



# Esempi di Strutture Dati

---



# Strutture dati in pratica

---

- Esempi
  - Record, array, tabelle hash, alberi, grafi, insiemi, ....
- Permettono di organizzare e memorizzare i dati
- Realizzazione
  - attraverso le strutture predefinite di un linguaggio di programmazione (array, record,...)
  - implementate dal programmatore (file, alberi, ...)
- La realizzazione trasforma una struttura dati astratta in un'altra struttura dati, di livello piu' basso



# Strutture dati in pratica II

---

Le strutture dati si valutano attraverso la

- complessita' spaziale
  - numero di celle di memoria usate per memorizzare i dati/programma
- complessita' in tempo
  - numero di operazione necessarie per effettuare la ricerca, l'inserimento ed eliminazione dei dati
- In questa parte del corso vedremo solo alcuni esempi di strutture dati

# Classificazioni strutture dati: accesso ai dati

## Accesso ai dati

- sequenziale, es. liste, file,
  - implementare un insieme di N oggetti con le liste
  - spazio occupato:  $O(N)$
  - ricerca: lineare  $O(N)$ , inserimento: costante, eliminazione: lineare  $O(N)$



```
struct node {  
    char value;  
    struct node *next };  
  
struct node *nodeAlloc(){  
    . . .  
}
```

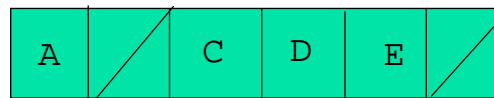
Scarselli

```
struct node *addNode(struct node *list, char c){  
    struct node *n=nodeAlloc();  
    n->value=c;  
    n->next=list;  
    return n;  
}
```

Fondamenti di Informatica II

# Classificazioni strutture dati: accesso ai dati II

- diretto, es. array
  - implementare un insieme di N oggetti con un array, supponendo di
    - assegnare ad ogni oggetto una posizione precisa nell'array
    - $O$  il numero di tutti i possibili oggetti ( $O \geq N$ )
  - spazio  $O(O)$
  - ricerca: costante, inserimento: costante, eliminazione: costante



```
char insieme[O]

void add(char v[], char c){
    v[atoi(c)]=c;
}
```

Scarselli

```
char insieme[O]

void del(char v[], char c){
    v[atoi(c)]='\0'
}
```

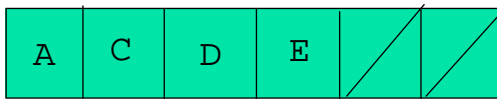
Si usa per  
rappresentare  
l'elemento  
vuoto

Fondamenti di Informatica II

5

# Classificazioni strutture dati: accesso ai dati III

- **ordinate**, es. array ordinati, B-Tree,
  - implementare l'insieme di N elementi con un array ordinato di M posizioni
  - spazio:  $O(M)$
  - ricerca: logaritmica  $O(\log(N))$ , inserimento: lineare  $O(N)$ , eliminazione: lineare  $O(N)$



```
struct set {  
    char values[];  
    int last;};
```

```
int find(char values[],char c,  
        int from, int to){  
    if (from == to) {if (values[from]==c) {return from;}  
                    else return -1;}  
  
    int middle=(to-from)/2+to;  
    if (value[middle]<=c) {find(values,c,middle+1,to);}  
    if (value[middle]>c) {find(values,c,from,middle-1);}  
}
```

# Classificazioni strutture dati: accesso ai dati IV

- **calcolato**, es. tabelle hash,
  - implementare l'insieme di N elementi con una tabella hash di M posizioni
  - spazio:  $O(M)$
  - ricerca: costante, inserimento: costante, eliminazione: costante

C		C	A	E	
---	--	---	---	---	--

```
char insieme[O]

void add(char v[], char c){
    v[myHash(c)] = c;
}
```

Scarselli

```
char insieme[O]

void del(char v[], char c){
    v[myHash(c)] = '\0';
}
```

Una funzione  
di hash

Fondamenti di Informatica II



# Classificazioni strutture dati: altre classificazioni

---

## Tipo di informazione memorizzata

- lineare
  - insiemi, file, ...
- strutturata
  - grafi, alberi, strutture generiche

## Luogo di memorizzazione

- alcune strutture dati sono adoperate per la memoria primaria, altre per la memoria secondaria
- La complessita' temporale degli accessi si calcola in modo diverso a seconda di dove la struttura dati è memorizzata





Grafi

---



# Grafi

## Un grafo

- un insieme  $V=\{v_1, v_2, \dots, v_n\}$  di nodi
- un insieme  $E \subseteq V \times V$  di archi

## Definizioni

- grafo (non) orientato: se gli archi (non) hanno una direzione
- figli di un nodo  $n_1$ :  $\{n: (n_1, n) \in E\}$
- padri di un nodo  $n_1$ :  $\{n: (n, n_1) \in E\}$
- cammino da un nodo  $n_1$  a un nodo  $n_2$ : una sequenza di archi che ci permette di arrivare da  $n_1$  a  $n_2$
- grafo ciclico: esiste un cammino non banale da un nodo  $n$  a se stesso

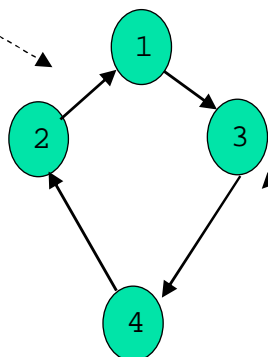
# Grafi: esempi

- 3, 4 sono figli di 2
- 3, 4 sono padri di 6

cammino (non) orientato da 1 a 6

- 1, 3, 4 sono vicini di 2

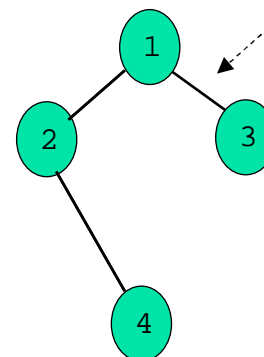
Grafi orientati



grafi ciclici

grafi non ciclici

Grafi non orientati

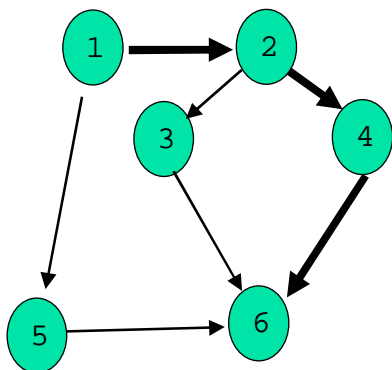


# Grafi: implementazione

Per mezzo di una matrice di adiacenza

- si costruisce una matrice  $n \times n$
- l'elemento  $i, j$  della matrice è 1 se  $(v_i, v_j) \in E$ , 0 altrimenti

Per grafi non diretti la matrice è simmetrica

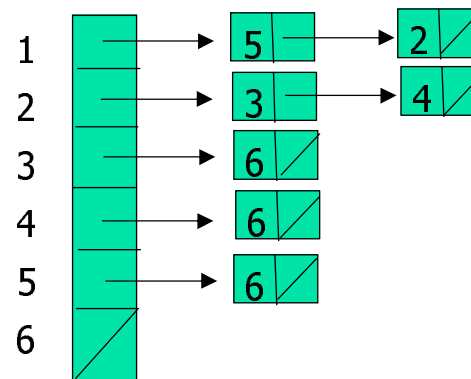
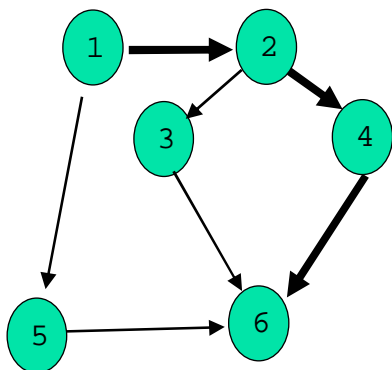


	1	2	3	4	5	6
1	0	1	0	0	1	0
2	0	0	1	1	0	0
3	0	0	0	0	0	1
4	0	0	0	0	0	1
5	0	0	0	0	0	1
6	0	0	0	0	0	0

# Grafi: implementazione II

Per mezzo di una liste di adiacenza

- Ogni nodo è associato ad un indice
- Per ogni nodi si memorizzano in una lista i figli





# Operazioni sui grafi

---

Verificare se esiste un arco dal nodo  $i$  al nodo  $j$

(si assume di avere  $n$  nodi e  $a$  archi)

- Con una matrice di adiacenza  $A$ : si verifica se  $A[i][j] = 1$ 
  - tempo costante  $O(1)$
- Con liste di adiacenza: si scorre la lista associata ad  $i$ 
  - nel caso peggiore  $O(n)$   
( $i$  è collegato a tutti i nodi e  $j$  è l'ultimo della lista)
  - nel caso medio  $O(1 + a/n)$



# Operazioni sui grafi II

---

## Trovare i figli del nodo $i$

- Con una matrice di adiacenza  $A$ : si scorre tutta l' $i$ -esima riga di  $A$ .
  - tempo lineare  $O(n)$
- Con liste di adiacenza: si scorre la lista associata ad  $i$ 
  - nel caso peggiore  $O(n)$   
( $i$  è collegato a tutti i nodi)
  - nel caso medio  $O(1+a/n)$

## Trovare i padri del nodo $i$

- Con una matrice di adiacenza  $A$ : si scorre tutta l' $i$ -esima colonna di  $A$ .
  - tempo lineare  $O(n)$
- Con liste di adiacenza: si scorre le liste associate a tutti i nodi
  - occorre  $O(a)$



## Operazioni sui grafi III

---

Lo spazio occupato dal grafo

- Con una matrice di adiacenza  $A$ : occorrono  $n^2$  bit
- Con liste di adiacenza: occorrono  $np + a$   
dove  $p$  è lo spazio occupato da un puntatore alla lista e  $a$  lo spazio occupato da un elemento della lista

Riassumendo: Le liste di adiacenza sono preferibili

- per grafi poco densi
- nel caso serva accedere spesso ai figli di un nodo





## Esempio: un grafo sparso

- Il Web è un grafo costituito
  - circa 3.000.000.000 di nodi (le pagine) e
  - 30.000.000.000 di archi (gli hyperlink)

	liste	matrice
Esiste (i,j)	10	1
Trovare i figli di i	10	3.000.000.000
Trovare i padri di i	30.000.000.000	3.000.000.000
Spazio occupato	30.000.000.000x8 Byte	9X10 <sup>18</sup> bit



