viene inserito nell'ultima posizione ( $\text{heap}[\text{++last}] = \text{elem}$ ) viene poi chiamata una funzione per mantenere le proprietà dello heap che scambia l'elemento con il padre finché il padre non risulta maggiore del figlio, o ci si trova in testa (si è divenuti la nuova radice)

## Estrazione

- Si restituisce il primo elemento (radice)
- L'ultimo elemento viene messo primo e  $\text{last}--$
- tramite una funzione si fa scendere il primo elemento (down)

down:  $O(\log n)$

son = figlio sinistro

- se son è l'ultimo dell'array ed è maggiore di i (ed è l'unico figlio) scambia, altrimenti termina
- Se ha 2 figli si prende l'indice del maggiore dei due e se son > i si scambiano; poi si chiama down(son) sulla nuova posizione di i.  
Altrimenti termina

Heapsort  $O(n \log n)$

- trasforma l'array in un heap  $(O(n))$
- esegui n volte l'estrazione scambiando il primo elemento con quello puntato da "last"  $(O(n \log n))$

build heap:

- Esegui la funzione down sulla prima metà degli elementi dell'array (l'altra metà sono foglie)
- Esegui down partendo dall'elemento centrale e tornando indietro fino al primo

Esempio

0	1	2	3	4
38	40	45	50	35

A

heapSort(A, 5)	38, 40, 45, 50, 35
buildHeap(A, 4)	38, 40, 45, 50, 35
down(A, 2, 4)	38, 40, 45, 50, 35
down(A, 1, 4)	38, 50, 45, 40, 35
down(A, 0, 4)	50, 40, 45, 38, 35
extract(A, 4)	45, 40, 35, 38, 50
extract(A, 3)	40, 38, 35, 45, 50
extract(A, 2)	38, 35, 40, 45, 50
extract(A, 1)	35, 38, 40, 45, 50

## Metodo di Ricerca Hash

- metodo di ricerca in array
- non basato sui confronti
- molto efficiente

$h$  (funzione hash) : InfoType  $\rightarrow$  indici

Sia  $n$  = numero massimo di elementi

$K$  = dimensione array

$h$  è iniettiva: (Accesso Diretto)

$h(x)$  = indirizzo hash dell'elemento che contiene  $x$

Complessità  $O(1)$

Problema:

$K$  può diventare enorme, con conseguente spreco di memoria

### Esempio

Sapere che cifre appaiono

apparizioni cifra	
1	0
0	1
2	2
0	3
0	4
0	5
0	6
1	7
0	8
0	9

$\{0, 2, 7\} \rightarrow$

$n = 5$  (insiemi di max 5 cifre)

$K = 10$

$n/K = 0,5$

$$h(x) = x$$

### Accesso Non Diretto

$$h(x_1) = h(x_2) \quad (h \text{ non è più iniettiva})$$

Ho due nuovi problemi:

- Come trovo elementi collisi?
- Come inserisco gli elementi?

## 1<sup>a</sup> Soluzione: hash ad indirizzamento aperto

- $h(x) = (x \% K)$

- Scansione lineare: Se non si trova l'elemento al suo posto, lo si cerca nei successivi (Idem per l'inserimento)

Esempio

$$n=K=5$$

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

$$n/K=1$$

$$\{0, 2, 7\}$$

$$h(0)=0$$

$$\begin{array}{r} 0 \\ 0 \end{array}$$

$$h(2)=2$$

$$\begin{array}{r} 1 \\ 1 \\ 2 \\ 2 \end{array}$$

$$h(7)=2$$

$$\begin{array}{r} 7 \\ 3 \\ 1 \\ 4 \end{array}$$

Conseguenze:

Agglomerato: gruppo di elementi con indirizzi hash uguali diversi

La presenza di collisioni ed agglomerati aumenta il tempo di ricerca

$$\text{Scansione\_lineare}(x) = (h(x) + \text{cost. } j) \% K$$

$$\text{scansione\_quadratica}(x) = (h(x) + j^2 * \text{cost}) \% K$$

La diversa lunghezza del passo di scansione riduce gli agglomerati

Il tempo medio di Ricerca dipende da:

