

Università di Pisa

NICOLETTA DE FRANCESCO

Algoritmi e strutture dati

a.a. 2019/2020

Nozione di Algoritmo

Origine del nome «algoritmo»



Muhammad ibn Musa **al-Khwarizmi**
matematico arabo dell'800 (libri scritti **813-830** circa)

notazione posizionale dei numeri e lo 0
(Algoritmi de numero Indorum)- algebra

Il nome algoritmo con il significato attuale
viene usato dal XIX secolo

Prime caratterizzazioni di algoritmo e computer

Alan Turing : Macchina di Turing (1936)

Alonso Church: lambda-calcolo (1936)

Concetto di algoritmo

- insieme **finito** di istruzioni teso a risolvere un problema
- Input e Output
- ogni istruzione deve essere ben definita ed eseguibile in un **tempo finito** da un **agente di calcolo**
- E' possibile utilizzare una memoria per i risultati intermedi

Primi algoritmi nella storia

Algoritmi aritmetici

Babilonesi (circa 2500 a.c.)

Egizi (circa 1500 a.c.)

Greci:

algoritmo di Euclide (300 a.c.)

setaccio di Eratostene (sieve, crivello) (I secolo d.c.)

Algoritmo di Euclide

Trovare il Massimo Comun Divisore fra due numeri interi non negativi

MCD(30,21)=3

TH: il MCD fra due numeri è uguale al MCD fra il più piccolo e la differenza fra i due

```
int MCD(int x, int y) {  
    while (x!=y)  
        if (x < y) y=y-x; else x=x-y;  
    return x;  
}
```

x= 30, y=21

x= 9, y=21

x = 9, y=12

x =9, y= 3

x =6, y=3

x= 3, y=3

Algoritmo di Euclide con il resto della divisione

TH: il MCD fra due numeri x e y è uguale al MCD fra y e il resto della divisione fra x e y

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

$x=30, y=21$

$x=21, y=9$

$x=9, y=3$

$x=3, y=3$

$x=3, y=0$

Numeri primi

Trovare tutti i numeri primi fino a un dato numero n

Algoritmo inefficiente: dividere ogni numero minore o uguale a n per tutti i suoi predecessori

Algoritmo poco più inefficiente: dividere ogni numero n per tutti i suoi predecessori fino a radice quadrata di n .

Algoritmo molto più efficiente

Setaccio di Eratostene

- 1. Elencare tutti i numeri fino a n**
- 2. Partendo dal primo numero primo, 2, cancellare dall'elenco tutti i multipli di 2**
- 3. Ripetere il procedimento con i numeri seguenti non ancora cancellati**

Setaccio di Eratostene

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Setaccio di Eratostene

Cancello i multipli di 2

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Setaccio di Eratostene

Cancello i multipli di 3 a partire da 9

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Setaccio di Eratostene

Cancello i multipli di 5 a partire da 25

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Setaccio di Eratostene

Cancello i multipli di 7 a partire da 49

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Stop perché $11 * 11 > 100$

Setaccio di Eratostene

```
void setaccio (int n){  
    bool primi[n]; primi[0] = primi[1] = false;  
    for (int i = 2; i < n; i++) primi[i] = true; //inizializza  
    int i = 1;  
    for (i++; i * i < n; i++){ //scorri i numeri successivi a partire da i*i *  
        while (!primi[i]) i++; // cerca il prossimo primo  
        for (int k=i * i; k< n; k+= i) primi[k]= false; //cancella multipli di i  
    }  
    for(int j = 2; j<n; j++)  
        if (primi[j]) cout <<j<< endl; //stampa tutti primi  
}  
  
* i numeri non cancellati precedenti i*i sono primi
```


Complessità degli algoritmi (programmi)

complessità di un algoritmo

funzione (sempre positiva) che associa alla **dimensione** del problema il **costo** della sua risoluzione

Costo: tempo, spazio (memoria),

dimensione: dipende dai dati

Per confrontare due algoritmi si confrontano le relative funzioni di complessità

complessità dei programmi : esempio

$T_P(n)$ = Complessità del **tempo** di esecuzione del programma P al variare di **n**:

```
int max(int a[], int n) {  
    int m=a[0];  
    for (int i=1; i < n;i++)  
        if (m < a [ i ]) m = a[i];  
    return m;  
}
```

Se tutti i tempi
costanti sono uguali
a 1:

$$T_{\max}(n) = 4n$$

E' necessario trovare un metodo di calcolo della complessità che misuri **l'efficienza come proprietà dell'algoritmo, cioè astragga**

- **dal computer su cui l'algoritmo è eseguito**
- **dal linguaggio in cui l'algoritmo è scritto**

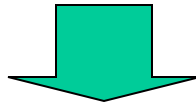
**L'efficienza deve essere misurata
indipendentemente anche da specifiche dimensioni
dei dati:**

**la funzione della complessità deve essere
analizzata nel suo comportamento asintotico**

tre programmi P, Q ed R

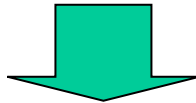
$$T_P(n) = 2n^2 \quad T_Q(n) = 100n \quad T_R(n) = 5n$$

Per $n \geq 50$, $T_Q(n) \leq T_P(n)$



$T_Q(n)$ ha complessità **minore o uguale a** $T_P(n)$
ma non vale il contrario

Per $n \geq 3$, $T_R(n) \leq T_P(n)$

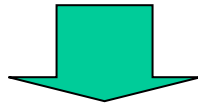


$T_R(n)$ ha complessità **minore o uguale a** $T_P(n)$
ma non vale il contrario

tre programmi P, Q ed R

$$T_P(n) = 2n^2 \quad T_Q(n) = 100n \quad T_R(n) = 5n$$

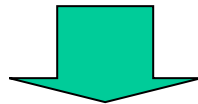
Per ogni n , $T_R(n) \leq T_Q(n)$



$T_R(n)$ ha complessità minore o uguale a $T_Q(n)$

Per ogni n , $T_Q(n) \leq 20T_R(n)$

$T_Q(n)$ ha complessità **minore o uguale** a $T_R(n)$



$T_Q(n)$ e $T_R(n)$ hanno la **stessa** complessità

Notazione O grande (limite asintotico superiore)

$f(n)$ è di ordine $O(g(n))$ se esistono
un intero n_0 ed una costante $c > 0$ tali che
per ogni $n \geq n_0$: $f(n) \leq c g(n)$

Notazioni

$f(n)$ è di ordine $O(g(n))$

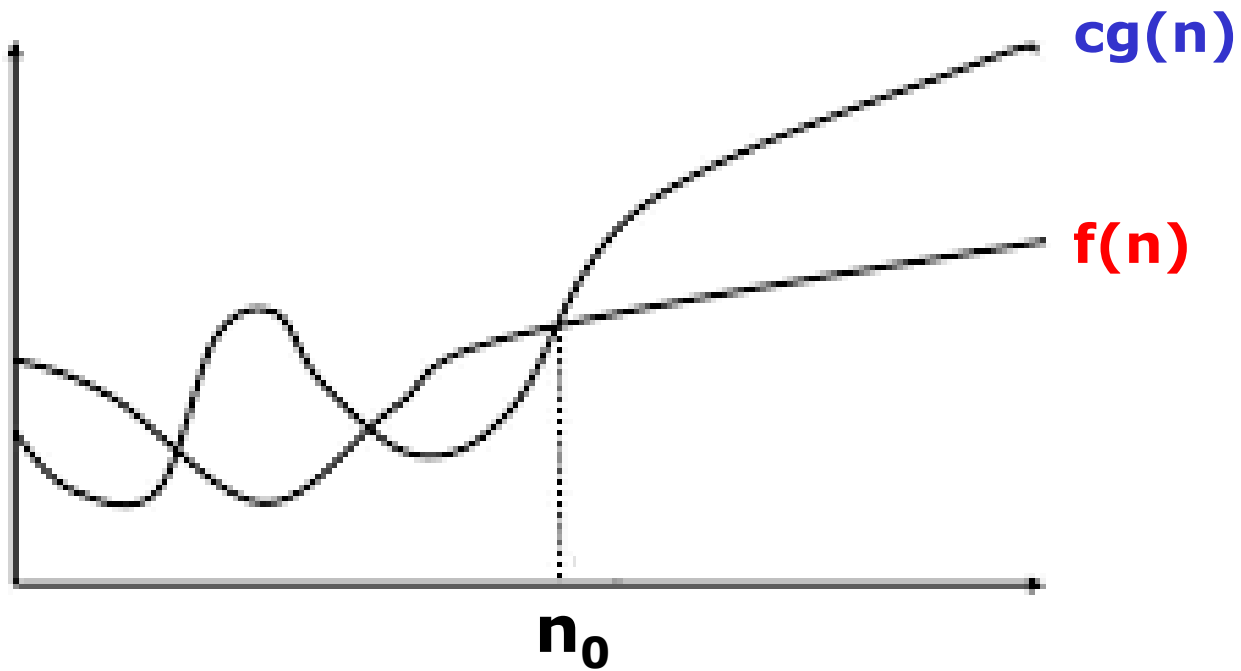
$f(n)$ è $O(g(n))$

$f(n) \in O(g(n))$

$f(n) = O(g(n))$ notazione ambigua

Notazione O grande

$f(n)$ è $O(g(n))$



Complessità computazionale

$T_Q(n)$ è $O(T_P(n))$ [$n_0=50, c=1$] oppure [$n_0=1, c= 50$]

$T_R(n)$ è $O(T_P(n))$ [$n_0=3, c=1$]

$T_R(n)$ è $O(T_Q(n))$ [$n_0=1, c=1$]

$T_Q(n)$ è $O(T_R(n))$ [$n_0=1, c=20$]

$$T_P(n) = 2n^2$$

$$T_Q(n) = 100n$$

$$T_R(n) = 5n$$

$T_P(n)$ non è $O(T_Q(n))$

$T_P(n)$ non è $O(T_R(n))$

Notazioni

$f(n)$ è di ordine $O(g(n))$

$f(n)$ è $O(g(n))$

$f(n) \in O(g(n))$

Una funzione $f(n)=\text{expr}$ si indica soltanto con expr

$f(n)= 3-n$



$3-n$

$f(n)=100n$ è $O(g(n)=5n)$



$100n$ è $O(5n)$

esempi

$$T_{\max}(n) = 4n \in O(n) \quad [n_0=1, c=4]$$

$$T_{\max}(n) = 4n \in O(n^2) \quad [n_0=4, c=1]$$

$$T_Q(n), T_R(n) \in O(n)$$

$$2^{n+10} \in O(2^n) \quad [n_0=1, c=2^{10}]$$

$$n^2 \in O(1/100 n^2) \quad [n_0=1, c=100]$$

$$n^2 \in O(2^n) \quad [n_0=4, c=1]$$

Complessità computazionale

**$O(f(n))$ = insieme delle funzioni
di ordine $O(f(n))$**

$$O(n) = \{ \text{costante}, n, 4n, 300n, 100 + n, .. \}$$

$$O(n^2) = O(n) \cup \{ n^2, 300 n^2, n + n^2, ... \}$$

REGOLA DEI FATTORI COSTANTI

Per ogni costante positiva k , $O(f(n)) = O(kf(n))$.

REGOLA DELLA SOMMA

Se $f(n)$ è $O(g(n))$, allora $f(n)+g(n)$ è $O(g(n))$.

REGOLA DELLA PRODOTTO

Se $f(n)$ è $O(f_1(n))$ e $g(n)$ è $O(g_1(n))$, allora $f(n)g(n)$ è $O(f_1(n)g_1(n))$.

regole

- Se $f(n)$ è $O(g(n))$ e $g(n)$ è $O(h(n))$, allora
 $f(n)$ è $O(h(n))$
- per ogni costante k , k è $O(1)$
- per $m \leq p$, n^m è $O(n^p)$
- Un polinomio di grado m è $O(n^m)$

esempi

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

due funzioni

$$\begin{array}{ll} f(n) = & \begin{array}{ll} n^3 & \text{se } n \text{ e' pari} \\ n^2 & \text{se } n \text{ e' dispari} \end{array} \\ g(n) = & \begin{array}{ll} n^2 & \text{se } n \text{ e' primo} \\ n^3 & \text{se } n \text{ e' composto} \end{array} \end{array}$$

$$f(n) \text{ è } O(g(n)) \quad n_0=3, \quad c=1$$

**non vale il contrario: esistono infiniti numeri
composti dispari**

funzioni incommensurabili

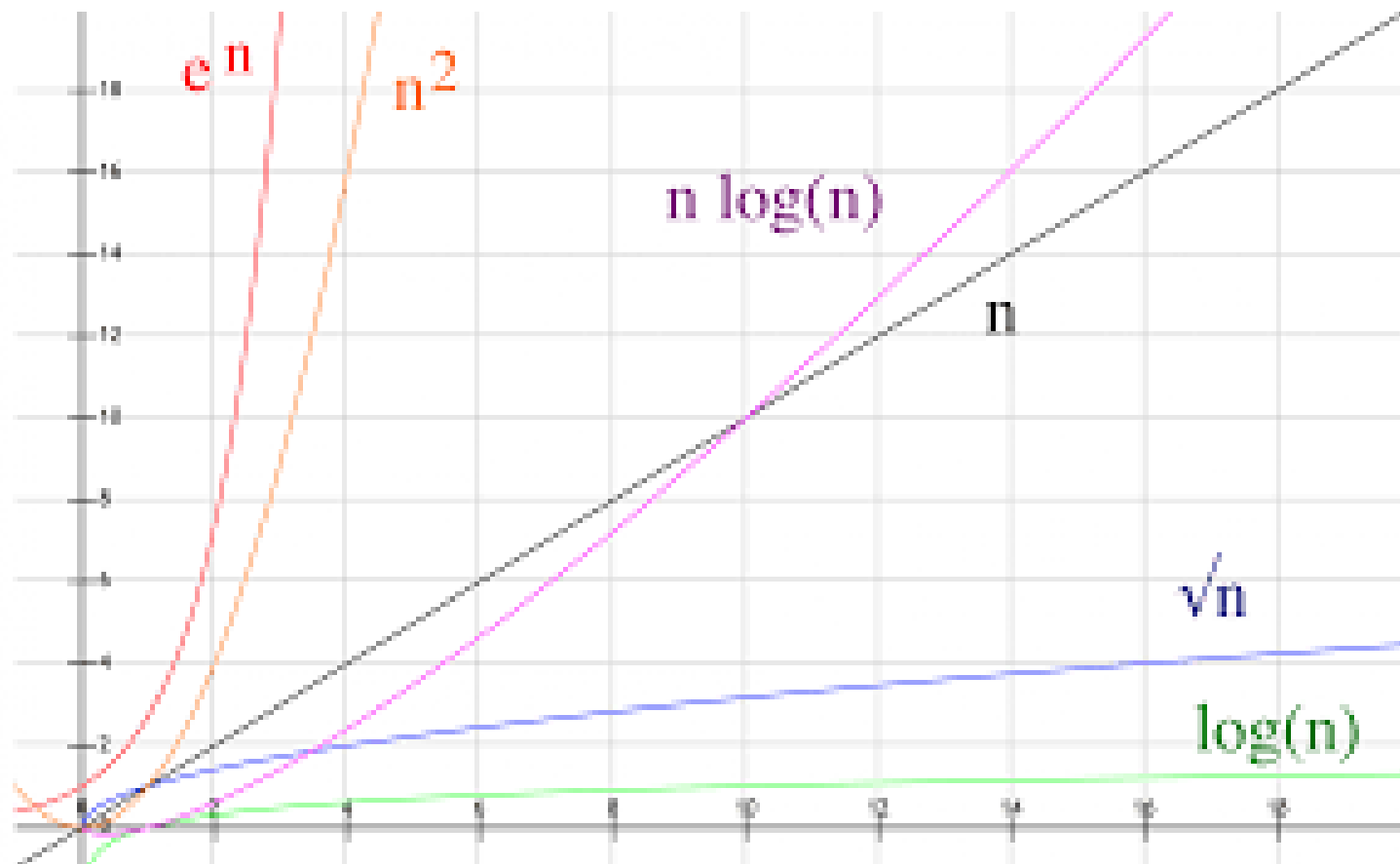
$$f(n) = \begin{cases} n & \text{se } n \text{ e' pari} \\ n^2 & \text{se } n \text{ e' dispari} \end{cases}$$

$$g(n) = \begin{cases} n^2 & \text{se } n \text{ e' pari} \\ n & \text{se } n \text{ e' dispari} \end{cases}$$

Classi di Complessità

$O(1)$	costante
$O(\log n)$	logaritmica ($\log_a n = \log_b n \log_a b$)
$O(n)$	lineare
$O(n \log n)$	nlogn
$O(n^2)$	quadratica
$O(n^3)$	cubica
..	
$O(n^p)$	polinomiale
$O(2^n)$	esponenziale
$O(n^n)$	esponenziale

Classi di Complessità



teorema

per ogni k , $n^k \in O(a^n)$, per ogni $a > 1$

Una qualsiasi funzione polinomiale ha minore complessità di una qualsiasi funzione esponenziale

crescita esponenziale



$2^{64}-1$ chicchi : 18.446.744.073.709.551.615 chicchi,
superiore ai raccolti di grano di tutto il mondo

Notazione Ω (omega grande) (limite asintotico inferiore)

$f(n)$ è $\Omega (g(n))$ se esistono

un intero n_0 ed una costante $c > 0$ tali che

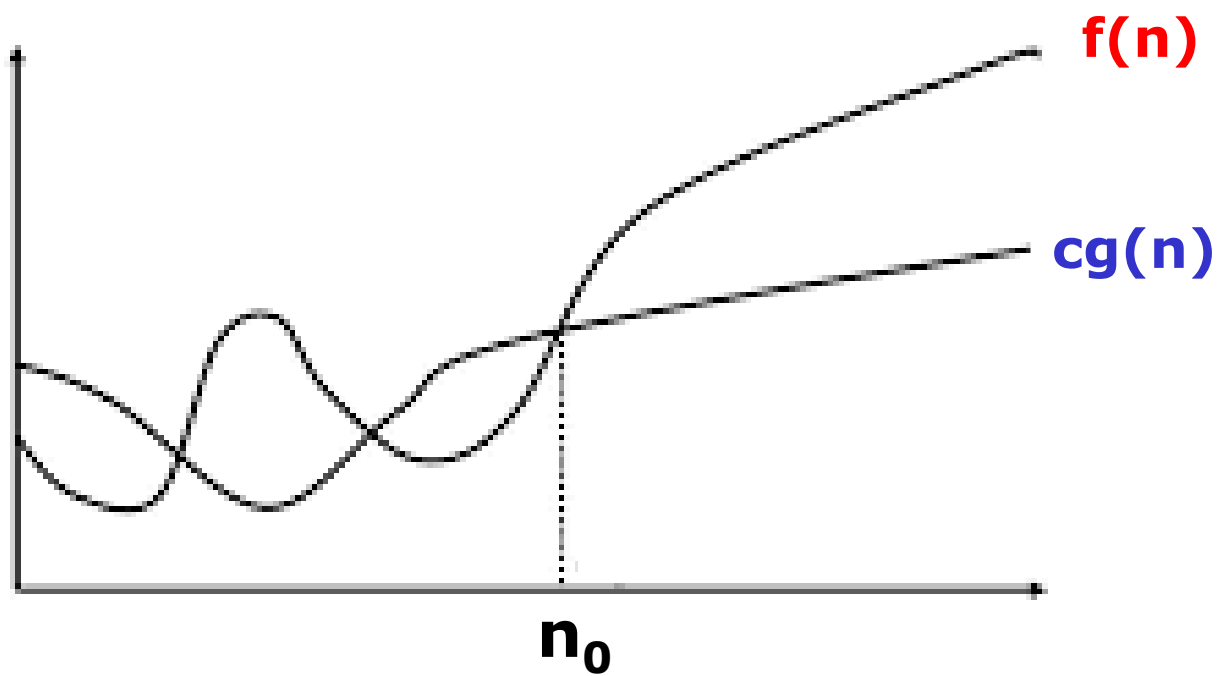
per ogni $n \geq n_0$: $f(n) \geq c g(n)$

Notazione Ω esempi

- $2n^2 + 3n + 2$ è $\Omega(n)$ $n_0=1, c=1$
- $2n^2 + 3n + 2$ è $\Omega(n^2)$ $n_0=1, c=1$
- n^2 è $\Omega(2n^2)$ $n_0=1, c=1/2$
- n^2 è $\Omega(100n)$ $n_0=101, c=1$ (oppure $n_0=1, c=1/100$)

Notazione Ω grande

$f(n)$ è $\Omega(g(n))$



Notazione Θ (theta grande) (limite asintotico stretto)

$f(n)$ è $\Theta(g(n))$ se

$f(n)$ è $O(g(n))$ e $f(n)$ è $\Omega(g(n))$

**$f(n)$ è $\Theta(g(n))$ quando f e g hanno lo
stesso ordine di complessità**

definizione alternativa di Θ

$f(n)$ è $\Theta (g(n))$ se esistono

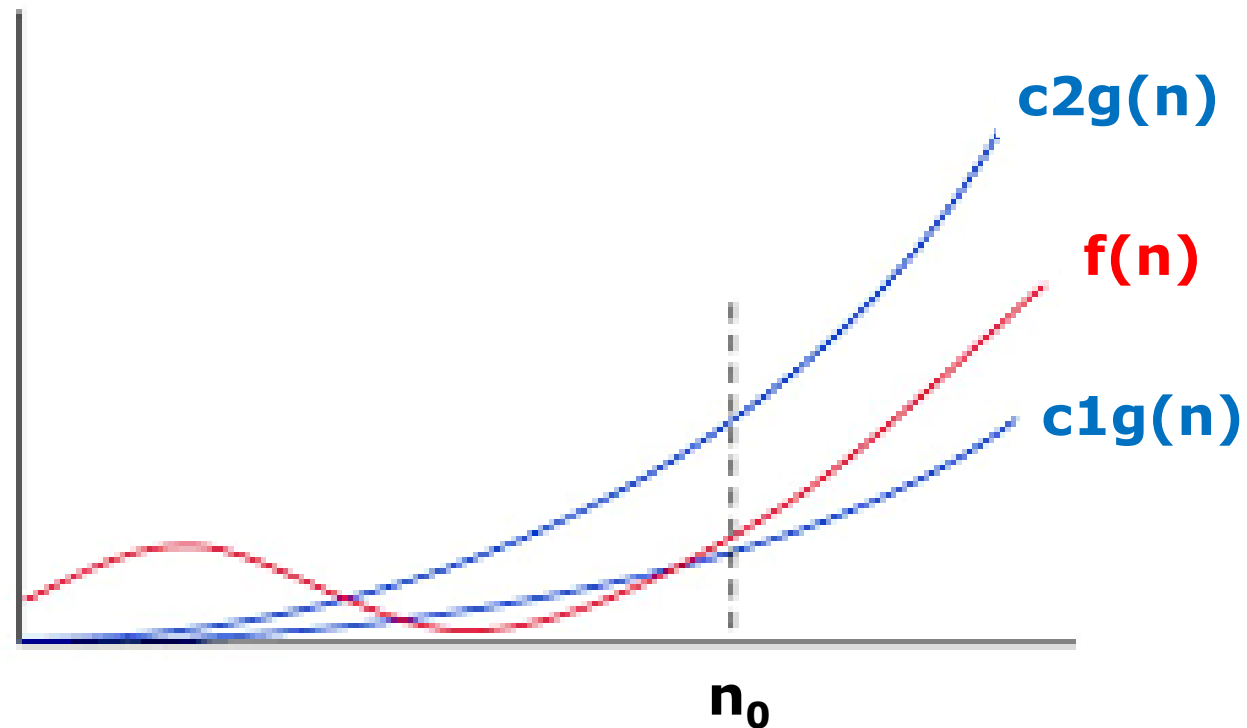
**un intero n_0 e due costante $c1, c2 > 0$ tali
che**

per ogni $n \geq n_0$:

$$c1 g(n) \leq f(n) \leq c2 g(n)$$

Notazione Θ

$f(n)$ è $\Theta (g(n))$



esempi

- $2n^2 + 3n + 2$ è $\Omega(n)$ ma non è $O(n)$, quindi non è $\Theta(n)$

- $2n^2 + 3n + 2$ è $O(n^2)$ [n0=1 , c=7]

- $2n^2 + 3n + 2$ è $\Omega(n^2)$ [n0=1 , c=1]

- $2n^2 + 3n + 2$ è $\Theta(n^2)$

- $f(n)$ è $O(g(n))$ se e solo se $g(n)$ è $\Omega(f(n))$
- se $f(n)$ è $\Theta(g(n))$ allora $g(n)$ è $\Theta(f(n))$
- Per Ω e Θ valgono le regole dei fattori costanti, del prodotto e della somma
- un polinomio di grado m è $\Theta(n^m)$

Complessità dei programmi iterativi

Programmi iterativi

C: costrutti del linguaggio -> **Classi di complessità**

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

$$C [V] = C [I] = O(1)$$

$$C [E1 \text{ op } E2] = C [E1] + C [E2]$$

$$C [I[E]] = C [E]$$

$$C [I=E;] = C [E]$$

$$C [I[E1] =E2;] = C [E1] + C [E2]$$

Programmi iterativi

$C [\text{return } E;] = C [E]$

$C [\text{if } (E) C] = C [E] + C [C]$

$C [\text{if } (E) C1 \text{ else } C2] =$
 $C [E] + C [C1] + C [C2]$

Programmi iterativi

$C [\text{for} (E1; E2; E3) C] =$

$C [E1] + C [E2] +$

$(C [C] + C [E2] + C [E3]) O(g(n))$

$g(n)$: numero di iterazioni

$C [\text{while} (E) C] =$

$C [E] + (C [C] + C [E]) O(g(n))$

Programmi iterativi

$$C [\{ C1 \dots Cn \}] = C [C1] + \dots + C [Cn]$$

$$C [F (E1, .. En)] =$$

$$C [E1] + \dots + C [En] + C [\{ C \dots C \}]$$

$$\text{se } T F (T1 I1, \dots, Tn In) \{ C \dots C \}$$

Selection sort

```
void exchange( int& x, int& y) {  
  O(1) int temp = x;  
  O(1) x = y;  
  O(1) y = temp;  
}
```

```
void selectionSort(int A[ ], int n) {  
  O(n2) for (int i=0; i< n-1; i++) {  
    O(1) int min= i;  
    O(n) for (int j=i+1; j< n; j++)  
      O(1) if (A[ j ] < A[min]) min=j;  
    O(1) exchange(A[i], A[min]);  
  }  
}
```

O(n²)

Bubblesort

```
void bubbleSort(int A[], int n) {  
     $O(n^2)$  for (int i=0; i < n-1; i++)  
         $O(n)$      for (int j=n-1; j >= i+1; j--)  
             $O(1)$      if (A[j] < A[j-1]) exchange(A[j], A[j-1]);  
}
```

numero di scambi = $O(n^2)$

con selectionSort numero di scambi = $O(n)$

Esempio (I)

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

risultato: $O(n)$

complessità: $O(1)$

```
int h (int x){  
    return x*x;  
}
```

risultato: $O(n^2)$

complessità: $O(1)$

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

risultato: $O(n)$

complessità: $O(n)$

Esempio (II)

```
void g (int n){    // n >= 0
    for (int i=1; i <= f(n); i++)
        cout << f(n);
}
```

complessità: $O(n)$

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

complessità: $O(n^2)$

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

complessità: $O(n^2)$

Esempio (III)

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

complessità: $O(n)$

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

complessità: $O(n^2)$

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

complessità: $O(n)$

Moltiplicazione fra matrici

```
void matrixMult (int A[N] [N], int B[N] [N], int C [N] [N]) {
```

```
     $O(n^3)$  for (int i=0; i < N; i++)
```

```
         $O(n^2)$      for (int j=0; j < N; j++) {
```

```
             $O(1)$          C[ i ] [ j ]=0;
```

```
             $O(n)$          for (int k=0; k < N; k++)
```

```
                 $O(1)$          C[ i ] [ j ]+=A[ i ] [ k ] * B[ k ] [ j ];
```

```
        }
```

```
    }
```

$O(n^3)$

Ricerca lineare e div2

```
int linearSearch (int A [], int n, int x) {  
    int b=0;  
    for (int i=0; !b & (i<n); i++)  
        if (A[ i ] == x) b=1;  
    return b;  
}
```

$O(n)$

```
int div2(int n) {  
    int i=0;  
    while (n > 1) {  
        n=n/2;  
        i++;  
    }  
    return i;  
}
```

$O(\log n)$

Complessità dei programmi ricorsivi

Fattoriale di un numero naturale : $n!$

$$0! = 1$$

$n! = 1 \times 2 \times \dots \times n$ se $n > 0$ **definizione iterativa**

$$0! = 1$$

$n! = n \times (n-1)!$ se $n > 0$ **definizione induttiva (o ricorsiva)**

fattoriale: algoritmo iterativo

$0! = 1$

$n! = 1 \times 2 \times \dots \times n$

```
int fact(int n) {  
    if (n == 0) return 1;  
    int a=1;  
    for (int i=1; i<=n; i++) a=a*i;  
    return a;  
}
```

fattoriale: algoritmo ricorsivo

$0! = 1$

$n! = n * (n-1)! \text{ se } n > 0$

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

Programmi ricorsivi: moltiplicazione

mult (0, y) = 0

mult (n,y)= y + mult (n-1,y) se n>0

```
int mult(int x, int y) {  
    if (x == 0) return 0;  
    return y + mult(x-1,y);  
}
```


Programmi ricorsivi : pari e massimo comun divisore

```
int pari(int x) {  
    if (x == 0) return 1;  
    if (x == 1) return 0;  
    return pari(x-2);  
}
```

Algoritmo di Euclide

```
int MCD (int x, int y) {  
    if (x == y) return x;  
    if (x < y) return MCD (x, y-x);  
    return MCD (x-y, y);  
}
```

Regole da rispettare

Regola 1

**individuare i casi base in cui la funzione è definita
immediatamente**

Regola 2

**effettuare le chiamate ricorsive su un insieme più
"piccolo" di dati**

Regola 3

**fare in modo che alla fine di ogni sequenza di chiamate
ricorsive, si ricada in uno dei casi base**

Regole da rispettare

```
int pari_errata(int x) {  
    if (x == 0) return 1;  
    return pari_errata(x-2);  
}
```

```
int MCD_errata(int x, int y) {  
    if (x == y) return x;  
    if (x < y) return MCD_errata(x, y-x);  
    return MCD_errata(x, y);  
}
```

definizione di **LISTA**

- **NULL** (sequenza vuota) è una **LISTA**
- un elemento seguito da una **LISTA** è una **LISTA**

```
struct Elem {  
    InfoType inf;  
    Elem*    next;  
};
```

programmi ricorsivi su liste

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

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

programmi ricorsivi su liste

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

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

programmi ricorsivi su liste

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

programmi ricorsivi su liste

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

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

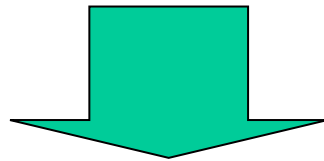

Induzione naturale

Sia P una proprietà sui naturali.

Base. P vale per 0

Passo induttivo. per ogni naturale n è vero che:

Se P vale per n allora P vale per $(n+1)$



P vale per tutti i naturali

Somma dei primi n numeri naturali

Dimostrare con il principio di induzione naturale che la somma dei primi n numeri è $n(n+1)/2$

$$\Sigma_{0..n} = n(n+1)/2$$

Base: $\Sigma_{0..0} = (0*1)/2 = 0$

Passo induttivo:

Ip: $\Sigma_{0..n} = n(n+1)/2$

Tesi: $\Sigma_{0..n+1} = [(n+1)(n+2)]/2$

Dim:

$$\begin{aligned}\Sigma_{0..n+1} &= \Sigma_{0..n} + (n+1) && \text{def} \\ &= n(n+1)/2 + (n+1) && \text{ip} \\ &= [n(n+1) + 2(n+1)]/2 \\ &= [(n+1)(n+2)]/2\end{aligned}$$

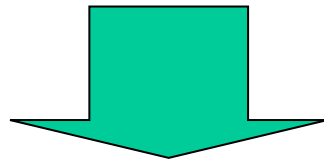
Induzione completa

Sia P una proprietà sui naturali.

Base. P vale per 0

Passo induttivo. per ogni naturale n è vero che:

Se P vale per ogni $m \leq n$ allora P vale per $(n+1)$



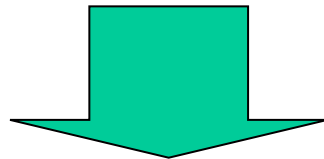
P vale per tutti i naturali

Insieme ordinato S

Base. P vale per i minimali di S

Passo induttivo. per ogni elemento E di S è vero che:

Se P vale per ogni **elemento minore** di E allora P vale per E



P vale per S

Complessità dei programmi ricorsivi

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

$$T(0) = a$$

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

Relazione di ricorrenza

soluzione

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

$$T(0) = a$$

$$T(1) = b + a$$

$$T(2) = b + b + a = 2b + a$$

$$T(3) = b + 2b + a = 3b + a$$

.

.

$$T(n) = nb + a$$

$T(n)$ è $O(n)$

selection sort ricorsiva

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

$$T(1) = a$$

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

soluzione

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

$$T(1) = a$$

$$T(2) = 2b + a$$

$$T(3) = 3b + 2b + a$$

.

.