## Esempio: un grafo denso

- Il grafo delle distanze stradali fra le 1000 città più grandi del mondo
  - 1000 di nodi (le pagine) e
  - si supponga 500.000 di archi (gli hyperlink)

	liste	matrice
Esiste (i,j)	500	1
Trovare i figli di i	500	1000
Trovare i padri di i	500.000	1000
Spazio occupato	500.000x8 Byte	1.000.000 bit



# Visite di un grafo: in profondita'

## Visita in profondita' (depth first)

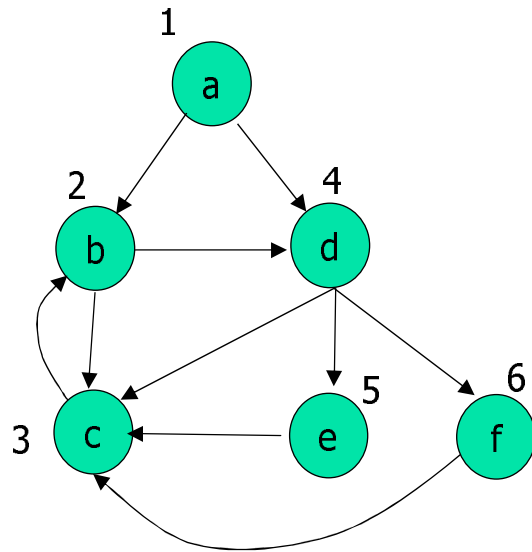
- partendo da un nodo si visitano ricorsivamente tutti i figli
- per evitare di ripassarci piu' volte i nodi devono essere marcati

```
typedef struct node{  
    int marked;  
    struct node **children;  
    int nSons;  
    int num;  
}
```

```
nNodes=1;  
void depthVisit(node *n){  
    int k;  
    n->marked=1;  
    n->num=nNodes;  
    nNodes++;  
    for (k=0;k<n->nSons,k++){  
        if(!(n->children[k]->marked))  
            depthVisit(n->children[k]);  
    }  
}
```

# Visite di un grafo: in profondita' II

- l'ordine di marcatura (nNodes) definisce un ordine dei nodi



sequenza della visita: a b c b d e d f ....

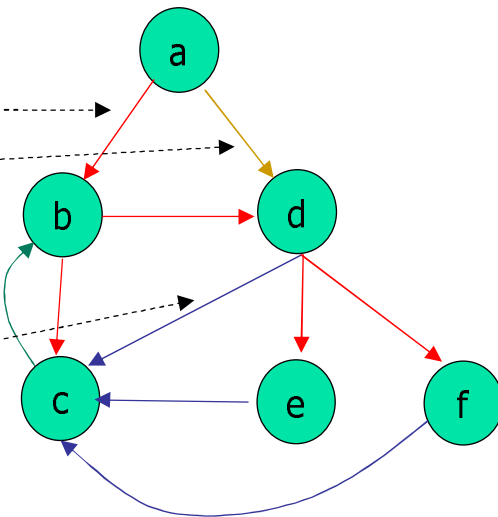
in rosso la  
marcatura  
dei nodi

ordinamento dei nodi: a b c d e f

# Albero di visita

## Albero di visita inprofondita'

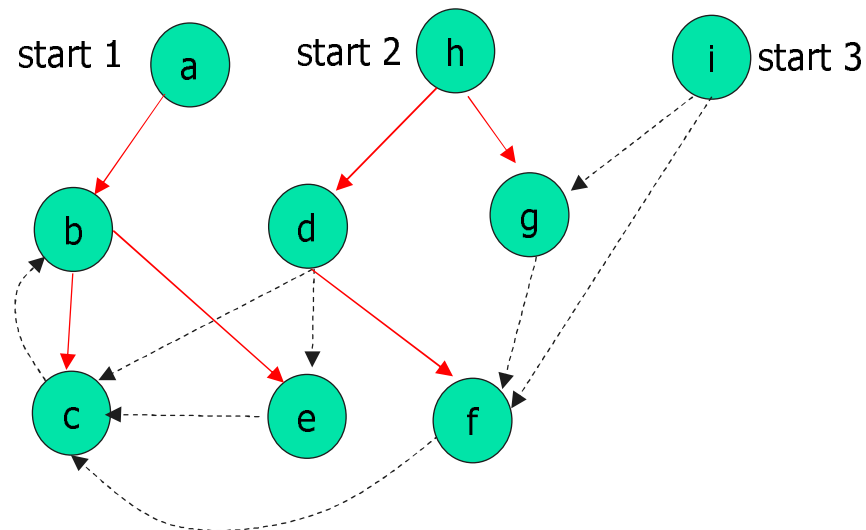
- quando `depthVisit` passa da un `n` ad uno `v` non marcato si aggiunge  $(n,v)$  all'albero
- gli archi si dividono
  - archi dell'albero
  - archi in avanti:  
 $(n,v)$  t.c.  $v$  è discendente di  $n$
  - archi all'indietro:  
 $(n,v)$  t.c.  $n$  è discendente di  $v$
  - archi trasversali:  
gli altri archi
- **Il grafo è ciclico se e solo se contiene archi all'indietro**



# Foresta di visita

Partendo da un nodo puo' capitare di **non visitare tutti i nodi**

- si sceglie un nodo non visitato e si riparte
- alla fine si ottiene un insieme di alberi di visita (foresta)





# Attraversamento in ordine posticipato

La visita in profondita' puo' servire a numerare i nodi in ordine posticipato

- il nodo viene numerato dopo che tutti i figli sono stati visitati

```
typedef struct node{  
    int marked;  
    struct node **children;  
    int nSons;  
    int num;  
}
```

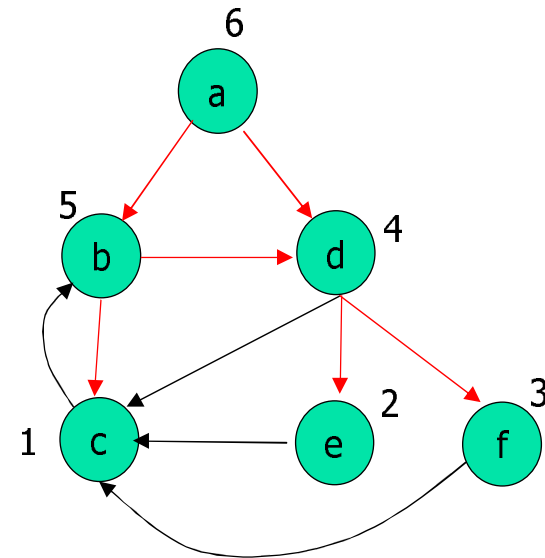
```
nNodes=1;  
void depthVisit(node *n){  
    int k;  
    n->marked=1;  
    for (k=0;k<n->nSons,k++){  
        if(!(n->children[k]->marked))  
            depthVisit(n->children[k]);  
    }  
    n->num=nNodes;  
    nNodes++;  
}
```

# Attraversamento in ordine posticipato II

ordinamento topologico:  
a b d f e c

Il numero d'ordine posticipato

- è minore per i figli rispetto ai padri (nell'albero di visita)
- per gli archi trasversali e in avanti la testa ha un numero più piccolo della coda
- per gli archi all'indietro la testa ha un numero più grande della coda



ordinamento posticipato:  
c e f d b a

Algoritmo per la verifica della ciclicità'

1. numera il grafo in ordine posticipato
2. scandisci tutti gli archi e verifica se esiste  $(n, v)$  dove  $v$  precede  $n$

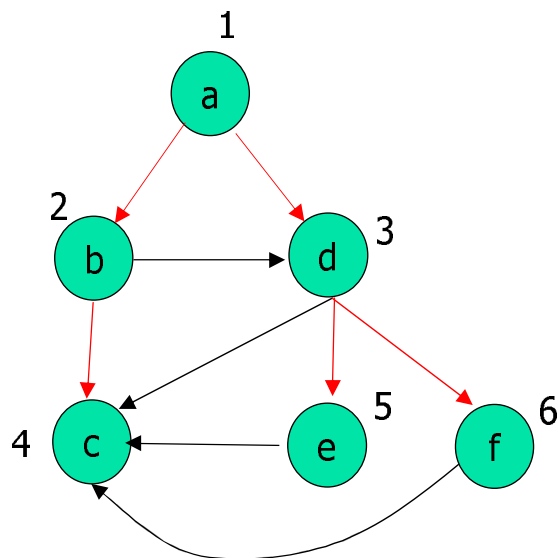
Richiede  $O(a)$   
operazioni



# Visita in ampiezza

## Visita in ampiezza (breadth first)

- prima si visita un nodo, poi tutti i figli, poi tutti i figli dei figli, ....

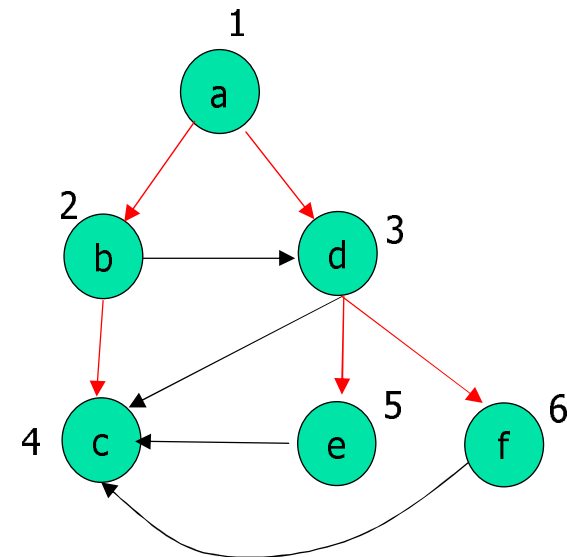


ordinamento in ampiezza:  
a b d c e f

# Visita in ampiezza II

- Per realizzare la visita in ampiezza occorre usare una cosa

```
nNodes=1;
void breadthVisit(node *n){
    int k;
    n->marked=1;
    n->num=nNodes;
    nNodes++;
    push(n,Q);
    while(!empty(Q)){
        v=pop(Q);
        for (k=0;k<v->nSons,k++){
            if(!(v->children[k]->marked)){
                n->children[k]->marked=1;
                n->num=nNodes;
                nNodes++;
                push(Q,n->children[k]);}}}
}
```



push(n,Q) : inserisce n in Q  
pop(n,Q) : ritorna la testa di Q  
empty(Q) : verifica se Q è vuota



