

Capitolo 1: Il Tipo di Dato Puntatore

PUNTATORI E STRUTTURE DATI DINAMICHE:
ALLOCAZIONE DELLA MEMORIA E
MODULARITÀ IN LINGUAGGIO C



Prerequisiti

DA TECNICHE DI PROGRAMMAZIONE

TP: risultati attesi

Nozioni elementari di architettura e di logica

- Conoscenza dell'architettura e del modo di funzionamento di una CPU e della memoria centrale, con particolare riferimento all'indirizzamento dei dati in una memoria RAM
- Algebra di Boole e funzioni logiche

TP: risultati attesi

Problem-solving

- Conoscenza di nozioni elementari di analisi della complessità
- Conoscenza di nozioni elementari di problem-solving come risoluzione di problemi progettuali, basata su caratterizzazione di problemi dal punto di vista del contesto applicativo e/o delle strategie algoritmiche adottate
- Conoscenza di strategie algoritmiche elementari, ad esempio per la risoluzione di problemi iterativi di verifica basati sul principio dei quantificatori universali ed esistenziali, sull'uso di vettori come contenitori per collezioni di dati e/o per la realizzazione di tabelle ad accesso diretto
- Conoscenza degli algoritmi di ordinamento iterativi
- Abilità nel passare dalla formulazione di un problema algoritmico alla sua soluzione, basata su passi di riconoscimento del tipo di problema e scelta di adeguate strutture dati e algoritmi risolutivi
- Abilità nella soluzione di problemi di tipo iterativo, basati su dati scalari, oppure sulla collezione di dati, sia ordinati che non ordinati.

TP: risultati attesi

Linguaggio C

- Conoscenza dei costrutti base della programmazione in linguaggio C, quali: tipi di dato scalari, input/output elementare su standard I/O e su file testo, costrutti iterativi e condizionali, uso di funzioni come sotto-programmi, tipi di dato strutturato, struct, vettori e matrici, manipolazione di stringhe di caratteri
- Conoscenza del tipo di dato puntatore come riferimento a dati e suo utilizzo per vettori e matrici, nonché come strumento per realizzare in C il passaggio di parametri a funzioni per riferimento (o meglio “by pointer”)
- Abilità nel realizzare algoritmi iterativi, in grado di manipolare sia dati scalari che aggregati in tipi struct e/o vettori e matrici come collezioni di dati
- Abilità nel realizzare programmi modulari, basati su scomposizione di un problema in sotto-problemi risolti mediante un efficace utilizzo di funzioni
- Abilità di utilizzo di strumenti di ausilio alla programmazione.

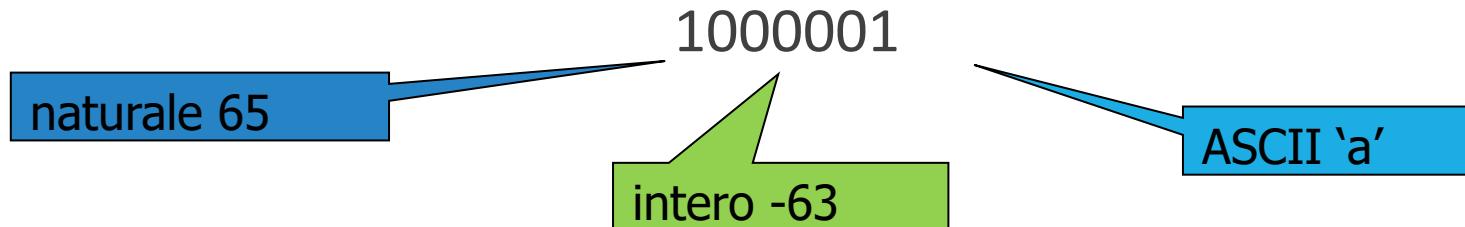
Il Tipo di Dato Puntatore

I dati in memoria centrale

Dati memorizzati come sequenze di 1 e 0 che codificano simboli di insiemi finiti:

- naturali, interi, razionali, caratteri

La sequenza ha significato solo se associata alla codifica:



Il modello della memoria

- **Memoria RAM:** matrice di bit con n righe e m colonne. Esempio: matrice da 128 bit:

- 32 righe x 4 colonne
- 16 righe x 8 colonne
- 8 righe x 16 colonne

In generale:

- n è una potenza di 2
- m è un multiplo di 8 (1 byte = 8 bit)

Cella e parola

■ Cella:

- gruppo di k bit cui si accede unitariamente
- in generale $k = 8 \Rightarrow 1$ byte
- cella da 1 byte \Rightarrow memoria byte-addressable
- identificata da un indirizzo: N celle \Rightarrow indirizzi tra 0 e $N-1$
- Indirizzo: stringa di $\lceil \log_2 N \rceil$ bit

■ Parola (word):

- raggruppamento di celle
- in generale occupa 4 o 8 byte
- può stare su 1 riga o su più righe successive
- raramente RAM word-addressable.

Big/Little Endian

Parole (word) su più celle:

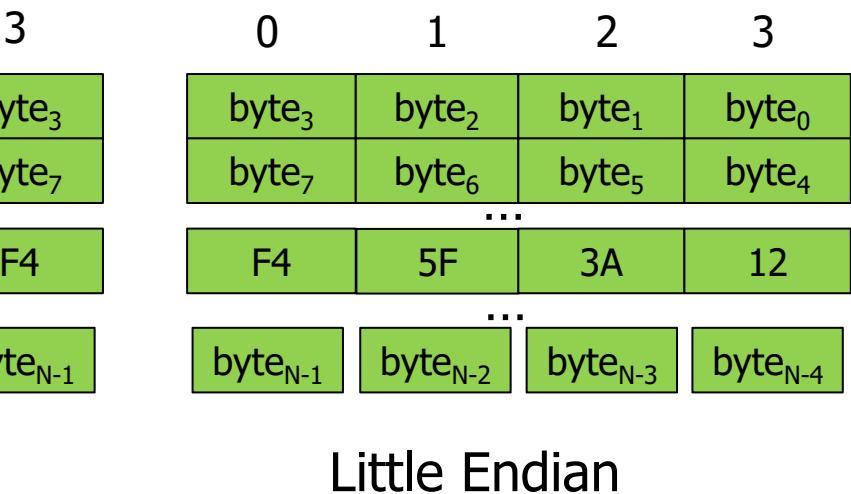
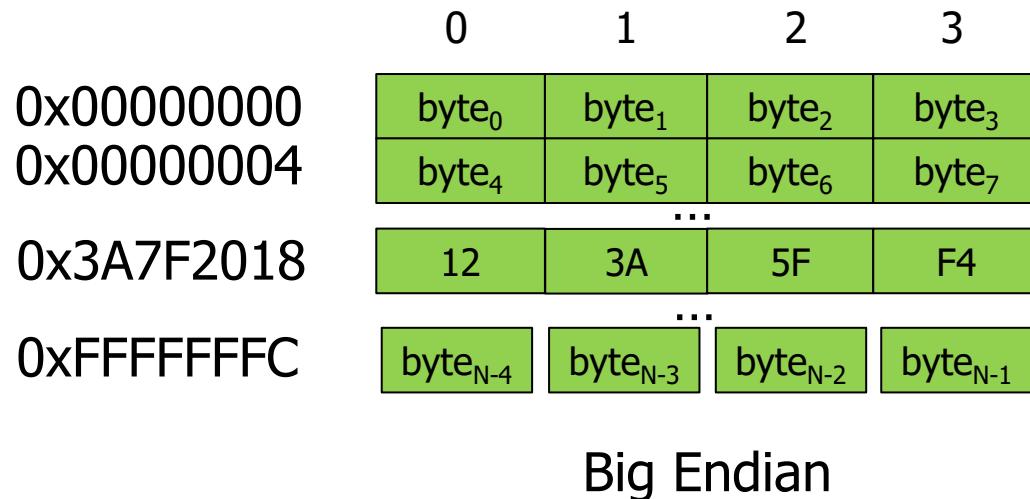
■ **BigEndian:**