- Fattore di carico:  $\alpha = n/K$  (must be  $< 1$ ) che è il numero medio di elementi per ogni posizione

- Legge di scansione: (doppio hash  $\rightarrow$  quadratica  $\rightarrow$  lineare)

Con la scansione lineare abbiamo un numero di accessi  $\leq \frac{1}{1-\alpha}$

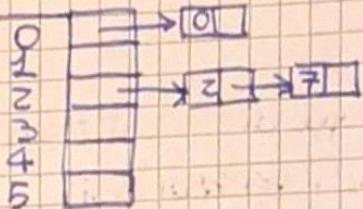
- Uniformità della funzione hash: (quanto sono equi probabili gli indici generati)

Problemi:

Molti inserimenti e cancellazioni degradano il tempo di ricerca, rendendo necessario "risistemare" l'array.

## Metodo di Concatenazione

- Array A di  $K \leq n$  puntatori ( $n/K \geq 1$ )
- elementi che collidono su i nella lista di puntatore  $A[i]$
- evita del tutto gli agglomerati



## Dizionari

chiave	info

Ricerca

Inserimento

Cancellazione

[Con la chiave = info ~~velocità~~ ]

## Algoritmi che lavorano su memoria non interna

Dando tener conto dei diversi tempi di accesso e delle dimensioni diverse dei livelli di memoria:

- Contiene meno i confronti tra elementi diversi
- Conta di più il numero di operazioni di I/O fra livelli diversi
- Necessità di tenere più dati possibili nelle memorie veloci

Ad esempio nel MergeSort conviene fondere 4 o più liste invece di 2.  
(mergeSort a più vie)

## PROGRAMMAZIONE DINAMICA

Si utilizza quando non è possibile applicare il metodo divide et impera

Metodo: Si risolvono tutti i sottoproblemi a partire dal basso e si conservano i risultati per poterli utilizzare successivamente (strategia bottom-up)

La complessità dipende dal numero di sottoproblemi

Quando si può applicare:

Sottostruzione ottima: una soluzione ottima del problema contiene la soluzione ottima dei sottoproblemi.

Sottoproblemi comuni: un algoritmo ricorsivo richiederebbe di risolvere lo stesso sottoproblema più volte.

PLSC       $O(n^2)$

più Lunga Sotto sequenza Comune

Esempio:  $\alpha = a b c a b b a$      $\beta = c b a b a c$     3 PLSC: baba

cbb a

caba

→ Se due elem. della mat. sono uguali +1 alto s<sub>x</sub>

→ Sennò il max tra sopra e sinistra

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

per trovare la sequenza, partire dal n. max e procedere a sx. per numeri uguali  $\downarrow$  in diag. per i diversi

osu

Risolvendo bottom-up tutti i sottoproblemi:

costruisco tutti gli  $L(i,j)$  di  $\alpha_1 \dots \alpha_i$  e  $\beta_1 \dots \beta_j$  a partire dagli indici più piccoli

lunghezza

## Algoritmi Greedy

- La soluzione ottima si ottiene mediante una sequenza di scelte
- In ogni punto dell'algoritmo viene scelta la strada che in quel momento sembra la migliore
- La scelta locale deve essere in accordo con la scelta globale.

Metodo: top-down. Non sempre si trova la soluzione ottima ma in certi casi si può trovare una soluzione approssimata.

## Codici di Complessione

Alfabeto: insieme di caratteri

Codifica: sostituisce ad ogni carattere del testo il corrispondente codice binario

Il codice può essere a lunghezza fissa o variabile:

### Codici prefissi

Lunghezza variabile, ma nessun codice può essere il prefisso di un altro.

Si cerca di dare la codifica più breve ai caratteri più frequenti.

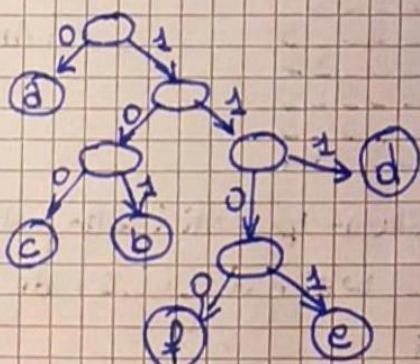
I codici prefissi si possono rappresentare con alberi binari; l'appresentazione ottima genera un albero pienamente binario.