# Tabelle hash

---



# Tabelle hash

Nelle tabelle hash la posizione  $f(k)$  di un oggetto dipende

- una chiave  $k$
- una funzione hash  $f$ : chiavi  $\rightarrow$  posizioni

Esempio: gli studenti

- la matricola ( $m$ ) è la chiave
- la funzione modulo ( $\text{mod}$ ) fornisce la posizione
- al posizione è:  $m \text{ mod } n$   
dove  $n$  è il numero di posizioni disponibili nella tabella

Matricola	Nome
01323123	Paolino paperino
03434394	Papreron dei Paperoni
23347966	Gastone
03453458	Pluto
35345400	Topolino



# Tabelle hash II

---

- **collisione**: evento a due oggetti viene assegnata la stessa posizione
- **fattore di carico**: rapporto  $O/n$  fra il numero oggetti allocati e il numero delle posizioni disponibili

## Il problema collisioni

- occorre un meccanismo per gestire le collisioni
- la funzione hash dovrebbe distribuire le chiavi in modo uniforme, in modo da minimizzare le collisioni
- il fattore di carico dovrebbe essere
  - abbastanza grande da non sprecare spazio
  - abbastanza piccolo da limitare le collisioni

# Gestione delle collisioni: ricerca lineare

La soluzione piu' semplice

- in caso di collisione si alloca l'oggetto nella **prima posizione** seguente disponibile

Matricola	Nome
35345400	Topolino
01323123	Paolino paperino
03434394	Papreron dei Paperoni
32423423	Gamba di legno
23347966	Gastone
03453458	Pluto

```
insert(object o){  
    int p,i;  
    p=hash(o);  
    while (!tab[i].free){  
        i=(i+1)%tab.numObjects;  
        if (i==p) {return(fail);}  
    }  
    tab[i].info=o;  
    tab[i].free=false;  
}
```

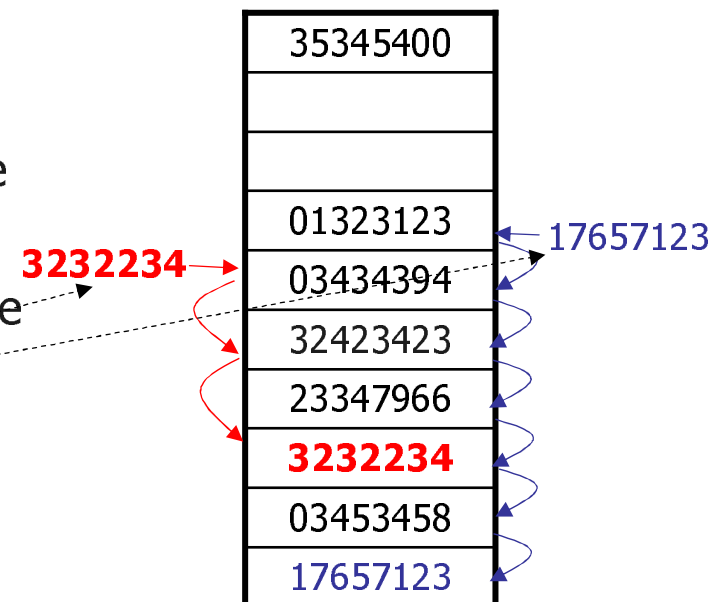
# Clustering

La ricerca di una posizione libera puo' richiedere molto tempo perche'

- la funzione di hash produce la stessa posizione per chiavi diverse (**clustering primario**)
- chiavi con per le quali la funzione hash produce valori vicini competono per le stesse posizioni (**clustering secondario**)

Soluzioni

- rehashing
- double hashing





# Gestione delle collisioni: Rehashing e double hashing

## Rehashing

- In caso di collisione, gli oggetti vengono messi non in sequenza ma in una posizione stabilita da una funzione di rehash
- **risolve il clustering secondario** non quello primario
  - due chiavi che collidono generano la stessa sequenza

## Esempio

- $\text{rehash}(i) = (i+q) \bmod n$   
con  $q$  e  $n$  primi fra loro
- con  $n=10$ ,  $q=3$ , hash iniziale 4 si genera 4,7,0,3,6,...

```
insert(object o){
    int p,i;
    p=hash(o);
    i=p;
    while (!tab[i].free){
        i=rehash(i);
        if (i==p) {return(fail);}
    }
    tab[i].info=o;
    tab[i].free=false;
}
```





# Rehashing e double hashing II

---

## Double hashing

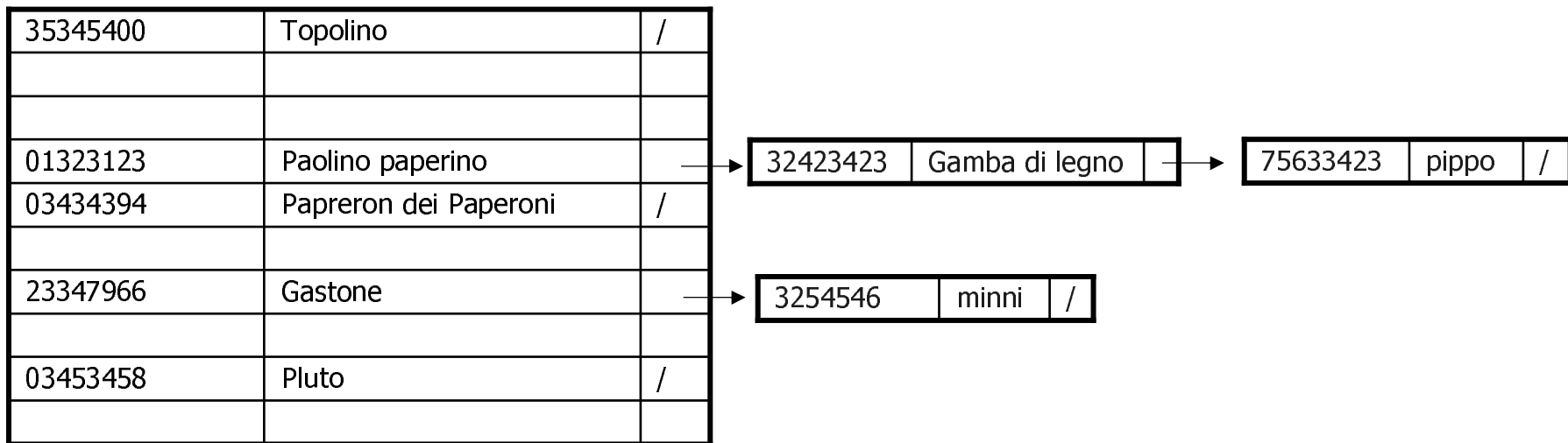
- la funzione di rehashing dipende anche dalla chiave iniziale:
- si applica alla chiave una seconda funzione di hash
- risolve il problema del clustering primario

## Esempio

- $\text{hash1}(k) = k \bmod 10$
- $\text{hash2}(k) = k \bmod 7$
- $\text{rehash}(i, k) = (i + \text{hash2}(k)) \bmod 10$
- Per le matricole 23, 13 si generano le sequenze  
23  $\rightarrow$  3, 5, 7, 9, ... ( $\text{hash1}(23)=3$ ,  $\text{hash2}(23)=2$ )  
13  $\rightarrow$  3, 9, 5, 1, ..... ( $\text{hash1}(13)=3$ ,  $\text{hash2}(13)=6$ )

# Gestione delle collisioni: concatenamento

- Per ogni posizione della tabella si mantiene una lista di “overflow” con gli oggetti che devono stare in quella posizione
- risolve il clustering secondario
- se il clustering primario è limitato, allora i tempi di ricerca sono limitati





# Funzioni hash: esempi

---

## Le funzioni hash

- dovrebbero generare una distribuzione **il piu' uniforme possibile** della chiavi

## Come funziona

- 1) Si trasforma la chiave  $k$  in un intero  $r$ 
  - se la chiave è una stringa si considera la sequenza di codici ASCII
- 2) Si applica ad  $r$  una funzione  $f$  che lo trasforma in una posizione  **$p=f(r) \bmod n$**

## Esempi

- **Trasformazione radicale**: si cambia la base di rappresentazione (es, da 10 a 11)  
 $f(232)=2*11^2+3*11^1+2=2.697, \quad p=2.697 \bmod 10=7$
- **Shift folding**: si dividono le cifre in gruppi che vengono sommati  
 $f(3\ 45\ 65\ 67)=3+45+65+67=180, \quad p=180 \bmod 10=0$



# File hash: hashing per la memoria secondaria

---

## Considerazioni generali

- il disco è **organizzato in settori**
  - non è possibile caricare in memoria principale un singolo oggetto
  - si caricano tutti gli oggetti di un blocco
- il disco è lentissimo rispetto alla memoria principale
  - il tempo speso nell'accesso alla memoria principale è spesso trascurabile
  - **occorre minimizzare il numero degli accessi la disco**

## File hash

- la funzione hash produce l'indirizzo di un blocco
- le tuple del blocco sono organizzate in modo sequenziale
- diminuisce la probabilita' di overflow fra pagine diverse



# File hash

File hash con

- 40 record
- fattore di blocco 10
- 5 blocchi
- 2 collisioni
  - numero medio di accessi: 1,05

60600
66005
116455
200205
205610
201260
102360
205460
200430
102690

205845

66301
25751
115541
200296
205796

205802
200902
116202
205912
205762
205617
205667
210522
205977
205887

206092

200268
205478
210533
200138
102338
205693
200498

200604
201159
200464
205619
205724
206049



# Confronto con tavola hash

- 40 record
  - **tavola hash** con 50 posizioni:
    - 1 collisione a 4
    - 2 collisioni a 3
    - 5 collisioni a 2
- numero medio di accessi: 1,425