- il Most Significant Byte occupa l'indirizzo di memoria più basso
- il Least Significant Byte occupa l'indirizzo di memoria più alto

□ **LittleEndian:**

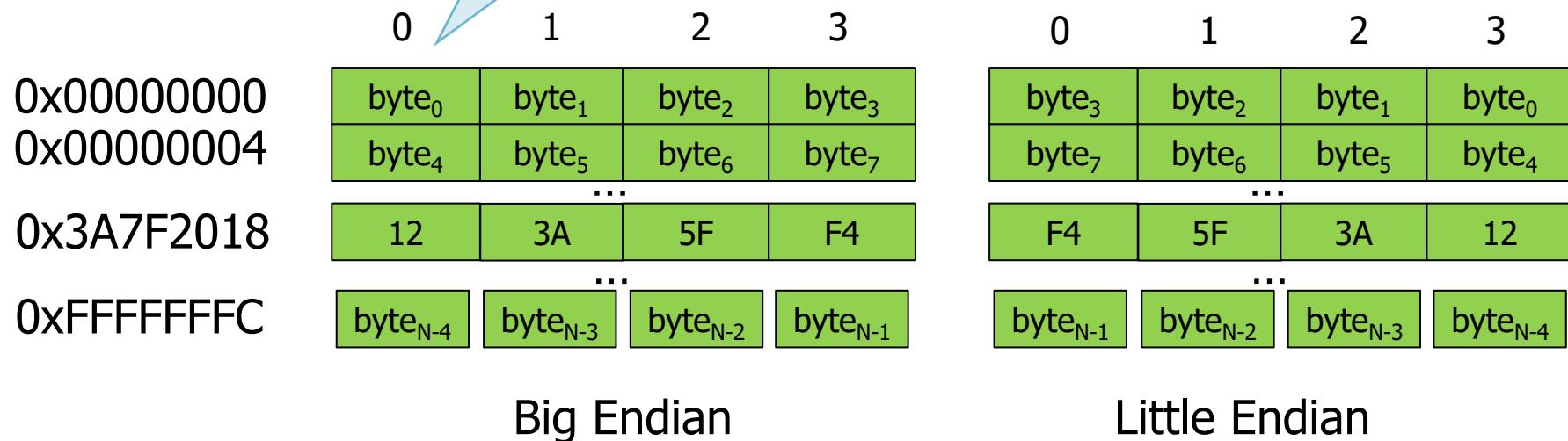
- il Most Significant Byte occupa l'indirizzo di memoria più alto
- il Least Significant Byte occupa l'indirizzo di memoria più basso.

- Memoria da 4 GB
- byte-addressable
- celle da 1 byte
- parole da 4 byte
- dato 0x123A5FF4 all'indirizzo 0x3A7F2018