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

$T(n)$ è $O(n^2)$

quicksort

quicksort(array A ,*inf*, *sup*)

Lavora sulla **porzione di array** compresa fra ***inf*** e ***sup***:

1. Scegli un **perno**
2. Dividi l'array in **due parti**: nella prima metti gli elementi **\leq perno** e nella seconda quelli **\geq perno**
3. chiama **quicksort** sulla prima parte (se contiene almeno 2 elementi)
4. chiama **quicksort** sulla seconda parte (se contiene almeno 2 elementi)

quicksort

la divisione in due parti dell'array avviene mediante scambi fra elementi utilizzando due **cursori s** e **d**, inizialmente posti uno su **inf** e l'altro su **sup**:

1. Fino a quando $s \leq d$:

```
2. {Porta avanti s fino a che  $A[s] < \text{perno}$ ;  
Porta avanti d fino a che  $A[d] > \text{perno}$ ;  
Scambia  $A[s]$  con  $A[d]$ ;  
S++;  
D--;  
}
```

3. Se $\text{inf} < d$ chiama quicksort sulla prima parte (da **inf** a **d**);

4. Se $s < \text{sup}$ chiama quicksort sulla seconda parte (da **s** a **sup**);

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;  
        exchange(A[s], A[d]);  
        s++;  
        d--;  
    };  
  
    if (inf < d)  
        quickSort(A, inf, d);  
    if (s < sup)  
        quickSort(A, s, sup);  
}
```

O(n)

Array A

0	1	2	3	4
2	5	3	1	7

quicksort(A, 0, 4)

Inf=0 , Sup=4, perno=3

0	1	2	3	4
2	5	3	1	7
s				d

while

0	1	2	3	4
2	5	3	1	7
	s		d	

Scambio, poi s++, d--

0	1	2	3	4
2	1	3	5	7
		s, d		

while

0	1	2	3	4
2	1	3	5	7

s, d

Scambio, poi s++, d--

0	1	2	3	4
2	1	3	5	7

d

s

s>d: fine del while

0	1	2	3	4
2	1	3	5	7
	d		s	

Due chiamate ricorsive:

```
quicksort(A, 0, 1) ;  
quicksort(A, 3, 4) ;
```



```
quicksort(A, 0, 1) ;
```

Inf=0 , Sup=1, perno=2

0	1	2	3	4
2	1	3	5	7
s	d			

while

0	1	2	3	4
2	1	3	5	7
s	d			

Scambio, poi s++, d--

0	1	2	3	4
1	2	3	5	7
d	s			

s>d: fine del while,
inf=d, sup=s:
nessuna chiamata ricorsiva

```
quicksort(A, 3, 4);
```

```
Inf=3 , Sup=4, perno=5
```

0	1	2	3	4
1	2	3	5	7
			s	d

while

0	1	2	3	4
1	2	3	5	7
			s,d	

Scambio, poi s++, d--

0	1	2	3	4
1	2	3	5	7
		d		s

s>d: fine del while,
inf>d, sup=s:
nessuna chiamata ricorsiva

QuickSort

quicksort([3,5,2,1,1], 0, 4)

quicksort([1,1,2,5,3], 0, 1)

quicksort([1,1,2,5,3], 3, 4)

Quicksort

$$T(1) = a$$

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

Se $k=1$:

$$T(1) = a$$

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

$$O(n^2)$$

Se $k=n/2$:

$$T(1) = a$$

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

soluzione

$$T(1) = a$$

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

$$T(1) = a$$

$$T(2) = 2b + 2a$$

$$T(4) = 4b + 4b + 4a$$

$$T(8) = 8b + 8b + 8b + 8a = 3(8b) + 8a$$

$$T(16) = 16b + 16b + 16b + + 16b + 16a = 4(16b) + 16a$$

.

.

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

$T(n)$ è $O(n \log n)$

quicksort

La complessità nel **caso medio** è uguale a quella nel caso migliore: $O(n \log n)$ (ma con una costante nascosta maggiore). Questo se tutti i possibili contenuti dell'array in input (tutte le permutazioni degli elementi) sono equiprobabili.

Per ottenere questo risultato indipendentemente dal particolare input, ci sono versioni "**randomizzate**" di quicksort in cui il perno ad ogni chiamata è scelto in modo casuale.

Ricerca in un insieme

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

$$T(0) = a$$

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

$$O(n)$$

Ricerca in un insieme

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

$$T(0) = a$$

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

soluzione

$$T(0) = a$$

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

$$T(0) = a$$

$$T(1) = b + a$$

$$T(2) = b + b + a$$

$$T(4) = b + b + b + a$$

·

·

$$T(n) = (\log n + 1) b + a$$

$T(n)$ è $O(\log n)$

Ricerca in un insieme

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

$$T(0) = a$$

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

soluzione

$$T(0) = a$$

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

$$T(0) = a$$

$$T(1) = b + 2a$$

$$T(2) = b + 2b + 4a = 3b + 4a$$

$$T(4) = b + 6b + 8a = 7b + 8a$$

.

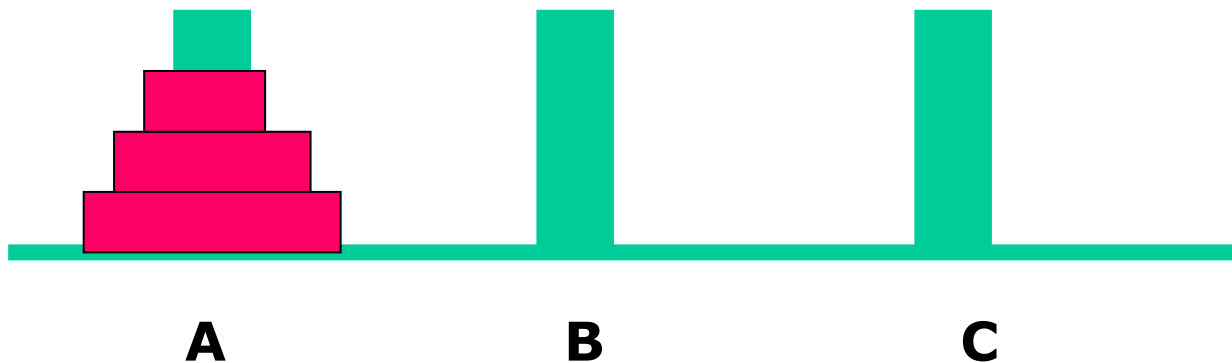
.

$$T(n) = (2^n - 1)b + 2^n a$$

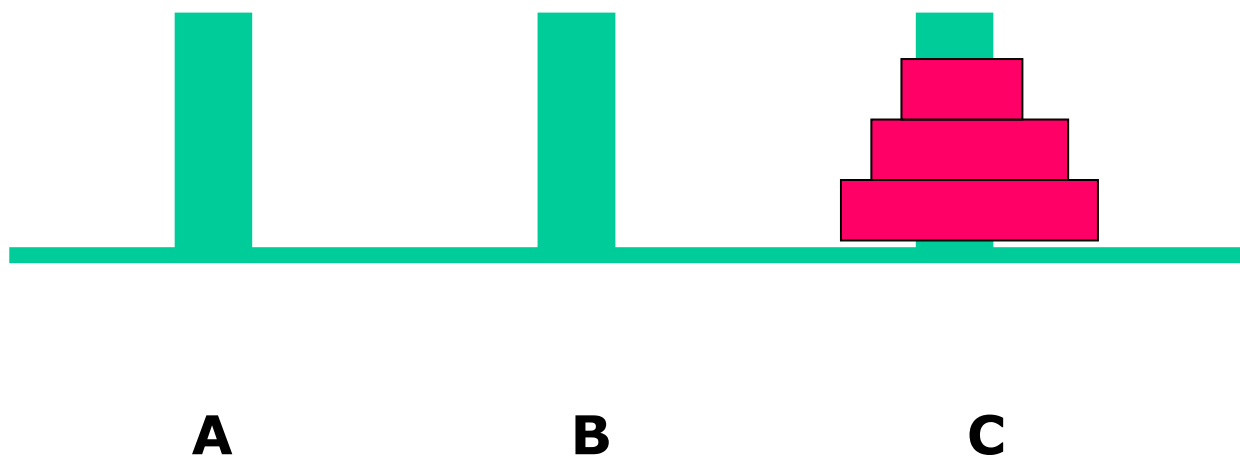
$T(n)$ è $O(n)$

Torre di Hanoi

- **3 paletti**
- **bisogna spostare la torre di n cerchi dal paletto sorgente A a quello destinatario C usando un paletto ausiliario B**
- **Un cerchio alla volta**
- **Mai un cerchio sopra uno piu' piccolo**



Torre di Hanoi



Torre di Hanoi

void **trasferisci** una torre di n cerchi da A a C {

Se $n=1$ **sposta** il cerchio dal A a C;

altrimenti

{ **trasferisci** la torre degli $n-1$ cerchi più piccoli da A a B
usando C come paletto ausiliario;

sposta il cerchio più grande dal A a C;

trasferisci la torre degli $n-1$ cerchi più piccoli da B a C
usando A come paletto ausiliario;

} }

Torre di Hanoi

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

$$T(1) = a$$

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

hanoi(3, A, B, C)

hanoi(2, A, C, B)

hanoi(1, A, B, C)

sposta(A, C);

sposta(A, B);

hanoi(1, C, A, B)

sposta(C, B);

sposta(A,C);

hanoi(2, B, A, C)

hanoi(1, B, C, A)

sposta(B, A);

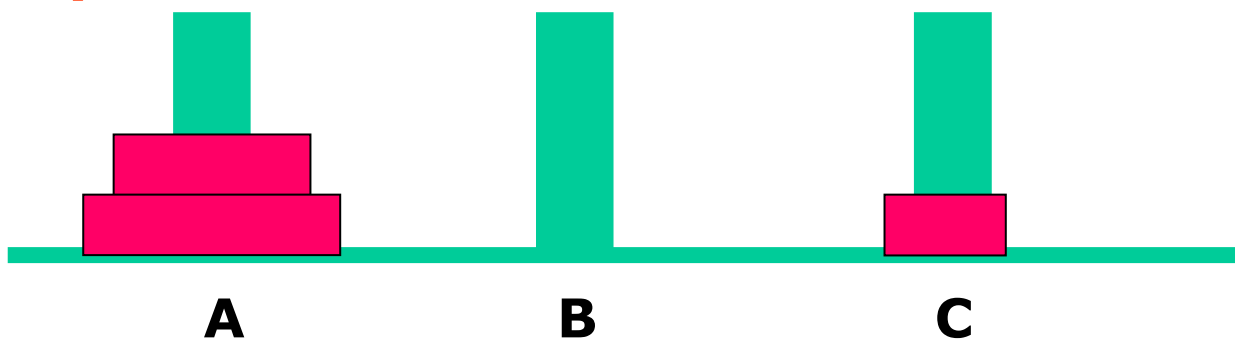
sposta(B, C);

hanoi(1, A, B,C)

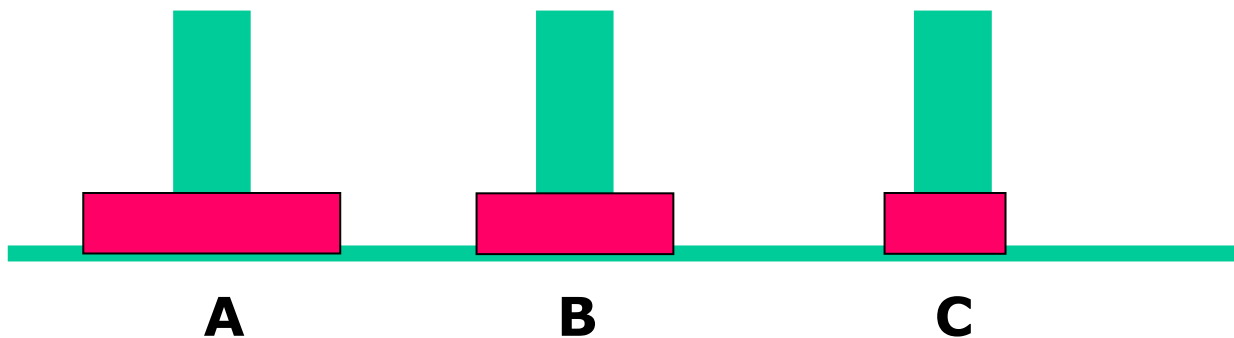
sposta(A,C);



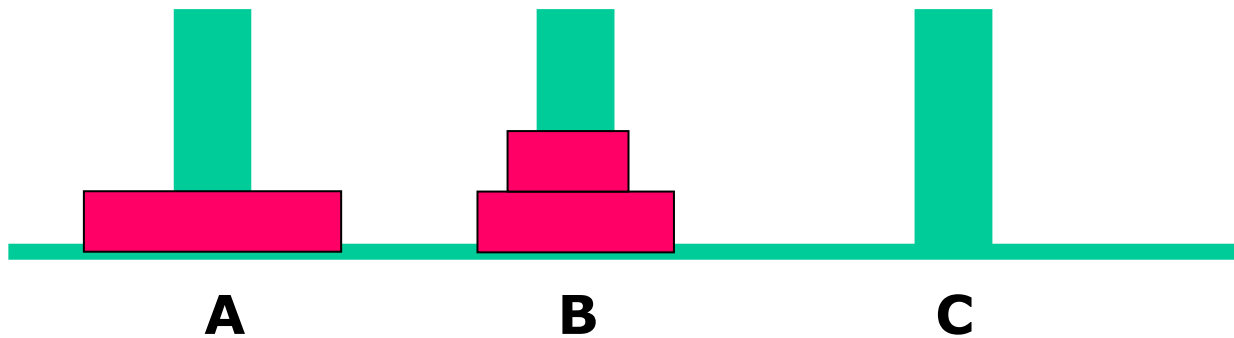
sposta(A, C);



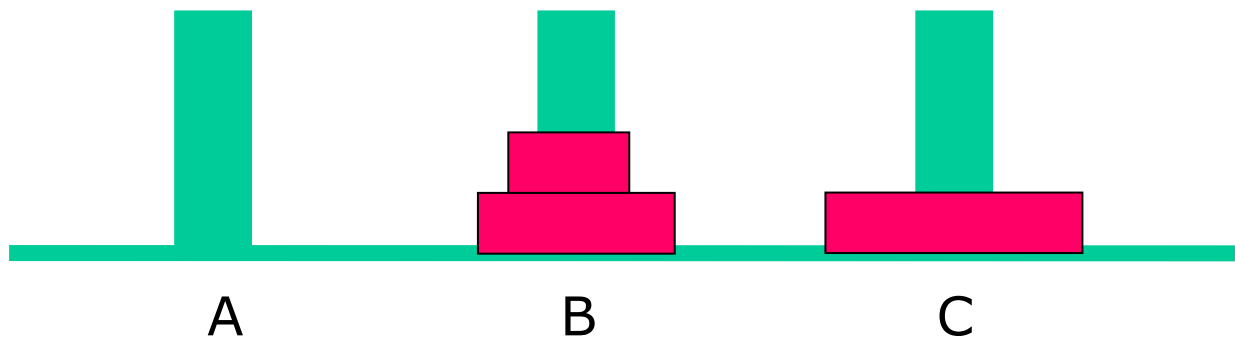
sposta(A, B);



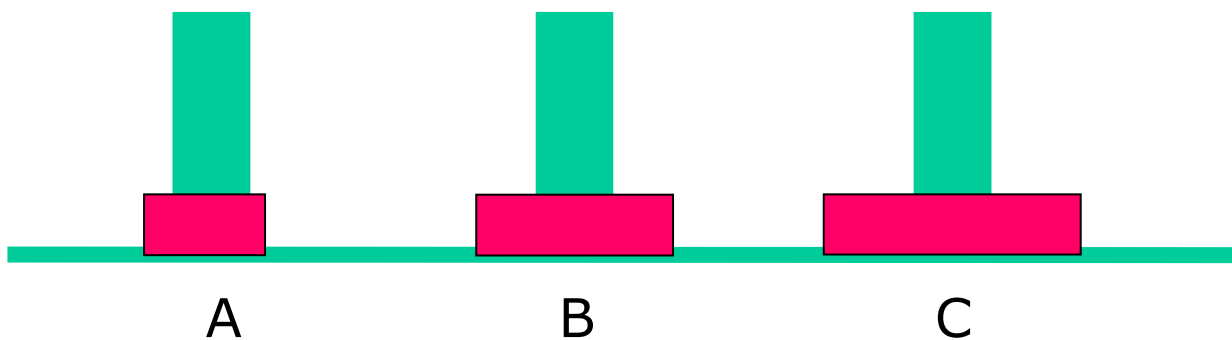
sposta(C, B);



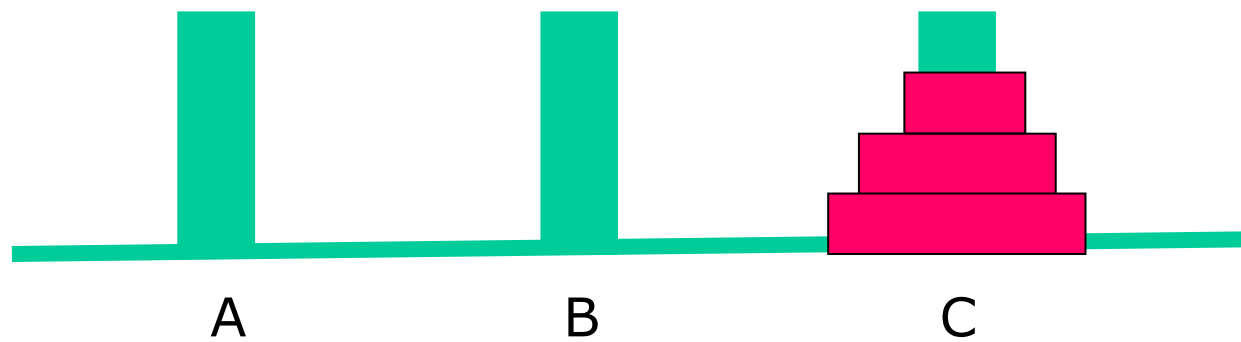
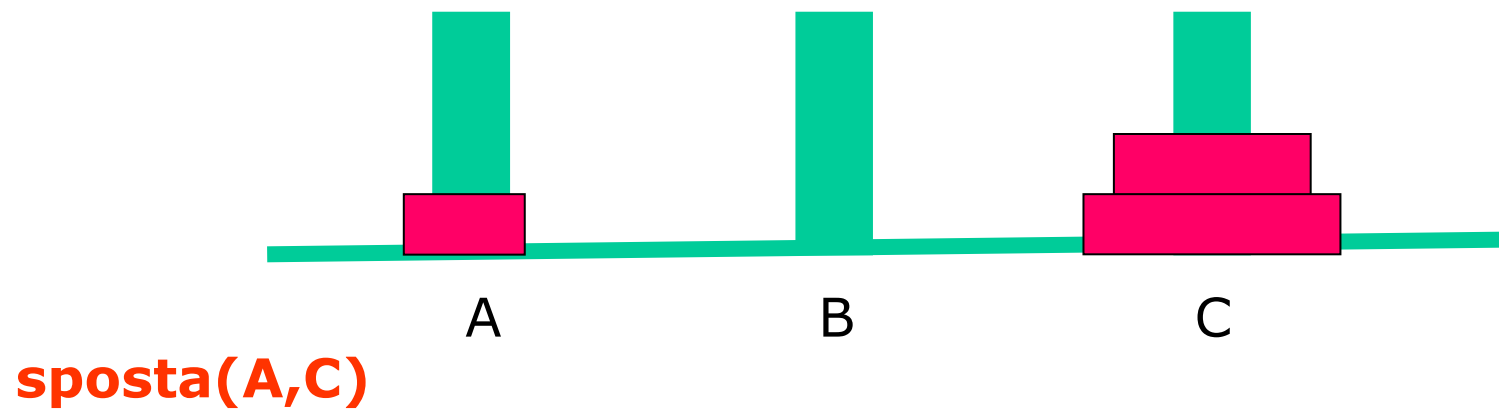
sposta(A,C)



sposta(B,A)



sposta(B,C)



soluzione

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

$$T(1) = a$$

$$T(2) = b + 2a$$

$$T(3) = b + 2b + 4a = 3b + 4a$$

$$T(4) = 7b + 8a$$

·

·

$$T(n) = (2^{n-1} - 1)b + 2^{n-1}a$$

$T(n)$ è $O(2^n)$

Metodo divide et impera

```
void dividetimpera( S ) {  
  
    if ( |S| <= m )  
        <risolvi direttamente il problema>;  
    else {  
        <dividi S in b sottoinsiemi  $S_1.. S_b$ >;  
        dividetimpera( $S_{i1}$  );  
        ...  
        dividetimpera( $S_{ia}$  );  
  
        <combina i risultati ottenuti>;  
    }  
}
```

Metodo divide et impera

$$T(0) = d$$

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

$$O(\log n)$$

$$T(0) = d$$

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

$$O(n)$$

$$T(0) = d$$

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

$$O(n \log n)$$

Metodo divide et impera

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

$$T(n) = hn^k + aT(n/b) \quad \text{se } n > m$$

$$h > 0$$

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

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

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

La complessità è calcolata prendendo come misura il **numero di cifre che compongono il numero**

Ad esempio:

- **L'addizione ha complessità $O(n)$**
- **la moltiplicazione che studiamo alle elementari ha complessità $O(n^2)$**

Moltiplicazione veloce fra interi non negativi

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

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

$$A=1325 = 13*10^2 + 25 \quad n=4$$

$n/2$	$n/2$
A_s	A_d

13	25
A_s	A_d

$$AB = A_s B_s 10^n + (A_s B_d + A_d B_s) 10^{n/2} + A_d B_d$$

$$(A_s + A_d)(B_s + B_d) = A_s B_d + A_d B_s + A_s B_s + A_d B_d$$

$$A_s B_d + A_d B_s = (A_s + A_d)(B_s + B_d) - A_s B_s - A_d B_d$$

$$AB = A_s B_s 10^n + ((A_s + A_d)(B_s + B_d) - A_s B_s - A_d B_d) 10^{n/2} + A_d B_d$$

$$AB = A_s B_s 10^n + ((A_s + A_d)(B_s + B_d) - A_s B_s - A_d B_d) 10^{n/2} + A_d B_d$$

numero mult (numero A, numero B, int n) {

if (n=1) return A * B; **O(1)**

else {

A_s = parte sinistra di A ; A_d = parte destra di A ; **O(n/2)**

B_s = parte sinistra di B ; B_d = parte destra di B ; **O(n/2)**

int x1= $A_s + A_d$; int x2= $B_s + B_d$; **O(n/2)** (somma di due numeri)

int **y1**= mult (x1, x2 ,n/2);

int **y2**= mult (A_s , B_s , n/2);

int **y3**= mult (A_d , B_d ,n/2);

int z1= left_shift(y2,n); **O(n)**

int z2= left_shift (y1-y2-y3, n/2); **O(n/2)**

return z1+z2+y3;

}

left_shift(h,n) scorre h
di n posti a sinistra
facendo entrare n 0
(*10ⁿ)

Moltiplicazione veloce

$$T(1) = d$$

$$T(n) = bn + 3T(n/2)$$

$$T(n) \in O(n^{\log_2 3})$$

$$T(n) \in O(n^{1.59})$$

Relazioni lineari

$$T(0) = d$$

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

$$O(n)$$

$$T(1) = a$$

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

$$O(n^2)$$

$$T(0) = d$$

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

$$O(2^n)$$

Relazioni lineari

$$T(0) = d$$

$$T(n) = bn^k + a_1T(n-1) + a_2T(n-2) + \dots + a_rT(n-r)$$

polinomiale solo se esiste al più un solo $a_i = 1$ e gli altri a_i sono tutti 0 (c'è una sola chiamata ricorsiva). Negli altri casi sempre **esponenziale.**

Soluzione di una classe di relazioni lineari

$$T(0) = d$$

$$T(n) = bn^k + T(n-1)$$

$$b > 0$$

$$T(n) \in O(n^{k+1})$$

Serie di Fibonacci

$$f_0 = 0$$

$$f_1 = 1$$





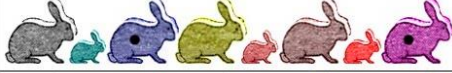


$$f_n = f_{n-1} + f_{n-2}$$

Ogni numero è uguale alla somma dei due precedenti

0, 1, 1, 2, 3, 5, 8, 13, 21, 34,

Serie di Fibonacci

LIBER ABBACI di Leonardo Fibonacci (1200 circa)

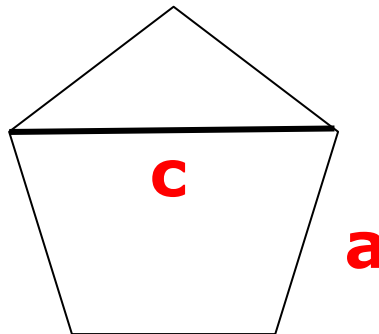
Mesi	Coppie nate	Coppie adulte *	Totale coppie	Evoluzione nascite nei primi sei mesi
Inizio	0	1	1	
1°	1	1	2	
2°	1	2	3	
3°	2	3	5	
4°	3	5	8	
5°	5	8	13	
6°	8	13	21	

Sezione aurea

Sezione aurea: limite del rapporto fra ogni numero della serie e il precedente

$$\phi = \lim_{n \rightarrow \infty} \frac{f_n}{f_{n-1}} = \frac{1 + \sqrt{5}}{2} = 1,61803 \dots$$

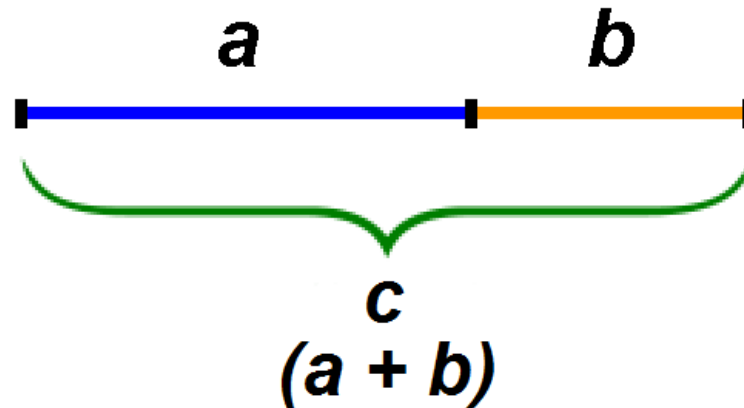
Φ è il rapporto fra la diagonale c e il lato a del pentagono



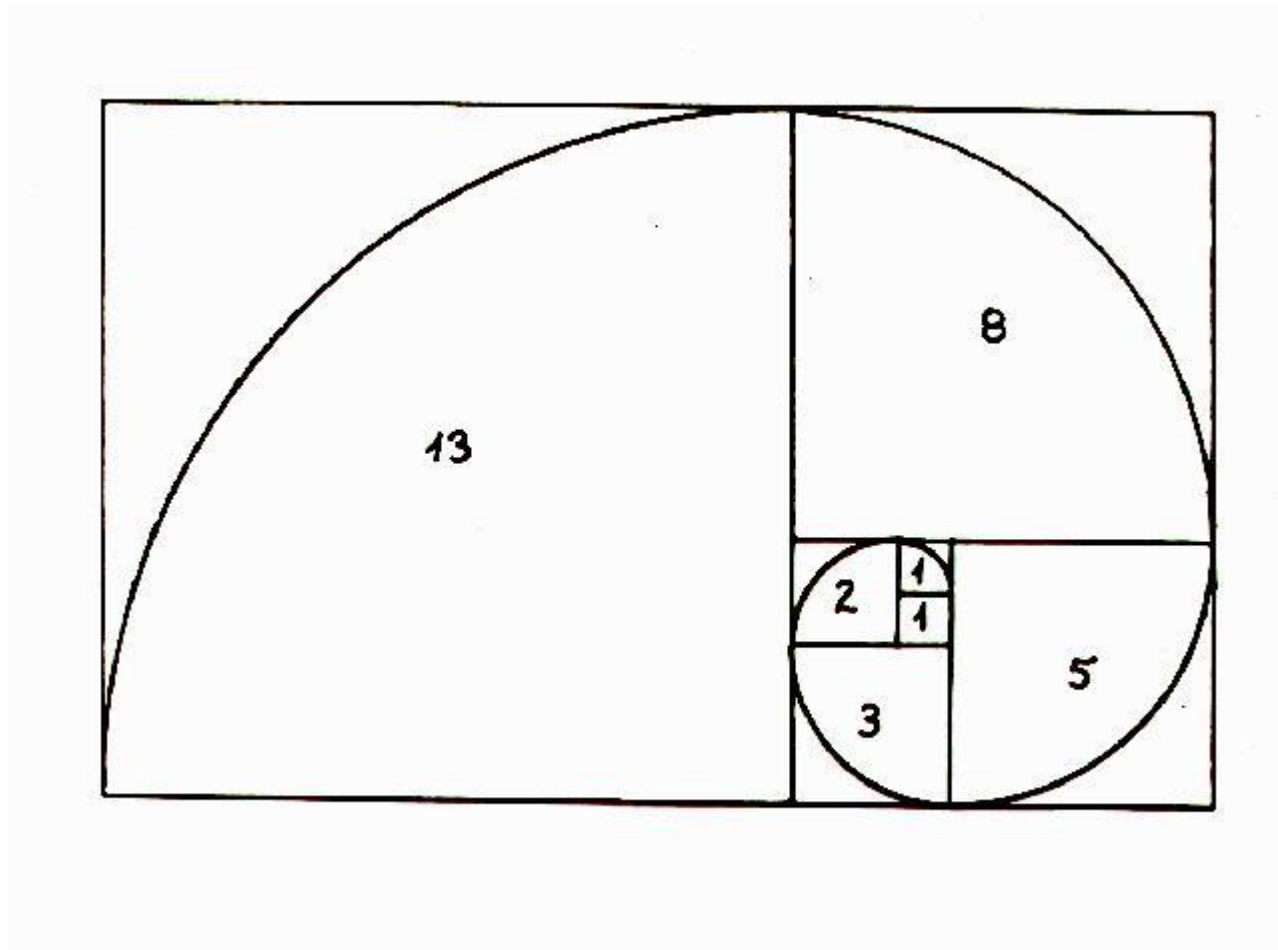
Sezione aurea

$$\frac{c=a+b}{a} = \frac{a}{b} = \varphi$$

a è medio proporzionale fra **c=a+b** e **b** ($a > b$)



spirale logaritmica



Serie di Fibonacci

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

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

$$T(0) = T(1) = d$$

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

$$T(n) \in O(2^n)$$

Serie di Fibonacci

```
int fibonacci(int n) {  
    int k; int j=0; int f=1;  
    for (int i=1; i<=n; i++) {  
        k=j; j=f; f=k+j;  
    }  
    return j;  
}
```

$$T(n) \in O(n)$$

Serie di Fibonacci

```
int fibonacci( int n, int a = 0, int b = 1 ) {  
    if (n == 0) return a;  
    return fibonacci( n-1, b, a+b );  
}
```

$$T(0) = d$$

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

$$T(n) \in O(n)$$

Mergesort

```
void mergeSort( sequenza S ) {  
    if ( |S| <= 1 )  
        return;  
    else {  
        < dividi S in 2 sottosequenze S1 e S2 di uguale  
            lunghezza >;  
        mergeSort(S1 );  
        mergeSort(S2 );  
        < fondi S1 e S2 >;  
    }  
}
```

mergesort

5	8	1	9	2	4	10	7
---	---	---	---	---	---	----	---

Dividi in due parti uguali

5	8	1	9
---	---	---	---

2	4	10	7
---	---	----	---

ordina la prima parte con
mergesort *

1	5	8	9
---	---	---	---

ordina la seconda parte
con **mergesort**

2	4	7	10
---	---	---	----

fondi

1	2	4	5	7	8	9	10
---	---	---	---	---	---	---	----

mergesort *

5	8	1	9
---	---	---	---

Dividi in due parti uguali

5	8
---	---

1	9
---	---

ordina la prima parte con
mergesort *

ordina la seconda parte
con **mergesort**

5	8
---	---

1	9
---	---

fondi

1	5	8	9
---	---	---	---

mergesort *



Dividi in due parti uguali

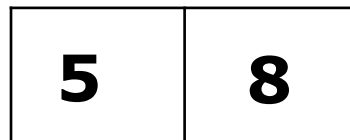


ordina la prima parte con
mergesort

ordina la seconda parte
con **mergesort**



fondi



Mergesort su liste

```
void mergeSort(Elem*& s1) {  
    if (s1 == NULL || s1->next == NULL) return;  
    Elem* s2 = NULL;  
  
    split (s1, s2);  
  
    mergeSort (s1);  
  
    mergeSort (s2);  
  
    merge (s1, s2);  
}
```

$$T(0) = T(1) = d$$

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

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

Mergesort : split

```
void split (Elem* & s1, Elem* & s2) {  
    if (s1 == NULL || s1->next == NULL)  
        return;  
    Elem* p = s1->next;  
    s1->next = p->next;  
    p->next = s2;  
    s2 = p;  
    split (s1->next, s2);  
}
```

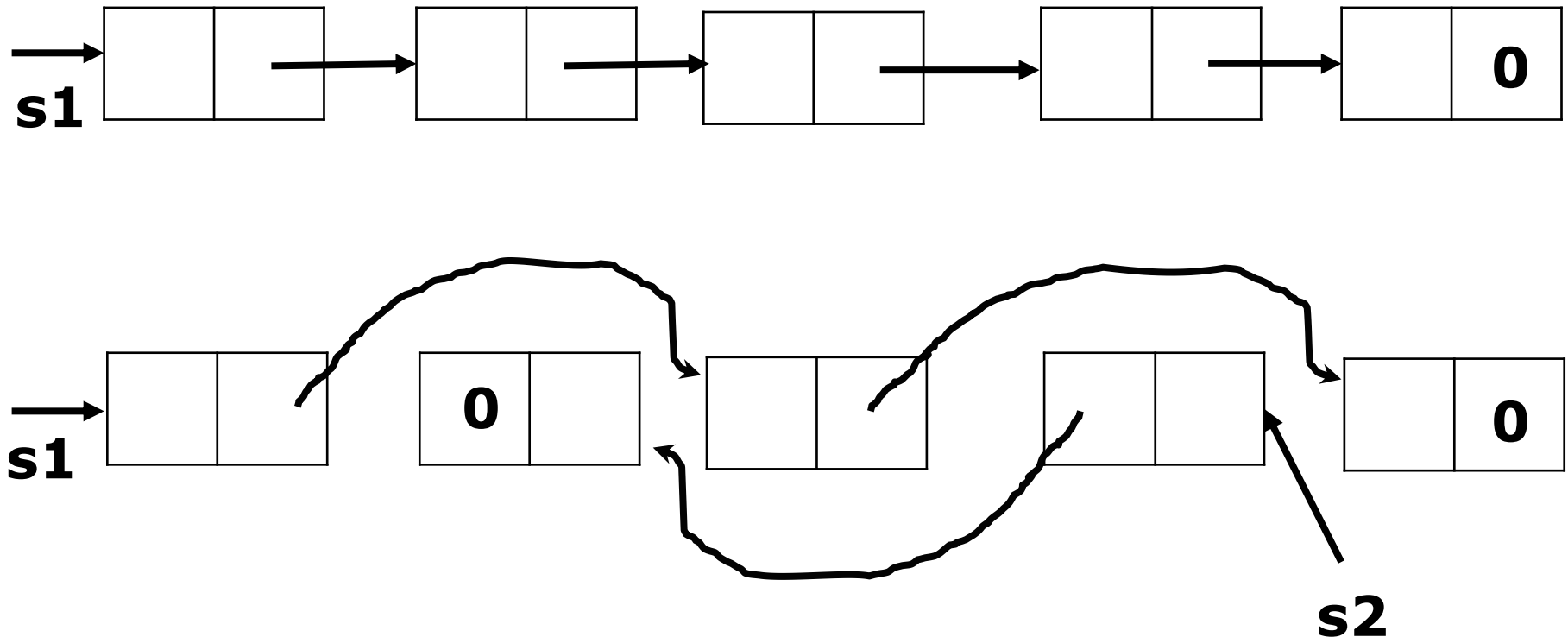
$$T(0) = T(1) = d$$

$$T(n) = b + T(n-2)$$

$$T(n) \in O(n)$$

Mergesort : split

Divide la lista in due liste mettendo gli elementi di posizione pari in una e quelli di posizione dispari nell'altra (rovesciati)



Mergesort : merge

```
void merge (Elem* & s1, Elem* s2) {  
    if (s2 == NULL)  
        return;  
    if (s1 == NULL) {  
        s1 = s2;  
        return;  
    }  
    if (s1->inf <= s2->inf)  
        merge (s1-> next, s2);  
  