M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49



Heap

---

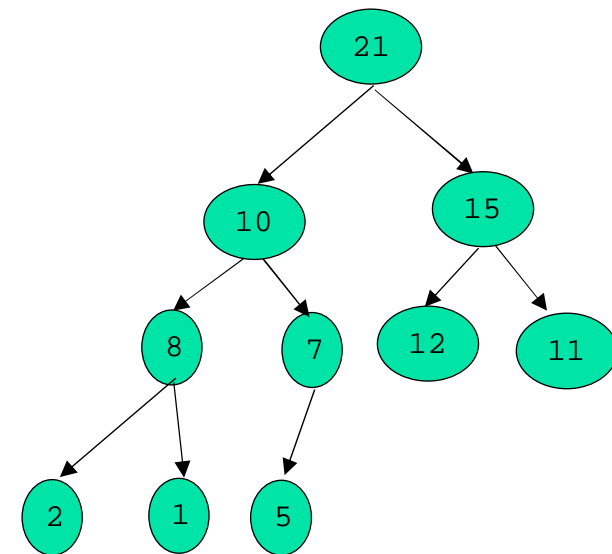
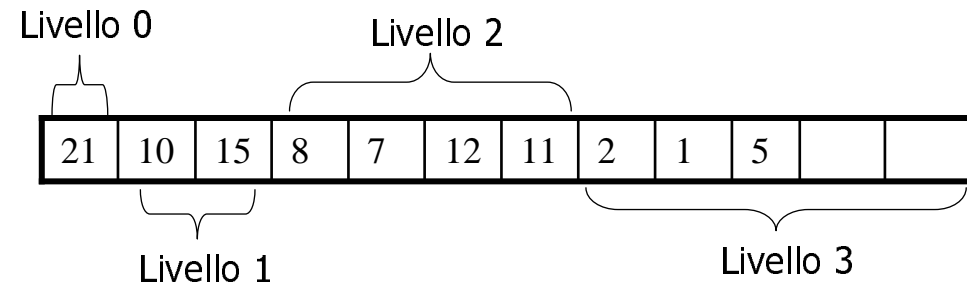
# Heap

Cos'è

- una struttura parzialmente ordinata
- implementa una coda con priorit 

Come funziona

- gli oggetti sono contenuti nei nodi di un albero
- nella radice c'  l'elemento piu' grande
- ogni nodo contiene un elemento maggiore di tutti gli elementi contenuti nel sotto albero
- l'albero   bilanciato
  - la struttura   fissata, la posizione degli elementi puo' variare
- l'heap viene memorizzato in un vettore





# Inserimenti

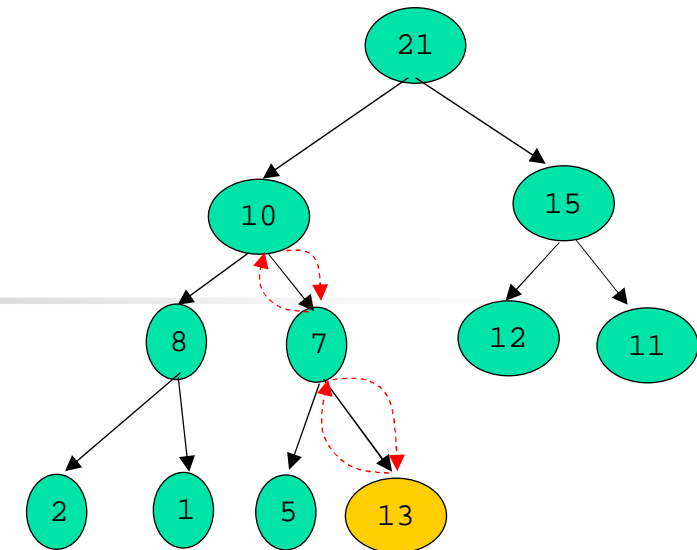
## Inserimenti

- si inserisce il nuovo elemento nell'ultimo livello
- si scambiano i nodi fino a quando l'ordinamento è rispettato
- un inserimento costa  $O(\log N)$

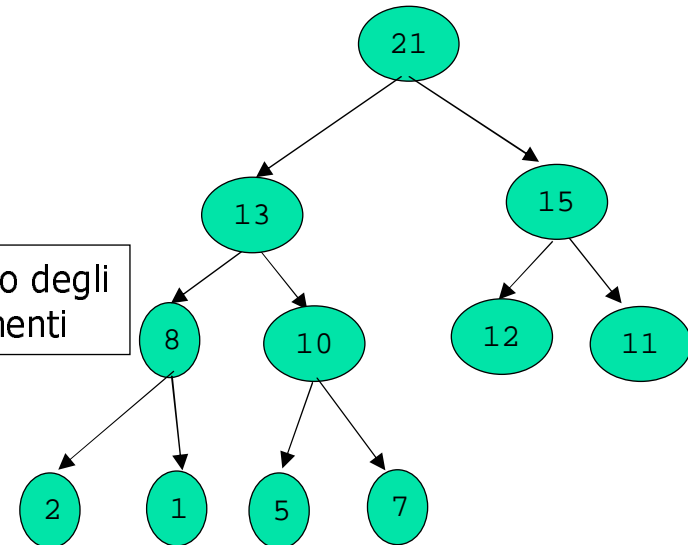
```
heapDim++;  
H[heapDim-1]=object;  
  
for(j=heapDim-1; j!=0; j=(j-1)/2){  
  
    if(H[j]>H[(j-1)/2]){  
        tmp=H[j];  
        H[j]=H[(j-1)/2];  
        H[(j-1)/2]=tmp;  
    }  
  
    else break;  
}  
}
```

Scarselli

Fondamenti di Informatica II



scambio degli  
elementi

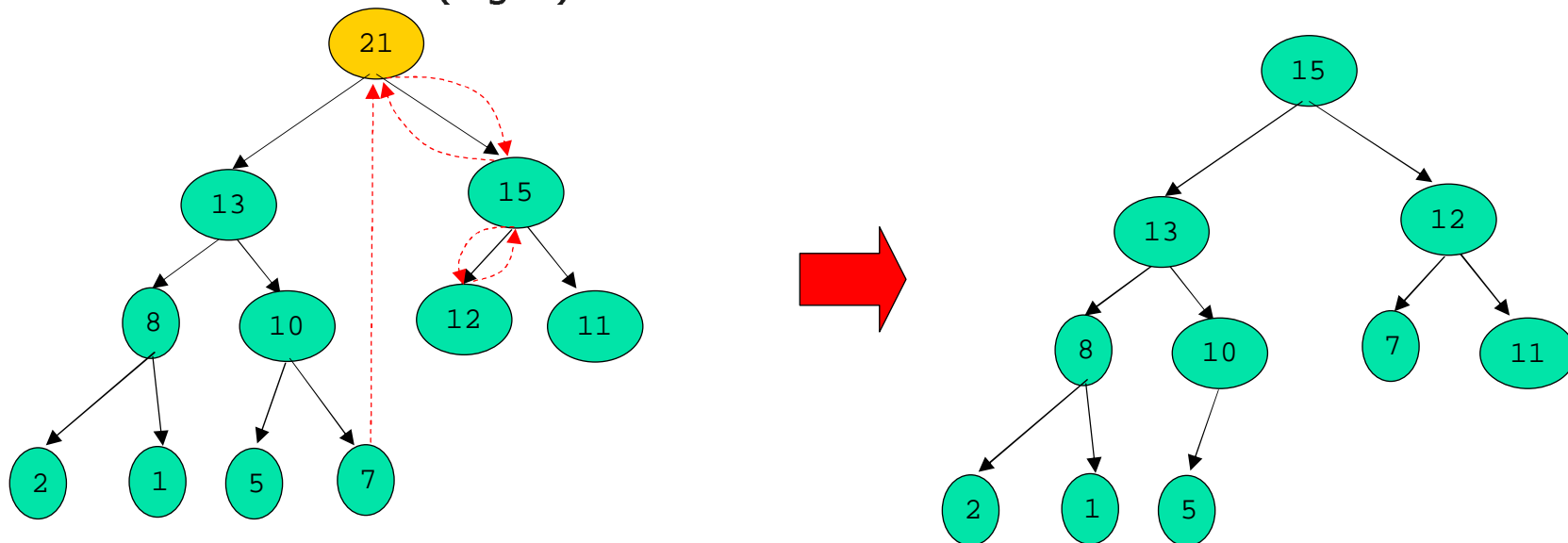


41

# Cancellazione della radice

## Rimozione

- si sostituisce la radice con l'ultima foglia
- si scambiano i nodi con il massimo dei figli fino quando l'ordinamento è rispettato
- una rimozione costa  $O(\log N)$





# Heapsort

---

## Ordinamento usando un heap

- 1) si costruisce un heap con gli elementi da ordinare
- 2) si estrae la radice
- 3) si ripete (2) fino a quando l'heap è vuoto

## Osservazioni

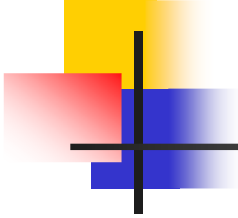
- il costo dell'algoritmo è  $O(N \log N)$
- l'algoritmo può usare efficientemente la memoria impiegando solo il vettore in cui memorizza l'heap

```
void heapSort(object H, int dim){  
    int i,j;  
    object root;  
    buildHeap(H,dim);  
  
    for(int i=dim-1;i>=1;i--){  
        root=extractFromHeap(H,i);  
        H[i]=root;  
    }  
}
```



# Complessita': esempi

---



# Complessita' computazionale concreta

---

## L'analisi della complessita'

- serve a valutare l'efficienza di un algoritmo nella risoluzione di un problema al crescere della dimensione del problema

## Cosa si misura

- la dimensione del problema  $n$ : la lunghezza dell'ingresso
- complessita' in tempo: il tempo impiegato nella soluzione del problema  $T(n)$
- complessita' in spazio: lo spazio di memoria usato nella soluzione del problema  $M(n)$

## Tipi di misure

- caso medio (complessita' media):  
la media della complessita' su tutti i problemi di lunghezza  $n$
- caso pessimo:  
la complessita' peggiore fra tutti i problemi di lunghezza  $n$



# Crescita asintotica

Si studia la complessita' per  $n \rightarrow \infty$

- $O(f(n))$  è l'insieme di funzioni  $g(n)$  tali che esiste  $c$  per cui  $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \leq c$ 
  - $O(f(n))$  fornisce un **limite superiore**:  $g(n)$  cresce al massimo come  $f$
- $o(f(n))$  è l'insieme di funzioni  $g(n)$  per cui  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c$ 
  - $O(f(n))$  fornisce un **limite inferiore**
- Per un algoritmo si vuol conoscere la  $f$  che cresce meno per cui vale
  - $T(n) \in O(f(n))$  e/o  $T(n) \in o(f(n))$

$$g_1(n) = 3n^2 + 7$$

$$g_1(n) \in O(x^2)$$

$$g_1(n) \in o(x^2)$$

$$g_2(n) = 5n + 1$$

$$g_2(n) \in O(x^2)$$

$$g_2(n) \notin o(x^2)$$

$$g_3(n) = n^3 - 10$$

$$g_3(n) \notin O(x^2)$$

$$g_3(n) \in o(x^2)$$



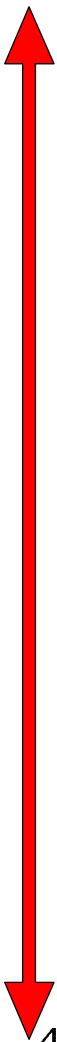
# Crescita asintotica II

## Classi di complessita'

- $O(1)$  costante
  - non dipende dall'ingresso
- $O(\log n)$  logaritmica
  - es. ricerca in strutture ordinate
- $O(n)$  lineare
  - es. somma di  $n$  numeri
- $O(n \log n)$  costante
  - es. mergesort
- $O(n^k)$  polinomiale
  - dato un insieme di  $n$  punti sul piano, si cercano quella che distano meno di  $L$
- $O(k^n)$  esponenziale
  - es. stampa di tutti gli interi di  $n$  cifre

problemi facili

problemi difficili





# Algoritmi ricorsivi

---





# Calcolo della complessita' per algoritmi divide et impera

Si supponga che

- l'algoritmo divide un problema di dimensione  $n$  in problemi di dimensione  $n_1, \dots, n_h$
- $D(n)$  tempo necessario a fare la suddivisione
- $C(n)$  tempo necessario a ricombinare i risultati
- $T(n)=c$  per  $n \leq k$ , dove  $c$  è costante