Una riga rappresenta 4 byte: indirizzi bassi a sinistra, alti a destra

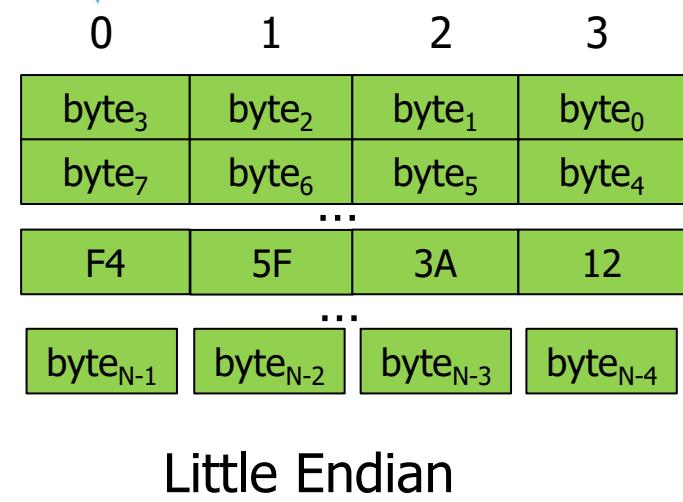
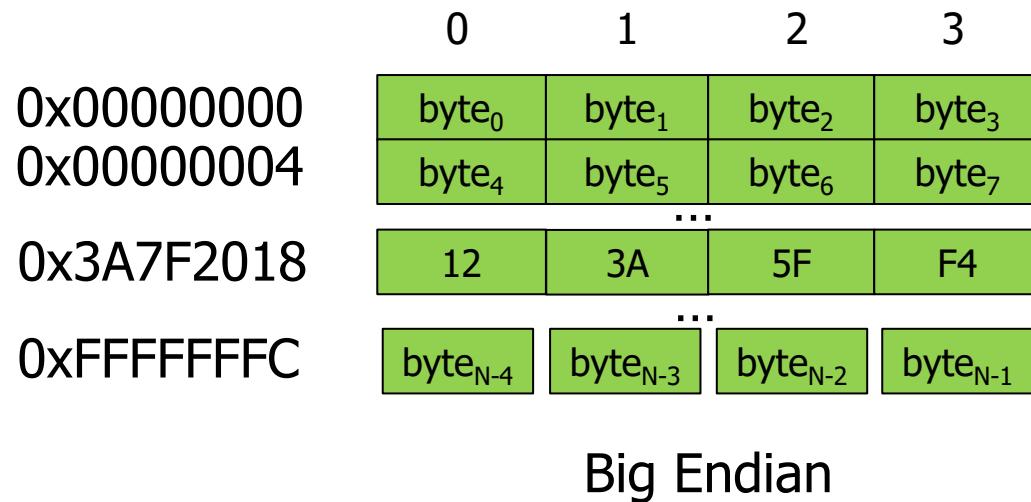
- Memoria da 4 parole
 - byte-addressing
 - celle da 1 byte
- parole da 4 byte
 - dato 0x123A5FF4 all'indirizzo 0x3A7F2018



Una riga rappresenta 4 byte: indirizzi bassi a sinistra, alti a destra

- Memoria da 4 GB
- byte-addressable
- celle da 1 byte

- pa... 4 byte
- dato 23A5FF4
- all'indirizzo 0x3A7F2018



Byte più significativo: 12

- celle da 1 byte

	0	1	2	3
0x00000000	byte ₀	byte ₁	byte ₂	byte ₃
0x00000004	byte ₄	byte ₅	byte ₆	byte ₇
0x3A7F2018	12	3A	5F	F4
0xFFFFFFF4	byte _{N-4}	byte _{N-3}	byte _{N-2}	byte _{N-1}

Big Endian

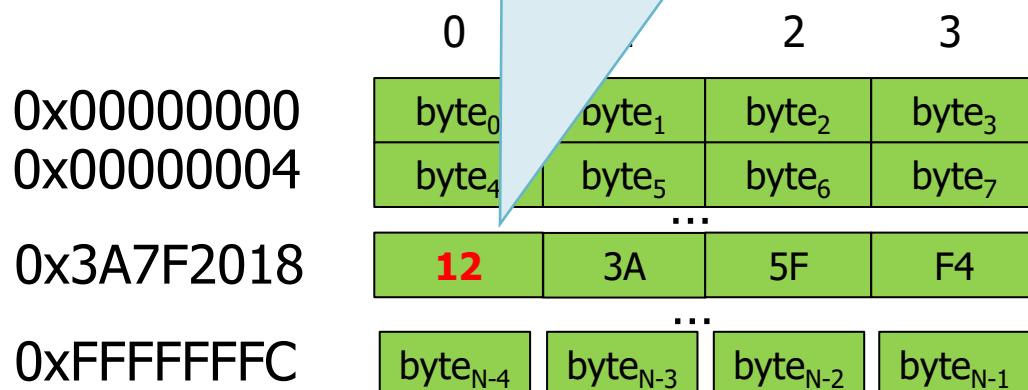
- parole da 4 byte dato 0x123A5FF4 all'indirizzo 0x3A7F2018