### Esempio

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

- L'albero ha tante foglie quanti sono i caratteri dell'alfabeto

- La decodifica trova un cammino dalla radice ad una foglia per ogni carattere riconosciuto.



## ALGORITMO DI HUFFMAN

Costruisce un codice ottimo dato un alfabeto e fa frequenza dei caratteri.

E' un algoritmo greedy

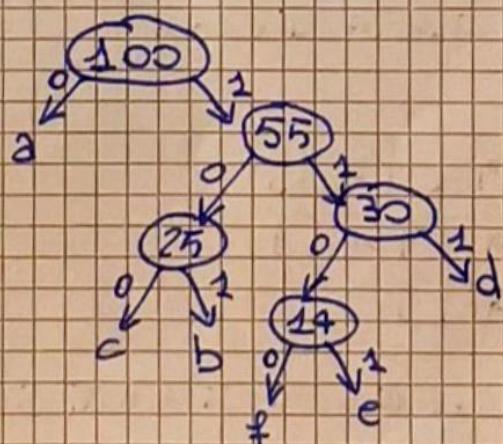
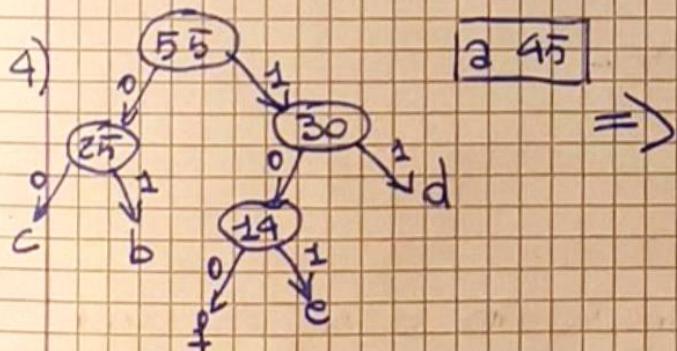
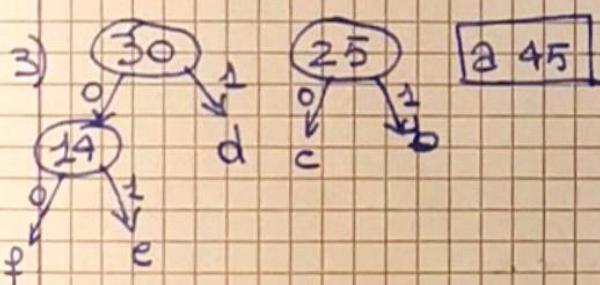
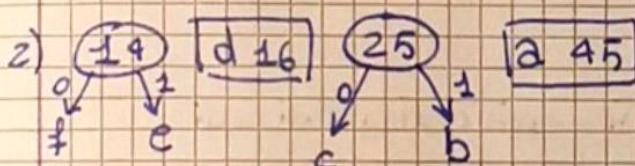
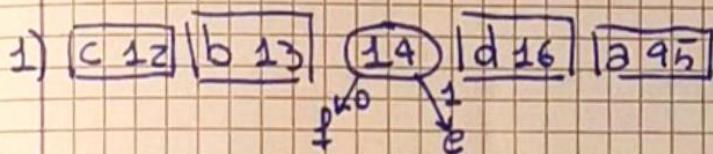
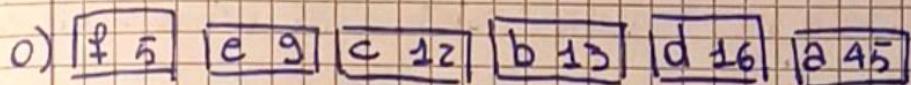
Costruisce l'albero binario in modo bottom-up

## Método:

App' inizio ci sono in alberi di un solo nodo con le frequenze dei cavattori.