Si ottiene il seguente sistema di equazioni

$$T(n) = c, n \leq k$$

$$T(n) = D(n) + C(n) + \sum_i^h T(n_i), n > k$$

# Calcolo della complessità: il max ricorsivo

Il max ricorsivo (contando solo i confronti)

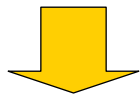
- divide un problema di dimensione  $n$  in 2 problemi di dimensione  $n/2$ 
  - profondità della ricorsione  $= k = \log_2 n$
- $D(n)=1$  confronto (+ 2 divisioni),  $C(n)=1$  confronto,  $T(1)=1$



si ottiene

$$T(1) = 1$$

$$T(n) = 2 + 2T(n/2), n \geq 2$$

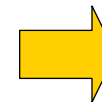


soluzione

$$\begin{aligned} T(n) &= 2 + 2T(n/2) = 2 + 2(2 + 2T(n/4)) = 2 + 2^2 + 2^2T(n/4) = \dots \\ &= 2 + \dots 2^{k-1} + 2^k + 2^k T(1) = 2^{k+1} - 2 + 2^k T(1) = 2n - 2 + n = 3n - 2 \end{aligned}$$

```
max(A, i, j) {  
    if (i == j) return (A[i]);  
    m1 = max(A, i, (i+j)/2);  
    m2 = max(A, (i+j)/2+1, j);  
    if (m1 > m2) return m1;  
    else return m2;  
}
```

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



max è  $O(n)$

# Ricerca binaria in un vettore ordinato

- si confronta il valore da ricercare  $o$  con la cella centrale  $A[m]$  del vettore
  - se  $o=A[m]$ , allora lo abbiamo trovato
  - se  $o<A[m]$ , allora cerchiamo nella parte destra del vettore
  - se  $o>A[m]$ , allora cerchiamo nella parte sinistra del vettore

```
search(A,o,i,j){  
    if(i>j) return(FAIL);  
    m=(i+j)/2;  
    if(A[m]==o) return(m);  
    if (o<A[m])  
        return(search(A,o,i,m-1));  
    else  
        return(search(A,o,m+1,j));  
}
```

**cercando 12**

3	7	9	12	15	16
---	---	---	----	----	----

12<9



12	15	16
----	----	----



12<15

12
----

12=12

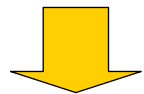
# Ricerca binaria in un vettore ordinato II

- divide un problema di dimensione  $n$  in 1 problemi di dimensione  $(n-1)/2$ 
  - profondità' della ricorsione  $\leq k = \log_2 n$

- $D(n) = \text{al più 3 confronti (+ 1 divisione)}$

- $C(n) = 0$

- $T(0) = 1$



si ottiene

$$T(0) = 1, T(n) = 3 + T\left(\frac{n-1}{2}\right), n \geq 2$$

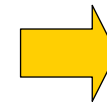


soluzione

$$T(n) \leq 3 + T(n/2) \leq 3 + (3 + T(n/2)) \leq \dots \leq 3k + kT(0)$$

$$T(n) = 3 \log n + \log n = 4 \log n$$

```
search(A,o,i,j){  
    if(i>j) return(FAIL);  
    m1=(i+j)/2;  
    if(A[m]==o) return(m);  
    if (o<A[m])  
        return(search(A,o,i,m-1));  
    else  
        return(search(A,o,m+1,j));  
}
```



search è  $O(\log(n))$



# Una classe di algoritmi ricorsivi: carico di combinazione costante

Max e la ricerca binaria appartengono ad algoritmi divide et impera

- per i quali il tempo di combinazione/divisione è costante (indipendente da  $n$ )
- per essi vale
  - $T(1)=d$
  - $T(n)=a T(n/b)$dove
  - $a$ = numero di sottoproblemi,  $n/b$ =dimensione dei sottoproblemi

Per tali algoritmi vale

$$T(n) \in O(\log n), \quad \text{se } a = 1$$

$$T(n) \in O(n^{\log_b a}), \quad \text{se } a > 1$$

- con max,  $d=1$ ,  $a=1$ ,  $b=2$
- con search,  $d=1$ ,  $a=2$ ,  $b=2$

# Ordinamento: mergesort

1. si divide il vettore  $v$  in due sottovettori  $v1$  e  $v2$  di uguali dimensioni
2. si ordinano ricorsivamente  $v1$  e  $v2$
3. i sottovettori  $v1$  e  $v2$  sono fusi scorrendoli in modo ordinato

```
mergesort(A,i,j){
```

```
  if((i==j)) return;
```

```
  if(i<j){
```

```
    m=(i+j)/2;
```

```
    mergesort(A,i,m);
```

```
    mergesort(A,m+1,j);
```

```
    merge(A,i,m,j);
```

```
  }
```

```
}
```

la dimensione del vettore è 1

divisione

ordinamento dei sottovettori

fusione dei sottovettori

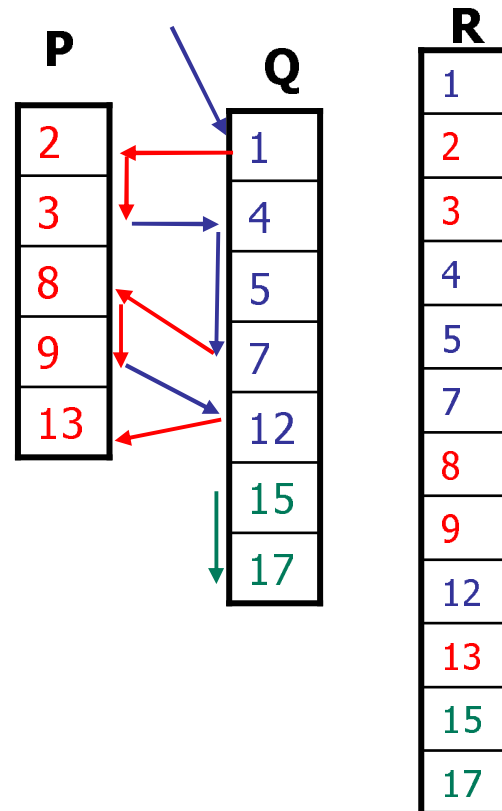
# la fusione di due vettori ordinati

- occorre un vettore d'appoggio
- i due sottovettori sono letti sequenzialmente usando due puntatori

```
1.  i, j, k=0

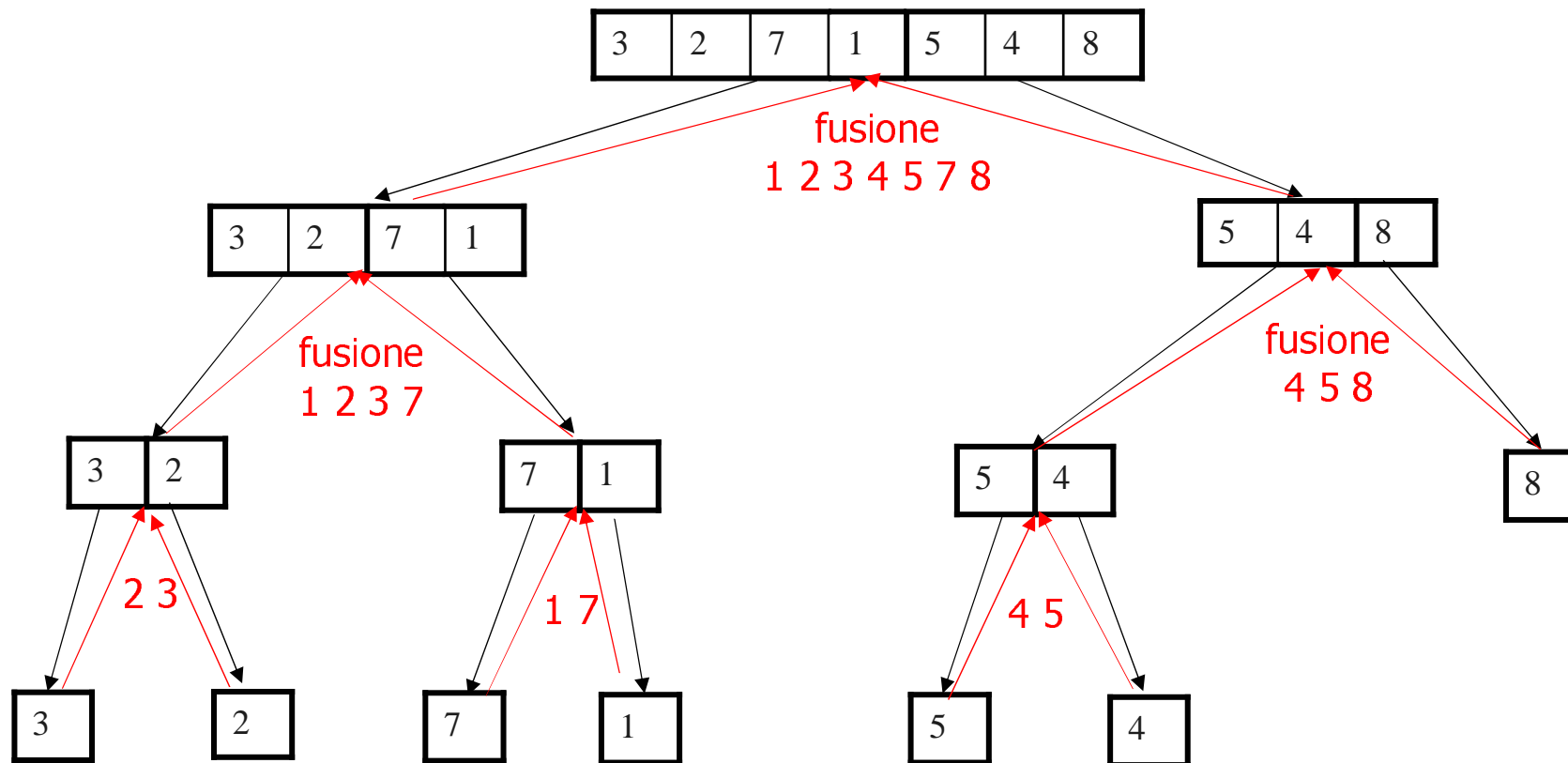
2.  while(i < |P| && j < |Q|){
    if (P[i] < Q[j]){
        R[k]=P[i];
        i++;
    }
    else {
        R[k]=Q[j];
        j++;
    }
    k++;
}

3.  Aggiungi a R la parte
    rimanenti di P o Q
```