    else {  
        merge (s2-> next, s1);  
        s1 = s2;  
    }  
}
```

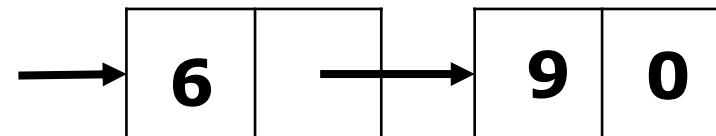
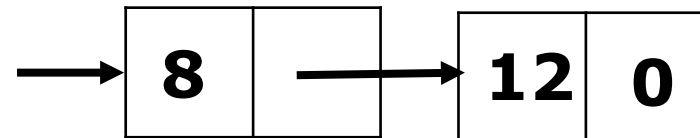
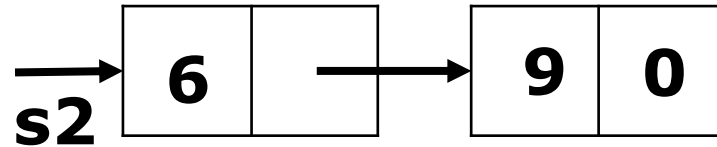
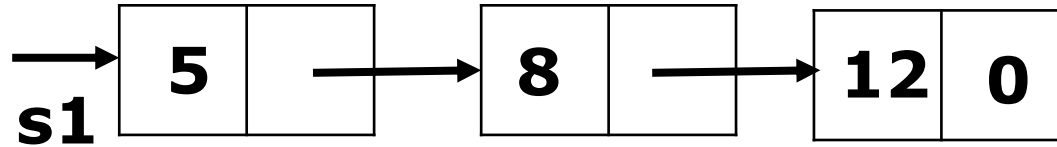
$$T(0) = d$$

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

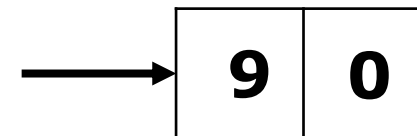
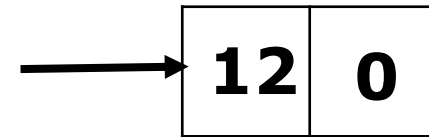
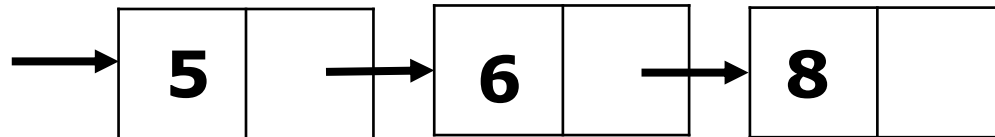
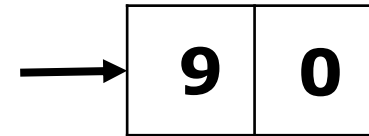
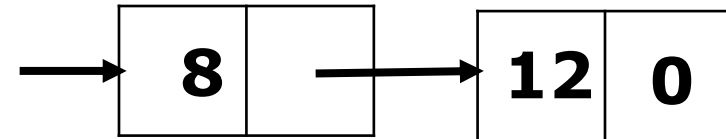
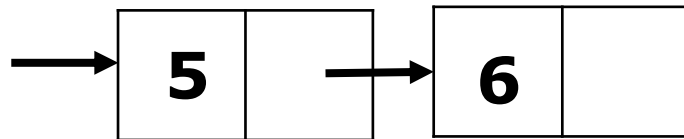
$$T(n) \in O(n)$$

Scorre in parallelo le due liste confrontando i loro primi elementi e scegliendo ogni volta il minore fra i due

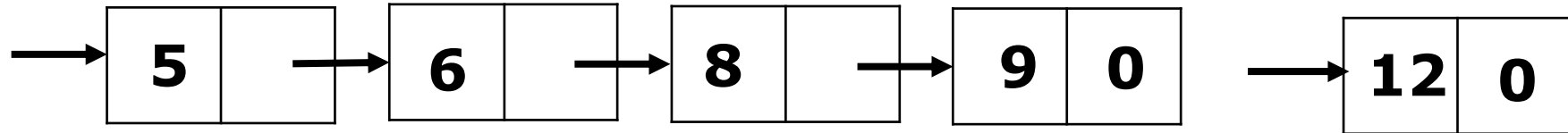
merge



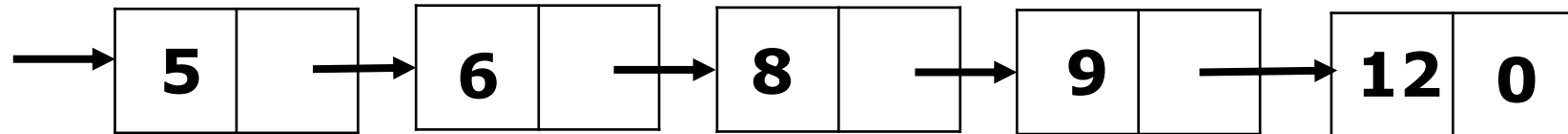
merge



merge



Lista vuota



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

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

mergeSort([2,3])

dividi([2,3])

mergeSort([2])

mergeSort([3])

fondi([2] , [3])

mergeSort([5,1])

dividi([5,1])

mergeSort([5])

mergeSort([1])

fondi([5] , [1])

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

Esempio mergesort

[2]

[3]

[2,3]

[5]

[1]

[1,5]

[1,2,3,5]

Alberi

Alberi binari

- **NULL** è un albero binario;
- un nodo **p** più **due alberi binari** **Bs** e **Bd** forma un albero binario

p è **radice**

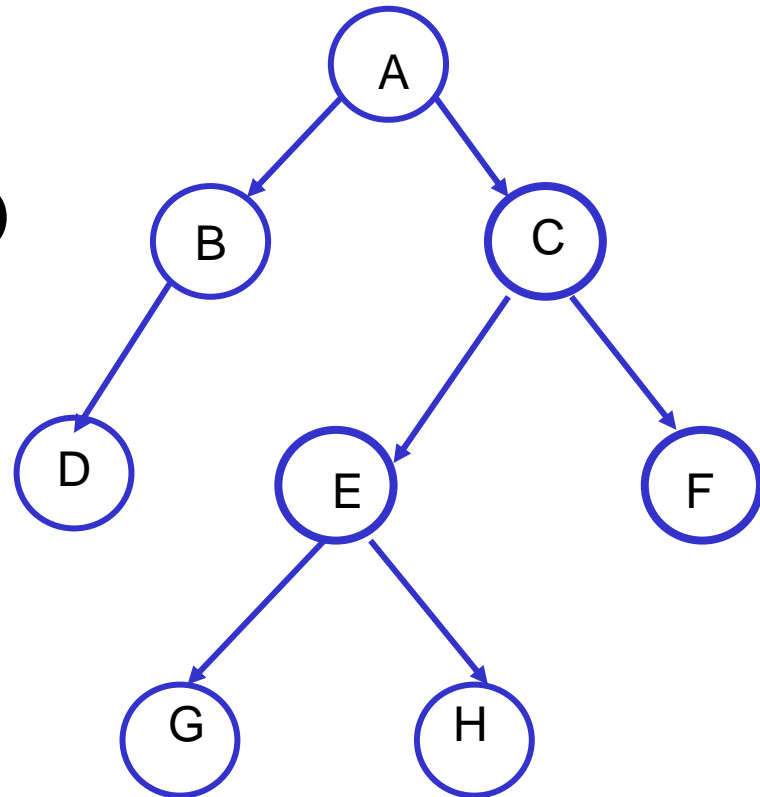
Bs è il **sottoalbero sinistro** di **p**

Bd il **sottoalbero destro** di **p**

alberi **etichettati**

Alberi binari

- **padre**
- **figlio sinistro (figlio destro)**
- **antecedente**
- **foglia**
- **discendente**
- **livello di un nodo**
- **livello dell'albero**



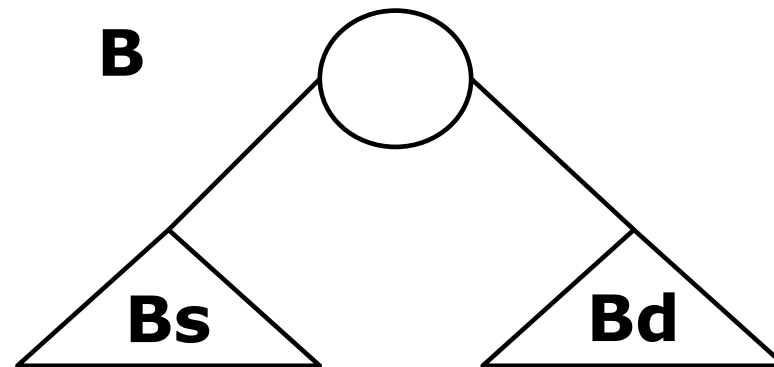
caso base

albero vuoto (NULL)

caso ricorsivo

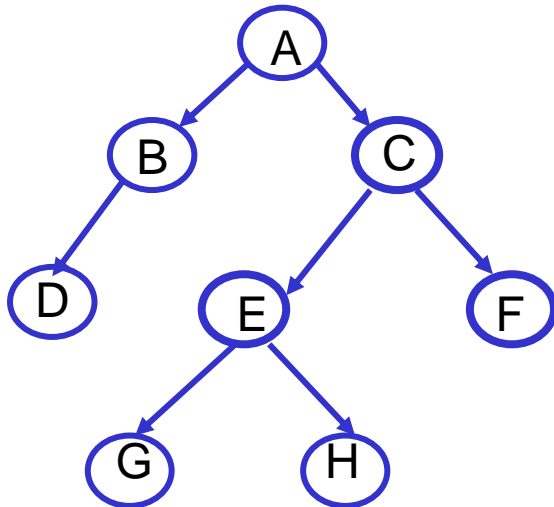
radice + due sottoalberi

B = vuoto



Visita anticipata (preorder)

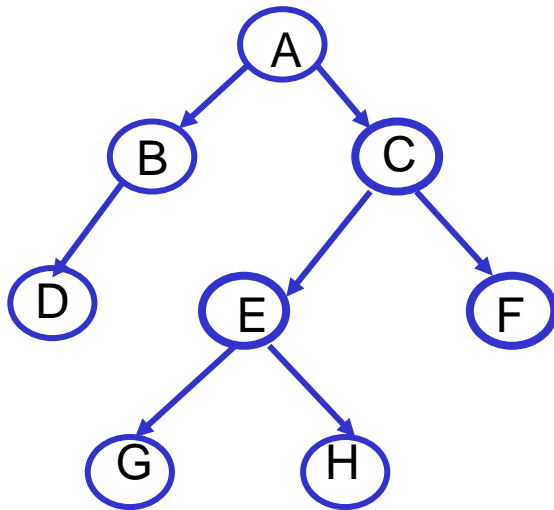
```
void preOrder ( albero ) {  
    se l'albero e' vuoto termina;  
    altrimenti {  
        esamina la radice;  
        preOrder ( sottoalbero sinistro);  
        preOrder ( sottoalbero destro);  
    }  
}
```



A B D C E G H F

Visita differita (postorder)

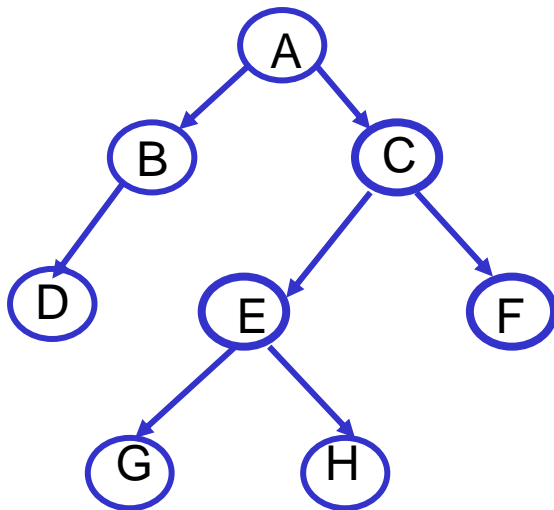
```
void postOrder ( albero ) {  
    se l'albero e' vuoto termina;  
    altrimenti {  
        postOrder ( sottoalbero sinistro);  
        postOrder ( sottoalbero destro);  
        esamina la radice;  
    }  
}
```



D B G H E F C A

Visita simmetrica (inorder)

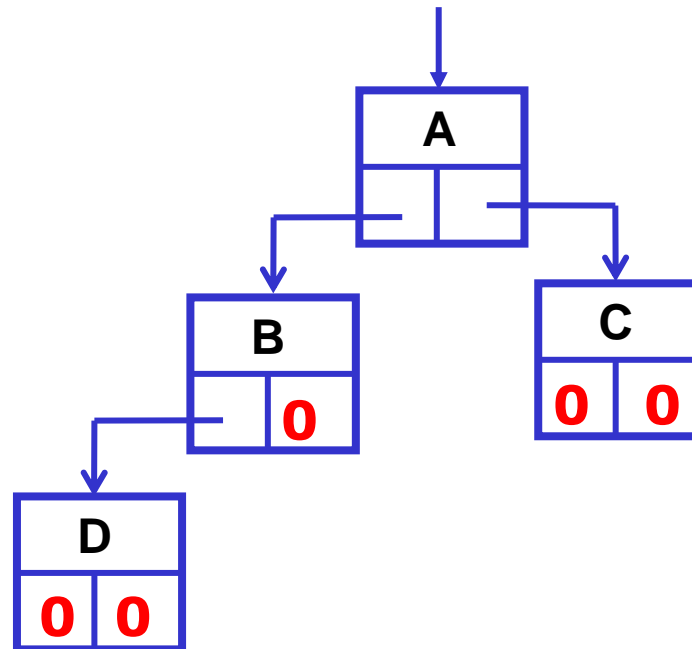
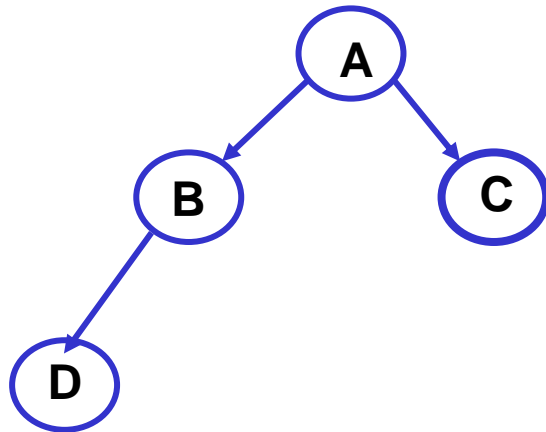
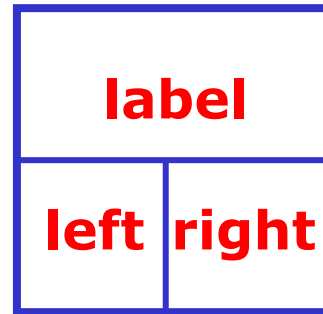
```
void inOrder ( albero ) {  
    se l'albero e' vuoto termina;  
    altrimenti {  
        inOrder ( sottoalbero sinistro);  
        esamina la radice;  
        inOrder ( sottoalbero destro);  
    }  
}
```



D B A G E H C F

Memorizzazione in lista multipla

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



```
void preOrder(Node* tree)
{  if (!tree) return;
   else {
       <esamina tree->label>;
       preOrder(tree->left);
       preOrder(tree->right);
   }
}
```

```
void preOrder(Node* tree) {
    if (!tree) return;
    else {
        cout << tree->label;
        preOrder(tree->left);
        preOrder(tree->right);
    }
}
```

Visite in C++

```
void postOrder(Node* tree) {  
    if (!tree) return;  
    else {  
        postOrder(tree->left);  
        postOrder(tree->right);  
        <esamina tree->label>;  
    }  
}
```

```
void inOrder(Node* tree) {  
    if (!tree) return;  
    else {  
        inOrder(tree->left);  
        <esamina tree->label>;  
        inOrder(tree->right);  
    }  
}
```

Complessità delle visite

Complessità in funzione del numero di nodi:

$$T(0) = a$$

$$T(n) = b + T(n_s) + T(n_d) \quad \text{con } n_s + n_d = n - 1 \quad n > 0$$

Caso particolare:

$$T(0) = a$$

$$T(n) = b + 2T((n-1)/2)$$

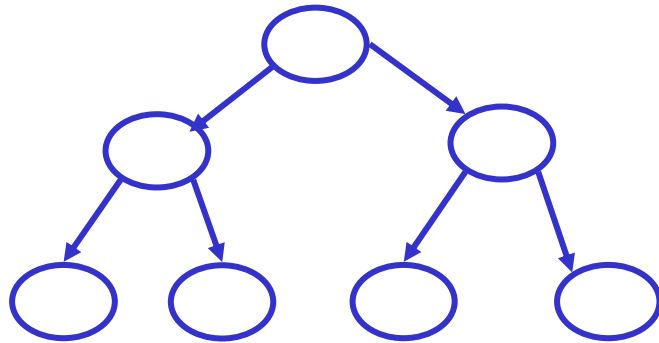
$$T(n) \in O(n)$$

Visita iterativa

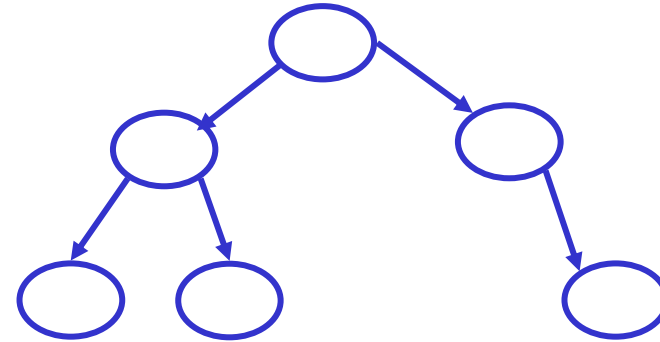
```
void preOrder(Node* tree) {  
    stack<Node*> miapila(100);  
    for (;;) {  
        while (tree) {  
            <esamina tree->label>;  
            miapila.push(tree);  
            tree=tree->left;  
        }  
        if (miapila.empty()) return;  
        tree=miapila.pop()->right;  
 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

Alberi binari bilanciati

i nodi di tutti i livelli tranne quelli dell'ultimo hanno due figli



bilanciato

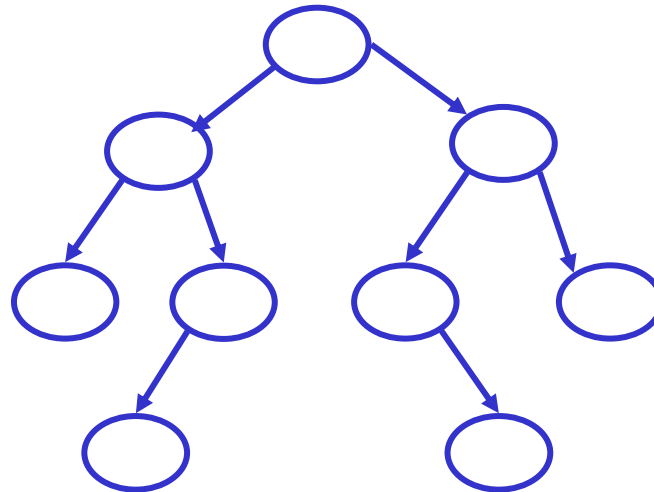


non bilanciato

Un albero binario bilanciato con livello k ha $2^{(k+1)} - 1$ nodi e 2^k foglie

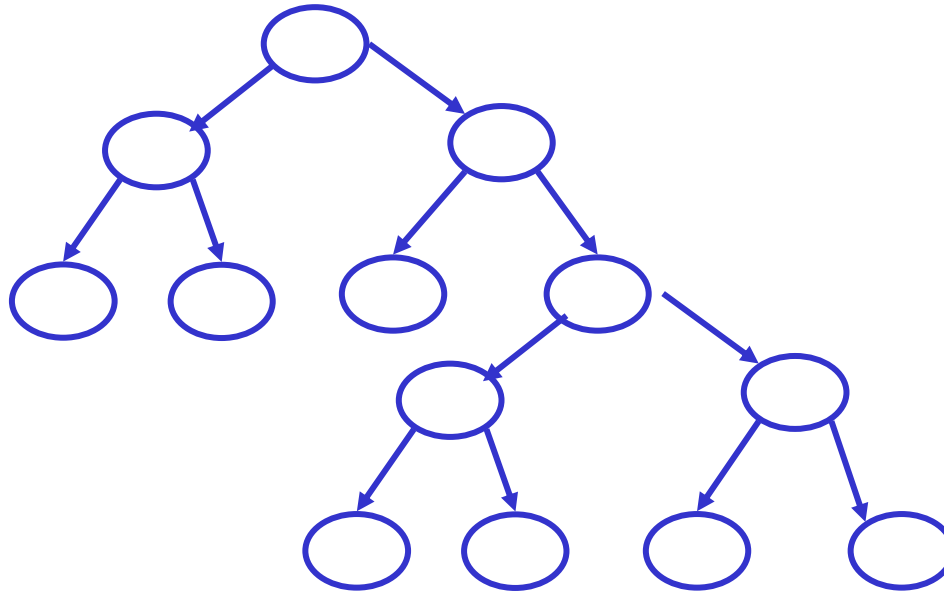
Alberi binari quasi bilanciati

**fino al penultimo livello è un albero bilanciato
(un albero bilanciato è anche quasi bilanciato)**



Alberi pienamente binari

Tutti i nodi tranne le foglie hanno 2 figli



Un albero binario pienamente binario ha tanti nodi interni quante sono le foglie meno 1

Complessità in funzione dei livelli (se l'albero è bilanciato):

$$T(0) = a$$

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

$$T(k) \in O(2^k)$$

Alberi binari: conta i nodi e le foglie

conta i nodi

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

conta le foglie

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

$$T(n) \in O(n)$$

Alberi binari: cerca un'etichetta

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

```
Node* findNode (Infotype n, Node*tree) {  
    if (!tree) return NULL;           //albero vuoto: l'etichetta non c'è  
    if (tree->label==n)                // trovata:ritorna il puntatore  
        return tree;  
    Node* a=findNode(n, tree->left);  // cerca a sinistra  
    if (a) return a;                  // se trovata ritorna il puntatore  
    else return findNode(n, tree->right); // cerca a destra  
}
```

Alberi binari: cancella tutto l'albero

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

alla fine il puntatore deve essere NULL

Alberi binari: inserisci un nodo

inserisce un nodo (**son**) come figlio di **father**, sinistro se **c='l'**, destro se **c='r'**. Ritorna 1 se l'operazione ha successo, 0 altrimenti. Se l'albero è vuoto, inserisce il nodo come radice

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

Alberi binari: inserisci un nodo (cont.)

```
Node* a=findNode(father,tree);    //cerca father
if (!a) return 0;                 //father non c'è
if (c=='l' && !a->left) {         //inserisci come figlio sinistro
    a->left=new Node;
    a->left->label=son;
    a->left->left =a->left->right=NULL;
    return 1;
}
```

Alberi binari: inserisci un nodo (cont.)

```
if (c=='r' && !a->right) {           //inserisci come figlio destro  
    a->right=new Node;  
    a->right->label=son;  
    a->right->left = a->right->right = NULL;  
    return 1;  
}  
return 0;                             //inserimento impossibile  
}
```

Class BinTree

```
template<class InfoType>
class BinTree {
    struct Node {
        InfoType label;
        Node *left, *right;
    };
    Node *root;
    Node* findNode(InfoType, Node*);
    void preOrder(Node*);
    void inOrder(Node*);
    void postOrder(Node*);
    void delTree(Node*&);
    int insertNode(Node*&, InfoType, InfoType, char)
```

Class BinTree

public:

BinTree() { **root** = NULL; };

~BinTree(){ **delTree**(root); };

int find(InfoType x) { return **findNode**(x, root); };

void pre() { **preOrder**(root); };

void post(){ **postOrder**(root); };

void in() { **inOrder**(root); };

int insert(InfoType son, InfoType father, char c) {
 insertNode(root,son, father,c);

};

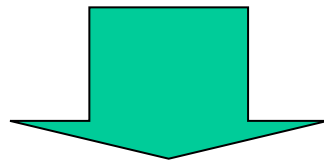
};

L'ordinamento e' basato sulla struttura

Base. P vale l'albero vuoto

Passo induttivo. Per un albero non vuoto B è vero
che:

Se P vale per B_s e per B_d allora vale per B



P vale per B

esempio

P: in ogni albero binario il numero dei sottoalberi vuoti è uguale al numero dei nodi +1

Base. Vero per l'albero vuoto: $\text{Nodi}=0$, $\text{Vuoti}=1$

Passo induttivo.

Ipotesi: $\text{Vuoti}_s = \text{Nodi}_s + 1$, $\text{Vuoti}_d = \text{Nodi}_d + 1$

Tesi: $\text{Vuoti}_B = \text{Nodi}_B + 1$

Dim. $\text{Nodi}_B = \text{Nodi}_s + \text{Nodi}_d + 1$

$\text{Vuoti}_B = \text{Vuoti}_s + \text{Vuoti}_d$

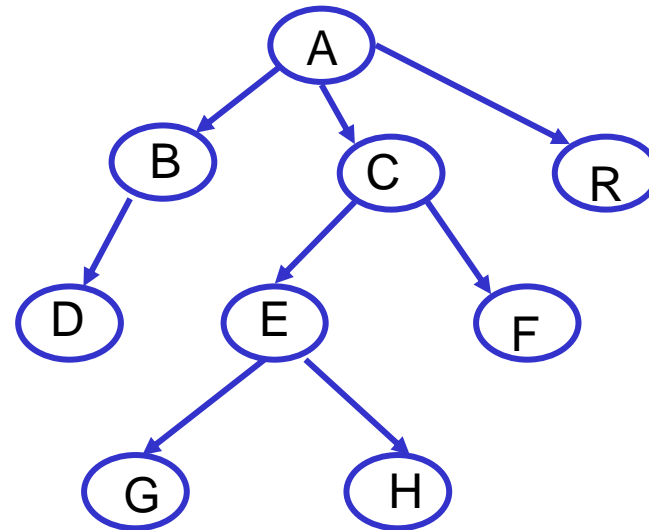
Usandi l'ip. induttiva:

$\text{Vuoti}_B = \text{Nodi}_s + 1 + \text{Nodi}_d + 1 = \text{Nodi}_B + 1$

Alberi generici: definizione

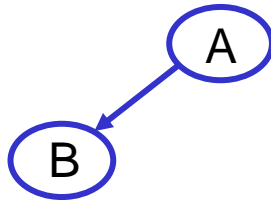
- un nodo p è un **albero**
- un nodo + una **sequenza di alberi** $A_1 \dots A_n$ è un albero

- radice
- padre
- i -esimo sottoalbero
- i -esimo figlio
- livello

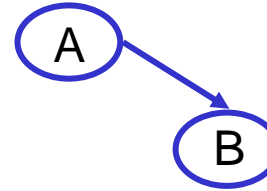


Alberi generici: differenza con alberi binari

alberi binari



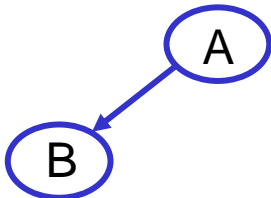
diverso da



sottoalbero **destro** vuoto

sottoalbero **sinistro** vuoto

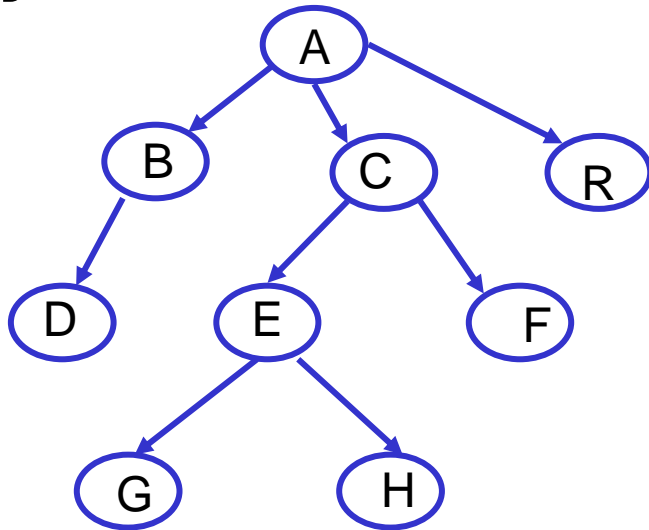
alberi generici



unico albero: radice: A, un sottoalbero

Alberi generici: visite

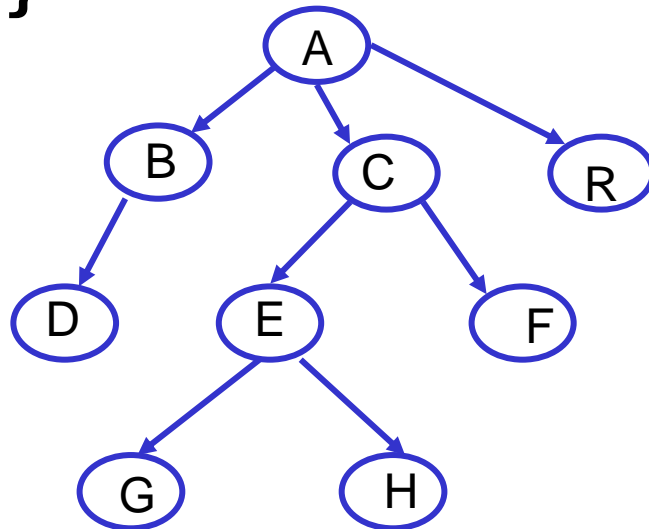
```
void preOrder ( albero ) {  
    esamina la radice;  
    se l'albero ha n sottoalberi {  
        preOrder ( primo sottoalbero);  
        ...  
        preOrder ( n-esimo sottoalbero);  
    }  
}
```



A B D C E G H F R

Alberi generici: visite

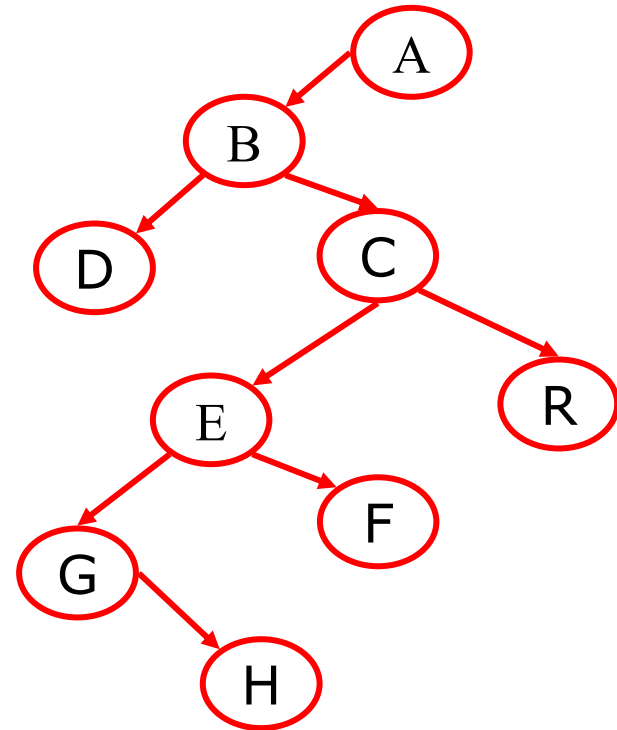
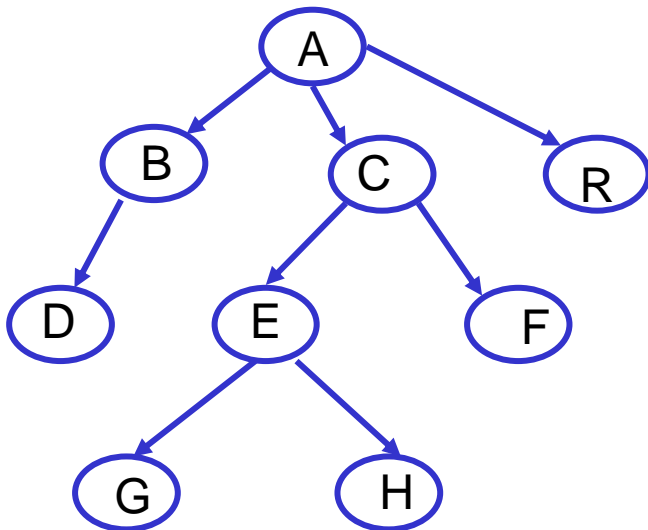
```
void postOrder ( albero ) {  
    se l'albero ha n sottoalberi {  
        postOrder ( primo sottoalbero);  
        ...  
        postOrder ( n-esimo sottoalbero);  
        esamina la radice;  
    }  
}
```



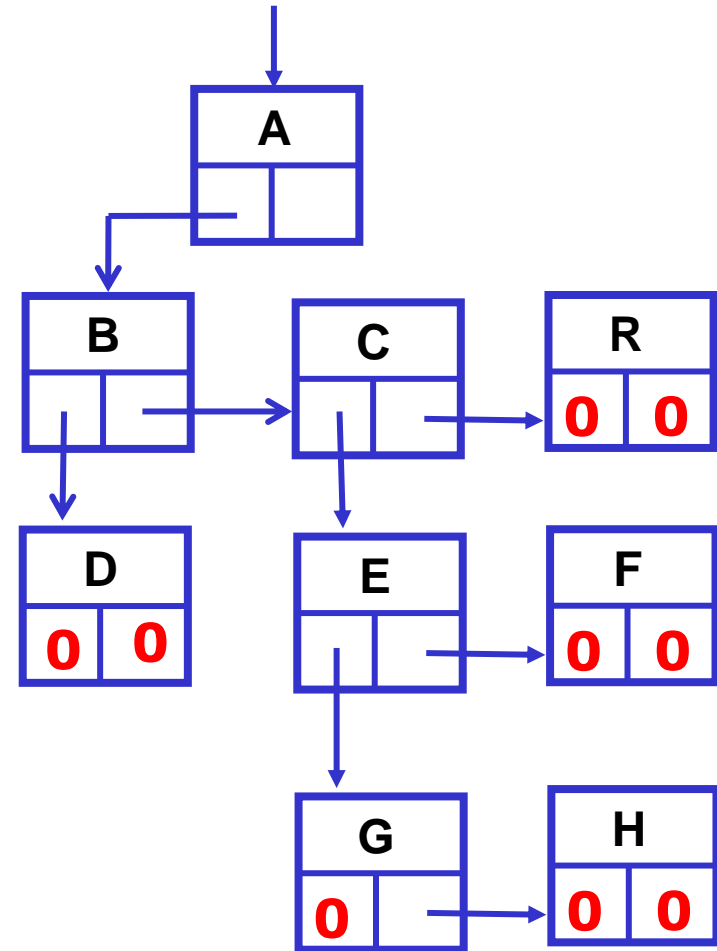
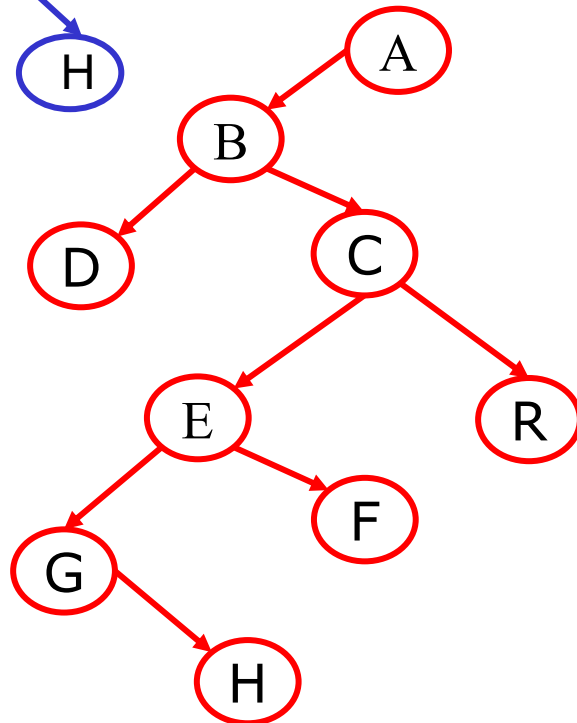
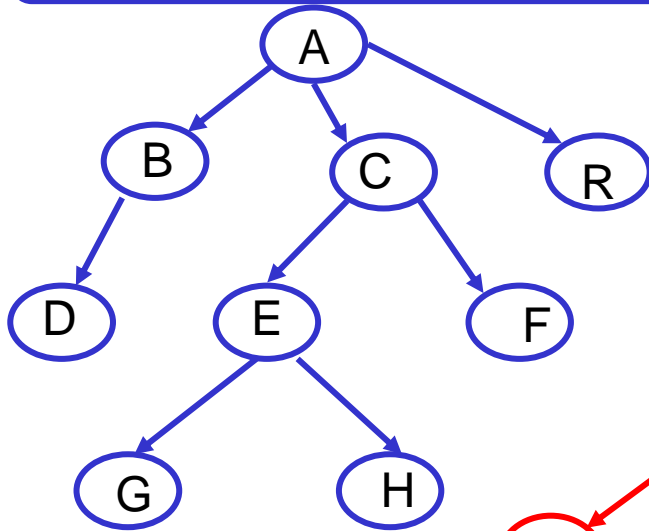
D B G H E F C R A

MEMORIZZAZIONE FIGLIO-FRATELLO

- **primo figlio a sinistra**
- **primo fratello a destra**



Alberi generici: memorizzazione



Utilizzando la memorizzazione figlio-fratello:

la visita **preorder** del trasformato corrisponde
alla visita **preorder** dell'albero generico

la visita **inorder** del trasformato corrisponde alla
visita **postorder** dell'albero generico

Esempi di programmi su alberi generici: conta nodi e foglie

conta i nodi (vedi albero binario)

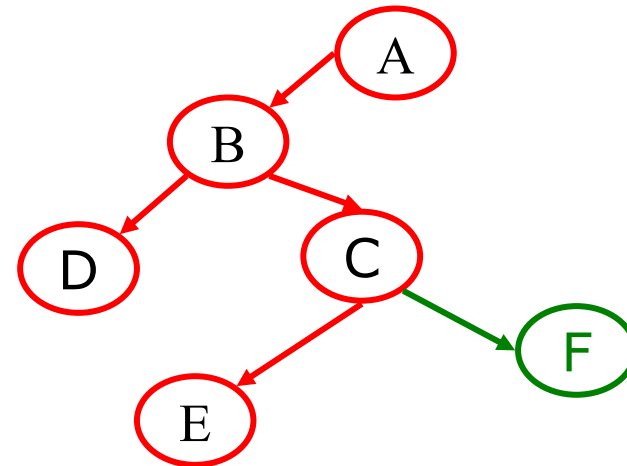
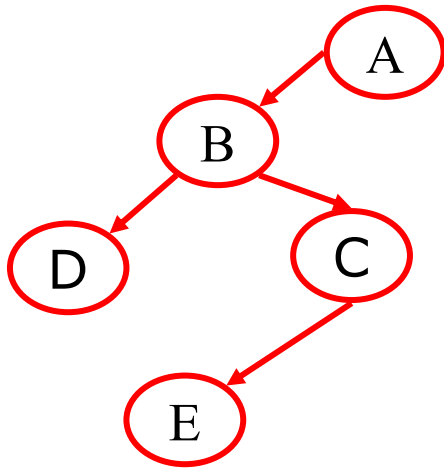
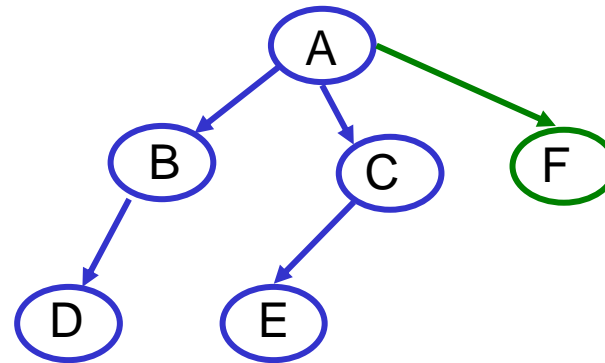
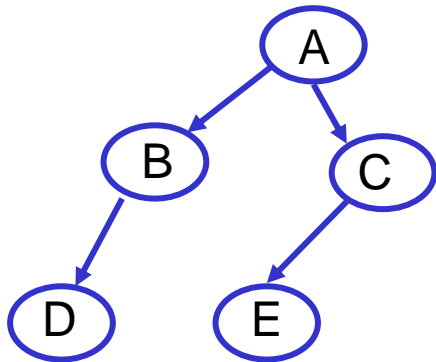
```
int nodes (Node* tree) {  
    if (!tree) return 0;  
    return 1+nodes(tree->left)+nodes(tree->right);  
}
```

conta le foglie

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

Esempi di programmi su alberi generici: inserimento

Inserisci F come ultimo figlio di A



Esempi di programmi su alberi generici: inserimento

inserisce un nodo in fondo a una lista di fratelli

```
void addSon(InfoType x, Node* &tree) {  
    if (!tree) { //lista vuota  
        tree=new Node;  
        tree->label=x;  
        tree->left = tree->right = NULL;  
    }  
    else //lista non vuota  
        addSon(x, tree->right);  
}
```

Esempi di programmi su alberi generici: inserimento

inserisce **son** come ultimo figlio di **father**.