	0	1	2	3
0x00000000	byte ₃	byte ₂	byte ₁	byte ₀
0x00000004	byte ₇	byte ₆	byte ₅	byte ₄
0x3A7F2018	F4	5F	3A	12
0xFFFFFFF4	byte _{N-1}	byte _{N-2}	byte _{N-3}	byte _{N-4}

Little Endian

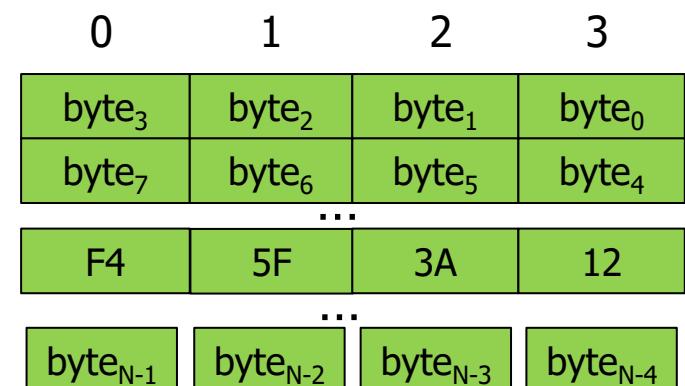
Byte più significativo: **12**

- celle da 1



Big Endian

- parole da 4 byte
- dato 0x**123A5FF4** all'indirizzo 0x3A7F2018



Little Endian

Byte più significativo: **12**

- celle da 1 byte

	0	1	2	3
0x00000000	byte ₀	byte ₁	byte ₂	byte ₃
0x00000004	byte ₄	byte ₅	byte ₆	byte ₇
0x3A7F2018	12	3A	5F	F4
0xFFFFFFF4	byte _{N-4}	byte _{N-3}	byte _{N-2}	byte _{N-1}

Big Endian

- parole da 4 byte
- dato 0x**123A5FF4** all'indirizzo 0x3A7F2018

	0	1	2	3
0x00000000	byte ₀	byte ₁	byte ₂	byte ₃
0x00000004	byte ₇	byte ₆	byte ₅	byte ₄
0x3A7F2018	F4	5F	3A	12
0xFFFFFFF4	byte _{N-1}	byte _{N-2}	byte _{N-3}	byte _{N-4}

Little Endian

Allineamento

“allineata”:

- parola di memoria che inizia ad un indirizzo divisibile per il numero di byte che compongono la parola stessa (la dimensione)

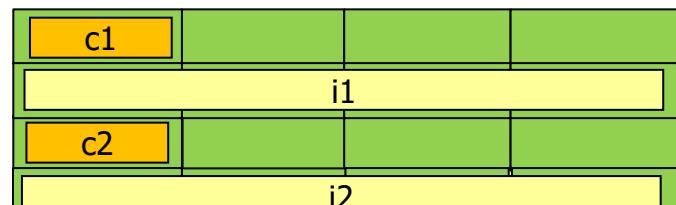
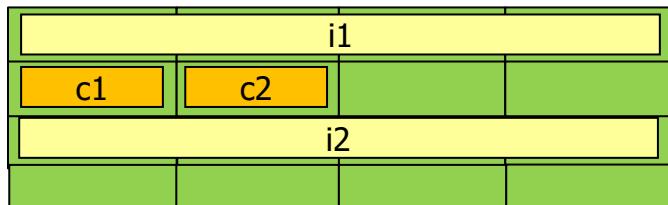
Esempio: memoria con 8 celle da 1 byte e parole da 2 byte con tecnica Big Endian: **allineata**, **non allineata**