La complessità della fusione è  $O(|P| + |Q|)$

# Mergesort: un esempio





# Calcolo della complessita': il mergesort

## Il mergesort

- divide un problema di dimensione  $n$  in 2 problemi di dimensione  $n/2$ 
  - profondita' della ricorsione =  $k = \log_2 n$
- $D(n)=1$ ,  $C(n)$ =“costo fusione”= $O(n)$ 
  - per semplicita', si assume  $D(n)+C(n)=n$
- $T(1)=0$

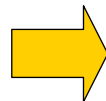


si ottiene

$$T(1) = 0$$

$$T(n) = n + 2T(n/2), n \geq 2$$

soluzione



$$T(n) = n + 2(n/2 + 2T(n/4)) = \dots = kn + 2^k T(1)$$

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



mergesort è  $O(n \log n)$

```
mergesort(A,i,j){  
    if((i==j)) return;  
    if(i<j){  
        m=(i+j)/2;  
        mergesort(A,i,m);  
        mergesort(A,m+1,j);  
        merge(A,i,m,j);  
    }  
}
```



# Una classe di algoritmi ricorsivi: carico di combinazione lineare

---

Mergesort appartiene ad una classe di algoritmi divide et impera

- per i quali il tempo di combinazione e divisione è lineare
  - per essi vale
    - $T(1)=d$
    - $T(n)=a T(n/b)+cn+e$
- dove
- $a$ = numero di sottoproblemi,  $n/b$ =dimensione dei sottoproblemi,

Per tali algoritmi vale

$$T(n) \in O(\log n), \quad \text{se } a < b$$

$$T(n) \in O(n \log n), \quad \text{se } a = b$$

$$T(n) \in O(n^{\log_b a}), \quad \text{se } a > b$$

- con mergesort,  $d=0$ ,  $a=2$ ,  $b=2$

# Ordinamento: quicksort

1. si sceglie un perno  $A[k]$  (es. il primo elemento del vettore)
2. si divide il vettore in due sottovettori  $v1$  e  $v2$  con i valori minori e maggiori di  $A[k]$
3. si applica il quicksort ai sottovettori  $v1$  e  $v2$

```
quicksort(A,i,j){  
    if((i==j)) return;  
    m=perno(A,i,j);  
    quicksort(A,i,p-1);  
    quicksort(A,p+1,j);  
}
```

la dimensione del vettore è 1

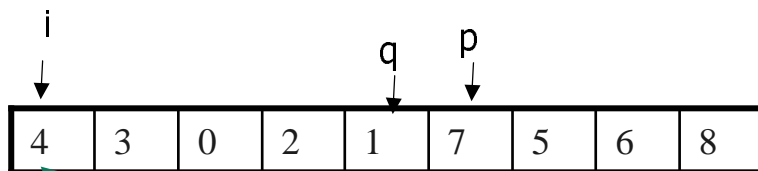
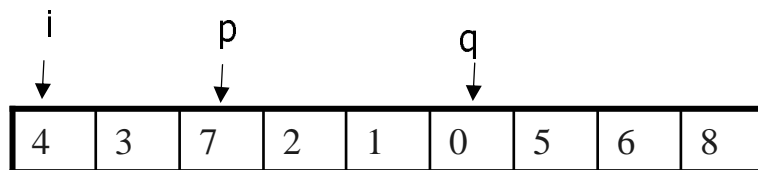
ordinamento dei sottovettori

```
perno(A,i,j){  
    p=i, q=j;  
    while(p<=i){  
        while(A[p]<A[i]) p++;  
        while(A[q]>A[i]) q++;  
        if(p<q) scambia(A,p,q);  
    }  
    scambia(A,i,q);  
}
```

scambia l'elemento in posizione p  
con quello in posizione q

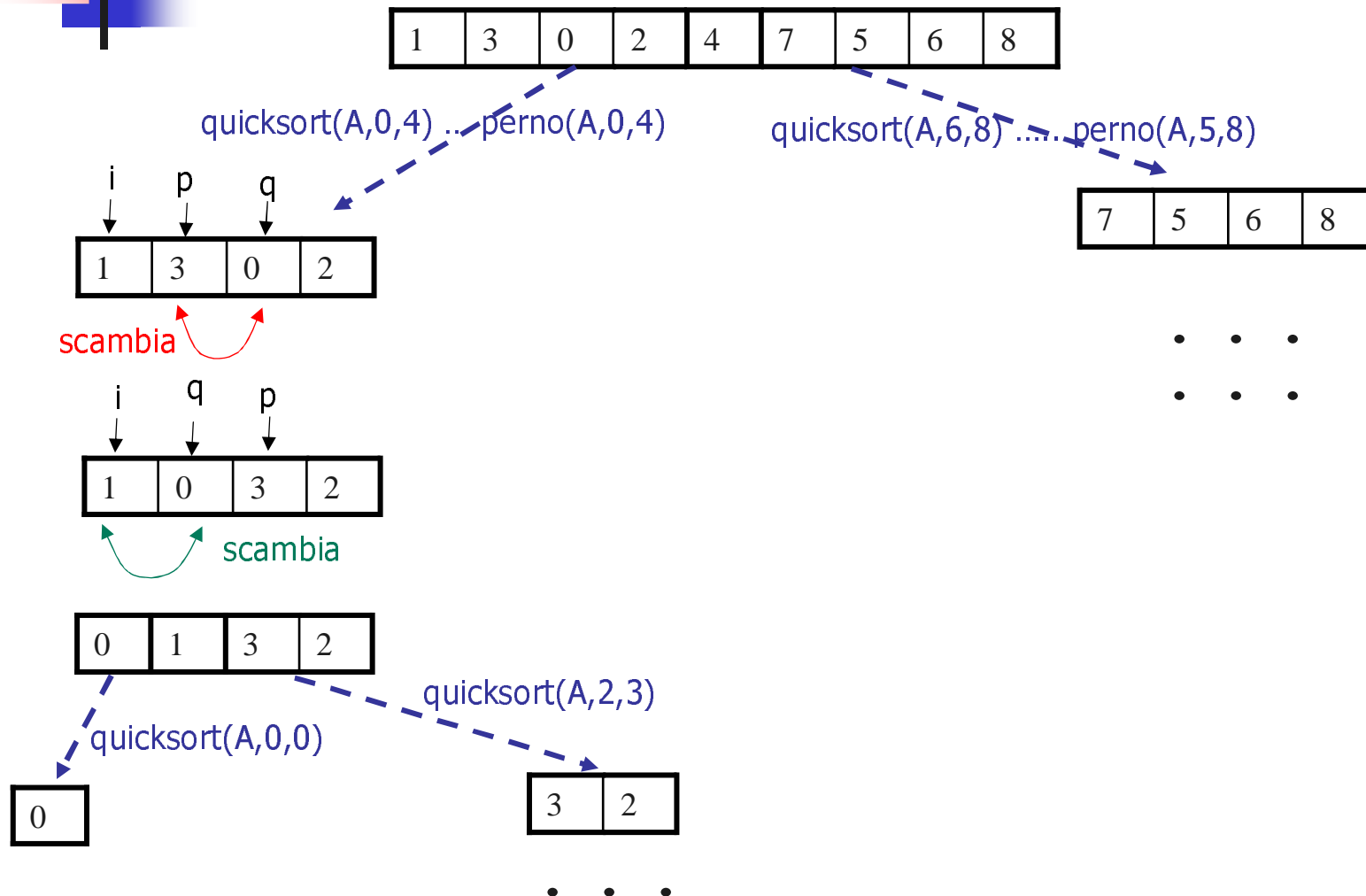
# Quicksort: un esempio

Inizialmente quicksort(A,0,8) chiama perno(A,0,8)



```
perno(A,i,j){  
  p=i, q=j;  
  while(p<=q){  
    while(A[p]<A[i]) p++;  
    while(A[q]>A[i]) q++;  
    if(p<q) scambia(A,p,q);  
  }  
  scambia(A,i,q);  
}
```

# Quicksort: un esempio II



# Calcolo della complessita': il quicksort, caso ottimo

Il caso ottimo si verifica quando il perno è nel centro

- divide un problema di dimensione  $n$  in 2 problemi di dimensione  $n/2$ 
  - profondita' della ricorsione  $k = \log_2 n$
- $D(n)=1$ ,  $C(n)$ ="costo fusione" $=O(n)$ 
  - per semplicita', si assume  $D(n)+C(n)=n$
- $T(1)=0$



si ottiene

- $a=b=2$
- $\text{quicksort} \in O(n \log n)$

```
quicksort(A,i,j){  
    if((i==j)) return;  
    m=perno(A,i,j);  
    quicksort(A,i,p-1);  
    quicksort(A,p+1,j);  
}
```

Il quicksort nel caso ottimo appartiene alla classe dei problemi ricorsivi con combinazione lineare, per cui si applicano le regole

$$T(n) \in O(\log n), \text{ se } a < b$$

$$T(n) \in O(n \log n), \text{ se } a = b$$

$$T(n) \in O(n^{\log_b a}), \text{ se } a > b$$

# Calcolo della complessita': il quicksort, caso pessimo

Il caso pessimo si ha quando il perno è uno degli estremi

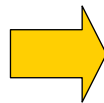
- divide un problema di dimensione  $n$  in un problema di dimensione  $n-1$ 
  - profondita' della ricorsione  $k=n$
- $D(n)=1$ ,  $C(n)$ ="costo fusione" $=O(n)$ 
  - per semplicita', si assume  $D(n)+C(n)=n$
- $T(1)=0$

si ottiene

$$T(1) = 0$$

$$T(n) = n + T(n-1), n \geq 2$$

soluzione



$$T(n) = n + (n-1) + (n-1) + \dots + 1 = n(n+1)/2$$



Il quicksort nel caso pessimo non appartiene alla classe dei problemi ricorsivi con combinazione lineare, perché  $b=n/(n-1)$  dipende da  $n$