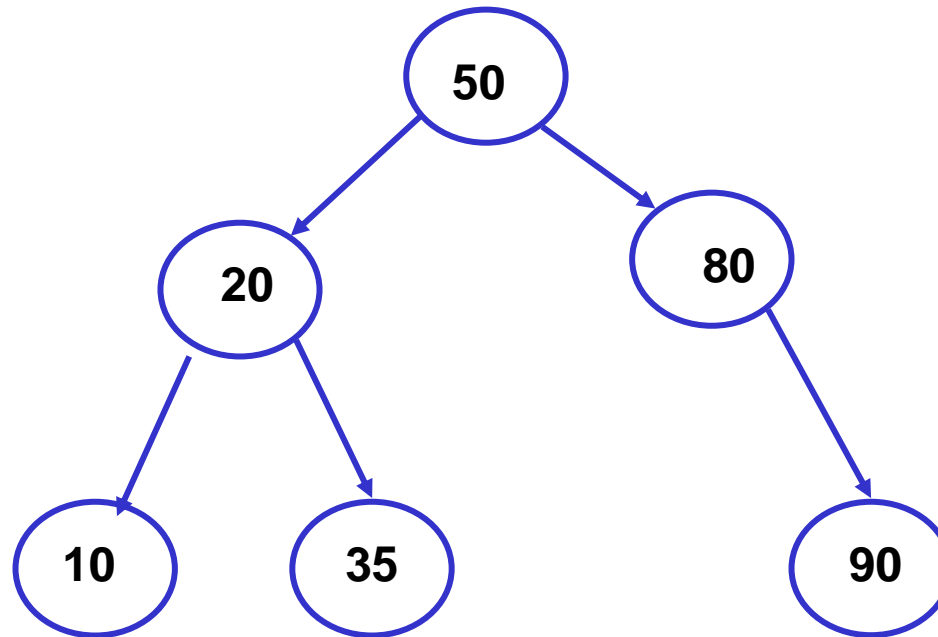
```
int insert(InfoType son, InfoType father, Node* &tree) {  
    Node* a=findNode(father, tree); // a: puntatore di father  
    if (!a) return 0;                // father non trovato  
    addSon(son, a->left);  
    return 1;  
}
```

Alberi binari di ricerca: definizione

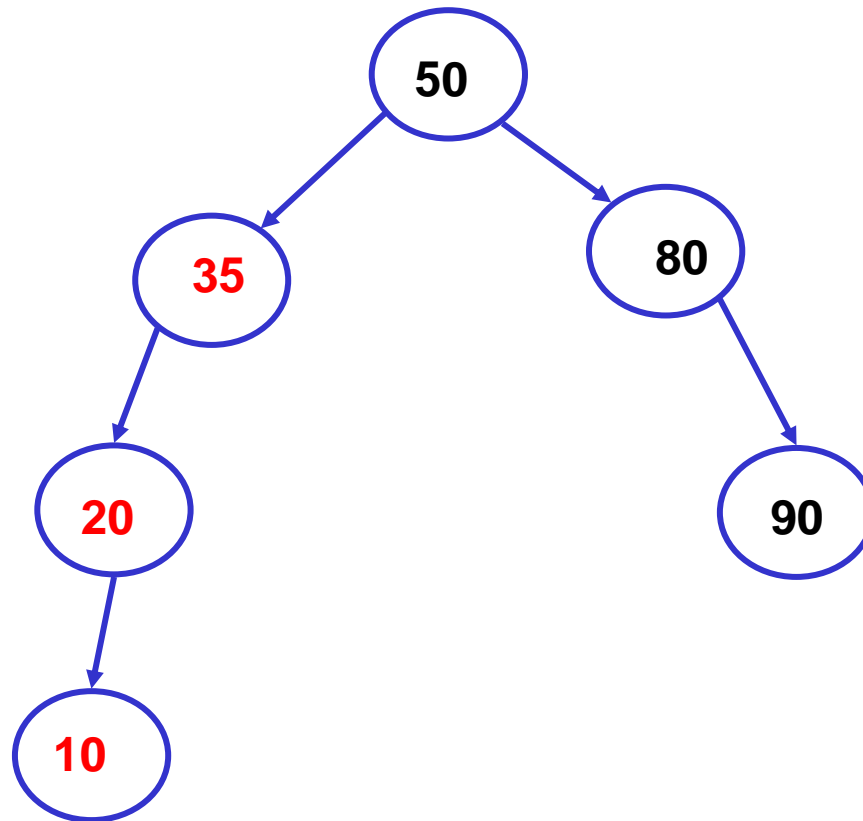
Un **albero binario di ricerca** è un albero binario tale che per ogni nodo p :

- i nodi del sottoalbero sinistro di p hanno etichetta **minore dell'etichetta di p**
- i nodi del sottoalbero destro di p hanno etichetta **maggiore dell'etichetta di p**

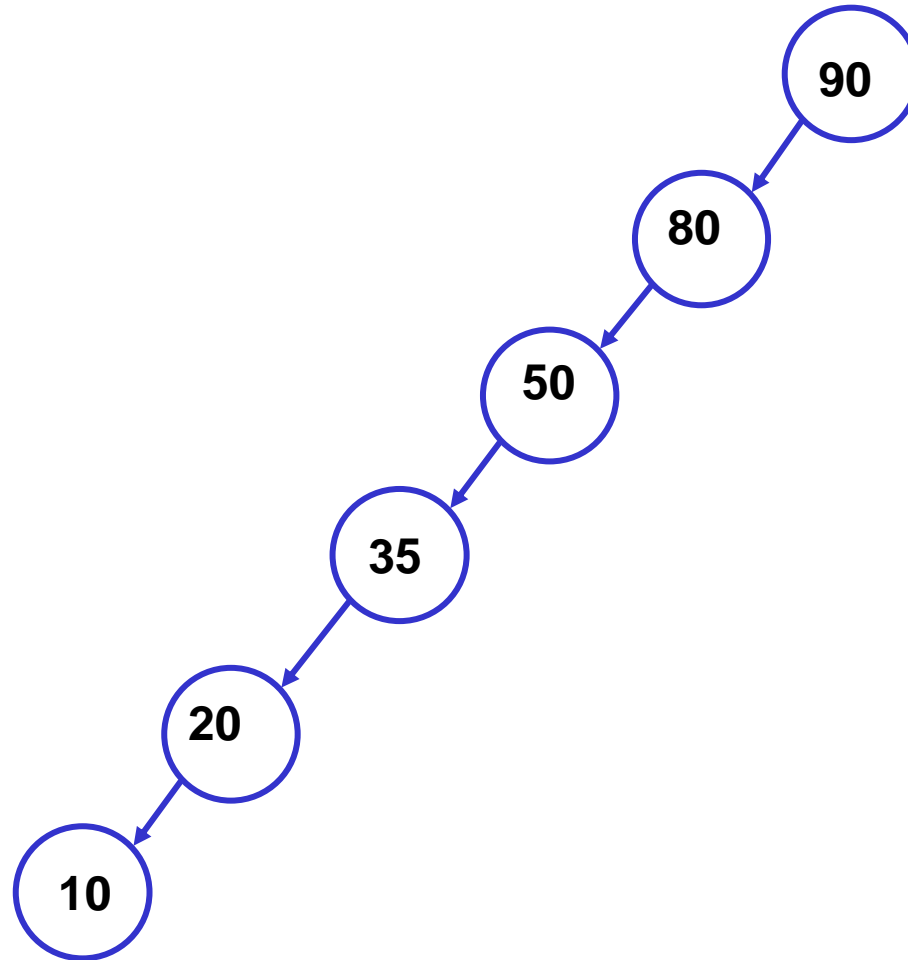
Un albero binario di ricerca



Un albero binario di ricerca con gli stessi nodi



Un albero binario di ricerca con gli stessi nodi



Alberi binari di ricerca: proprietà e operazioni

- non ci sono doppioni
- la visita simmetrica elenca le etichette in **ordine crescente**

OPERAZIONI

- **ricerca** di un nodo
- **inserimento** di un nodo
- **cancellazione** di un nodo

```
Node* findNode (InfoType n, Node* tree) {  
    if (!tree) return 0;                // albero vuoto  
  
    if (n == tree->label) return tree;   // n=radice  
  
    if (n < tree->label)                  // n < radice  
        return findNode(n, tree->left);  
  
    return findNode(n, tree->right);     // n > radice  
}
```


$$T(0)=a$$

$$T(n)= b + T(k) \quad k < n$$

$$T(0)=a$$

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

$$O(\log n)$$

$$T(0)=a$$

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

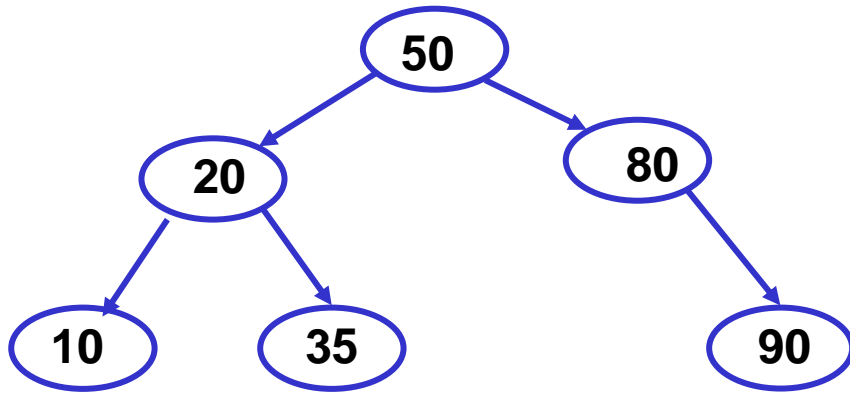
$$O(n)$$

in media : $O(\log n)$

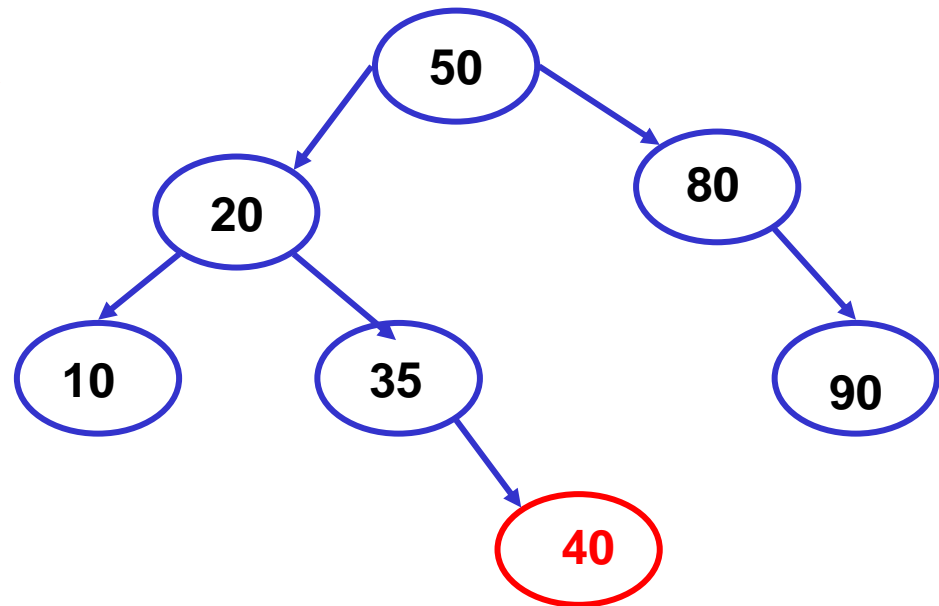
```
void insertNode (InfoType n, Node* &tree) {  
    if (!tree) { // albero vuoto: creazione nodo  
        tree=new Node;  
        tree->label=n;  
        tree->left = tree->right = NULL; return;  
    }  
    if (n<tree->label) // n<radice  
        insertNode (n, tree->left);  
    if (n>tree->label) // n>radice  
        insertNode (n, tree->right);  
}
```

$O(\log n)$

Esempio di inserimento



inserisco 40



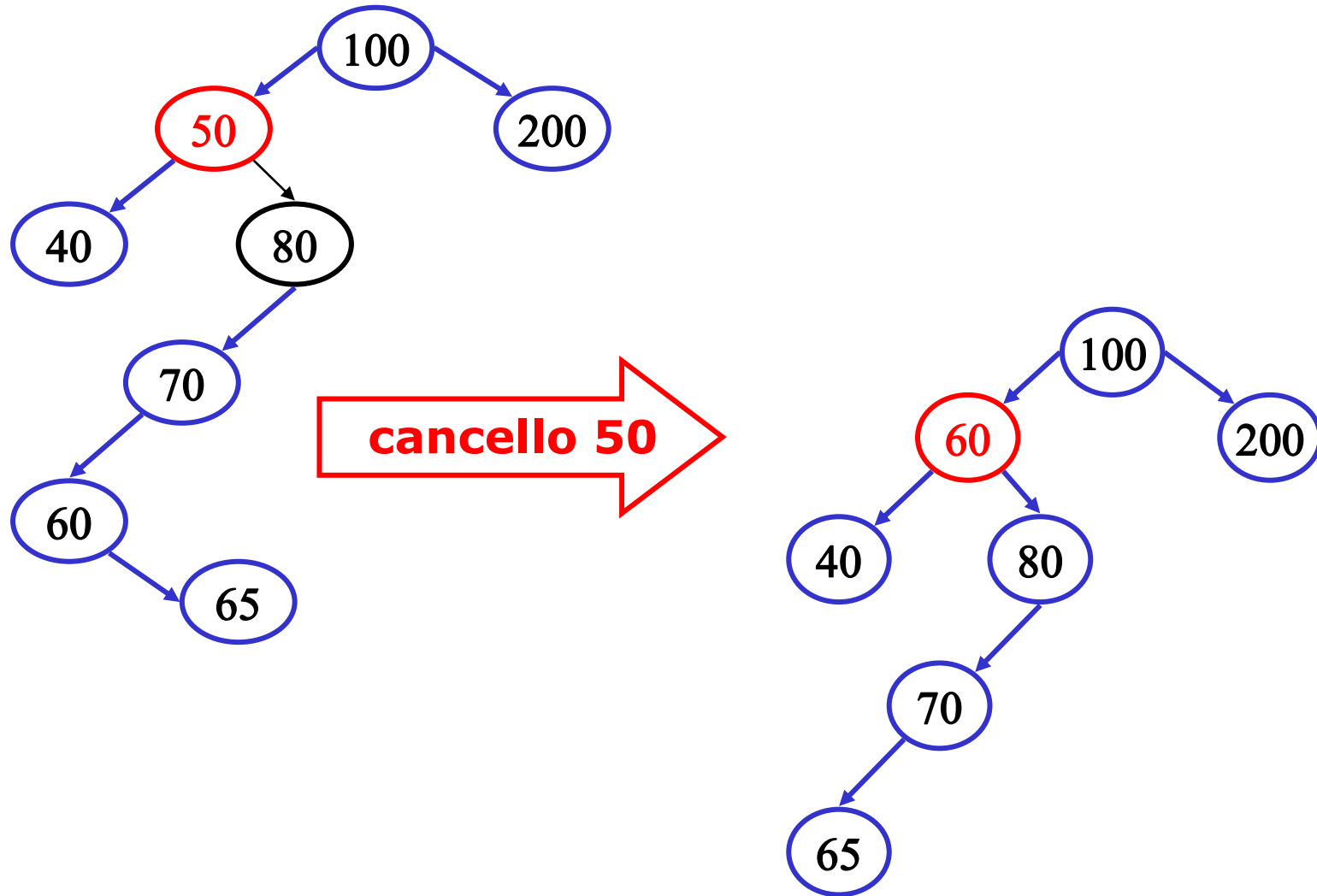
restituisce l'etichetta del nodo più piccolo di un albero ed elimina il nodo che la contiene

```
void deleteMin (Node* &tree, InfoType &m) {  
    if (tree->left)                //c'è un nodo più piccolo  
        deleteMin(tree->left, m);  
    else {  
        m=tree->label;              //restituisco l'etichetta  
        Node* a=tree;  
        tree=tree->right;           //connetto il sottoalbero destro di  
                                    // m al padre di m  
        delete a;                  //elimino il nodo  
    }  
}
```

```
void deleteNode(InfoType n, Node* &tree) {  
    if (tree)  
        if (n < tree->label)                //n minore della radice  
            { deleteNode(n, tree->left); return; }  
        if (n > tree->label)                //n maggiore della radice  
            { deleteNode(n, tree->right); return; }  
        if (!tree->left)                    //n non ha figlio sinistro  
            { Node* a=tree; tree=tree->right; delete a;return;}  
        if (!tree->right)                   //n non ha figlio destro  
            { Node* a=tree; tree=tree->left; delete a; return;}  
        deleteMin (tree->right, tree->label); //n ha entrambi i figli  
}
```

$O(\log n)$

Esempio di cancellazione



Limiti inferiori per i problemi

Un problema è di ordine $\Omega (f(n))$ se non è possibile trovare un algoritmo che lo risolva con complessità minore di $f(n)$ (tutti gli algoritmi che lo risolvono hanno complessità $\Omega (f(n))$)

Si applica soltanto agli algoritmi

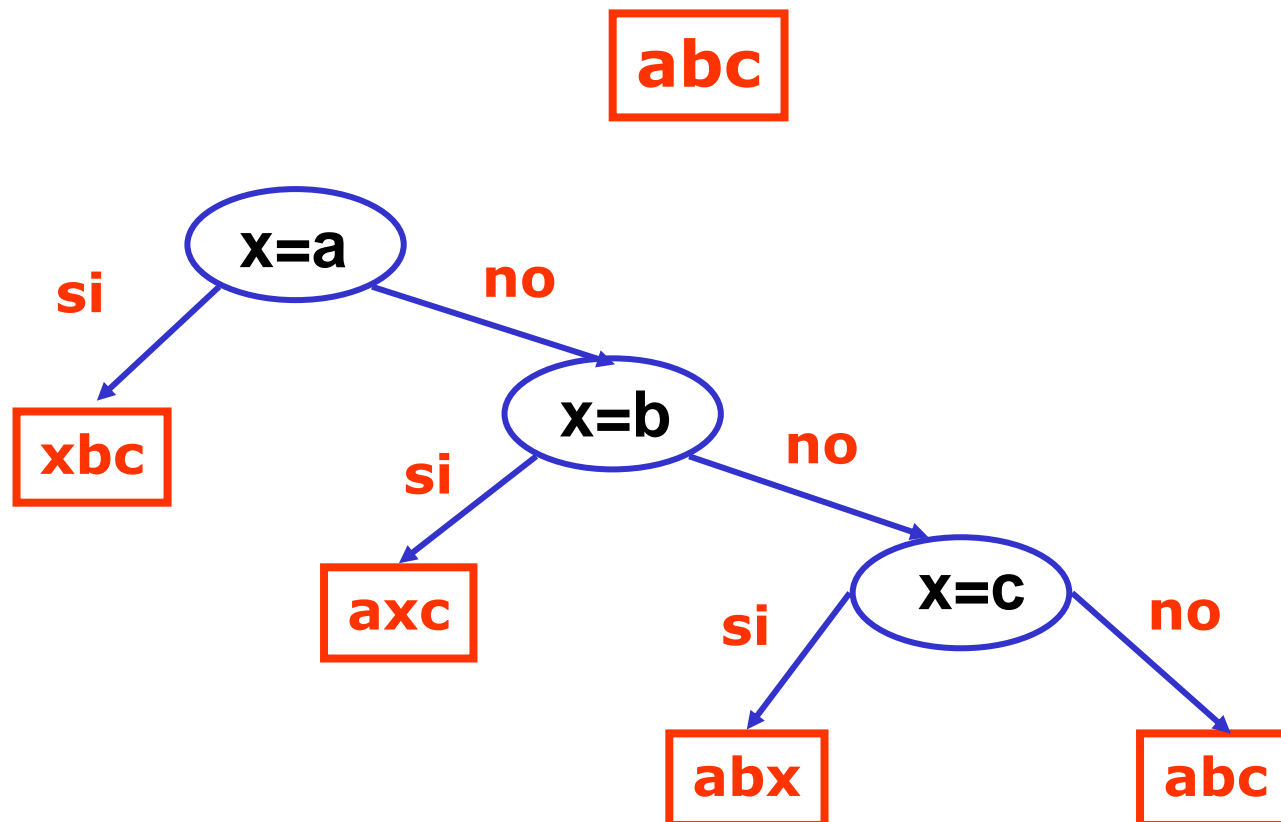
- **basati su confronti**
- **che hanno complessità proporzionale al numero di confronti che vengono effettuati durante l'esecuzione dell'algoritmo**

Limiti inferiori: alberi di decisione

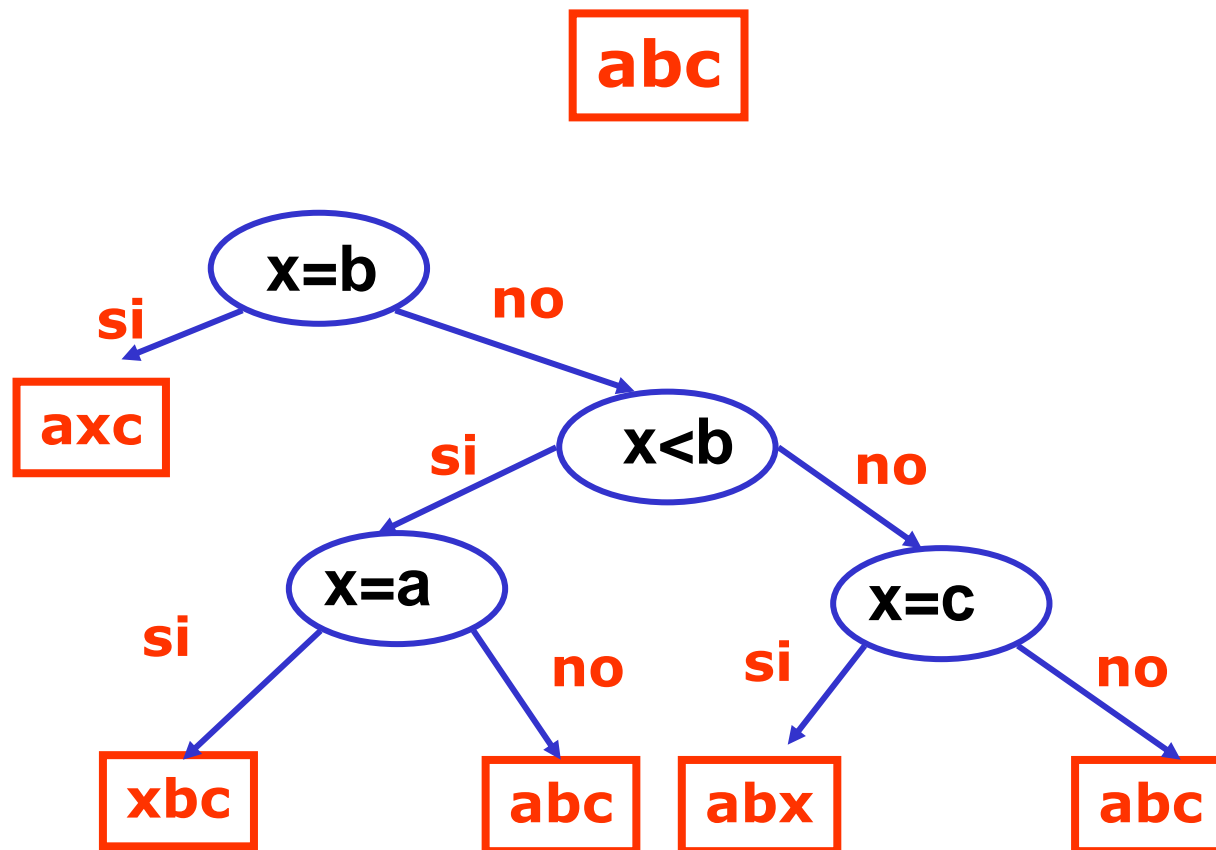
albero binario che corrisponde all'algoritmo:

- 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

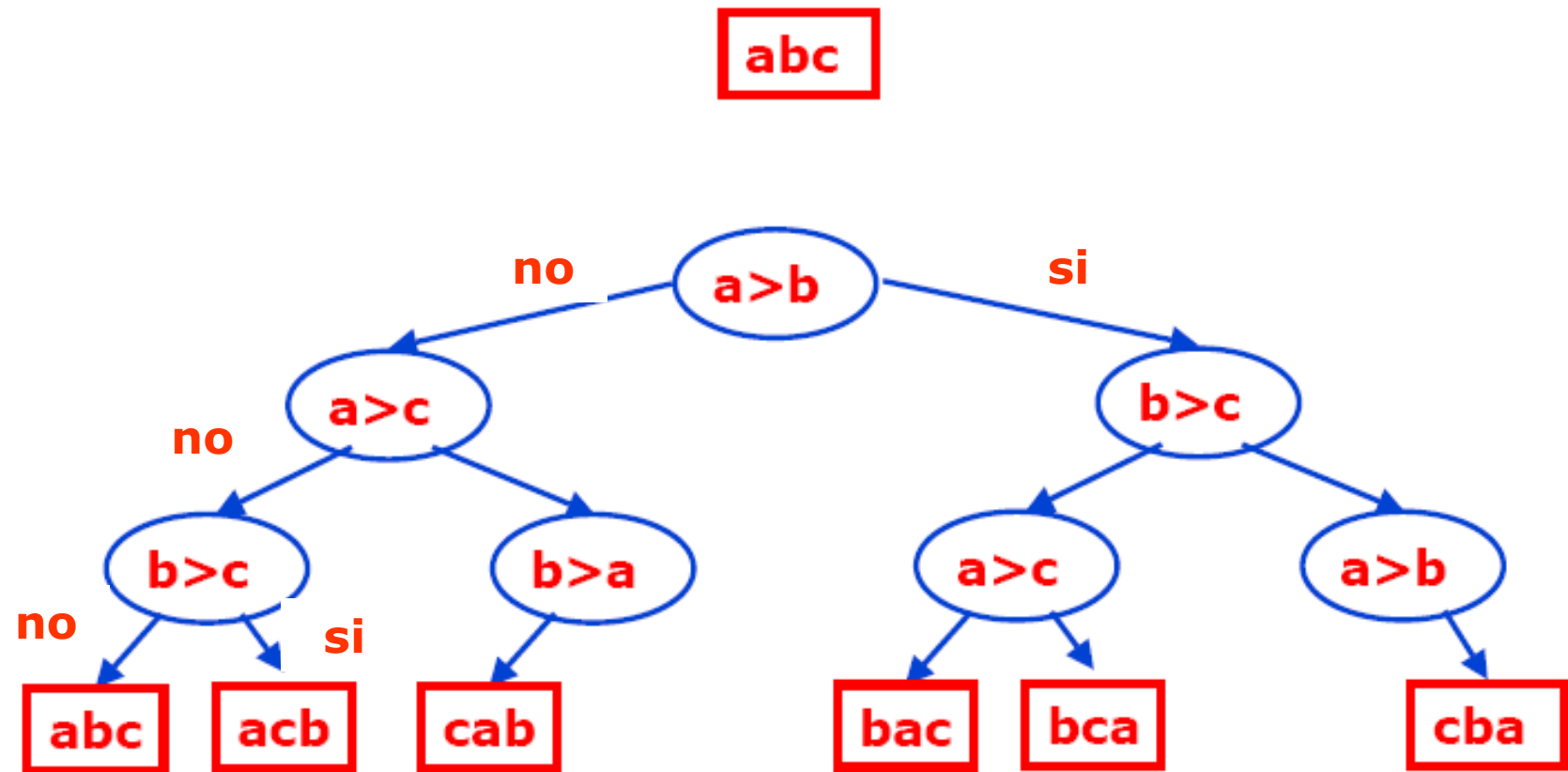
albero di decisione per la ricerca lineare



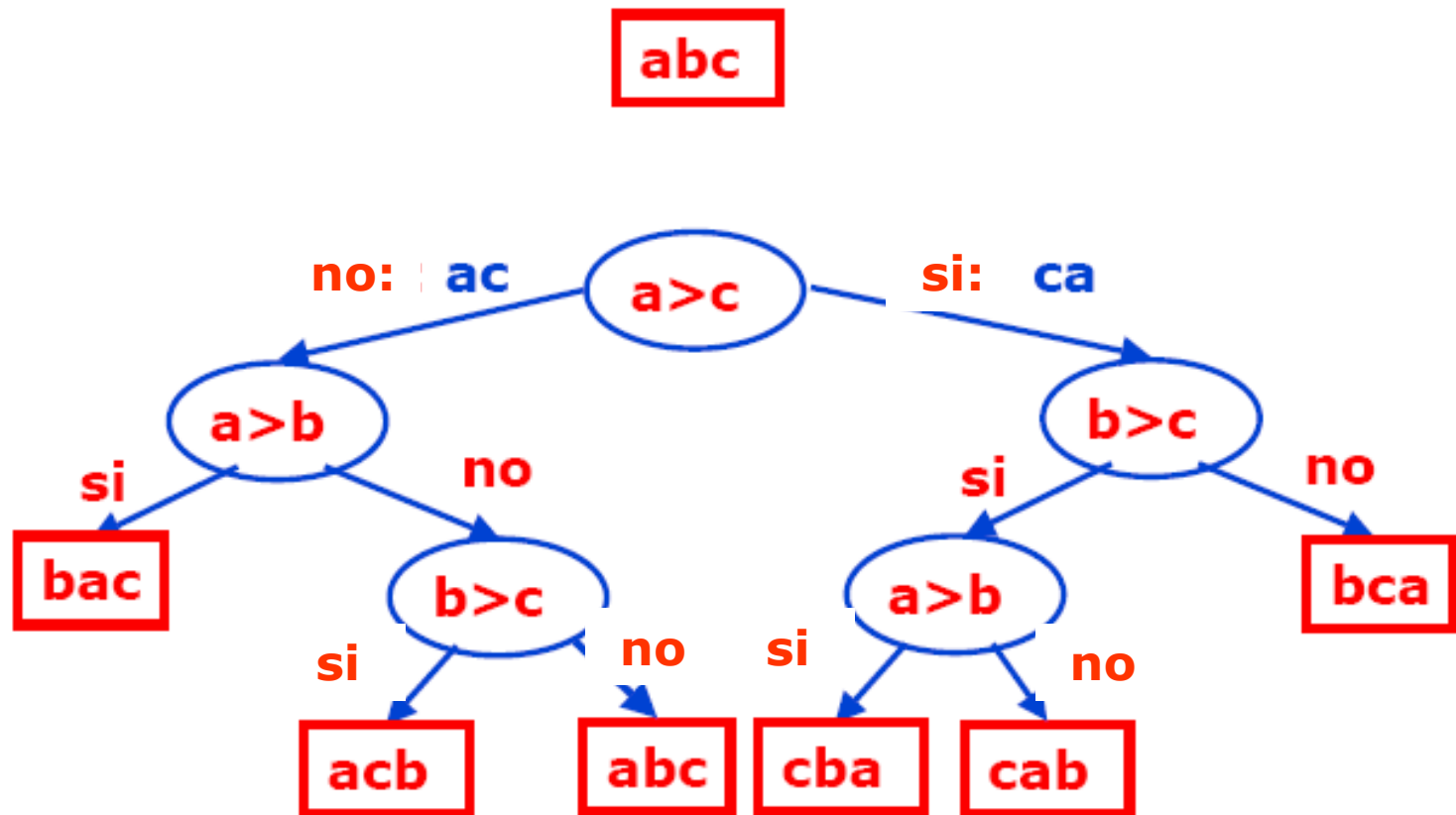
albero di decisione per la ricerca binaria



Albero del selection sort con 3 elementi



Albero del mergesort con 3 elementi



Limiti inferiori: alberi di decisione

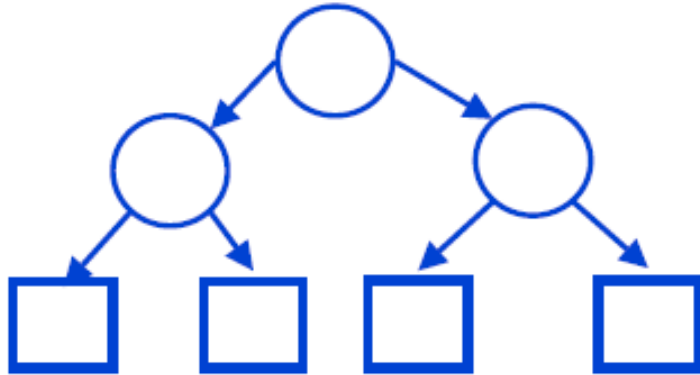
Ogni algoritmo che risolve un problema che ha $s(n)$ soluzioni ha un albero di decisione con almeno $s(n)$ foglie.

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

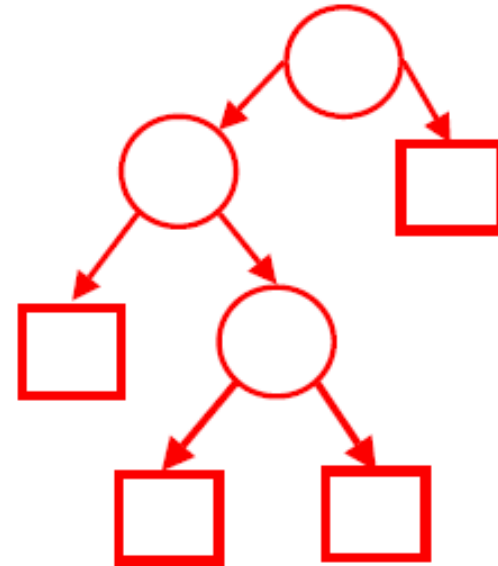
fatti

- Un albero binario con k livelli ha al massimo 2^k foglie (ce l'ha quando è bilanciato)
- Un albero binario con s foglie ha almeno $\log_2 s$ livelli
- Gli alberi binari bilanciati minimizzano sia il caso peggiore che quello medio: hanno $\log s(n)$ livelli.

Confronto fra algoritmi con 4 soluzioni



cammino max :2
cammino medio: 2



cammino max : 3
cammino medio: 2,25

$$(1+2+2*3)/4=9/4=2,25$$

Numero soluzioni : $n!$

$$n! = (n/e)^n$$

cammino medio e max: $\log(n!) \approx n \log n$

- **Mergesort è ottimo**
- **Quicksort è ottimo nel caso medio**
- **Non sempre il limite è raggiungibile
(la ricerca è $\Omega(\log n)$)**

Ordinamenti con complessità minore di $O(n \log n)$

- **counting sort**
- **radix sort**

counting sort

- Ordina una sequenza di **interi**
- Si può usare quando si conoscono i valori minimo e massimo degli elementi da ordinare
- Per ogni valore presente nell'array, si contano gli elementi con quel valore utilizzando un array ausiliario avente come dimensione **l'intervallo dei valori**
- Successivamente si ordinano i valori tenendo conto dell'array ausiliario

Esempio

A=

0	1	2	3	4	5	6	7	8	9	10	11	12	13
7	7	4	4	7	5	4	7	4	5	1	1	0	1

C=

0	1	2	3	4	5	6	7
1	3	0	0	4	2	0	4

n=14
minimo=0
massimo=7
possibili valori:8

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	1	1	4	4	4	4	5	5	7	7	7	7

counting sort

```
void counting_sort(int A[], int k, int n)
    // 0 è il minimo, k il massimo, C array ausiliario
{   int i, j; int C[k+1];
    for (i=0; i<=k; i++) C[i] = 0;           // O(k)
    for (j=0; j<n; j++) C[A[j]] ++;          // O(n)
    j=0;
    for (i=0; i<=k; i++)                     // O(n+k)
        while (C[i]>0){
            A[j]=i;
            C[i]--;
            j++;
        }
```

counting sort

Non basato su confronti

Complessità $O(n+k)$ (nel caso sopra citato $O(n+8)$)

Conveniente quando k è $O(n)$

Necessaria memoria ausiliaria

Si può usare per ordinare in ordine alfabetico sequenze di caratteri

radix sort

- **Ordina una sequenza di interi**
- **Si può usare quando si conosce la lunghezza massima (numero di cifre) d 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 di radix sort con cifre decimali

Numeri da ordinare ($d=3$, $k=10$): 190, 051, 054, 207, 088, 010

1° passata ($O(n)$)

Si inseriscono i numeri nei contenitori in base al valore dell'ultima cifra (la meno significativa)

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

Si estraggono i numeri rilegendoli da sinistra a destra e dal basso verso l'alto:

190, 010, 051, 054, 207, 088

Esempio di Radix sort con cifre decimali

190, 010, 051, 054, 207, 088

2° passata ($O(n)$)

Si inseriscono i numeri nei contenitori in base al valore della **penultima** cifra

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

Si estraggono i numeri rileggendoli da sinistra a destra e dal basso verso l'alto:

207, 010, 051, 054, 088, 190

Esempio di Radix sort con cifre decimali

207, 010, 051, 054, 088, 190

3° e ultima passata ($O(n)$)

Si inseriscono i numeri nei contenitori in base al valore della **prima** cifra

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

Si estraggono i numeri rileggendoli da sinistra a destra e dal basso verso l'alto:

010, 051, 054, 088, 190, 207

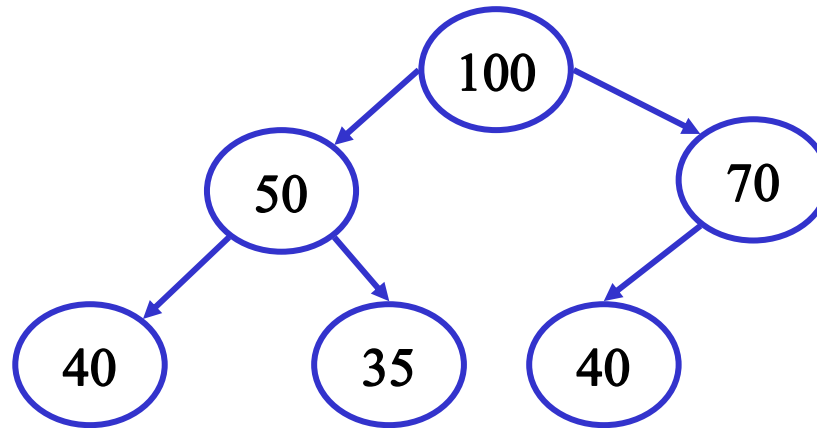
Radix sort

- **Non basato su confronti**
- **E' fondamentale partire dalla cifra meno significativa**
- **La complessità è $O(d(n+k))$ dove d è la lunghezza delle sequenze e k è il numero dei possibili valori di ogni cifra (nel caso dell'esempio $O(3(n+10))$)**
- **Necessaria memoria ausiliaria**
- **Conveniente quando d è molto minore di n**
- **Si può usare per ordinare in ordine alfabetico sequenze di caratteri**

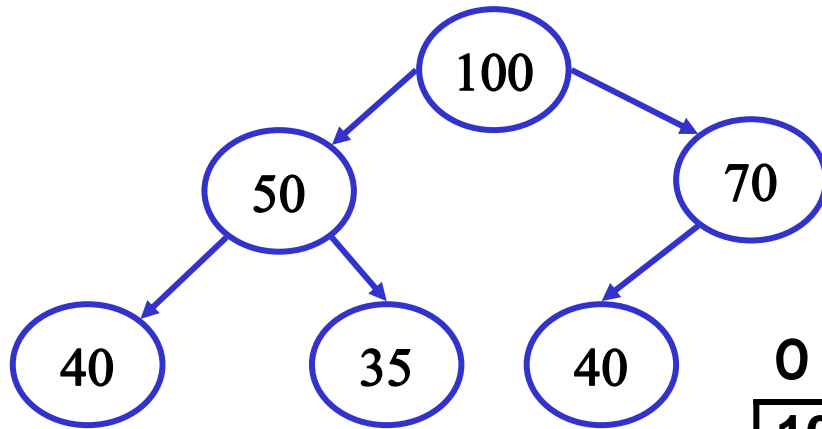
Heap: definizione

Heap: albero binario **quasi bilanciato** con le proprietà:

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



Heap: 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 (radice)

Classe Heap

```
class Heap {  
    int * h;  
    int last; //indice dell'ultimo elemento  
    void up(int);  
    void down(int);  
    void exchange(int i, int j){  
        int k=h[i]; h[i]=h[j];h[j]=k;  
    }  
public:  
    Heap(int);  
    ~Heap();  
    void insert(int);  
    int extract();  
};
```

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

last=5

Heap: costruttore e distruttore

```
Heap::Heap(int n){  
    h=new int[n];  
    last=-1;  
}
```

```
Heap::~~Heap() {  
    delete [] h;  
}
```


Heap: inserimento

- memorizza l'elemento nella prima posizione libera dell'array
- fai risalire l'elemento tramite scambi figlio-padre per mantenere la proprietà dello heap

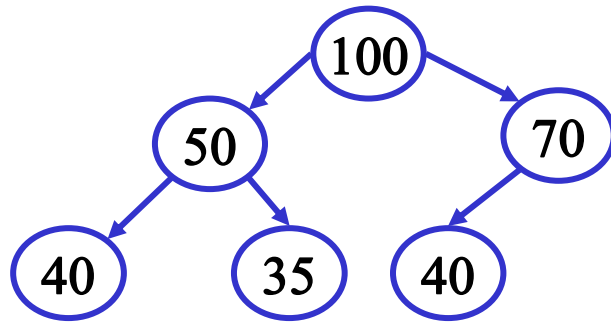
```
void Heap::insert (int x) {  
    h[++last]=x;  
    up(last);  
}
```