0x0	MSB	LSB
0x2		
0x4		MSB
0x6	LSB	

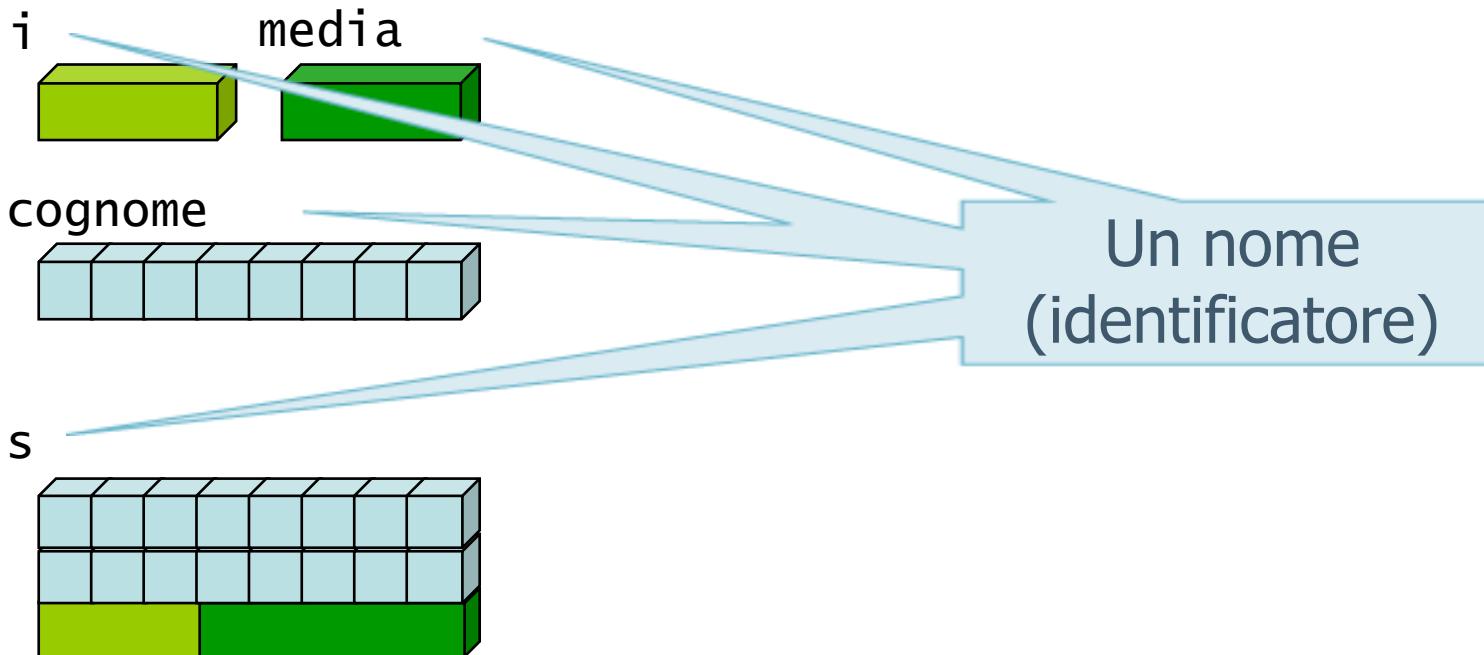
```
typedef struct item1_s {  
    int i1;  
    char c1, c2;  
    int i2;  
} Item1;
```

```
typedef struct item2_s {  
    char c1;  
    int i1;  
    char c2;  
    int i2;  
} Item2;
```

0x0028FEF4
0x0028FEF8
0x0028FEFC
0x0028FF00



Come si identifica una variabile?