Ad ogni passo si fondono i 2 minsh introducendo una nuova radice avente come etichetta la somma delle due radici

### Esempio:



Gli alberi sono memorizzati in un min heap (heap con ordinamento inverso, la radice è il più piccolo).

Si fa un ciclo dove in ogni iterazione:

- Si estraggono i due alberi con radice minore
- Si fondono
- Si inseriscono nel min heap

Abbiamo un ciclo con  $n$  iterazioni, ognuna  $O(\log n)$  (estrazione e un inserimento). L'algoritmo quindi è  $O(n \log n)$ .

La scelta locale è consistente con l'globale; sistemandolo prima i minori essi appariranno all'ultimo livello.

## [GRAFI]

### [GRAFI ORIENTATI]

$(N, A)$  con  $N =$  insieme di nodi

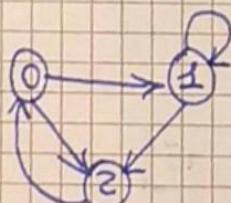
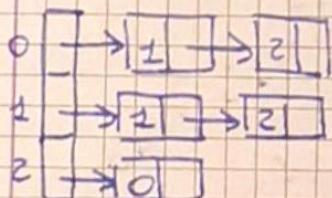
$A \subseteq N \times N =$  insieme di archi (coppie ordinate di nodi)

$$n = |N|; m = |A|$$

un grafo con  $n$  nodi ha max  $n^2$  archi

### Memorizzazioni

#### 1) Liste di adiacenza



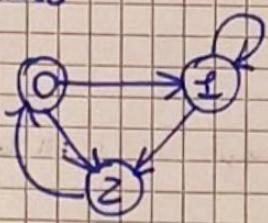
Nel caso di archi etichettati si aggiunge un campo idoneo nella struct

dell'Node ed un vettore con i nomi dei nodi

0	A
1	B
2	C

## 2) Matrici di adiacenza

	0	1	2
0	0	1	1
1	0	1	1
2	1	0	0



Nel caso di archi etichettati si mette l'etichetta al posto di "1" se l'arco è presente, si aggiunge poi un vettore con i nomi dei nodi

### Visita in profondità

```
Void Node visit (node) {
    esamina node;
    marchia node;
    Node visit (Successori di node non marchiati);
}
```

```
Void Depth Visit (grafo) {
    per tutti i nodi:
        node visit (su nodo non marchiato);
}
```

$O(m) + O(n)$

### GRAFI NON ORIENTATI

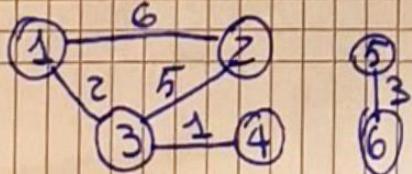
$(N, A)$  con  $N =$  insieme di Nodi

$A =$  insieme di coppie non ordinate di nodi

$n$  nodi  $\rightarrow$  max  $\frac{n(n-1)}{2}$  archi.

Si rappresentano in memoria come gli orientati, considerando che ogni connessione corrisponde a 2 archi nelle due direzioni opposte

### Esempio:



## Multi - Grafi non Orientati

(N, A) con N = insieme di nodi

A = multi-insieme di coppie non ordinate di nodi

Nessuna relazione tra n ed m

Analogamente si definiscono i multi-grafi orientati.

- Un grafo non orientato è connesso se esiste un cammino tra due nodi qualsiasi del grafo
- Componente连通子图: un sottografo connesso
- Componente连通子图 massimale: nessun nodo è connesso ad un'altra componente connessa
- Albero di copertura: insieme di componenti connesse massimali acicliche
- Minimo albero di copertura: la somma dei pesi sugli archi è minima.