Heap: inserimento funzione **up**

```
void Heap::up(int i) { // i è l'indice dell'elemento da far risalire
    if (i > 0) // se non sono sulla radice
        if (h[i] > h[(i-1)/ 2]) { // se l'elemento è maggiore del padre
            exchange(i,(i-1)/2); // scambia il figlio col padre
            up((i-1)/2); // e chiama up sulla nuova posizione
        } // altrimenti termina
    }
```

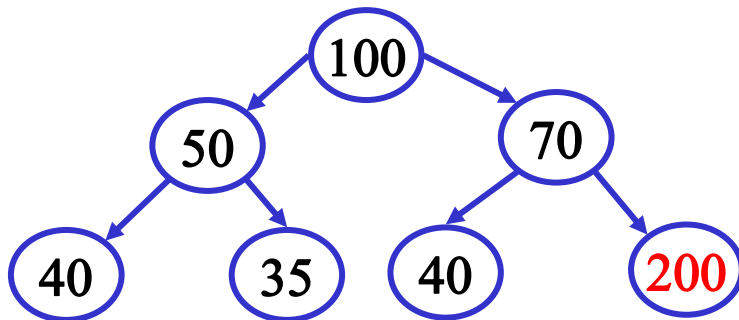
- la funzione termina o quando viene chiamata con l'indice 0 (radice) o quando l'elemento è inferiore al padre
- La complessità è $O(\log n)$ perchè ogni chiamata risale di un livello

Heap: esempio di inserimento



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

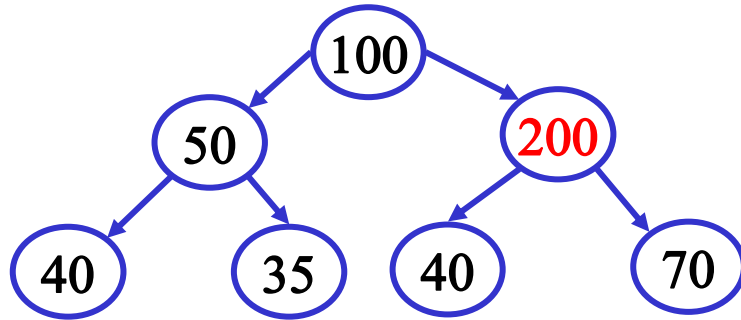
insert(200)



0	1	2	3	4	5	6	7
100	50	70	40	35	40	200	

Heap: esempio di inserimento

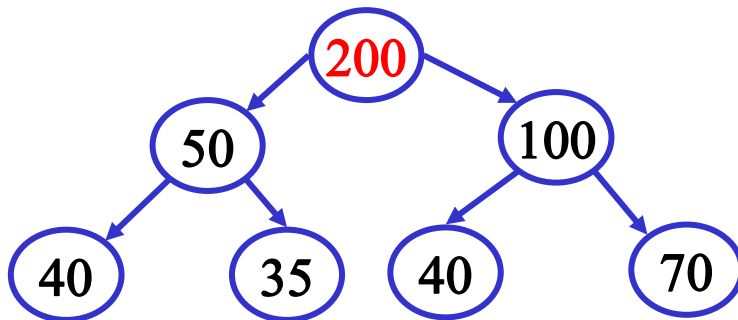
up(6)



0 1 2 3 4 5 6 7

100	50	200	40	35	40	70	
-----	----	-----	----	----	----	----	--

up(2)



0 1 2 3 4 5 6 7

200	50	100	40	35	40	70	
-----	----	-----	----	----	----	----	--

up(0)

Heap: estrazione

- restituisci il primo elemento dell'array
- metti l'ultimo elemento al posto della radice e decrementa last
- fai scendere l'elemento tramite scambi padre-figlio per mantenere la proprietà dello heap

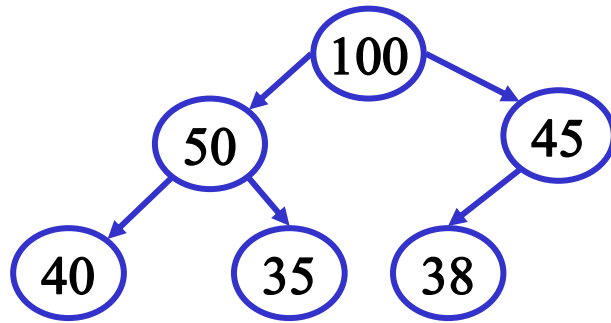
```
int Heap::extract() {  
    int r=h[0];  
    h[0]=h[last--];  
    down(0);  
    return r;  
}
```

Heap: estrazione funzione **down**

```
void Heap::down(int i) { // i è l'indice dell'elemento da far scendere
    int son=2*i+1;       // son = indice del figlio sinistro (se esiste)
    if (son == last) {   // se i ha un solo figlio (è l'ultimo dell'array)
        if (h[son] > h[i]) // se il figlio è maggiore del padre
            exchange(i,last); // fai lo scambio, altrimenti termina
    }
    else if (son < last) { // se i ha entrambi i figli
        if (h[son] < h[son+1]) son++; // son = indice del maggiore fra i due
        if (h[son] > h[i]) { // se il figlio è maggiore del padre
            exchange(i,son); // fai lo scambio
            down(son); // e chiama down sulla nuova posizione
        } // altrimenti termina (termina anche se i non ha figli)
    }
}
```

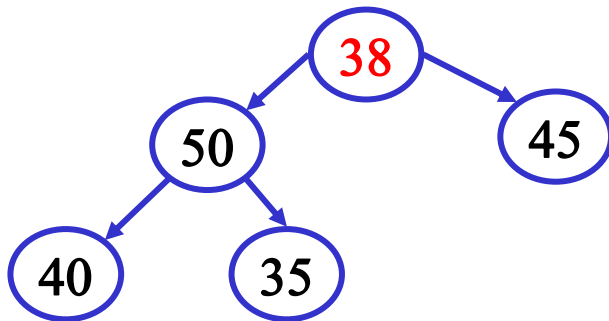
complessità : $O(\log n)$

Heap: esempio di estrazione



extract() -> **100**

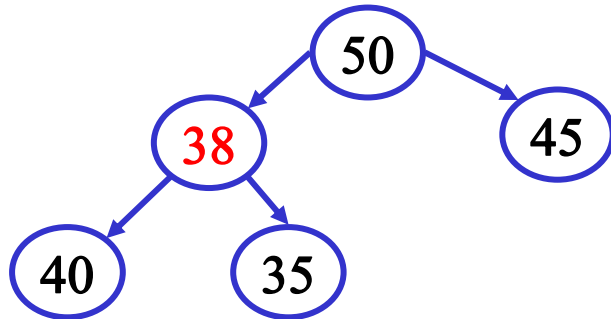
0	1	2	3	4	5	6	7
100	50	45	40	35	38		



0	1	2	3	4	5	6	7
38	50	45	40	35	38		

Heap: esempio di estrazione

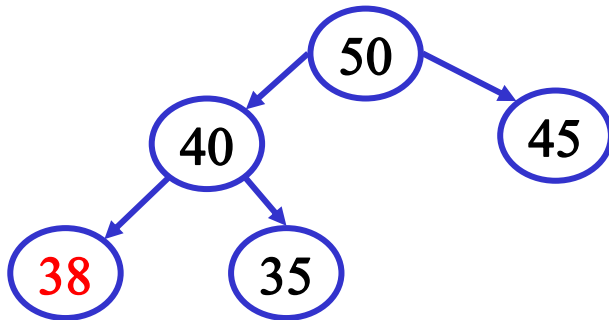
down(0)



0 1 2 3 4 5 6 7

50	38	45	40	35	38		
----	----	----	----	----	----	--	--

down(1)



0 1 2 3 4 5 6 7

50	40	45	38	35	38		
----	----	----	----	----	----	--	--

down(3)

Algoritmo di ordinamento Heapsort

- trasforma l'array in uno heap (**buildheap**)
- esegui **n** volte l'estrazione scambiando ogni volta il primo elemento dell'array con quello puntato da **last**

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

down modificata

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

$O(\log n)$

**I parametri sono l'array, l'indice
dell'elemento da far scendere,
l'ultimo elemento dello heap**

Estratt modificata

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

- I parametri sono l'array e l'ultimo elemento dello heap
- L'ultimo elemento viene scambiato con il primo
- Non si restituisce nulla

$O(\log n)$

Trasforma l'array in uno heap (buildheap)

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

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

$O(n)$

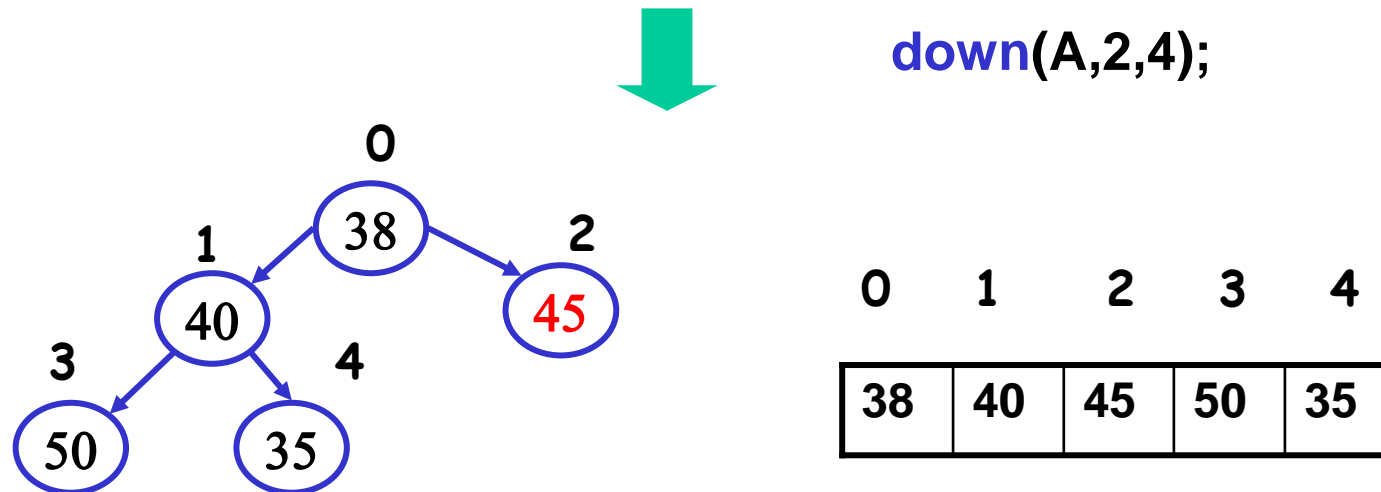
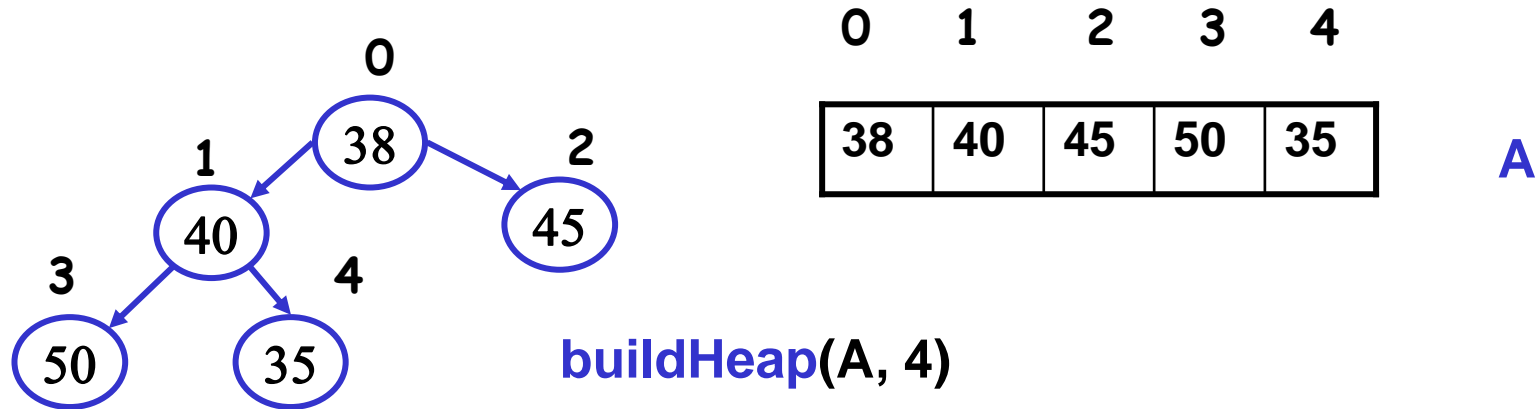
Esempio di heapsort

heapSort(A, int 5)

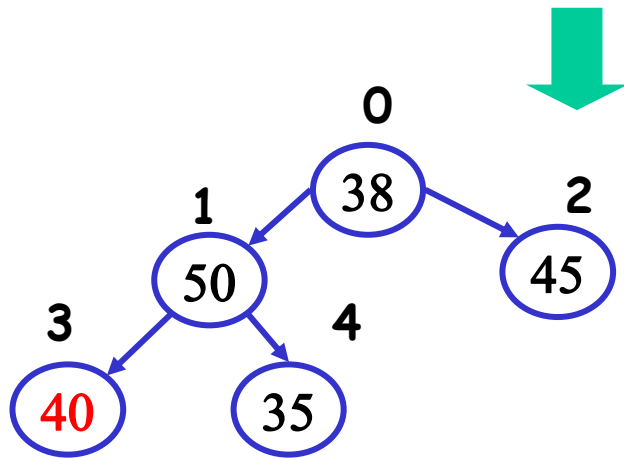
0	1	2	3	4
38	40	45	50	35

A

Esempio di heapsort: **buildHeap**

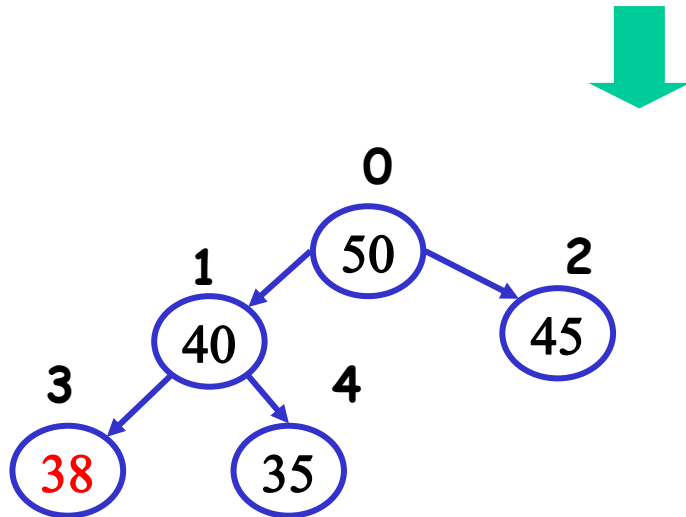


Esempio di heapsort: **buildHeap**



down(A,1,4);

0	1	2	3	4
38	50	45	40	35



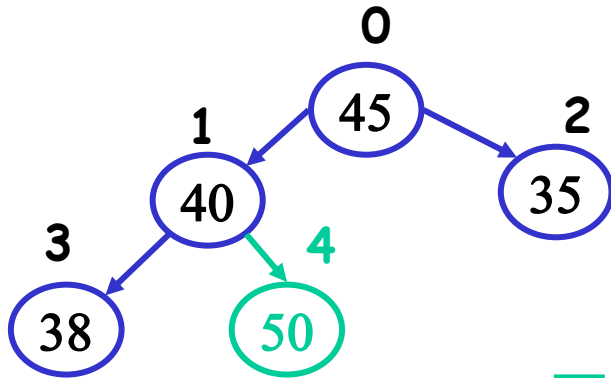
down(A,0,4);

0	1	2	3	4
50	40	45	38	35

Esempio di heapsort: estrazioni



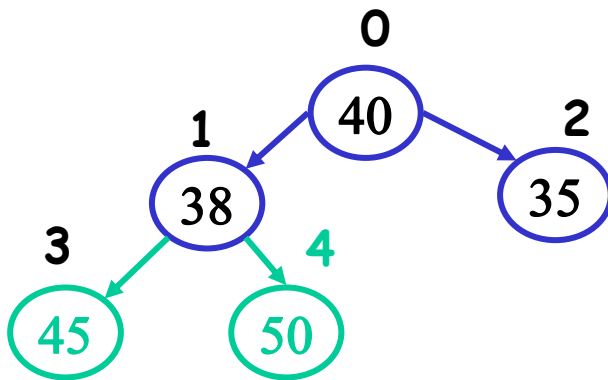
`extract(A,4);`



0	1	2	3	4
45	40	35	38	50



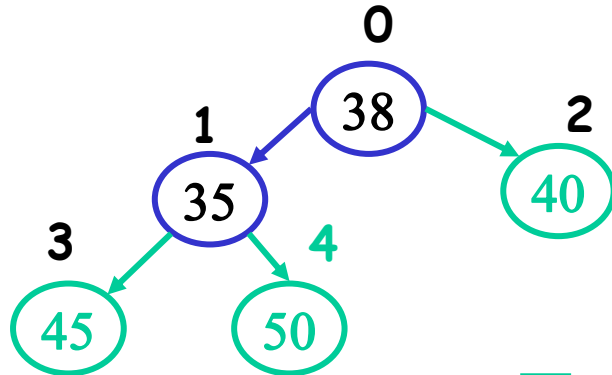
`extract(A,3);`



0	1	2	3	4
40	38	35	45	50

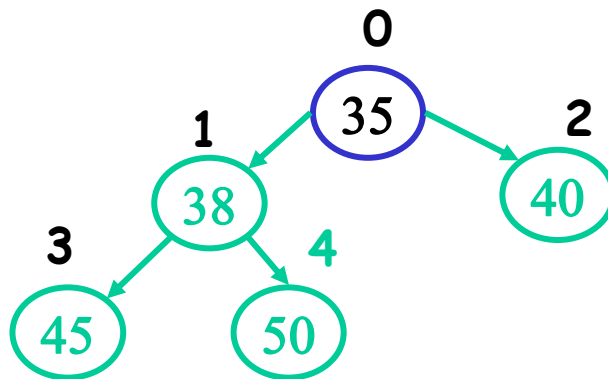
Esempio di heapsort: estrazioni

extract(A,2);



0	1	2	3	4
38	35	40	45	50

extract(A,1);



0	1	2	3	4
35	38	40	45	50

Metodo di ricerca hash

- **metodo di ricerca in array**
- **non basato su confronti**
- **molto efficiente**

Metodo hash

h (funzione hash) : InfoType \rightarrow indici

$x \rightarrow h(x)$

$n \leq k$

n = numero massimo di elementi
 k = dimensione dell'array

0

1

■

■

■

$k-1$



Metodo hash: accesso diretto

h iniettiva:

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

complessità : $O(1)$

Problema: memoria (k può essere molto grande)

Insiemi di al massimo 5 cifre decimali

```
bool hashSearch (int *A , int k, int x) {
```

```
    int i=h(x);
```

```
    if (A[i] == 1) return true;
```

```
    else return false ;
```

```
}
```

{ 0, 2, 7 }

n = 5

k = 10

n/k = 0,5

h(x) = x

h(0) = 0

h(2) = 2

h(7) = 7

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

NB: non è necessario memorizzare l'elemento

Metodo hash senza accesso diretto

Si rilascia l'iniettività e si permette che due elementi diversi abbiano lo stesso indirizzo hash:

$$h(x1) = h(x2) \text{ collisione}$$

Bisogna gestire le seguenti situazioni:

- **Come si cerca un elemento se si trova il suo posto occupato da un altro**
- **come si inseriscono gli elementi**

Metodo hash ad indirizzamento aperto

Una prima soluzione:

funzione hash modulare: $h(x) = (x \% k)$

(siamo sicuri che vengono generati tutti e soli gli indici dell'array)

Legge di scansione lineare: se non si trova l'elemento al suo posto, lo si cerca nelle posizioni successive fino a trovarlo o ad incontrare una posizione vuota

L'inserimento è fatto con lo stesso criterio

Esempio: insieme di al massimo 5 cifre

$n=k=5$

$h(x) = x \% k$

$n/k=1$

$h(0) = 0$

$h(2) = 2$

$h(7) = 2$

$\{ 0, 2, 7 \}$

0

0

1

-1

2

2

3

7

4

-1

Conseguenze

Agglomerato: gruppo di elementi con indirizzi hash diversi

La presenza di collisioni ed agglomerati aumenta il tempo di ricerca

esempio

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

$k = 99$

$h(x)$

$h(99) = h(198) = h(297) = 0$

$h(199) = h(100) = 1$

99, 198, 199, 297, 100

0	99
1	198
2	199
3	297
	100
98	

Metodo hash: ricerca con scansione lineare

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

-1: posizione vuota

Metodo hash : inserimento

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

Metodo hash: inserimento in presenza di cancellazioni

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

-1: posizione vuota
-2: posizione disponibile

Scansioni

scansione_lineare(x) = $(h(x) + \text{cost} * j) \bmod k$

Es: $(h(x) + j) \bmod k, j=1, 2, \dots$

scansione_quadratica(x; j) = $(h(x) + \text{cost} * j^2) \bmod k$

Es: $(h(x) + j^2) \bmod k, j=1, 2, \dots$

La diversa lunghezza del passo di scansione riduce gli agglomerati, ma è necessario controllare che la scansione visiti tutte le possibili celle vuote dell'array, per evitare che l'inserimento fallisca anche in presenza di array non pieno.

Tempo medio di ricerca per l'indirizzamento aperto

Il tempo medio di ricerca (numero medio di confronti) dipende da

- **Fattore di carico:** rapporto $\alpha = n/k$ (sempre < 1) : numero medio di elementi per ogni posizione
- **Legge di scansione** (migliore con la scansione quadratica e altre più sofisticate)
- **Uniformità della funzione hash** (genera gli indici con uguale probabilità)

Stima del numero medio di accessi

numero medio di accessi con la scansione lineare

$$\leq 1/(1 - \alpha)$$

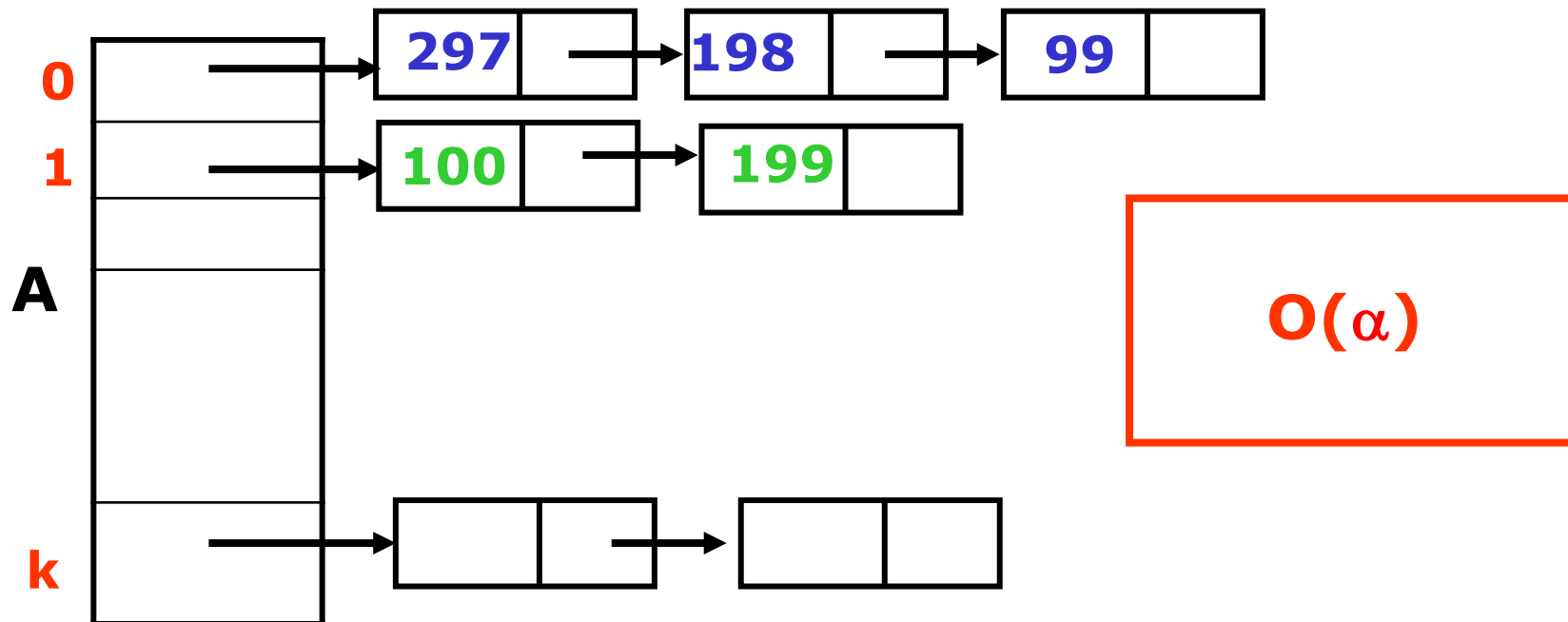
Per esempio per $\alpha = 0,5$, abbiamo 2 accessi, per $\alpha = 0,9$ ne abbiamo 10

Problemi con l'indirizzamento aperto

Molti inserimenti e cancellazioni degradano il tempo di ricerca a causa degli agglomerati. E' necessario periodicamente "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 (tabelle)

chiave	informazione

Ricerca
Inserimento
Cancellazione

Con **h(chiave)** si raggiunge l'informazione

Es: rubrica telefonica, studenti (chiave: matricola)

Motori di ricerca
Big Data
social networks
IOT

Livelli di memoria:

- **cache:** pochi megabyte, nanosecondi
- **memoria interna:** decine di gigabyte, decine di nanosecondi
- **dischi:** decine di terabyte, qualche millisecondo (SSD)
- **cloud:** decine di kilobyte, qualche secondo

Algoritmi di ricerca e ordinamento di dati non in memoria interna

Gli algoritmi devono tenere conto dei tempi diversi di accesso e delle dimensioni diverse dei livelli di memoria

- **Contano meno i confronti fra elementi diversi**
- **Conta di più il numero di operazioni di I/O fra livelli diversi di memoria**
- **Bisogna cercare di tenere più dati possibili nelle memorie più veloci**
- **Per esempio nel caso del mergesort, conviene fare la fusione, invece che fra due liste, fra 4 o di più (mergesort a più vie)**

Programmazione dinamica e algoritmi greedy

Programmazione dinamica

Si può usare quando non è possibile applicare il metodo del divide et impera (non si sa con esattezza quali sottoproblemi risolvere e non è possibile partizionare l'insieme in sottoinsiemi disgiunti)

Metodo: si risolvono **tutti i sottoproblemi a partire dal basso e si conservano i risultati ottenuti per poterli usare successivamente. (strategia bottom-up)**

La complessità del metodo dipende dal numero dei sottoproblemi

Quando si può applicare

sottostruttura 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

Più lunga sottosequenza comune (PLSC)

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

$\alpha = \alpha_1 \ \alpha_2 \ \alpha_3 \ \alpha_4 \ \alpha_5 \ \alpha_6 \ \alpha_7$

$\beta = \beta_1 \ \beta_2 \ \beta_3 \ \beta_4 \ \beta_5 \ \beta_6$

3 PLSC: **baba, cbba, caba**

Lunghezza delle PLSC = 4

$L(i,j)$ = lunghezza delle PLSC di $\alpha_1 \dots \alpha_i$ e $\beta_1 \dots \beta_j$

$$L(0,0)=L(i,0)=L(0,j)=0$$

$$L(i,j)=L(i-1,j-1)+1 \quad \text{se } \alpha_i = \beta_j$$

$$L(i,j)=\max(L(i,j-1),L(i-1,j)) \quad \text{se } \alpha_i \neq \beta_j$$

```
int length(char* a, char* b, int i, int j) {  
    if (i==0 || j==0) return 0;  
    if (a[i]==b[j]) return length(a, b, i-1, j-1)+1;  
    else  
        return max(length(a,b,i,j-1),length(a,b,i-1,j));  
};
```

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

**Costruisce tutti gli $L(i,j)$ a partire dagli
indici più piccoli (bottom-up):**

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

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

\dots

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

Algoritmo di programmazione dinamica

```
const int m=7; const int n=6;
int L [m+1][n+1];
int quickLength(char *a, char *b) {
    for (int j=0; j<=n; j++) L[ 0 ][ j ]=0; // prima riga

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

$T(n) \in O(nm)$

$T(n) \in O(n^2)$

PLSC

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

PLSC

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

cbba