Il quicksort è comunque spesso preferito al mergesort perché

- non ha bisogno di memoria aggiuntiva
- è  $O(n \log n)$  nel caso medio

il quick sort è  $O(n^2)$  nel caso pessimo



# Algoritmi di enumerazione

---





# Problema dello zaino: Knapsack

## Definizione

- Dato un insieme  $I=\{a_1, a_2, a_3, a_4, \dots, a_n\}$  di interi, trovare i sottoinsiemi  $S$  di  $I$  i cui elementi hanno somma minore di una costante  $K$

## Es.

- $I=\{2, 4, 8, 9\}$ ,  $K=9$
- soluzioni:  $\{2, 4\}$ ,  $\{2, 8\}$ ,  $\{9\}$ ,  $\{8\}$ ,  $\{4\}$ ,  $\{2\}$

## Algoritmo

- si generano **tutte le soluzioni possibili**

Si generano i sottoinsiemi **con**  $a_k$

Si generano i sottoinsiemi **senza**  $a_k$

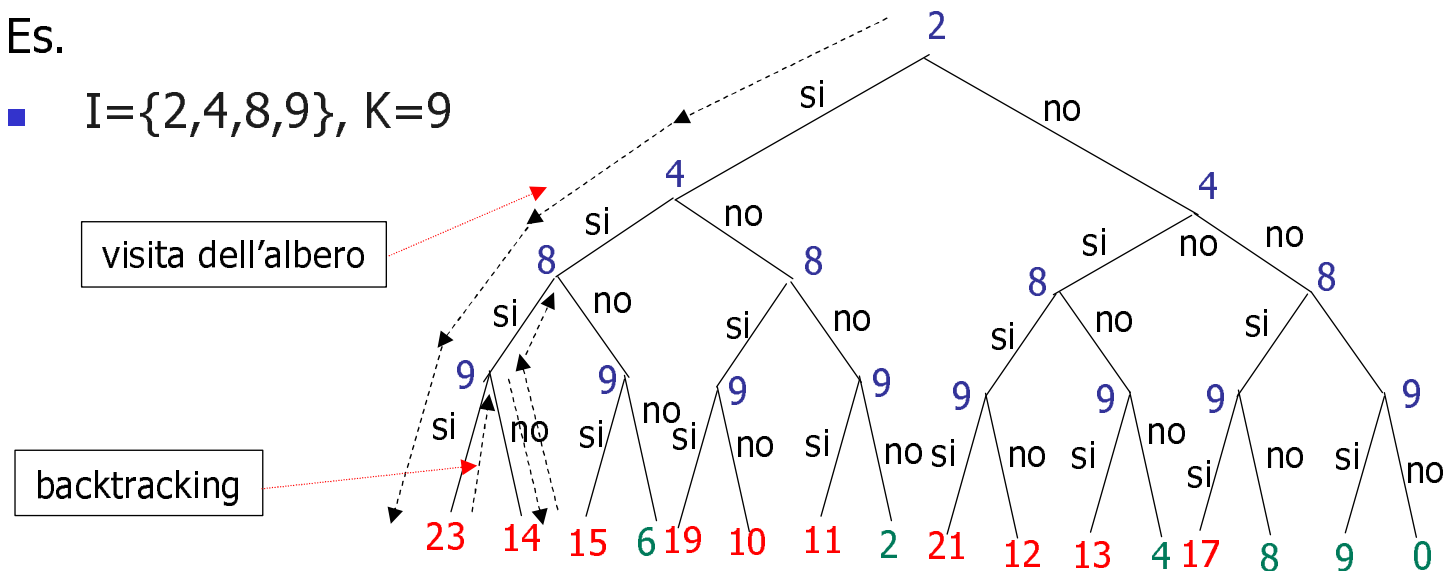
```
S=∅;  
genera(1);  
genera(i){  
    if(i==n && somma(S) < K)  
        stampa(S);  
    ◀ inserisci(S, I[i]);  
        genera(i+1);  
    ▶ rimuovi(I[i]);  
        genera(i+1);  
}
```

# Knapsack: l'albero delle soluzioni

- Il funzionamento dell'algoritmo di enumerazioni si può rappresentare attraverso l'albero delle soluzioni

Es.

- $I = \{2, 4, 8, 9\}$ ,  $K = 9$



Occorre generare  $2^n$  soluzioni: l'algoritmo è esponenziale



# Algoritmi di enumerazione

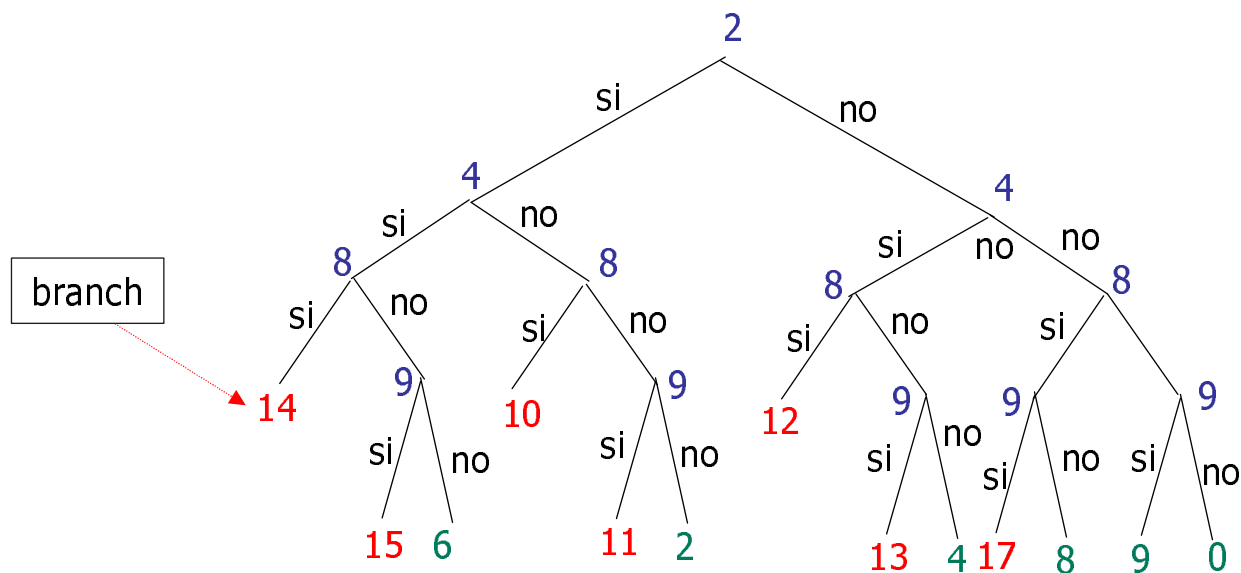
---

- Tentano di risolvere un problema enumerando (esaurendo) tutte le soluzioni
- Spesso sono algoritmi esponenziali
- Gli algoritmi di enumerazione costruiscono l'albero delle soluzioni
  - ad ogni livello genera una soluzione piu' complessa
  - visitato un figlio si torna indietro e si passa ad un altro ([backtracking](#))
- Se si cerca una soluzione specifica (quella ottima)
  - strategia [branch and bound](#)
  - per ogni nodo interno dell'albero si valuta il costo della soluzione ([bound](#))
  - eventualmente si puo' evitare di visitare il sotto albero ([branch](#))

# Knapsack: l'albero delle soluzioni

Es.

- $I=\{2,4,8,9\}$ ,  $K=9$
- Trovare il sottoinsieme  $S$  con il maggiore numero di elementi per cui la somma degli elementi di  $S$  è minore di  $K$



```
S=∅;  
genera(1);  
genera(i){  
    if(somma(S)<k){  
        inserisci(S,I[i]);  
        genera(i+1);  
  
        rimuovi(I[i]);  
        genera(i+1); }  
}
```



# Il problema delle otto regine

## Il problema

- Porre otto regine sulla scacchiera in maniera che nessuna sia sotto scacco
- La soluzione può essere rappresentata con vettore
- Si può estendere al caso generale di una scacchiere  $n \times n$