## KRUSKAL

Trova il minimo albero di copertura

Metodo:

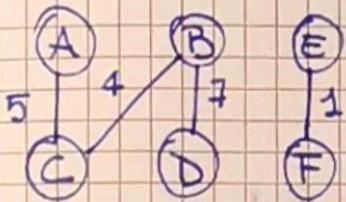
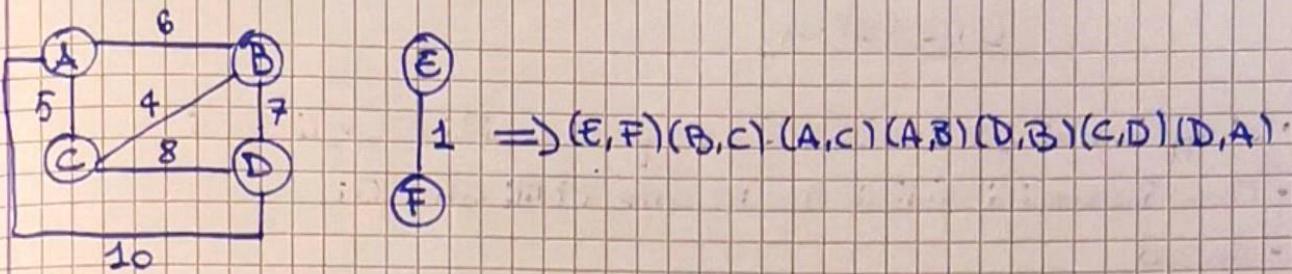
- 1) Elenca gli archi in ordine crescente, considera una componente per nodo
- 2) Scorri l'elenco e, per ogni arco  $a$   
if ( $a$  connette due componenti non connesse), unifica le componenti

$$O(m \log m) + O(m \log n) \Rightarrow O(m \log n) \quad (m \leq n^2)$$

1) num iterazioni:  $O(m)$

2) controllo + unificazione  $O(\log(n))$

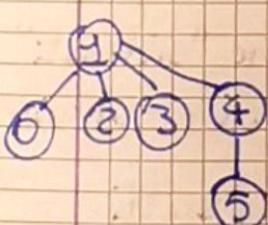
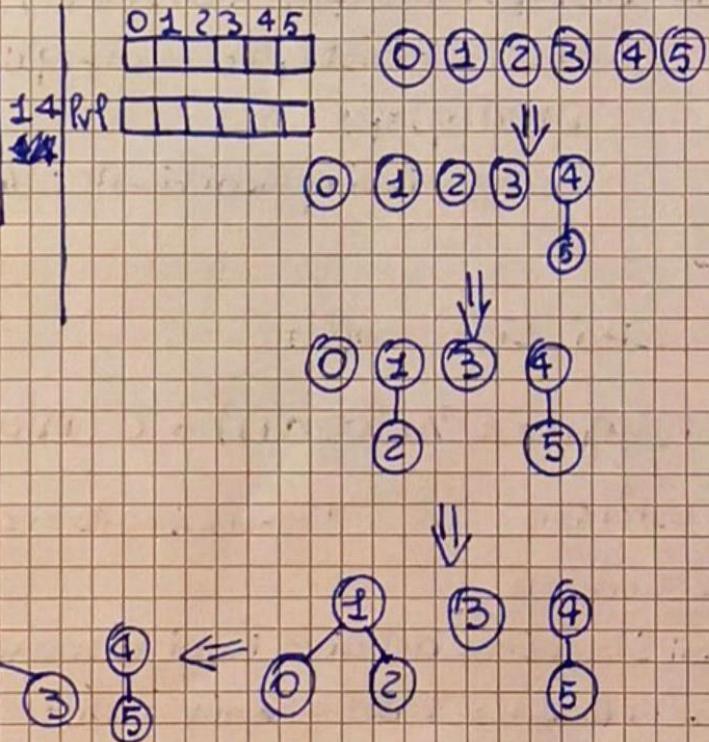
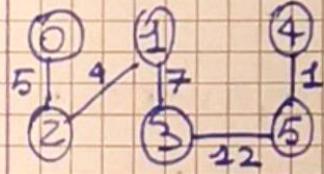
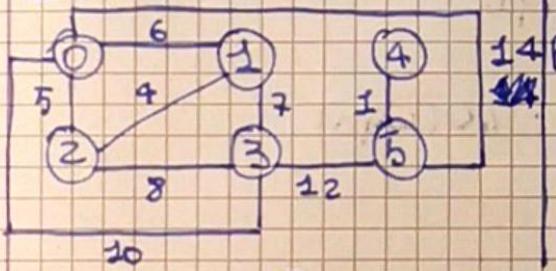
### Esempio esecuzione