```
void print(char *a, char *b, int i=m, int j=n){  
    if ((i==0) || (j==0) ) return;  
    if (a[i]==b[j]) {  
        print(a,b, i-1, j-1);  
        cout << a[i];  
    }  
    else if (L[i][j] == L[i-1][j])  
        print(a,b, i-1, j);  
    else print(a,b, i, j-1);  
}
```


Algoritmi greedy (golosi)

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: scegliendo ad ogni passo l'alternativa che sembra la migliore non si perdono alternative che potrebbero rivelarsi migliori nel seguito.

Metodo top-down

Non sempre si trova la soluzione ottima ma in certi casi si può trovare una soluzione approssimata (esempio del problema dello zaino)

codici di compressione

Alfabeto : insieme di caratteri (es: a, b, c, d, e, f)

Codice binario: assegna ad ogni carattere una stringa binaria

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

Decodifica: ricostruire il testo originario.

Il codice può essere a lunghezza fissa o a lunghezza variabile

codici di compressione

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

codici

Codifica di **abc** con codice a lunghezza fissa: :

000 001 010 (9 bit)

Codifica di **abc** con codice a lunghezza variabile :

0 101 100 (7 bit)

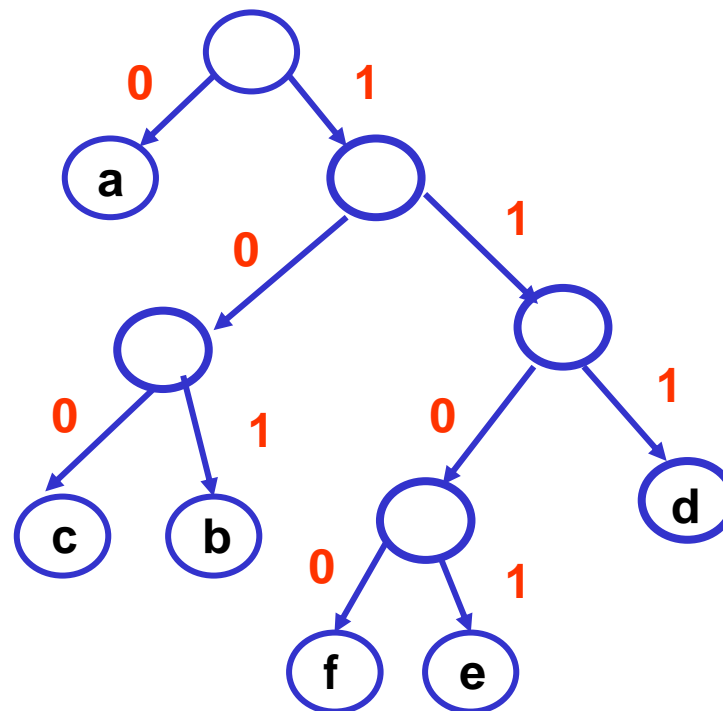
Problema della decodifica

Codice prefisso: nessun codice può essere il prefisso di un altro codice

codici prefissi

**I codici prefissi
possono essere
rappresentati con
alberi binari**

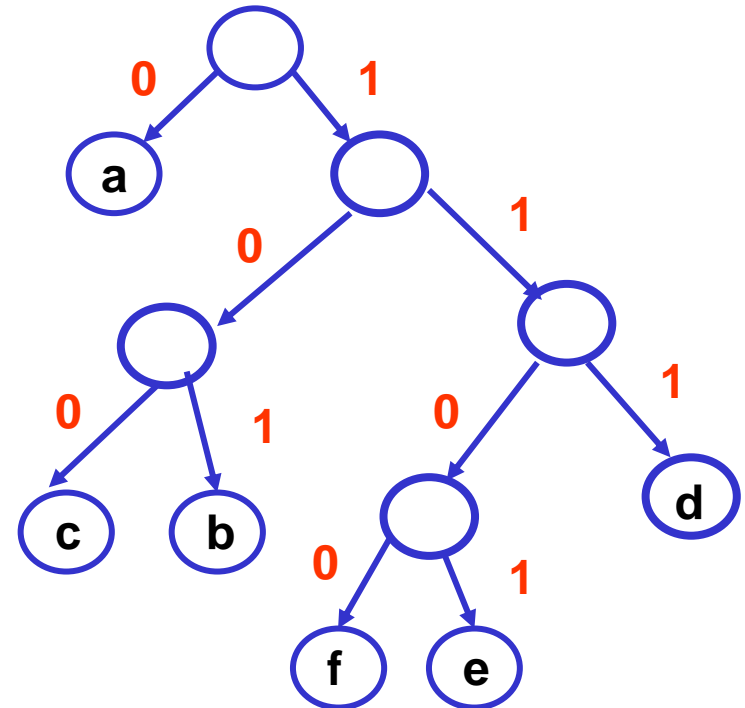
**Rappresentazione
ottima: albero
pienamente binario**



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

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

L'algoritmo di decodifica trova un cammino dalla radice ad una foglia per ogni carattere riconosciuto



Problema: dato un alfabeto e la frequenza dei suoi caratteri, costruire un codice ottimo (che minimizza la lunghezza in bit delle codifiche)

Algoritmo di Huffman

Costruisce l'albero binario in modo bottom-up

È un algoritmo **greedy**

algoritmo di Huffman

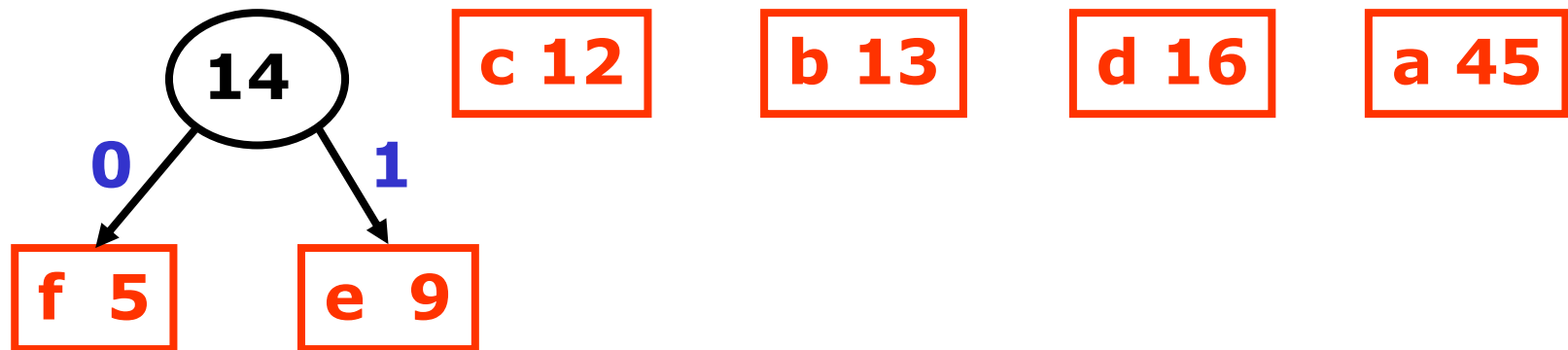
Gestisce un foresta di alberi

All'inizio ci sono n alberi di un solo nodo con le frequenze dei caratteri.

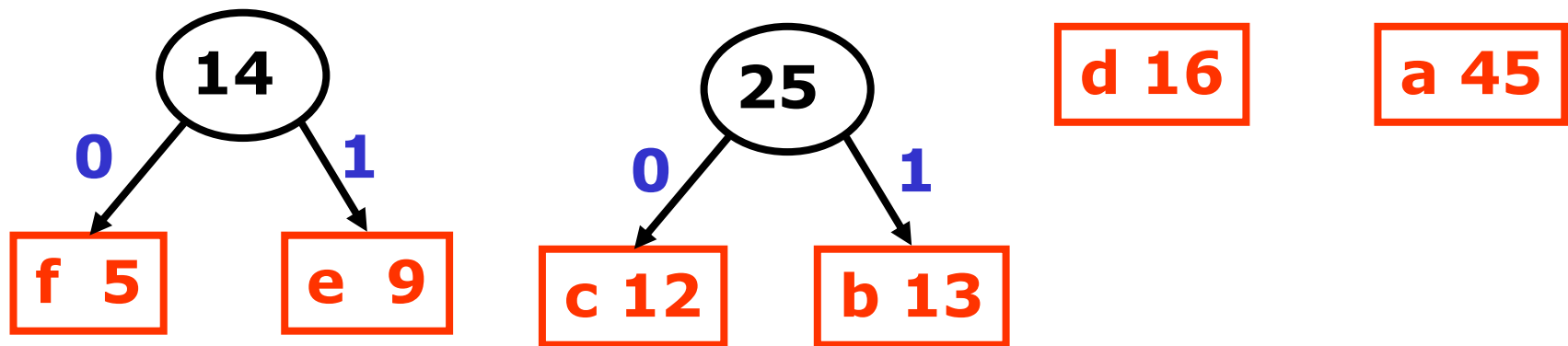
Ad ogni passo

- **vengono fusi i due alberi con radice minore introducendo una nuova radice avente come etichetta la somma delle due radici**

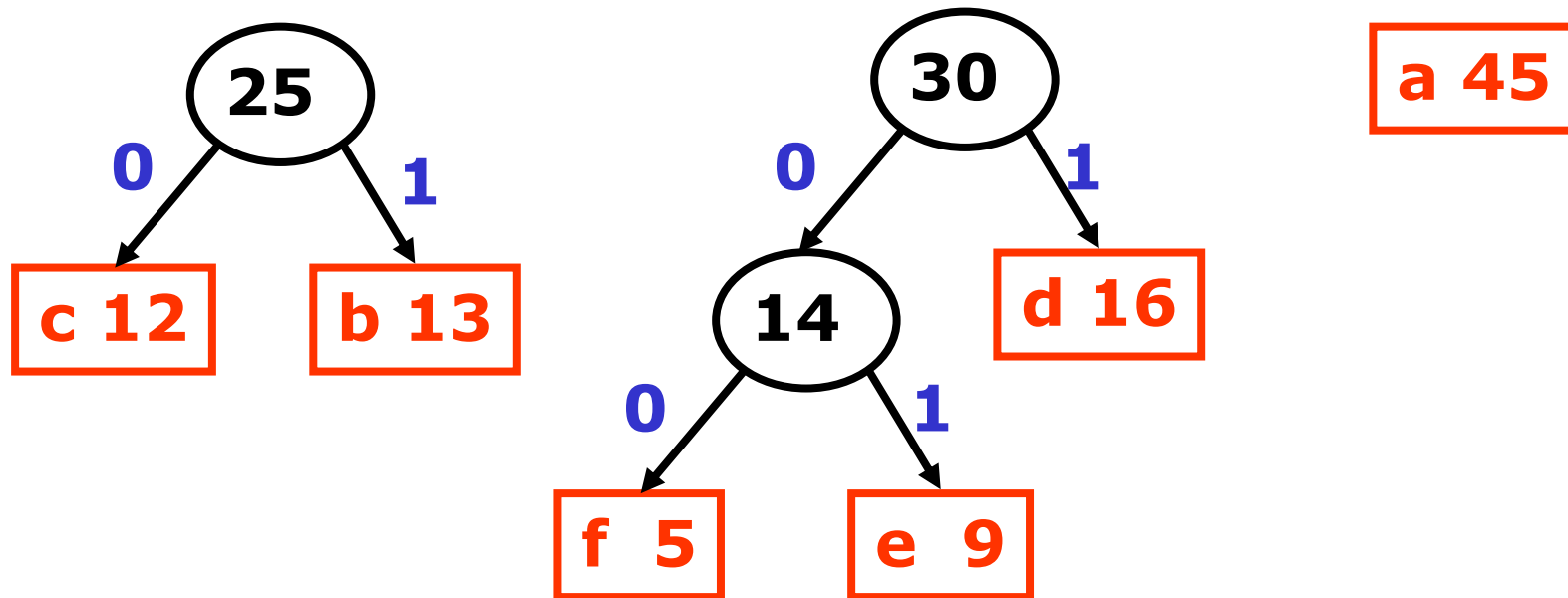
algoritmo di Huffman



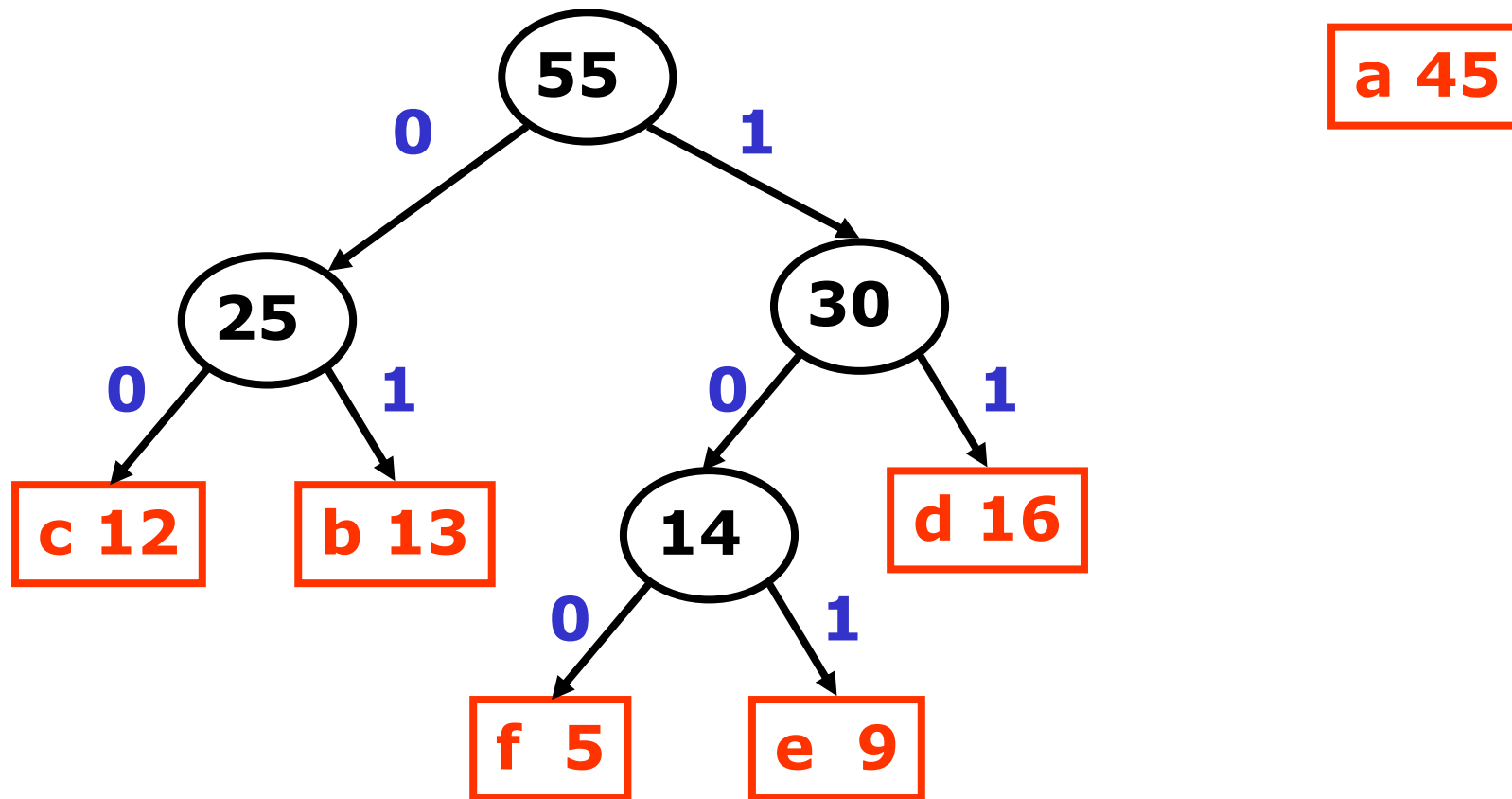
algoritmo di Huffman



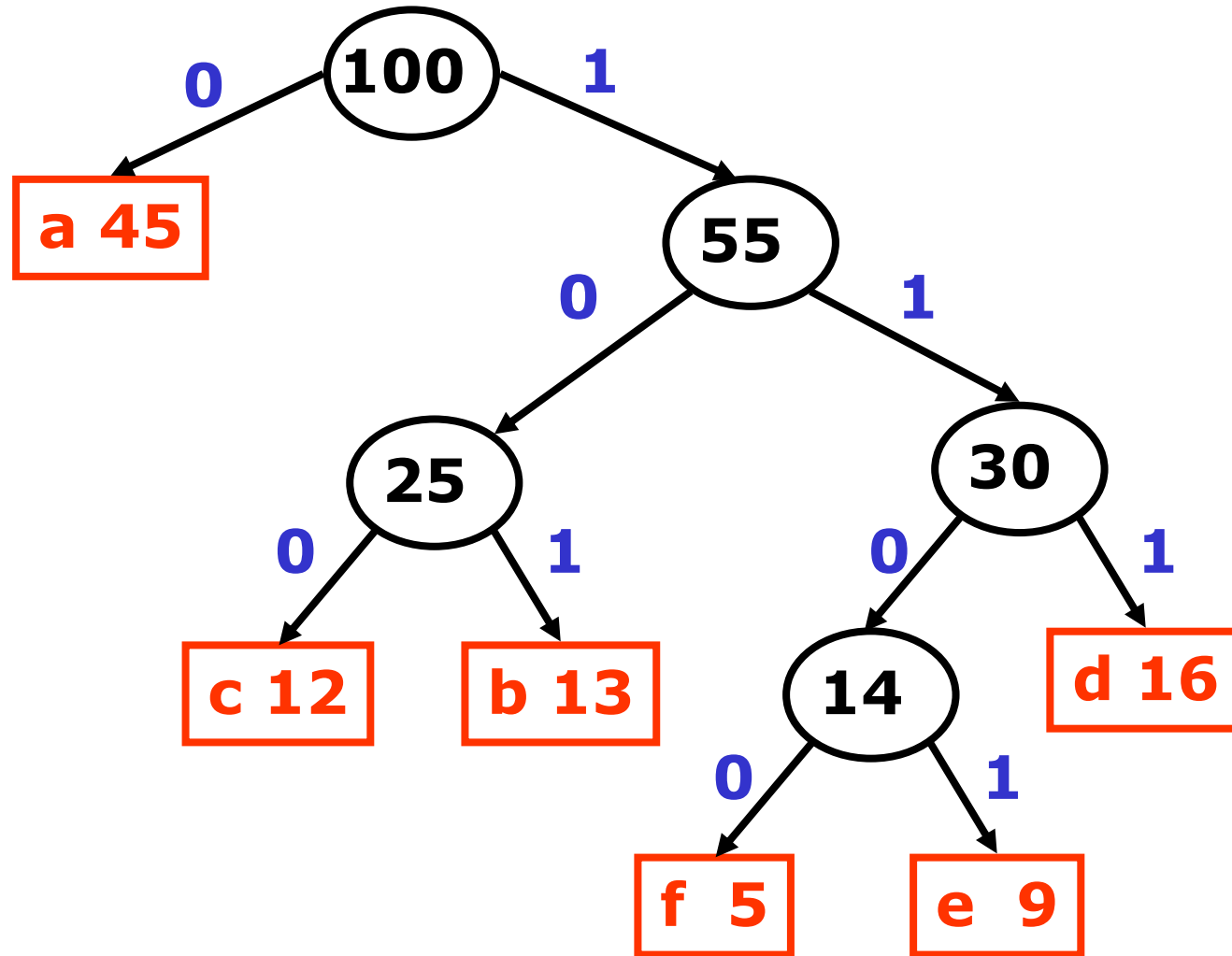
algoritmo di Huffman



algoritmo di Huffman



algoritmo di Huffman



algoritmo di Huffman: complessità

Gli alberi sono memorizzati in un **minheap** (heap con ordinamento inverso : la radice è il più piccolo)

Si fa un ciclo dove in ogni iterazione:

- vengono estratti i due alberi con **radice minore**
- vengono fusi in un nuovo albero avente come etichetta della radice la somma delle due **radici**
- l'albero risultante e' inserito nello heap

il ciclo ha **n** iterazioni ed ogni iterazione ha complessità **$O(\log n)$** (si eseguono 2 estrazioni e un inserimento)

$O(n \log n)$

La scelta locale è consistente con la situazione globale:

sistemando prima i nodi con minore frequenza, questi apparterranno ai livelli più alti dell'albero

Grafi

GRAFO ORIENTATO = (N , A)

N = insieme di **nodi**

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

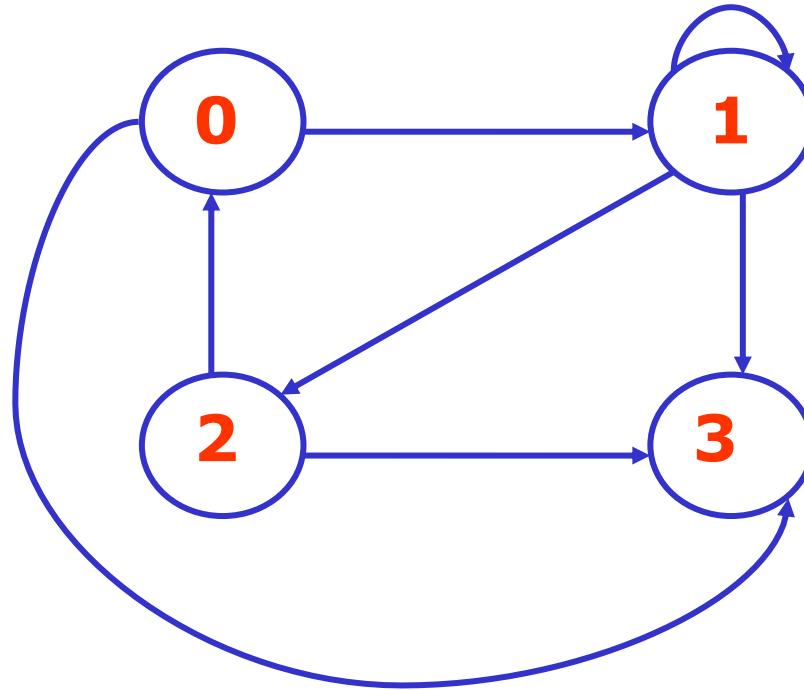
- predecessore
- successore
- cammino (sequenza di nodi-lunghezza = numero di archi)
- ciclo
- grafo aciclico

$n = |N|$ numero dei nodi

$m = |A|$ numero degli archi.

Un grafo orientato con n nodi ha al massimo n^2 archi

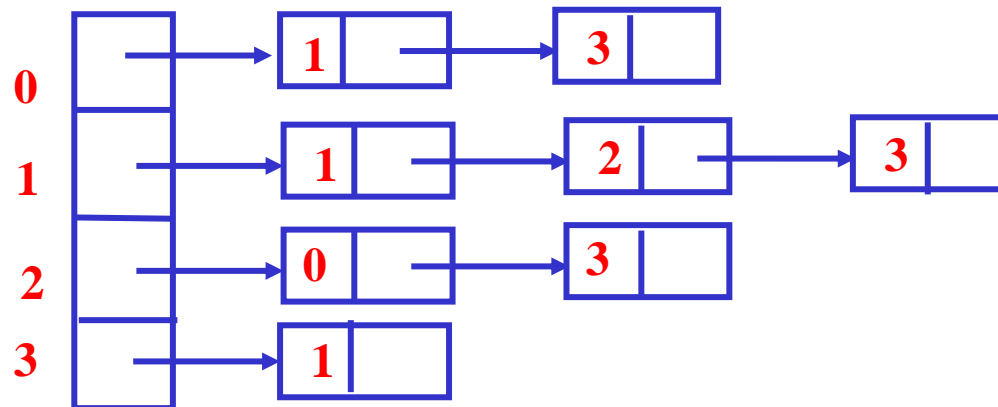
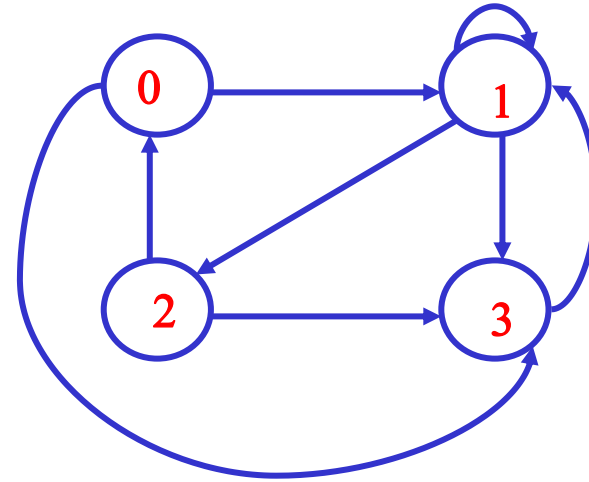
Esempio di grafo



rappresentazione in memoria dei grafi: **liste di adiacenza**

```
struct Node{  
    int NodeNumber;  
    Node * next;  
};
```

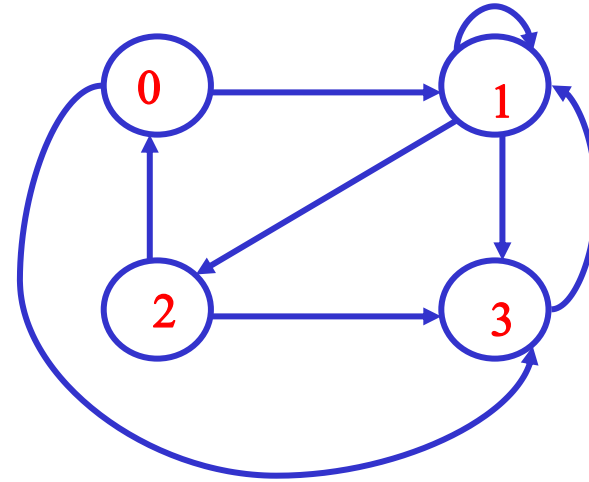
```
Node *graph[N];
```



rappresentazione in memoria dei grafi: **matrici di adiacenza**

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

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

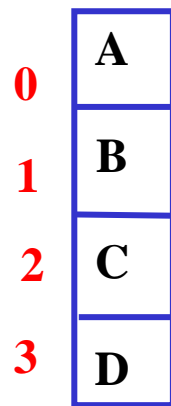
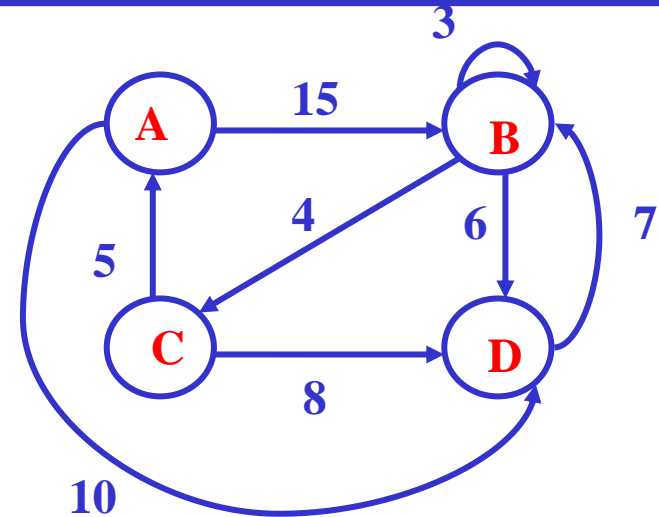


Grafi con nodi e archi etichettati : Liste di adiacenza

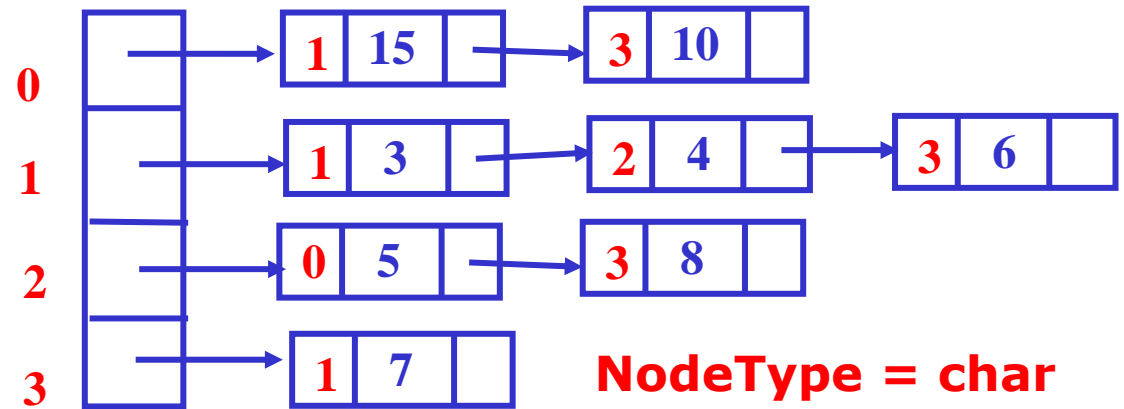
```
struct Node{
    int NodeNumber;
    ArcType arcLabel;
    Node * next;
};
```

```
Node * graph[N];
```

```
NodeType nodeLabels [N];
```



nodeLabels



graph

NodeType = char
ArcType = int

Con nodi e archi etichettati : **matrici di adiacenza**

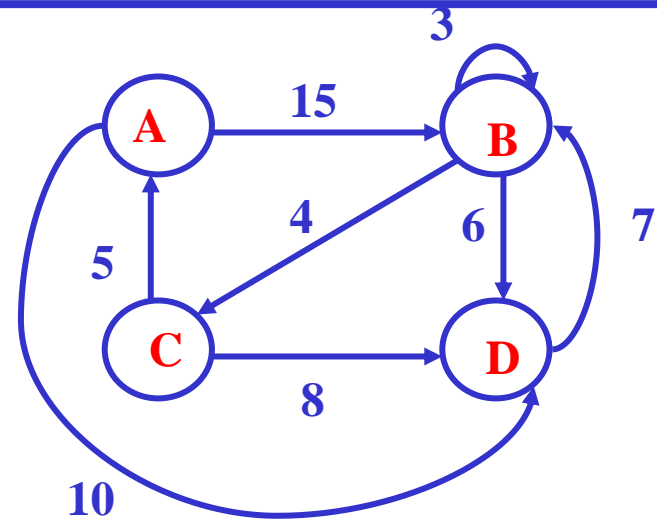
ArcType graph [N][N];

NodeType nodeLabels [N];

0	A
1	B
2	C
3	D

nodeLabels

	0	1	2	3
0	0	15	0	10
1	0	3	4	6
2	5	0	0	8
3	0	7	0	0



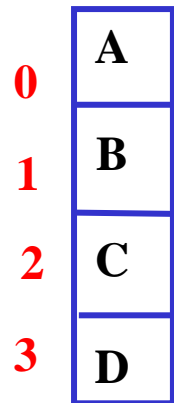
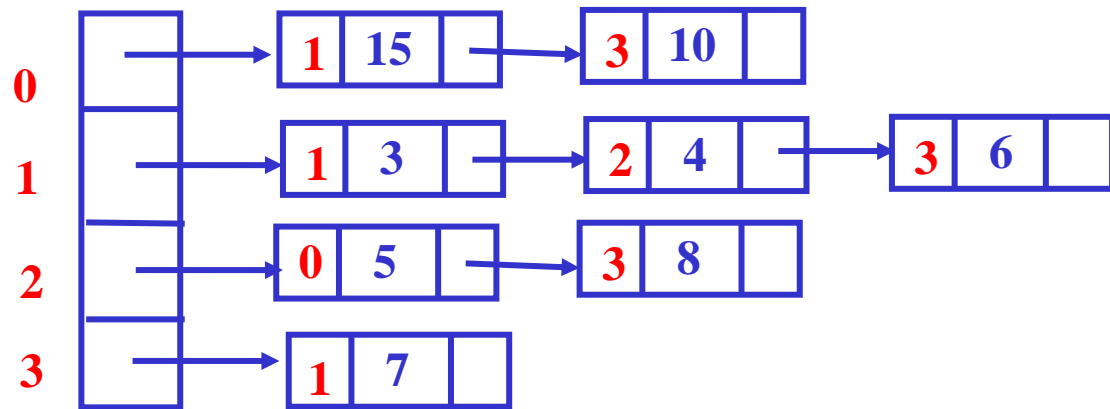
visita in profondità

```
void NodeVisit (nodo) {  
    esamina il nodo;  
    marca il nodo;  
    applica NodeVisit ai successori non marcati del nodo;  
}
```

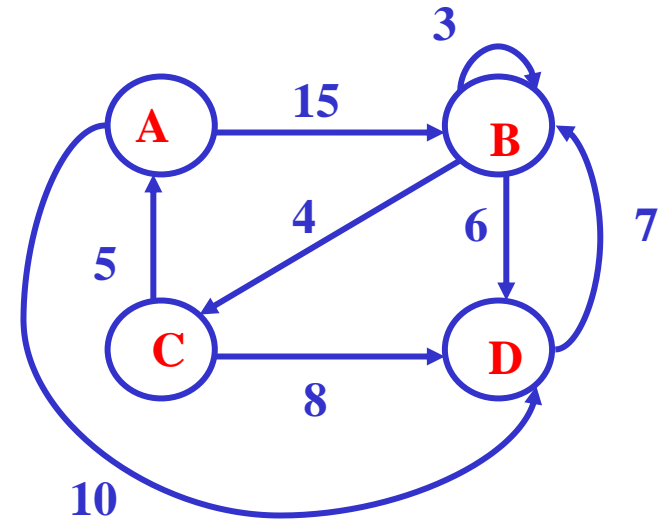
```
Void DepthVisit Graph(h) {  
    per tutti i nodi:  
        se il nodo non è marcato applica nodeVisit;  
}
```

$O(m) + O(n)$

visita in profondità: esempio



0 1 2 3
A B C D



Una classe per i grafi

```
class Graph{
  struct Node {
    int nodeNumber;
    Node* next;
  };
  Node* graph [N];
  NodeType nodeLabels [N];
  int mark[N];
  void nodeVisit( int i) {
    mark[i]=1;
    <esamina nodeLabels[i]>;
    Node* g; int j;
    for (g=graph[i]; g; g=g->next){
      j=g->nodeNumber;
      if (!mark[j]) nodeVisit(j);
    }
  }
}
```