Soluzione: 3,5,2,8,1,7,4,6

	1	2	3	4	5	6	7	8
1					Q			
2			Q					
3	Q							
4							Q	
5		Q						
6								Q
7						Q		
8				Q				



# Il problema delle n regine

- Una soluzione ricorsiva

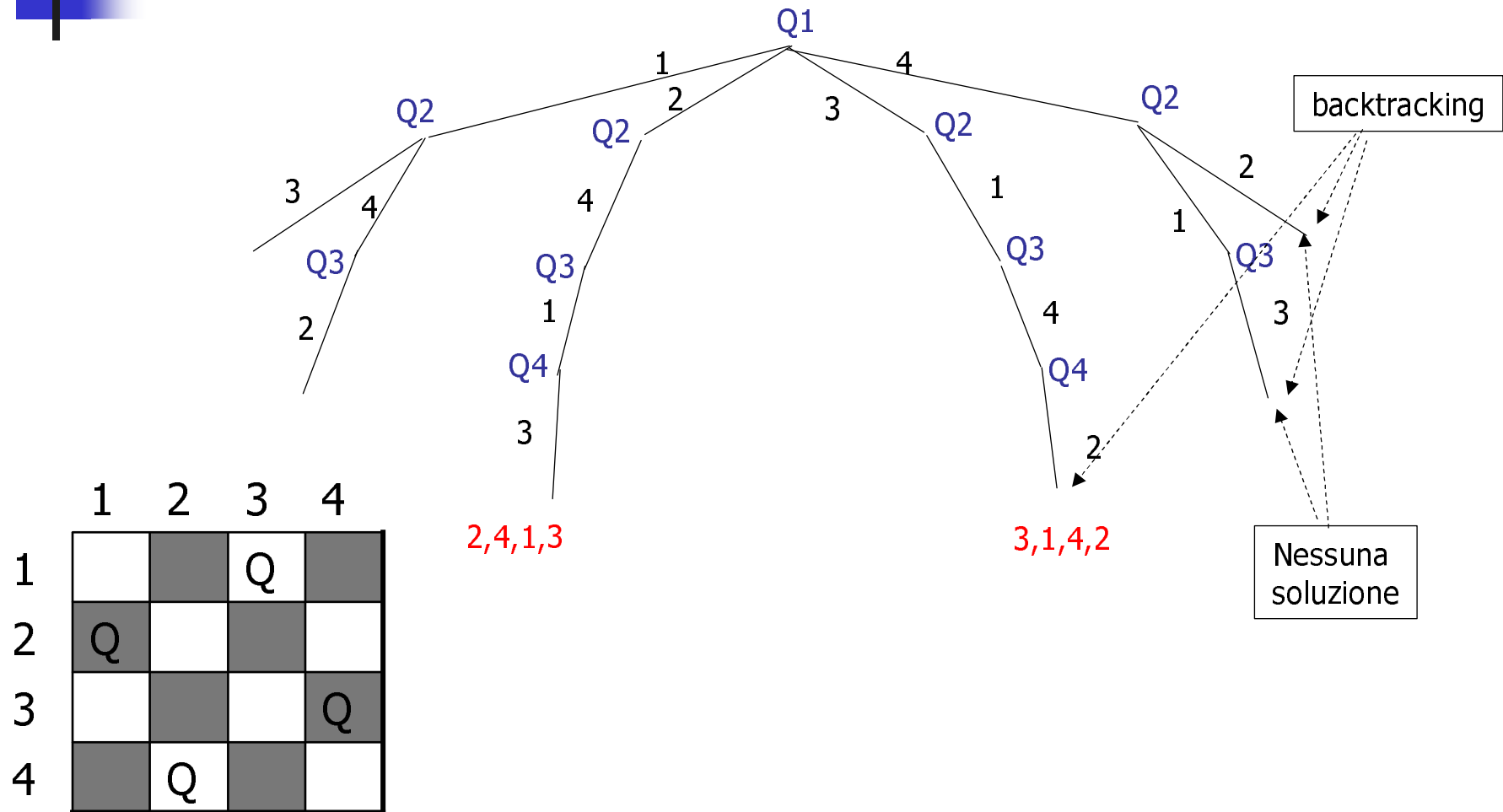
```
nRegine(k){  
  calcola le posizioni  $L_k$  che  
  non sono sotto scacco  
  if( $L_k == 0$ ) return;  
  j=1;  
  while( $j < |L_k|$ ){  
     $P(k) = L_k(j)$ ;  
    if( $k == N$ ) scrivi ( $P(1), \dots, P(k)$ );  
    else nRegine(k+1);  
    j++;  
  }  
}
```

Non ci sono soluzioni

Si prova a mettere  
una regina in ogni posizione  
che non è sotto scacco

Si sono posizionate  
tutte le regine

# Il problema delle 4 regine: l'albero delle decisioni





# Limiti inferiori per la complessita' di un problema

---

- Servono a stabilire se possono esistere **algoritmi migliori** per la risoluzione di un problema
- Un algoritmo è **ottimo** se la sua complessita'  $O(f(n))$  è uguale al limite inferiore  $\Omega(f(n))$  per il problema

## Esempio

- Un algoritmo di ordinamento deve analizzare tutti gli elementi, quindi è almeno  $\Omega(n)$
- **Trovare limiti inferiori significativi è molto difficile**





# Limiti inferiori: criteri

---

## Dimensione dei dati

- si devono analizzare tutti i dati
  - massimo su un vettore lungo  $n$ :  $\Omega(n)$
  - somma  $M+N$  di matrici  $n \times n$ :  $\Omega(n^2)$
  - prodotto  $M*N$  di matrici  $n \times n$ :  $\Omega(n^2)$

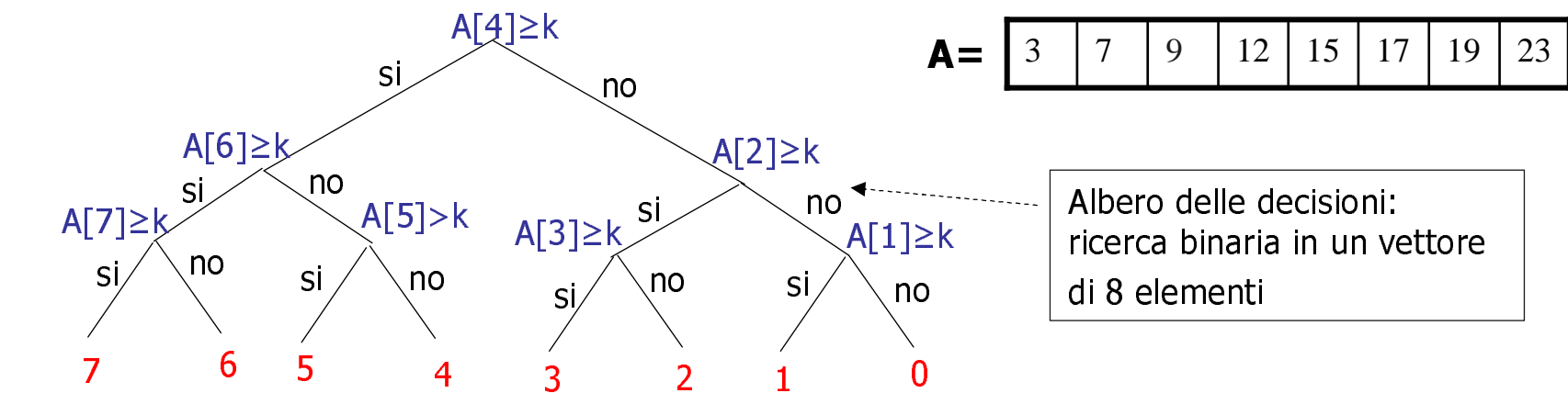
## Eventi contabili

- si devono ripetere alcune operazioni
  - stampare tutte le permutazioni  $n$  elementi:  $\Omega(n!)$

# Alberi di decisione

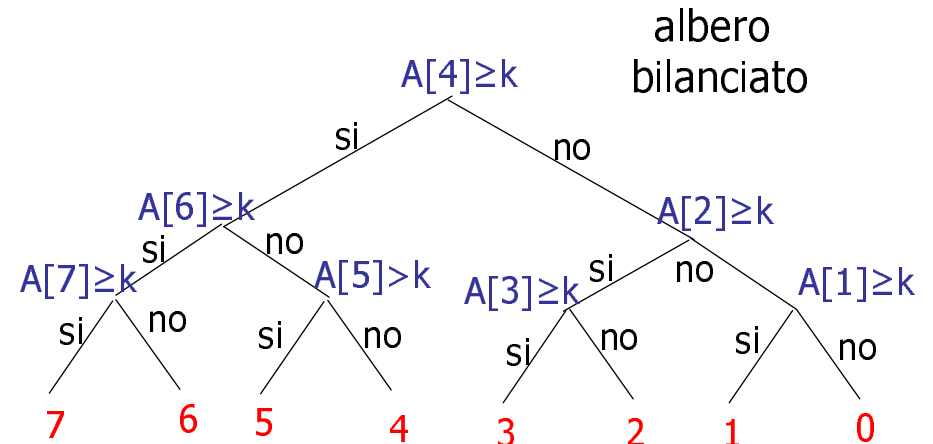
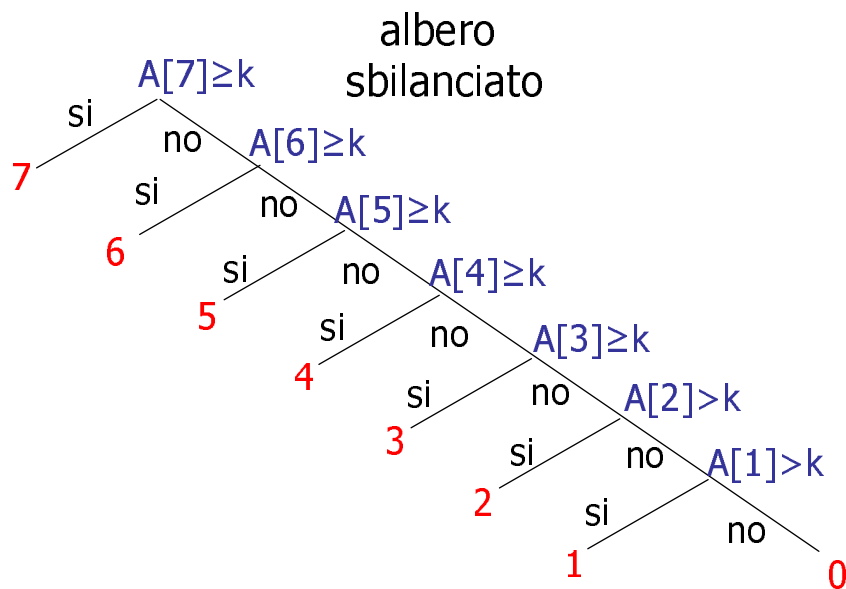
## Intuitivamente

- ogni algoritmo è basato su una **successione di decisioni**
- ogni decisione elimina alcune delle soluzioni possibili
- l'algoritmo può essere rappresentato dall'**albero delle decisioni**
  - i nodi interni rappresentano **decisioni da prendere**
  - le foglie rappresentano le **soluzioni**
  - i cammini dalla radice alle foglie rappresentano un'esecuzione dell'algoritmo
  - la profondità dei cammini indicano il **numero di operazioni**



# Alberi di decisione II

- Se l'albero è perfettamente bilanciato
  - la profondità massima e quella media sono  $O(\log S)$ , dove  $S$  è il numero delle soluzioni
- Se l'albero è il più possibile sbilanciato
  - la profondità massima e quella media sono  $O(S)$





# Limiti inferiori: alberi di decisione

---

## Assunzione

- Un programma può prendere solo un numero di decisioni finite ad ogni istante (if a then ..)
- Ogni programma è rappresentabile come un albero di decisione
- Il programma perfetto (se esiste) è rappresentabile come un albero di decisione perfettamente bilanciato

## Quindi

- nessun algoritmo ha complessità inferiore a  $\log(S)$ , dove  $S$  è il numero delle soluzioni



# Limiti inferiori: l'ordinamento

---

Ordinamento di un vettore di dimensione  $n$

- esistono  $n!$  soluzioni
- $\log(n!)$  è un limite inferiore

$$\log(n!) \cong \log\left(\sqrt{2\pi n}\left(\frac{n}{2}\right)^n\right) = \log(\sqrt{2\pi n}) + n \log\left(\frac{n}{2}\right)$$

Quindi

- $n \log(n)$  è un limite inferiore per l'ordinamento
- il mergesort e l'heapsort sono **algoritmi ottimi** (non è possibile fare meglio)