### Implementazione:

- I nodi sono numerati;
- Le componenti sono n'alberi;
- Memorizzate in Array, ogni nodo punta al padre;
- Se due nodi appartengono alla stessa componente si trova un antenato comune;
- 2 alberi sono unificati: inserendo il meno profondo come figlio della radice del più profondo

### Esempio



## Dijkstra

- Si applica ai grafici orientati con peso positivo sugli archi
- Trova i cammini minimi da un nodo a tutti gli altri
- Greedy

Metodo:

Utilizza 2 tabelle, dist e pred con n elementi.

Esegue in passi, in cui:

- si sceglie il nodo con distanza minore in dist
- si aggiornano pred e dist per i suoi immediati successori

I valori di dist si memorizzano in un minheap.

Scelto il nodo di partenza si pone la sua distanza = 0 e la dist.  
degli altri =  $\infty$  //  $O(n)$

Finché ci sono più di un nodo, si estrae quello con dist minore e per  
ogni successore:

```
if (dist(p) + arco(p,q) < dist(q)) {  
    dist(q) = dist(p) + arco(p,q);  
    pred(q) = p;  
    Re-inserisci q modificato; //  $O(\log n)$   
}
```

Complessità ciclo completo

$$O(n(\log n + (m/n \log n))) = O(m \log n + m \log n) = O(m \log n) \text{ se } m \gg n$$

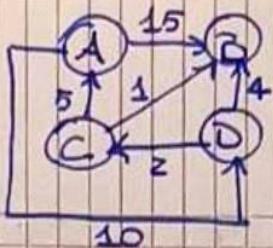
$\downarrow$  estrazione       $\downarrow$  re-inserimento medio dei successori

Perché funziona?

In ogni iterazione del ciclo i nodi inizialmente estratti sono "sistematici".  
Abbiamo la lunghezza del cammino in dist e pred permette di ricostruirlo.

Inoltre il cammino per questi nodi passa soltanto da nodi già scelti.

Esempio



dist

A	B	C	D
0	inf	inf	inf

pred

A	B	C	D
-	-	-	-

$$Q = A, B, C, D$$



A	B	C	D
0	15	inf	10

-	A	-	A
---	---	---	---

$$Q = B, C, D$$



A	B	C	D
0	14	12	10

A	B	C	D
-	D	D	A

$$Q = B, C$$



A	B	C	D
0	13	12	10

-	C	D	A
---	---	---	---

$$Q = B$$

PROBLEMI DIFFICILI  
 Ed NP-complettezza

Problemi difficili:

• Zaino: ottimizzare il riempimento dello zaino sapendo che:

- ogni oggetto ha un peso ed un valore.

Vogliamo la somma dei pesi  $\leq$  capacità e il valore totale massimo

• Comesso viaggiatore: Trovare il percorso minore che il viaggiatore deve fare per visitare tutte le città una ed una sola volta per poi tornare alla città di partenza.

• Ciclo Hamiltoniano: determinare se un multi-grafo possiede un ciclo Hamiltoniano, ovvero un ciclo che tocca tutti i nodi una ed una sola volta.

• Sat: data una formula nella logica dei predicati "F" con "n" variabili trovare, se esiste, una combinazione di valori booleani assegnati alle n variabili, verificano F

Questi (ed altri) sono definiti problemi difficili poiché ad oggi gli unici algoritmi conosciuti consistono nel provare tutte le combinazioni. Questo porta ad una complessità esponenziale.

### Problema del ciclo Euleriano

Dato un multi-grafo, trovare, se esiste, un ciclo che percorre tutti gli archi una ed una sola volta.