```
public:
  void depthVisit() {
    for (int i=0; i<N; i++)
      mark[i]=0;
    for (i=0; i<N; i++)
      if (! mark[i])
        nodeVisit (i);
  }
  ..
};
```

Grafi non orientati

grafo non orientato = (**N**, **A**),

N = insieme di **nodi**

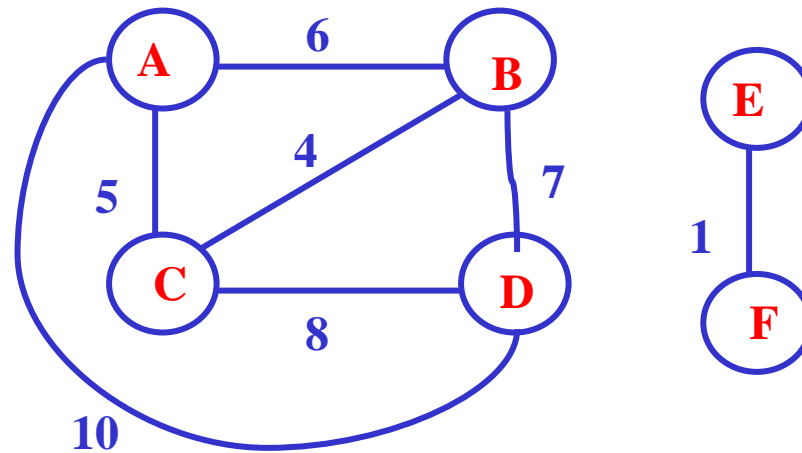
A = insieme di **coppie non ordinate di nodi**

- **nodi adiacenti**
- **ciclo (almeno 3 nodi)**

Un grafo non orientato con **n nodi ha al massimo $n(n-1)/2$ archi**

**Come un grafo orientato considerando
che ogni connessione corrisponde a due
archi orientati nelle due direzioni
opposte**

Esempio di grafo non orientato



Multi-grafo non orientato = $(\mathbf{N}, \mathbf{A}),$

\mathbf{N} = insieme di nodi

\mathbf{A} = **multi-insieme di coppie** non ordinate di nodi

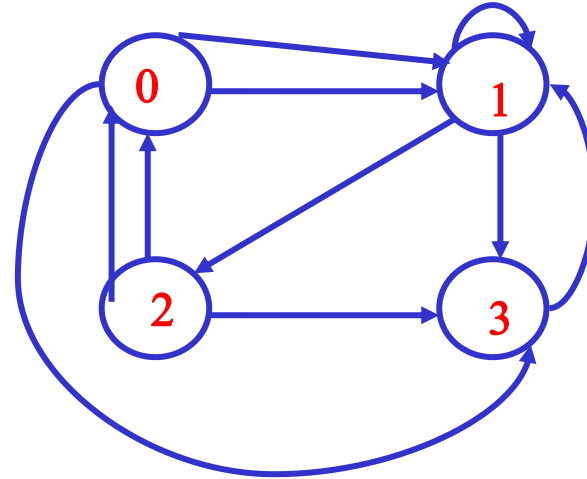
Non c'è relazione fra il numero di nodi e il numero di archi

Analogamente si definiscono i **multi-grafi non orientati**

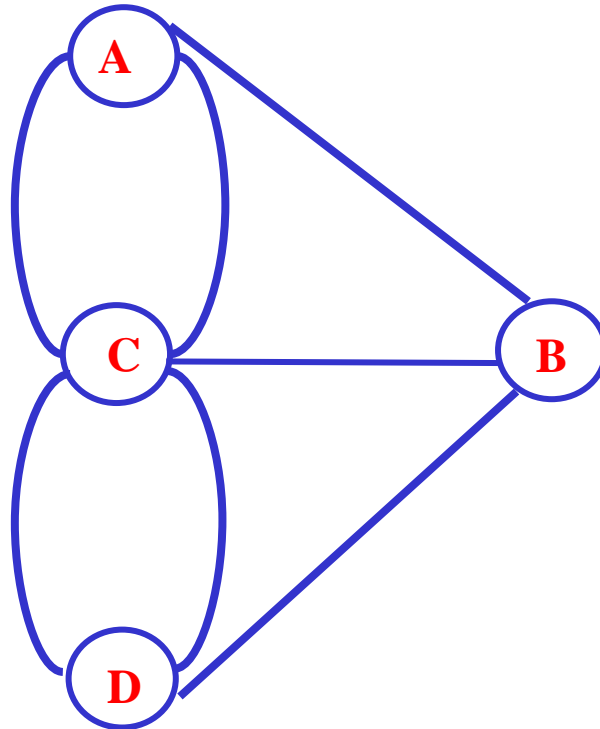
Multi-grafi orientati : matrice di adiacenza

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

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



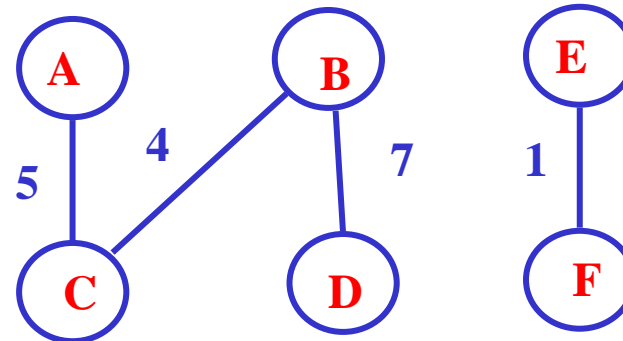
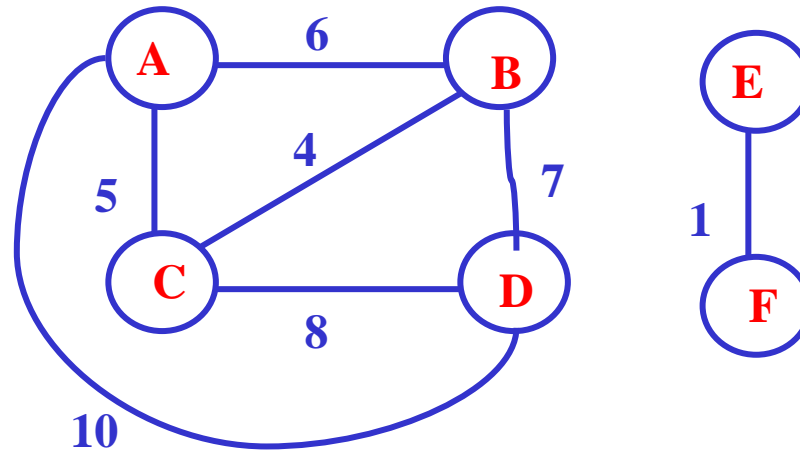
Esempio di multi-grafo non orientato



Minimo albero di copertura

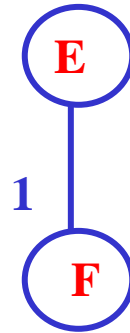
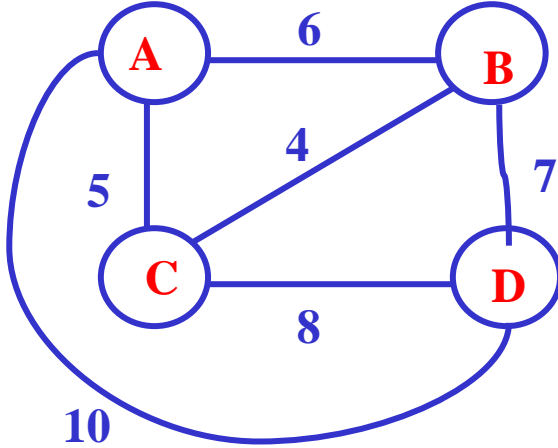
- Un grafo non orientato è **connesso** se esiste un cammino fra due nodi qualsiasi del grafo
- **Componente connessa**: sottografo connesso
- Componente connessa **massimale**: nessun nodo è connesso ad un'altra componente connessa
- **Albero di copertura**: insieme di componenti connesse massimali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

Minimo albero di copertura



algoritmo di **Kruskal** per trovare il minimo albero di copertura

1. **Ordina** gli archi del grafo in ordine crescente, considera una componente per nodo
2. **Scorri** l'elenco ordinato degli archi:
per ogni arco **a**
 if (**a** connette due componenti non connesse) {
 scegli **a**;
 unifica le componenti;
 }



(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)



{A} {B} {C} {D} {E} {F}



A

B

E

(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)

C

D

F

{A} {B} {C} {D} {E, F}

A

B

E

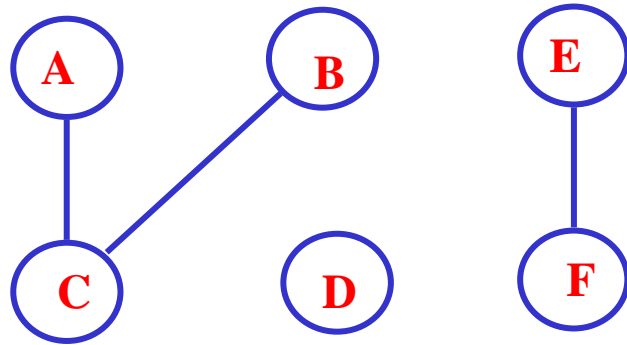
(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)

C

D

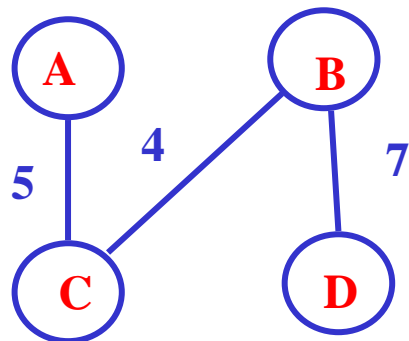
F

{A} {B, C} {D} {E, F}



(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)

{A, B, C} {D} {E, F}

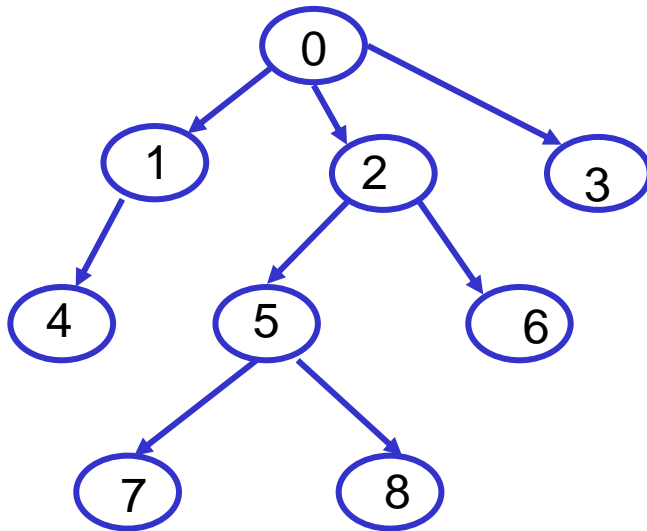


(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)

{A, B, C, D} {E, F}

Lunghezza: 17

Rappresentazione di un albero generico mediante un array



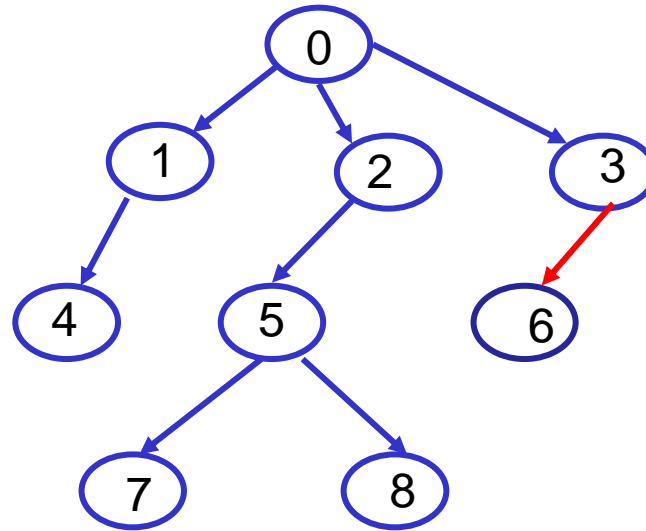
tree[i] è il padre di i

indice

tree

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

Esempio: 6 diventa figlio di 3



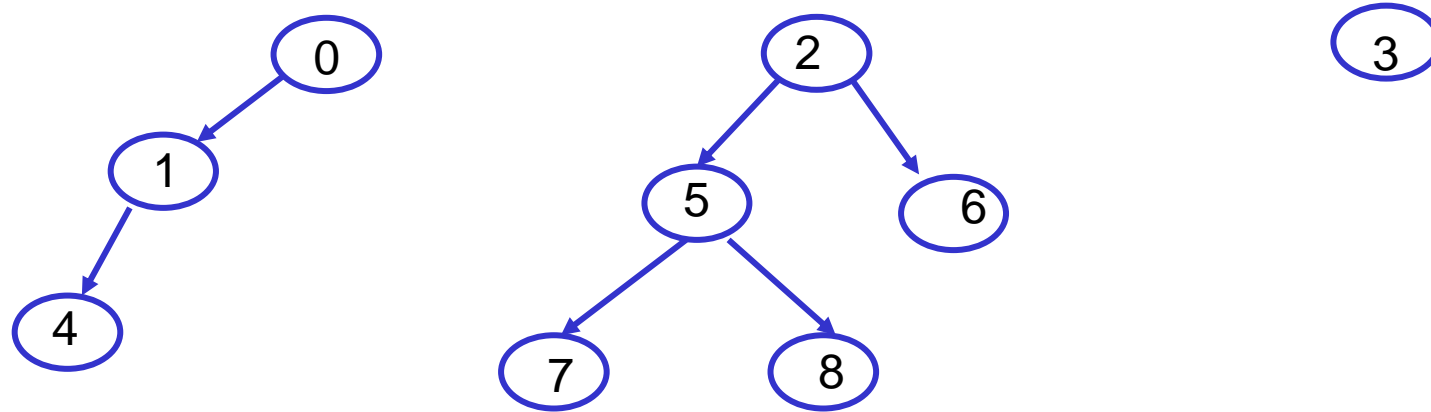
tree[6]=3;

indice

tree

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

Rappresentazione di una foresta di alberi mediante un array



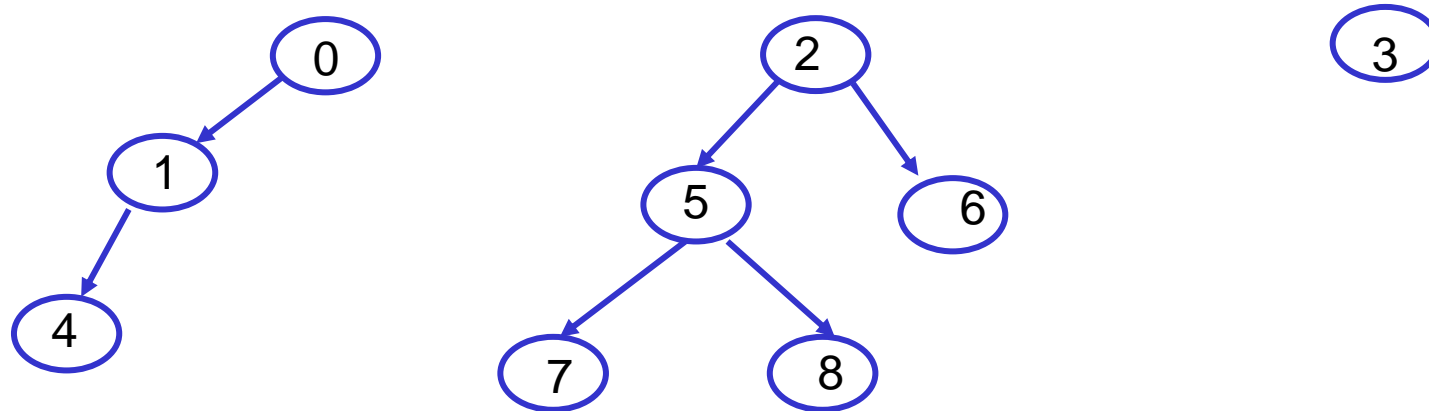
indice

tree

livello

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

Controllo di appartenenza allo stesso albero



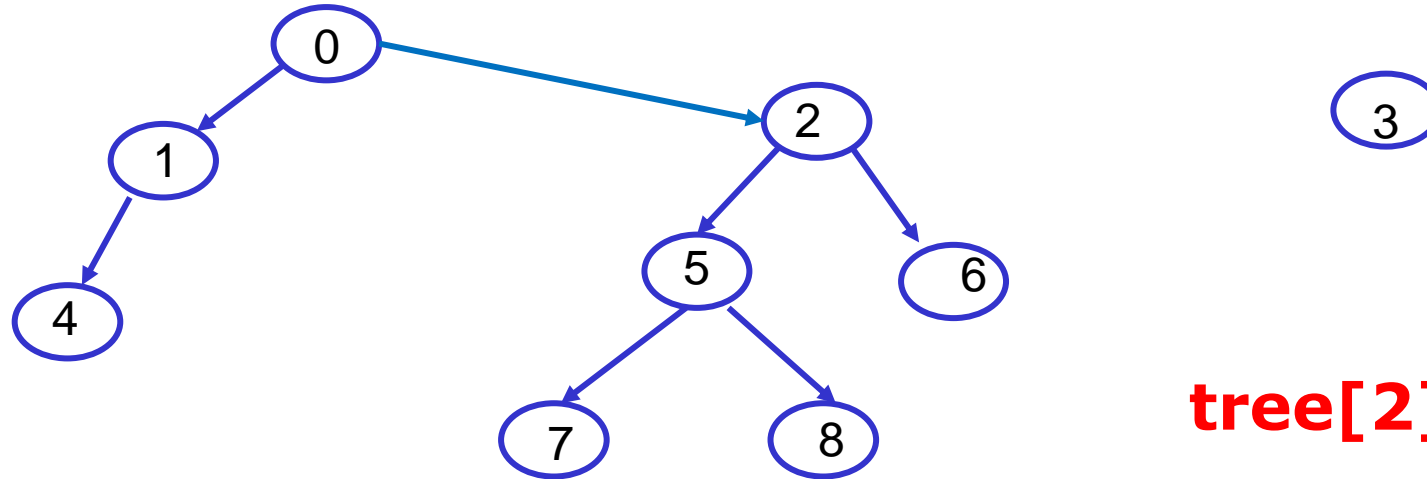
Per scoprire se due nodi appartengono allo stesso albero
risalgo da ogni nodo ai suoi antecedenti:

ok se trovo un antecedente comune

no se arrivo a due radici diverse

Con la rappresentazione mediante array vengono eseguiti
al massimo **tanti passi quanti sono i livelli dell'albero che
ha livello maggiore**

Fusione di alberi



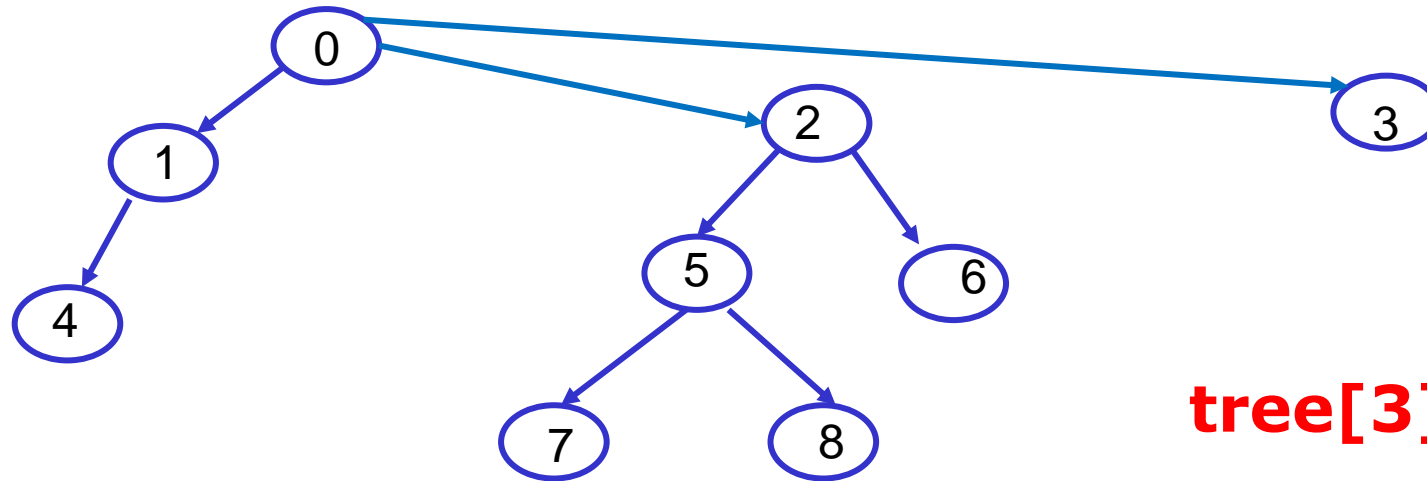
indice

tree

livello

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

Fusione alberi



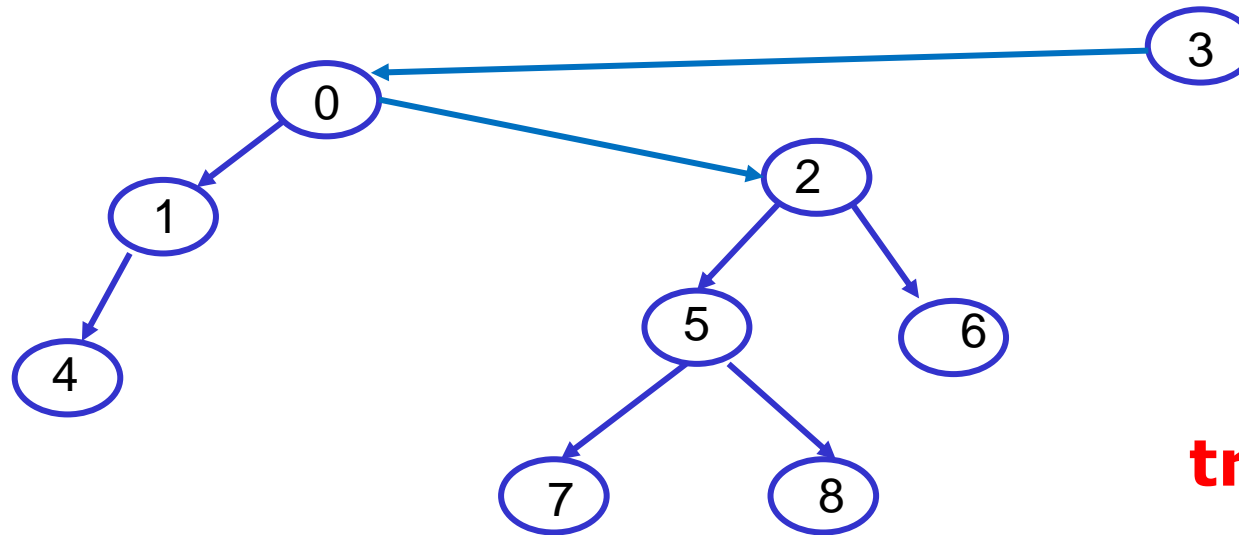
indice

tree

livello

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

Fusione di alberi



tree[0]=3;

indice

tree

livello

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

Fusione di alberi

**Inserire l'albero con livello minore come
sottoalbero della radice di quello con livello
maggiore mantiene gli alberi meno profondi**

Implementazione Kruskal

I nodi sono numerati

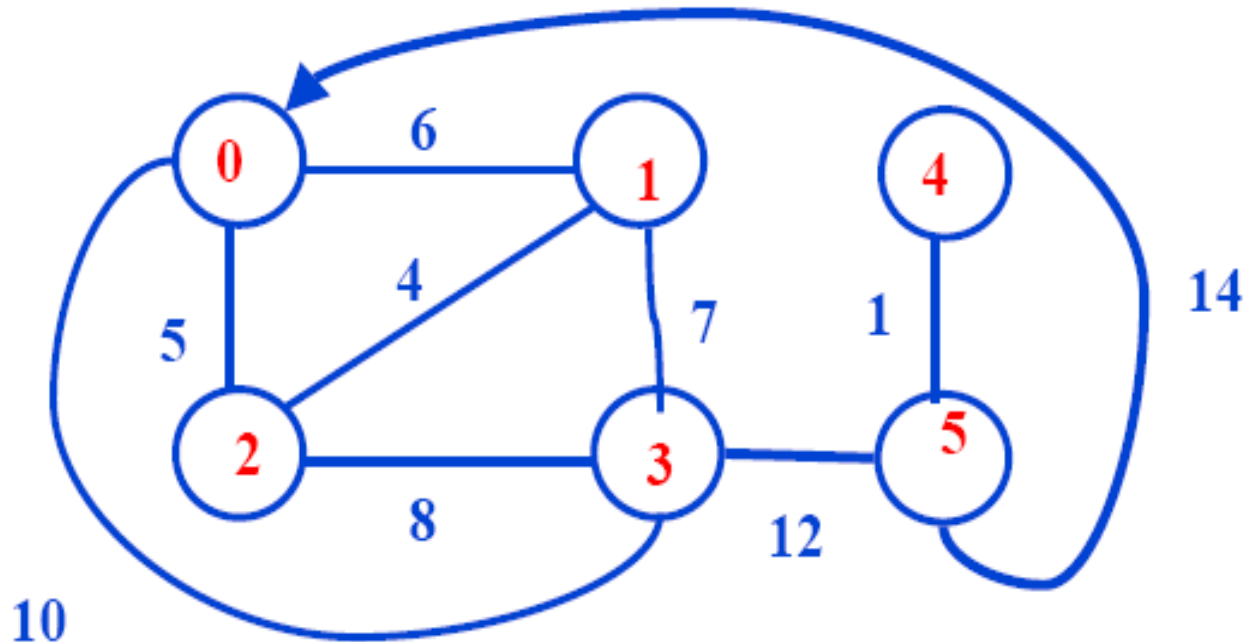
**Le componenti connesse sono memorizzate come
insiemi di alberi generici**

Sono memorizzate in array: ogni nodo punta al padre

**Se due nodi appartengono alla stessa componente
risalendo si incontra un antenato comune**

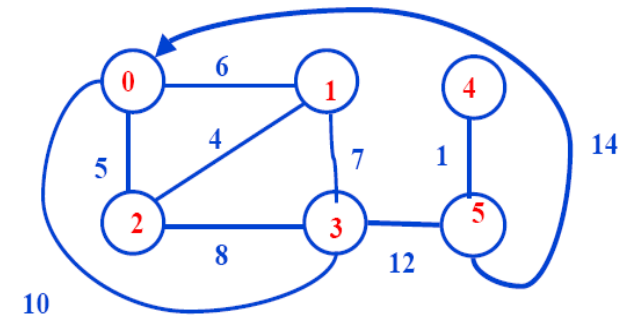
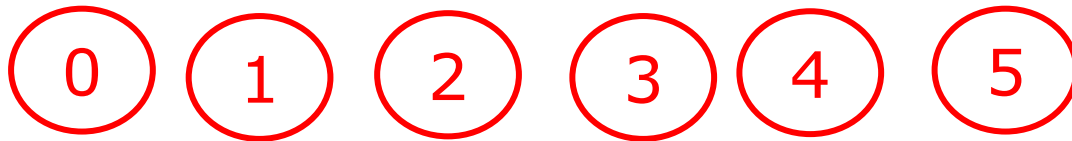
**Due alberi sono unificati inserendo quello meno
profondo (con livello minore) come sottoalbero della
radice di quello più profondo**

esempio



esempio

{0} {1} {2} {3} {4} {5}



indice

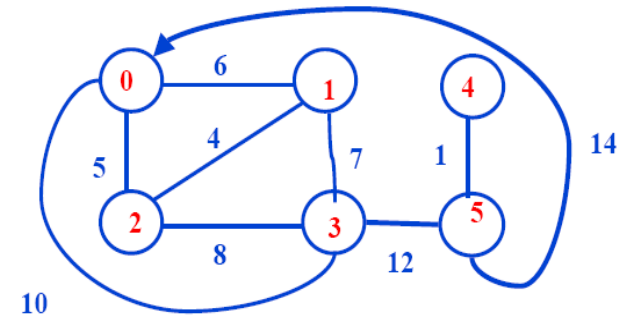
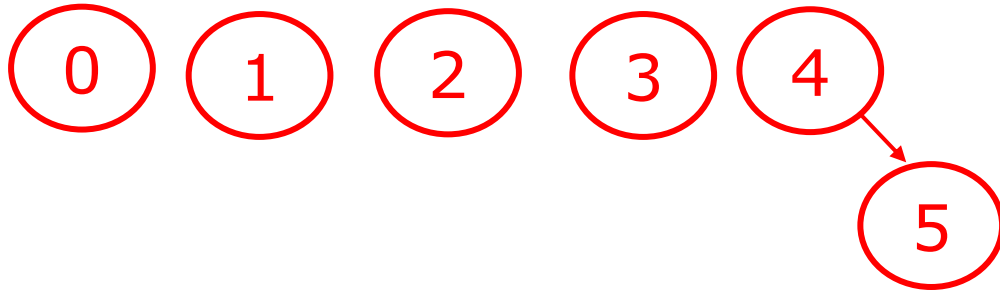
tree

livello

0	1	2	3	4	5
-	-	-	-	-	-
0	0	0	0	0	0

esempio

{0} {1} {2} {3} {4, 5}



Arco (4, 5)

indice

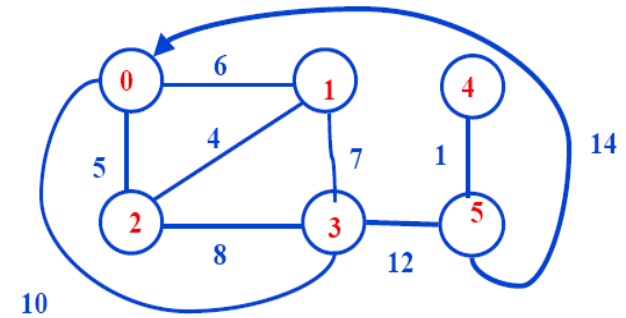
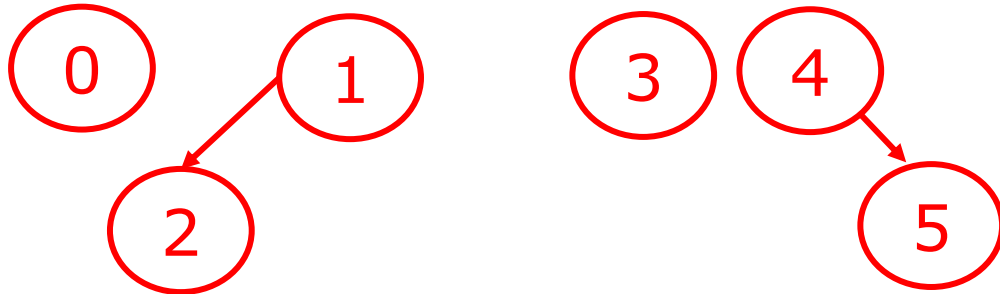
tree

livello

0	1	2	3	4	5
-	-	-	-	-	4
0	0	0	0	1	

esempio

{0} {1, 2} {3} {4, 5}



Arco (1, 2)

indice

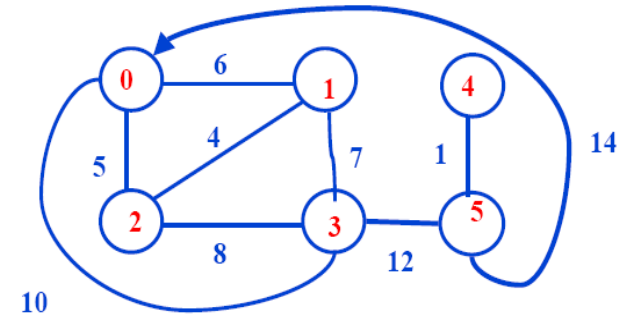
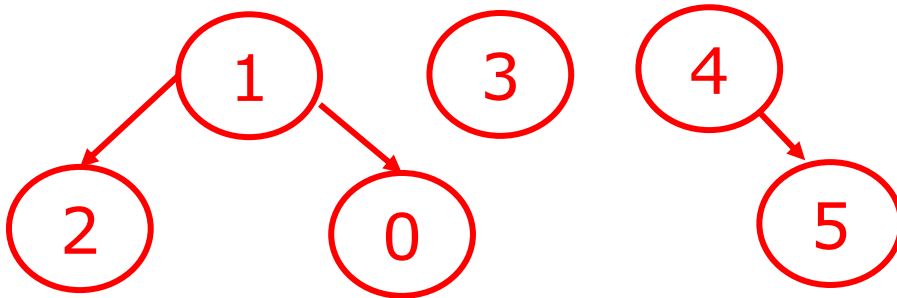
tree

livello

0	1	2	3	4	5
-	-	1	-	-	4
0	1		0	1	

esempio

{0, 1, 2} {3} {4, 5}



Arco (0, 2)

indice

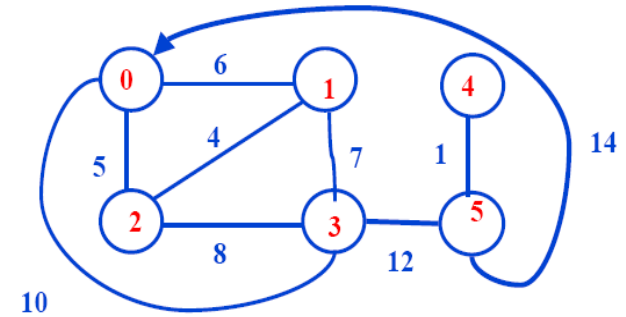
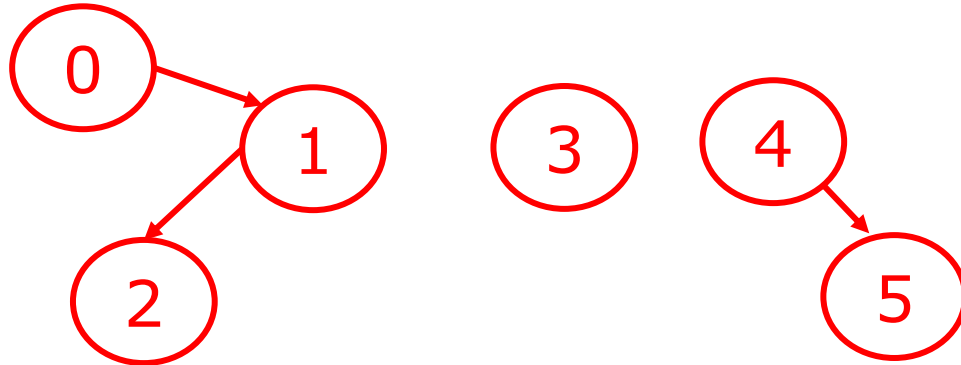
tree

livello

0	1	2	3	4	5
1	-	1	-	-	4
	1		0	1	

Scelta sbagliata (aumenta il livello)

{0, 1, 2} {3} {4, 5}



Arco (0, 2)

indice

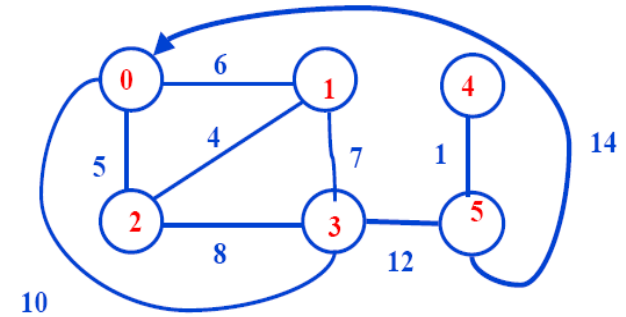
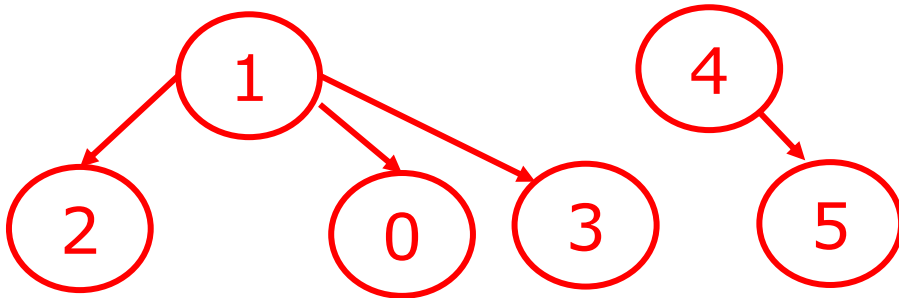
tree

livello

0	1	2	3	4	5
-	0	1	-	-	4
2			0	1	

esempio

{0, 1, 2, 3} {4, 5}



Arco (1,3)

indice

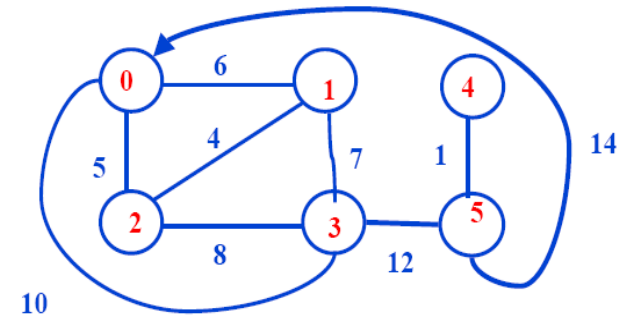
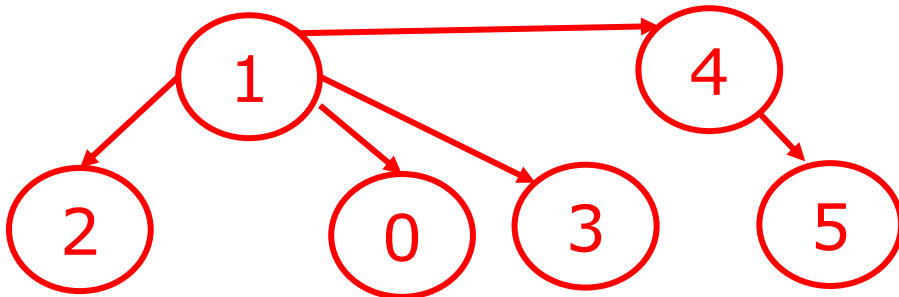
tree

livello

0	1	2	3	4	5
1	-	1	1	-	4
	1			1	

esempio

{0, 1, 2, 3, 4, 5}



Arco (3, 5)

indice

tree

livello

0	1	2	3	4	5
1	-	1	1	1	4
	2				

Inserire l'albero con livello minore come sottoalbero della radice di quello con livello maggiore ha come conseguenza che il controllo di appartenenza ad una stessa componente ha complessità $O(\log n)$

Complessità dell'algoritmo di Kruskal

1. **Ordina** gli archi del grafo in ordine crescente, considera una componente per nodo $O(m \log m)$
2. **Scorri** l'elenco ordinato degli archi:
per ogni arco a
if (a connette due componenti non connesse) { $O(\log n)$
scegli a ;
unifica le componenti; $O(1)$
}

Numero iterazioni del ciclo: $O(m)$

$O(m \log m) + O(m \log n)$

$O(m \log n)$

(m è $O(n^2)$)

algoritmo di Dijkstra

Si applica ai grafi orientati che hanno peso positive sugli archi

Trova i cammini minimi da un nodo di partenza a tutti gli altri nodi

Basato sulla metodologia greedy

algoritmo di Dijkstra

I cammini minimi, inizialmente posti a “infinito”, vengono aggiornati con stime via via più precise tramite un ciclo nel quale, ad ogni iterazione, un nuovo nodo viene “sistemato”, nel senso che il cammino minimo per lui viene stabilizzato. Alla fine tutti i nodi sono sistemati.

algoritmo di Dijkstra

Utilizza due tabelle **dist** (distanza) e **pred** (predecessore) con **n** elementi (**n**=numero dei nodi)

Per ogni nodo **A**, **dist (A)** contiene in ogni momento la lunghezza di un cammino dal nodo iniziale ad **A** e **pred(A)** il predecessore di **A** in questo cammino.

I nodi sono divisi in due gruppi: quelli già sistemati, per i quali i valori delle tabelle **dist** e **pred** sono definitivi, e quelli da sistemare (insieme **Q**). Per i nodi sistemati, **dist** contiene la lunghezza del minimo cammino e **pred** permette di ricostruirlo. Per gli altri, **dist** e **pred** contengono dati relativi al cammino trovato fino a quel momento, che eventualmente possono essere cambiati se si trova un cammino più corto.

algoritmo di Dijkstra

Inizialmente nessun nodo è sistemato: Q contiene tutti i nodi.

Si esegue poi un ciclo di n passi. Ad ogni passo

- 1. si considera "sistemato" il nodo, fra quelli di Q, con dist minore e lo si toglie da Q;**
- 2. si aggiornano pred e dist per gli immediati successori di questo nodo;**

Il ciclo termina dopo n-1 passi, quando Q contiene un solo nodo.

algoritmo di Dijkstra

algoritmo di Dijkstra

```
1  Q = N;  
2  per ogni nodo p diverso da p0 {           // O(n)  
    dist(p)=infinito, pred(p)=vuoto;  
}  
dist(p0)=0;
```

algoritmo di Dijkstra