#### Teorema di Euler:

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

Per sapere se un ciclo esiste basta verificare il Teorema con complessità di verifica O(m)

Trovarlo ha complessità polinomiale

### Teoria della NP-complettezza

Classifica un insieme di problemi difficili ma non dimostrati come esponenziali. Si applica a problemi decisionali: con risposta Sì o No (ogni problema può assumere questa forma)

La complessità del decisionale è  $\leq$  complessità non decisionale (stesso problema)

- Zaino: dato un valore  $N$ , esiste un riempimento con valore  $N' \geq N$ ?

- Commissario: Dato un intero  $K$ , esiste nel grafo un ciclo senza ripetizioni di nodi con lunghezza  $< K$ ?

- Ciclo Ham.: Dato un grafo, esiste un ciclo Hamiltoniano?

- SAT: Data una formula, esiste un assegnamento alle variabili che la soddisfi?

## ALGORITMI NON DETERMINISTICI

Si aggiunge un comando (immaginario)

choice(I)

Dove I è un insieme da cui choice sceglie non deterministicamente.

Esempi di algoritmi nondeterministici:

Sat (O(n))

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

Ricerca

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

Per ogni algoritmo non deterministico ne esiste uno deterministico che lo simula, esplorando l' spazio delle soluzioni, fino a trovare un successo.

Possibili soluzioni in numero esponenziale  $\rightarrow$  complessità esponenziale.

|P=NP?|

P = tutti i problemi decisionali risolvibili deterministicamente con tempo polinomiale

NP = tutti i problemi decisionali risolvibili nondeterministicamente con tempo polinomiale

Oppure

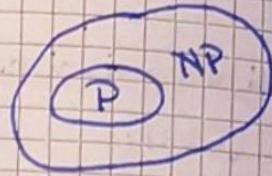
$R \in NP \Leftrightarrow$  Verifica di  $R \in$  comp. polinomiale  
di verifica di una soluzione (certificato)

$R \in NP \Leftrightarrow$  Verifica di  $R \in$  comp. polinomiale

Quindi

$P =$  Problemi decisionali facili da risolvere

$NP =$  Problemi decisionali facili da verificare.



### Riducibilità

È un metodo per convertire l'istanza di un problema  $P_1$  in quella di un problema  $P_2$ , utilizzando la soluzione di  $P_2$  per risolvere  $P_1$ .

Un problema  $P_1$  si riduce a  $P_2$  se ogni soluzione di  $P_1$  si ottiene deterministicamente in tempo polinomiale da quelle di  $P_2$ .

$$P_1 \leq P_2$$

Se si riduce  $P_1$  a  $P_2$  e  $P_2$  è risolvibile in tempo polinomiale, lo è anche  $P_1$ .

### Teorema di Cook

$\#NP \subseteq R \leq SAT$

SAT è dunque il problema NP più difficile.

### NP completezza

Un problema è NP-completo se, sia  $R$  il problema,

•  $R \in NP$

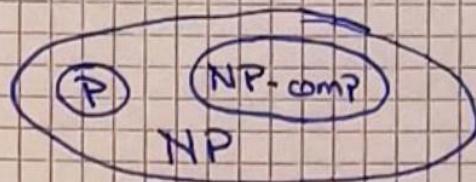
•  $SAT \leq R$

Se un problema è NP-completo posso usare nella dimostrazione dell'NP-completezza di altri problemi.

Se si trovasse un algoritmo polinomiale per un problema NP-completo allora  $P$  sarebbe = NP, ovvero ogni problema in NP si potrebbe risolvere in tempo polinomiale.

Alcuni problemi dimostrati NP-completi sono:

- Commissario viaggiatore
- Zaino
- Ciclo Hamiltoniano.



Per dimostrare che un problema è NP-completo,

1) dimostrare che  $R \in NP$

2) dimostrare che esiste un problema NP-completo che si riduce ad R.

### PROBLEMA FATTORIZZAZIONE DI UN NUMERO

FATT: Scomposizione in fattori primi

- Per ora si conoscono solo algoritmi esponenziali
- $FATT \in NP$  (La verifica è la moltiplicazione)
- Su FATT si basa la crittografia a chiavi pubblica

Esistono inoltre problemi risolvibili con solo  $O(n!)$  e problemi non risolvibili con un algoritmo