```
4  while (Q contiene più di un nodo) {  
5      estrai da Q il nodo p con minima dist(p);  // O(logn)  
6      per ogni nodo q successore di p {  
          lpq=lunghezza dell'arco (p,q);  
          if (dist(p)+lpq < dist(q)) {                //O(1)  
              dist(q)=dist(p)+lpq;                    //O(1)  
              pred(q)=p;                               //O(1)  
7          re-inserisci in Q il nodo q modificato; // O(logn)  
      }  
  }
```


algoritmo di Dijkstra

L'insieme Q è memorizzato in un **min-heap**. Di conseguenza i comandi

5 estrai da Q il nodo **p** con minima $\text{dist}(\mathbf{p})$;

7 re-inserisci in Q il nodo **q** modificato;

hanno complessità **$O(\log n)$**

Numero iterazioni del ciclo while : n

Complessità iterazione: $C[5] + m/n C[7]$

$= O(\log n + (m/n) \log n)$

Complessità del ciclo: $O(n(\log n + (m/n) \log n)) =$

$O(n \log n + m \log n)$

Perchè l'algoritmo di Dijkstra funziona

Ad ogni iterazione del ciclo i nodi già scelti (eliminati da Q) sono "sistemati":

per i nodi già scelti *dist* contiene la lunghezza del cammino minimo e *pred* permette di ricostruirlo.

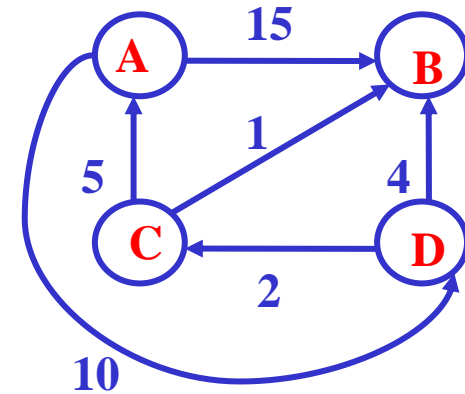
Il cammino minimo per i nodi già scelti *passa soltanto da nodi già scelti*

esempio

dist/pred

A	B	C	D
0 -	inf -	inf -	inf -

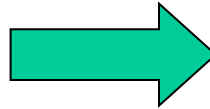
$Q = \{A, B, C, D\}$



estraggo **A**: $\text{dist}(A)=0$

$\text{dist}(A) + |(A, B)| < \text{dist}(B)$
 $0 + 15 < \text{inf.}$

$\text{dist}(A) + |(A, D)| < \text{dist}(D)$
 $0 + 10 < \text{inf.}$



$\text{dist}(B)=15, \text{pred}(B)=A$



$\text{dist}(D)=10, \text{pred}(D)=A$

A	B	C	D
0 -	15 A	inf -	10 A

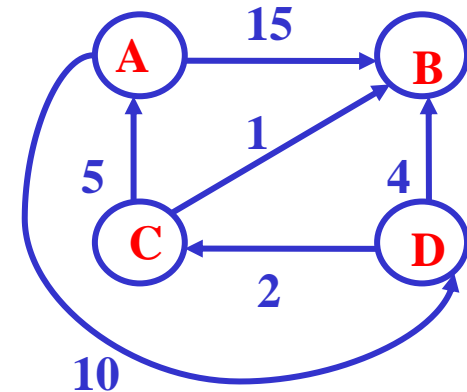
$Q = \{B, C, D\}$

esempio

dist/pred

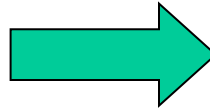
$Q = \{ B, C, D \}$

A	B	C	D
0 -	15 A	inf -	10 A



estraggo **D**: $\text{dist}(D)=10$

$\text{dist}(D) + |(D, \mathbf{B})| < \text{dist}(B)$
 $10 + 4 < 15$



$\text{dist}(B)=14, \text{pred}(B)=D$

$\text{dist}(D) + |(D, \mathbf{C})| < \text{dist}(C)$
 $10 + 2 < \text{inf.}$



$\text{dist}(C)=12, \text{pred}(C)=D$

A	B	C	D
0 -	14 D	12 D	10 A

$Q = \{ B, C \}$

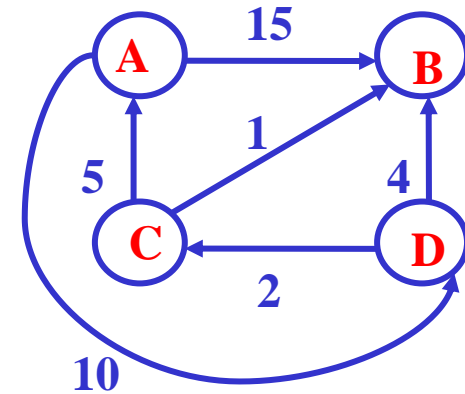
esempio

dist/pred

A B C D

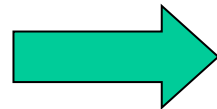
0 -	14 D	12 D	10 A
-------	--------	--------	--------

$Q = \{ B, C \}$



estraggo **C**: $\text{dist}(C)=12$

$\text{dist}(C) + |(C,B)| < \text{dist}(B)$
 $12 + 1 < 14$



$\text{dist}(B)=13, \text{pred}(B)=C$

A B C D

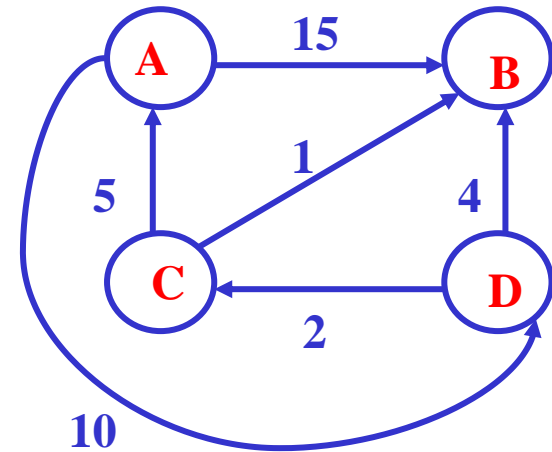
0 -	13 C	12 D	10 A
-------	---------------	--------	--------

$Q = \{ B \}$

soluzione

A	B	C	D
0 -	13 C	12 D	10 A

da A a B: A->D->C ->B **lung=13**
da A a C: A->D->C **lung=12**
da A a D: A->D **lung=10**



Nodo scelto	Q	A	B	C	D
	A, B, C, D	0 /-	i /-	i /-	i /-
A	B, C, D	0 /-	15/A	i /-	10/A
D	B, C	0 /-	14/D	12/D	10/A
C	B	0/-	13/C	12/D	10/A

dist/pred

Algoritmo PageRank di Google (cenni)

- **Serve al motore di ricerca per trovare le pagine web di interesse per una interrogazione**
- **Si basa sulle connessioni (link) fra le pagine**
- **Considera la rete come un **grafo** in cui le pagine sono i nodi e i link sono gli archi**

Algoritmo PageRank di Google

- Calcola il «**rango**» (rilevanza) di una pagina web P : **$R(P)$**
- La rilevanza di P dipende da quanto sono rilevanti le pagine che puntano a P (hanno un link verso P) ma anche da quanti link escono da queste pagine

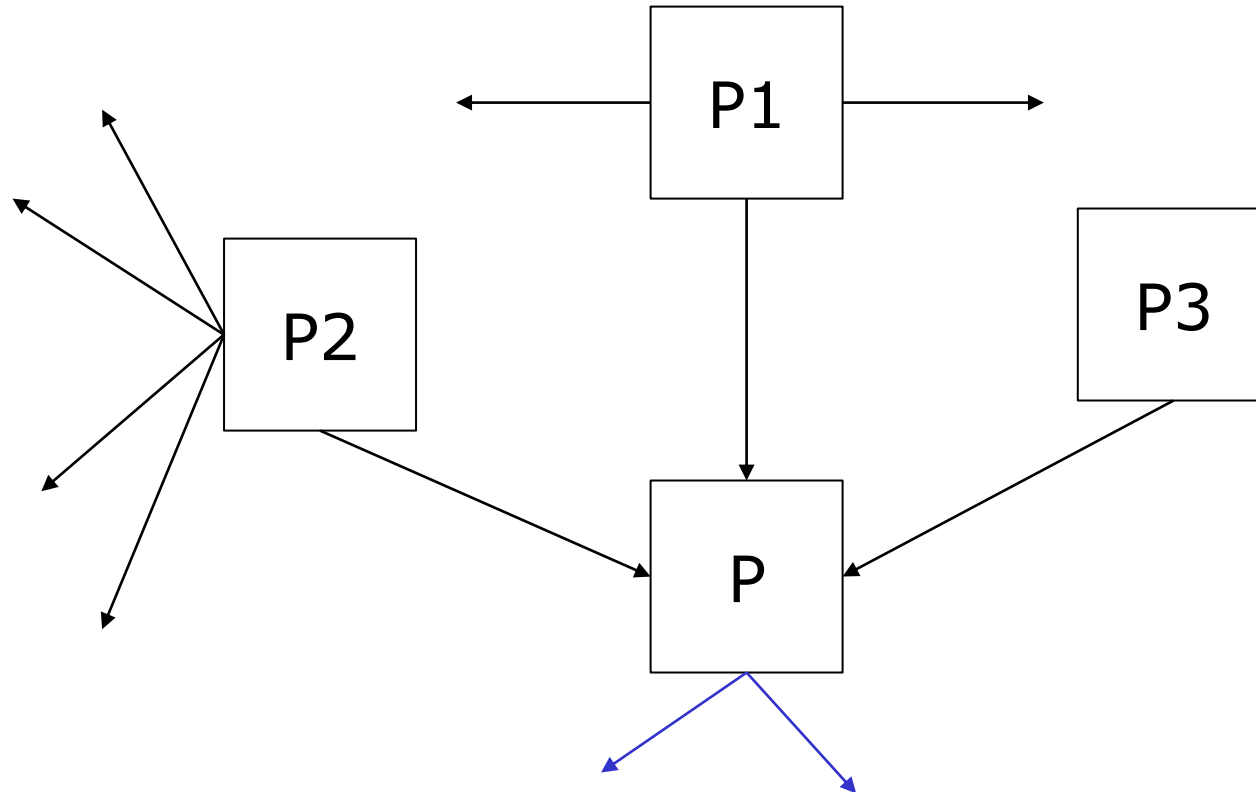
Algoritmo PageRank: formula base

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

$|Q|$ = numero di link uscenti da Q

$Q \rightarrow P$: link da Q a P

Algoritmo PageRank: esempio



$$R(P1) = 10$$

$$R(P2) = 10$$

$$R(P3) = 6$$

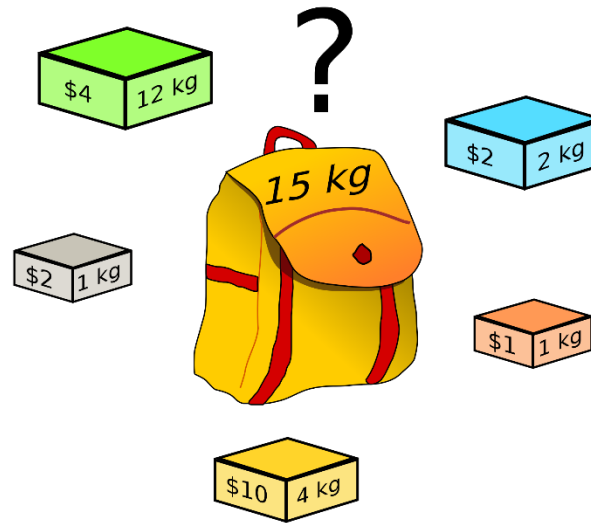
$$R(P) = 10/3 + 10/5 + 6/1 = 11,3$$

Algoritmo PageRank

- **Il calcolo viene fatto calcolando il rango dei nodi in modo iterativo utilizzando la matrice di adiacenza e partendo da un valore del rango uguale per tutti i nodi**
- **Sono necessari aggiustamenti della formula per assicurarne la convergenza (cicli, nodi pozzo)**

Problemi difficili: cenni alla NP-completezza

Problemi difficili: zaino



Ottimizzare il riempimento dello zaino:

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

Problemi difficili : commesso viaggiatore



trovare il percorso di minore lunghezza che un commesso viaggiatore deve seguire per visitare tutte le città una e una sola volta per poi tornare alla città di partenza

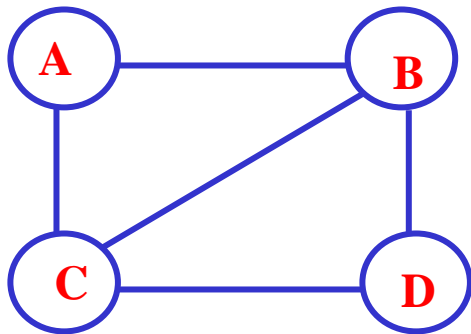
Problemi difficili : ciclo Hamiltoniano

Willian Rowan Hamilton (1859)

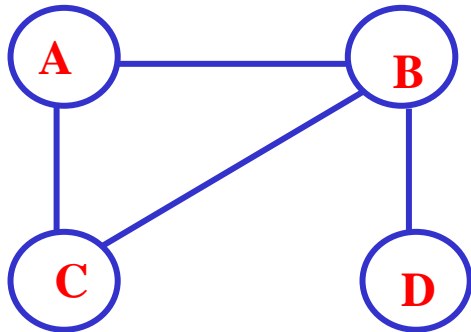
HC: Dato un multi-grafo, trovare, se esiste, un ciclo che tocca tutti i **nodi una e una sola volta**

Un grafo è Hamiltoniano se possiede un ciclo Hamiltoniano

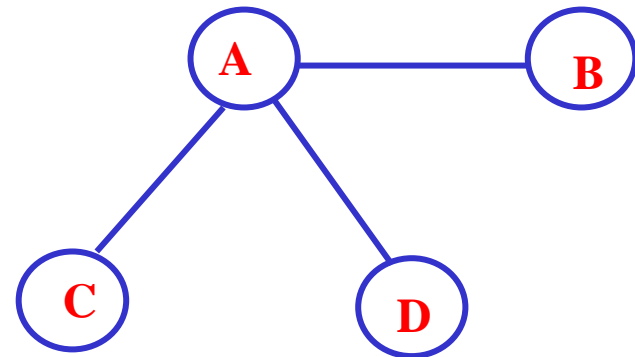
ciclo Hamiltoniano



ABDCA

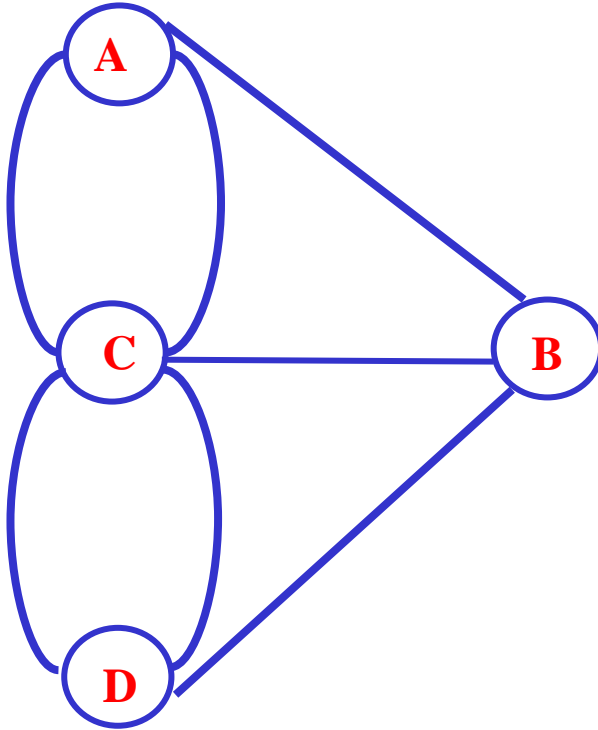


NO



NO

ciclo Hamiltoniano



ACDBA

SAT : soddisfattibilità di una formula nella logica dei predicati

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

Es.

$F = (x \text{ and not } x) \text{ or } (y \text{ and not } y)$ $n=2$ non sodd.

$F = (x \text{ and not } y) \text{ or } (\text{not } x \text{ and } y)$ $n=2$ $x=0, y=1$

Problemi difficili

algoritmi conosciuti oggi per zaino, commesso viaggiatore, ciclo Hamiltoniano e SAT:

provare tutte (o quasi) le combinazioni



complessità esponenziale

**Se le variabili che compaiono nella formula sono n ,
si provano tutte le combinazioni di valori delle
variabili, che sono 2^n**

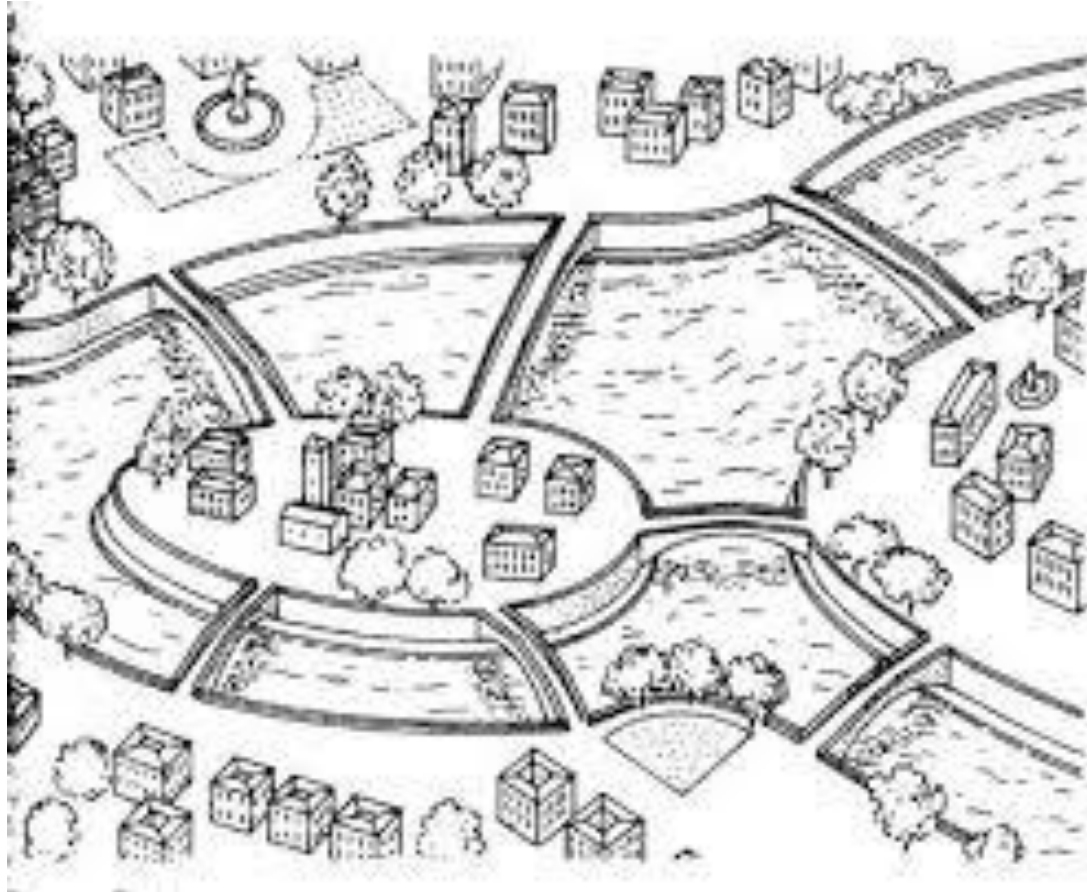
Problema del ciclo Euleriano

Eulero (1736)

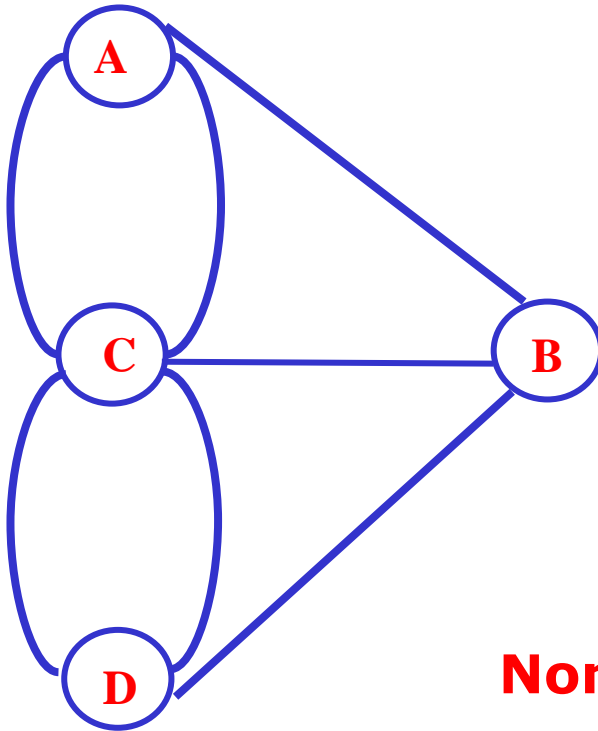
Dato un multi-grafo, trovare, se esiste, un ciclo che percorre tutti gli **archi una e una sola volta**

Un grafo è Euleriano se possiede un ciclo Euleriano

I ponti di Königsberg



I ponti di Königsberg: esiste un ciclo Euleriano?



Non esiste un ciclo Euleriano

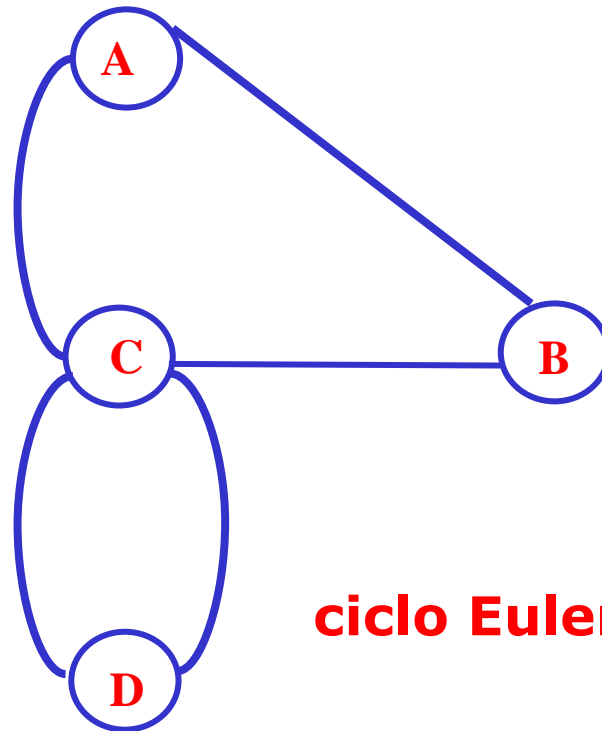
TEOREMA DI EULERO

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

Quindi basta controllare questa proprietà sul grafo ($O(m)$) per sapere se un ciclo esiste.

Trovarlo ha complessità polinomiale.

ciclo Euleriano



ciclo Euleriano: ABCDCA

Teoria della NP-completezza

Classifica un insieme di problemi difficili ma non dimostrati come esponenziali (come Hanoi, permutazioni, ...)

Si applica a problemi decisionali: con risposta SI o NO

Ogni problema può essere riformulato come problema decisionale.

Il problema decisionale ha complessità minore o uguale al problema non decisionale corrispondente.

Quindi se il problema decisionale è difficile, a maggior ragione lo sarà il corrispondente.

Problemi decisionali

Commesso viaggiatore: dato un intero k , esiste nel grafo un ciclo senza ripetizione di nodi di lunghezza minore di k ?

Zaino: dato un valore v , esiste un riempimento dello zaino con valore maggiore o uguale a v ?

Ciclo Hamiltoniano: dato un grafo, esiste un ciclo Hamiltoniano?

Ciclo Euleriano: dato un grafo, esiste un ciclo Euleriano? $O(m)$

Formula logica: data una formula, esiste un assegnamento alle variabili che rende vera la formula?

Algoritmi **nondeterministici**

Si aggiunge il comando

choice(I)

dove **I** è un insieme

choice(I) sceglie nondeterministicamente un elemento dell'insieme **I**

Un algoritmo nondeterministico per la **soddisfattibilità**

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

$O(n)$

Ritorna 1 se esiste almeno una scelta che con risultato 1

Un algoritmo nondeterministico di **ricerca** in array

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

$O(1)$

Un algoritmo nondeterministico di **ordinamento**

```
int nsort(int* a, int n) {  
    int b [n];  
    for (int i=0; i<n; i++)  
        b[i]=a[i];  
    for (int i=0; i<n; i++)  
        a[i]=b[choice({0..n-1})];  
    if (ordinato(a))  
        return 1;  
    return 0;  
}
```

$O(n)$

Relazione fra determinismo e nondeterminismo

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

Se le soluzioni sono in numero esponenziale, l'algoritmo **deterministico** avrà complessità esponenziale.

P e NP

P = insieme di tutti i problemi decisionali risolubili in tempo **Polinomiale** con un algoritmo deterministico

NP = insieme di tutti i problemi decisionali risolubili in tempo **Polinomiale** con un algoritmo **Nondeterministico**

NP : **N**ondeterministico **P**olinomiale

P e NP

P = { ricerca, ordinamento, ciclo Euleriano ... }

NP = { ricerca, ordinamento, fattorizzazione,
soddisfattibilità, zaino, commesso viaggiatore, ciclo
Hamiltoniano... }

Caratterizzazione alternativa della classe NP

NP = insieme di tutti i problemi decisionali che ammettono un **algoritmo polinomiale di verifica di una soluzione (certificato)**

Per dimostrare che un problema **R appartiene a NP si dimostra che la verifica di una soluzione di **R** è fatta in tempo polinomiale**

Esempi di verifica

Ciclo Hamiltoniano: si può verificare con complessità $O(n)$ se un cammino è un ciclo Hamiltoniano.

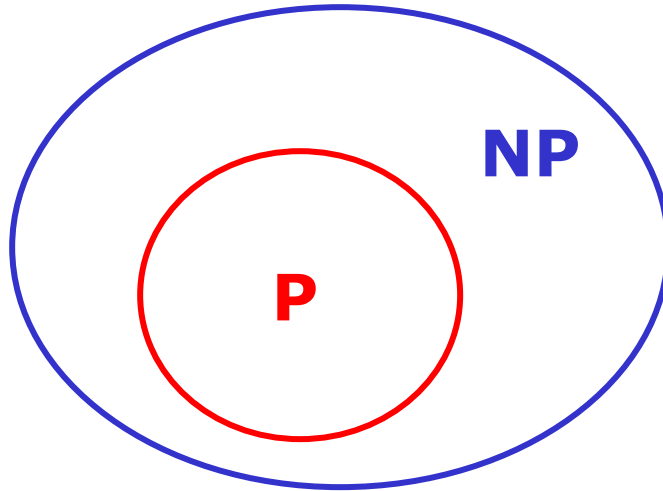
Formula logica: si può controllare in tempo $O(n)$ se dato un assegnamento di valori booleani alle variabili rende vera la formula

P e NP

P = problemi decisionali facili da **risolvere**

NP = problemi decisionali facili da **verificare**

P e NP



$P \subseteq NP$

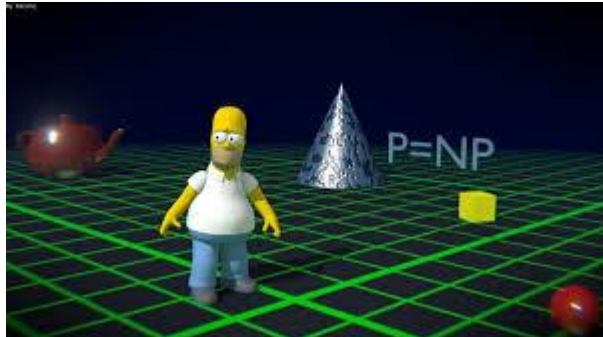
$P = NP ?$

P = NP?

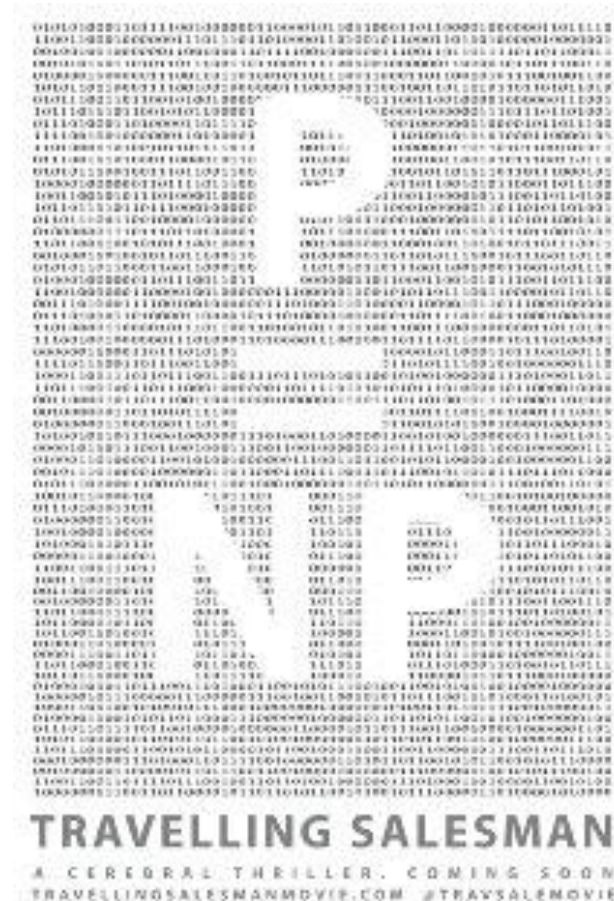
Uno dei 7 problemi del millennio (Clay Mathematical Institute, 2000)

Un milione di dollari per chi lo risolve (in un senso o nell'altro)

$$P = NP$$



«Macchine come me»
di Ian McEwan (2019)



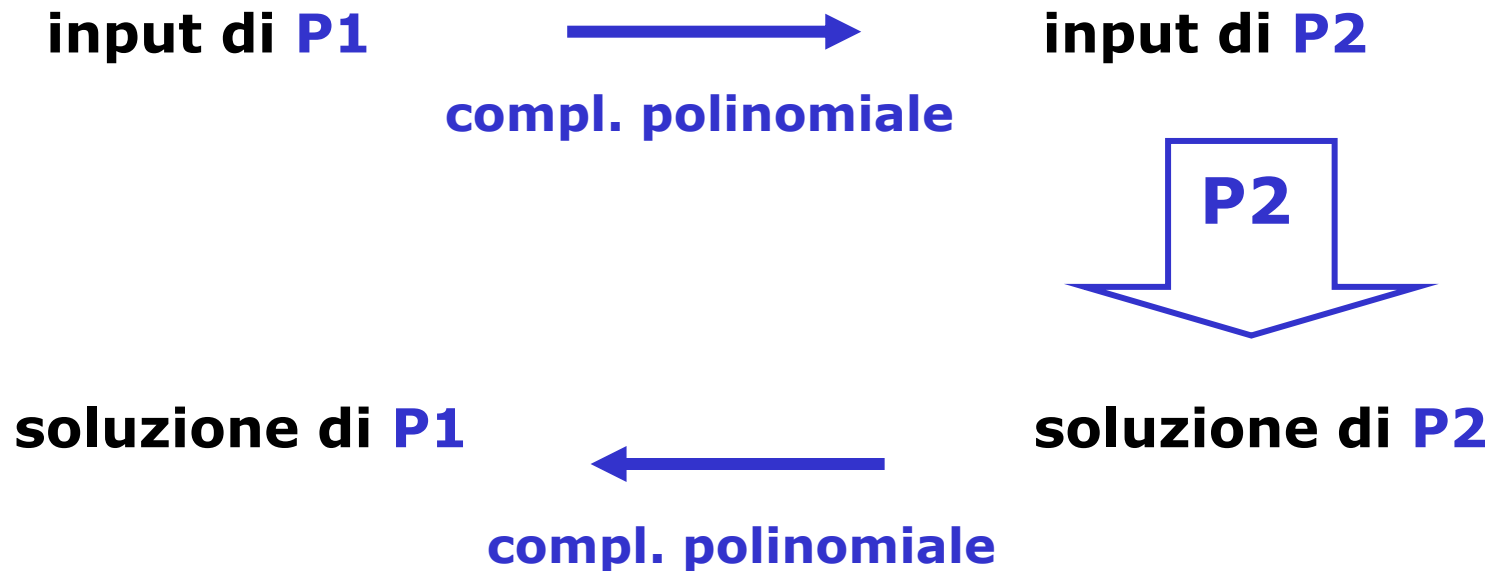
Film del 2012

riducibilità

La riducibilità è un metodo per convertire l'istanza di un problema **P1 in un'istanza di un problema **P2** e utilizzare la soluzione di quest'ultimo per ottenere la soluzione di **P1****

Riducibilità

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



$$\mathbf{P1 \leq P2}$$

Conseguenze della riducibilità

- **$P1 \leq P2$**
- **$P2$ è risolubile in tempo polinomiale**



$P1$ è risolubile in tempo polinomiale

Teorema di Cook

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

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

Quindi **SAT** è più difficile di tutti i problemi in NP

NP-completezza

Un problema **R** è **NP-completo** se

- **R** \in **NP** e
- **SAT** \leq **R**

NP-completezza

Se un problema è NP-completo, è **difficile tanto quanto SAT e può essere usato al posto di SAT nella dimostrazione di NP-completezza di un altro problema.**

I problemi NP-completi hanno tutti la stessa difficoltà e sono i più difficili della classe NP

Se si trovasse un algoritmo polinomiale per **SAT o per qualsiasi altro problema NP-completo, allora tutti i problemi in NP sarebbero risolubili in tempo polinomiale e quindi **P sarebbe uguale ad NP****

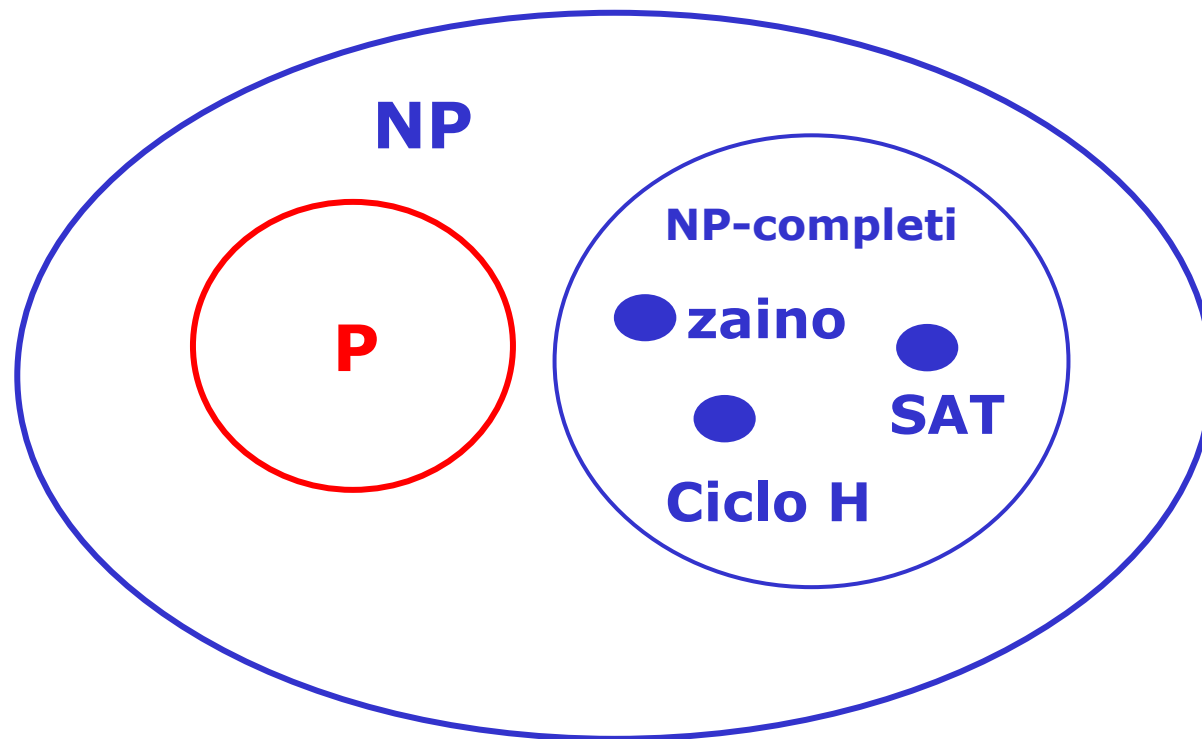
Problemi NP-completi

E' stato dimostrato che i seguenti problemi decisionali sono NP-completi:

- **Commesso viaggiatore**
- **Zaino**
- **Ciclo hamiltoniano**

Moltissimi altri problemi sono stati dimostrati NP-completi

Problemi NP-completi



Problemi NP-completi

Per dimostrare che un problema **R è NP-completo:**

dimostrare che R appartiene ad NP

individuare un algoritmo polinomiale nondeterministico per risolvere **R** (oppure dimostrare che la verifica di una soluzione di **R** può essere fatta in tempo polinomiale)

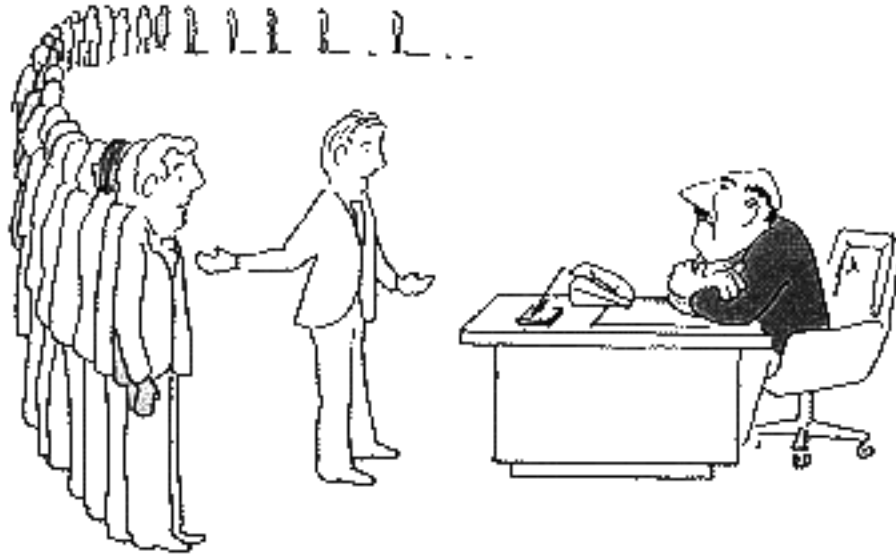
dimostrare che esiste un problema NP-completo che si riduce a R

se ne sceglie uno fra i problemi NP-completi noti che sia facilmente riducibile a **R**

Perché serve dimostrare che un problema è NP-completo?

Perché non riusciamo a risolverlo con un algoritmo polinomiale e vogliamo dimostrare che non ci si riesce a meno che P non sia uguale ad NP , problema tuttora non risolto

Utilizzo



I can't find an efficient algorithm, but neither can all these famous people.

Il problema della Fattorizzazione di un numero

FATT (Fattorizzazione): Scomposizione di un numero in fattori primi

Es: $150 = 2 \times 3 \times 5^2$

Come in tutti problemi di teoria dei numeri, la complessità si calcola in funzione del numero di cifre del numero da fattorizzare (dimensione dell'input)

Il problema della Fattorizzazione

La moltiplicazione è $O(n^{\log_2 3})$ o $O(n^2)$

1.634.733.645.809.253.848.443.133.883.865.090.859.841.783.670.033.0
92.312.181.110.842.389.333.100.104.508.151.212.118.167.511.579 **X**

1.900.871.281.664.822.113.126.851.573.935.413.975.471.896.789.968.5
15.493.666.638.539.088.027.103.802.104.498.957.191.261.465.571

=

3.107.418.240.490.043.721.350.750.035.888.567.930.037.346.022.842.7
27.545.720.161.948.823.206.440.518.081.504.556.346.829.671.723.286.
782.437.916.272.838.033.415.471.073.108.501.919.548.529.007.337.724
.822.783.525.742.386.454.014.691.736.602.477.652.346.609

Richiede meno di un secondo di tempo di calcolo!

Il problema della Fattorizzazione

L'inverso della moltiplicazione è difficile

Trovare A e B tali che

$A * B = 3.107.418.240.490.043.721.350.750.035.888.567.930.037.$
346.022.842.727.545.720.161.948.823.206.440.518.081.504.556.346.
829.671.723.286.782.437.916.272.838.033.415.471.073.108.501.919.
548.529.007.337.724.822.783.525.742.386.454.014.691.736.602.477.
652.346.609

A=1.634.733.645.809.253.848.443.133.883.865.090.859.841.783.670
.033.092.312.181.110.842.389.333.100.104.508.151.212.118.167.511
.579

B= 1.900.871.281.664.822.113.126.851.573.935.413.975.471.
896.789.968.515.493.666.638.539.088.027.103.802.104.498.957.191.
261.465.571

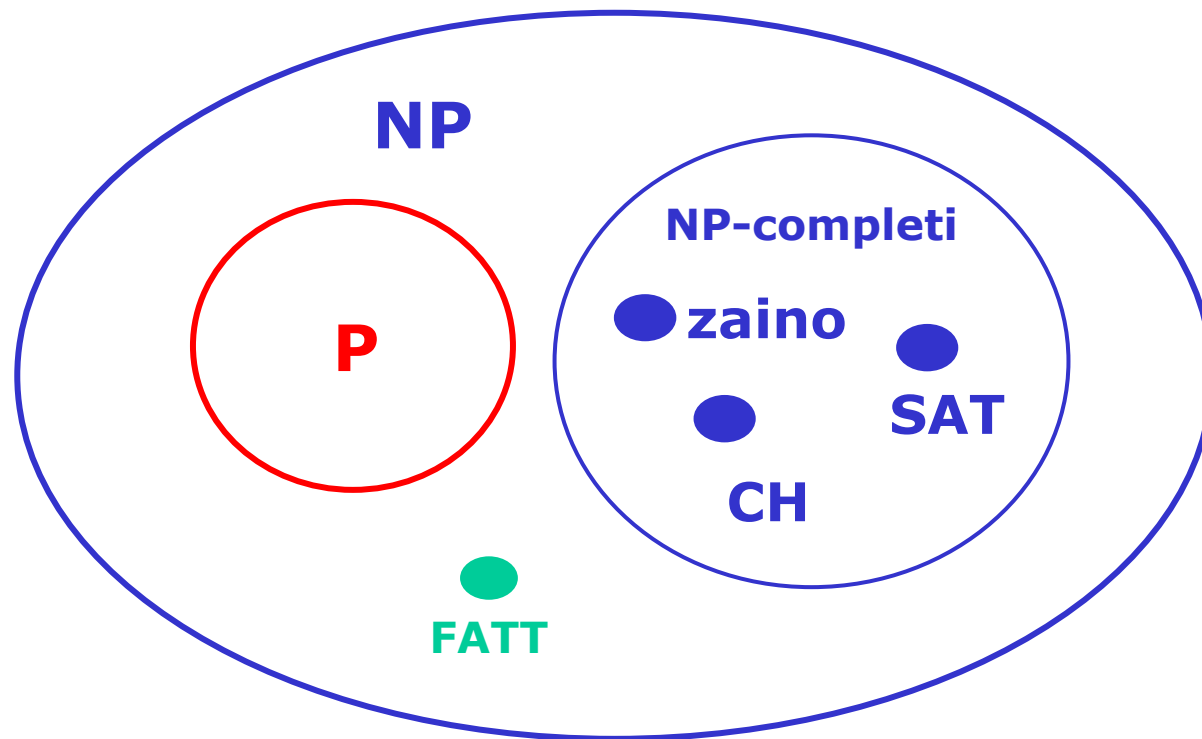
Oggi richiede più di una decina di anni di tempo di calcolo!

Il problema della Fattorizzazione

- Per ora si conoscono soltanto algoritmi **esponenziali** per FATT
- FATT è in NP: la verifica è polinomiale (moltiplicazione)
- E' questione aperta se FATT sia in P, ma quasi sicuramente non è NP-completo
- Praticamente impossibile scomporre un numero di 200 o più cifre decimali con gli algoritmi attuali

Il problema della Fattorizzazione

- Sulla difficoltà di FATT si basano i meccanismi della **crittografia a chiave pubblica** (operazioni facili con inversa difficile) che si basano su numeri primi molto grandi
- Un algoritmo polinomiale per FATT non dimostrerebbe $P=NP$, ma metterebbe in forte crisi i meccanismi della crittografia
- algoritmo polinomiale di Shor (1994) basato su **Quantum Computing**



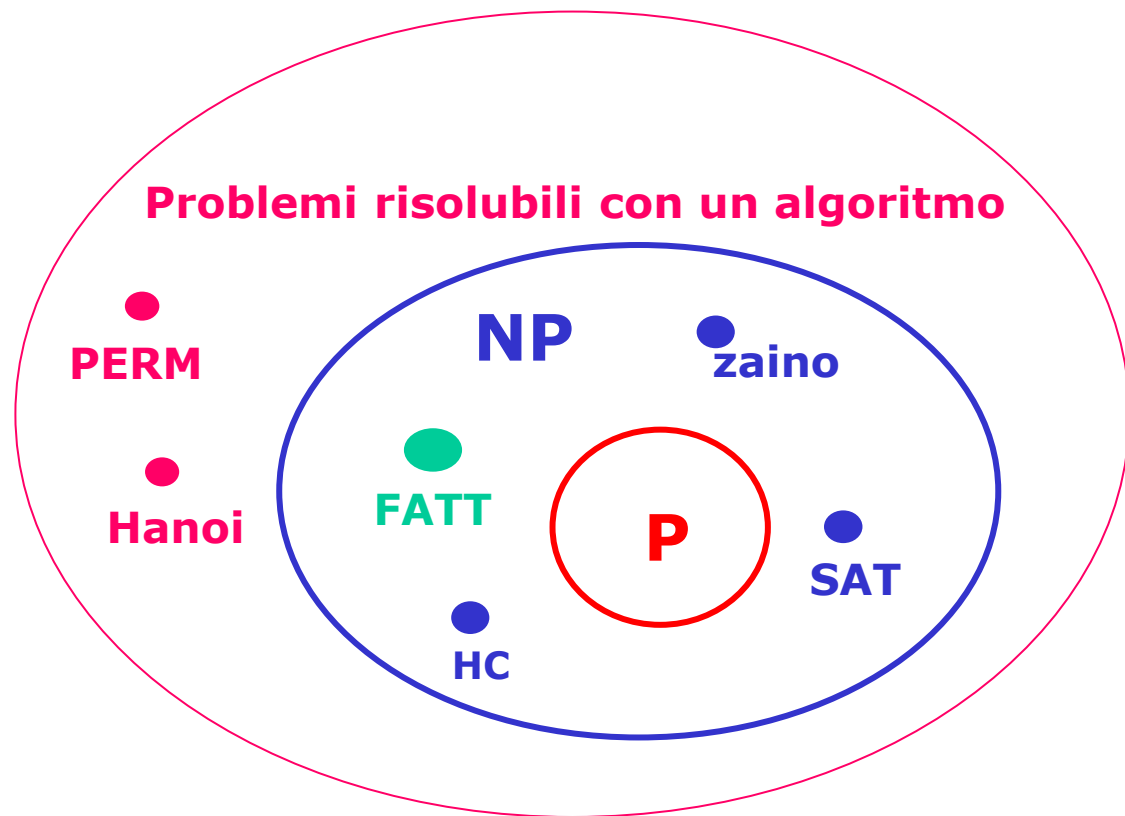
metodologie per affrontare i problemi difficili

- **Algoritmi di approssimazione**
- **Algoritmi probabilistici**
- **Algoritmi paralleli**
- **Reti neurali**
- **Quantum Computing**

Problemi non in NP

PERM: Trovare tutte le permutazioni di un insieme (n!)

Torre di Hanoi



Problemi **non risolubili** con un algoritmo

- **TERM: Decidere la terminazione di un programma su un input**
- **SAT-I: soddisfattibilità di una formula nella logica del I ordine**
- **EQ: Decidere l'equivalenza di due programmi**

Problemi