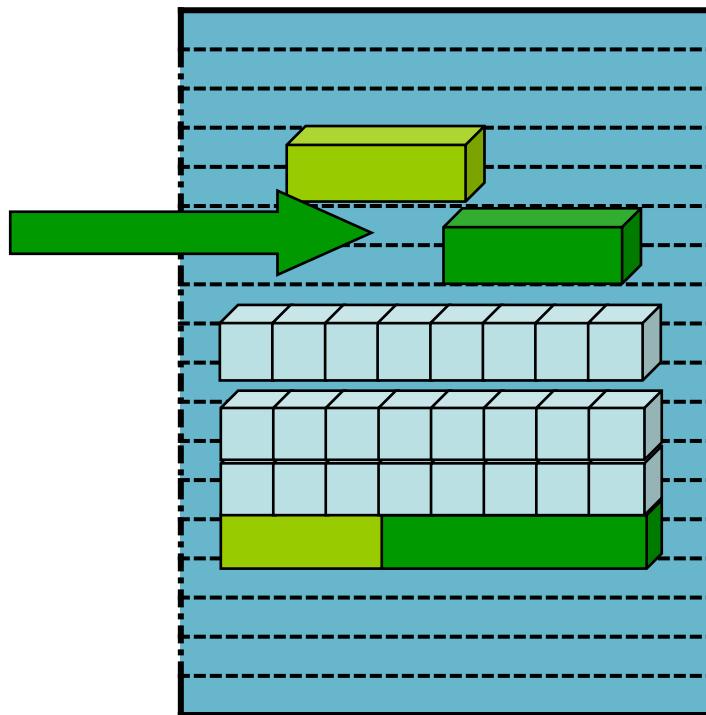


Il puntatore

1. Strumento alternativo alle variabili per l'accesso ai dati.

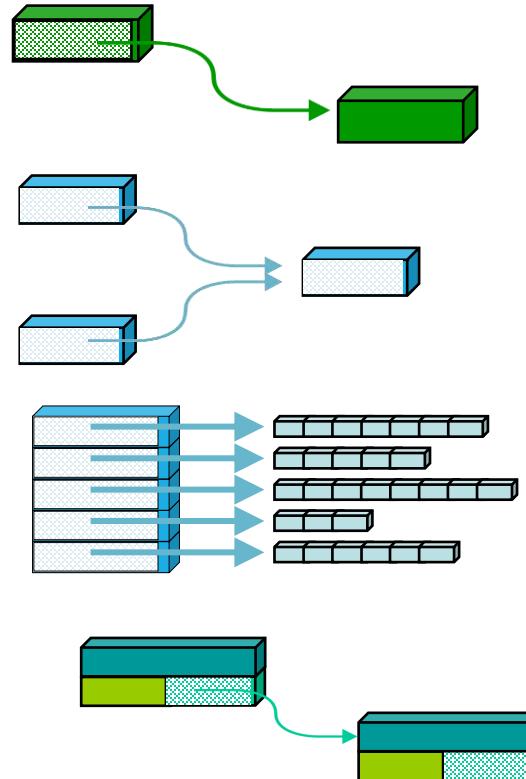
Informazioni necessarie:

- dove si trova il dato in memoria (indirizzo)
- come è codificato (tipo di dato)



Il puntatore

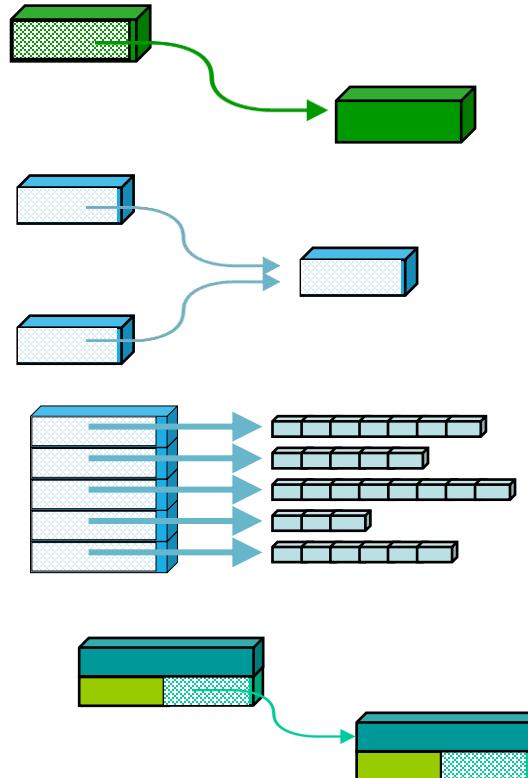
- Il puntatore è un'informazione manipolabile (si può calcolare, modificare, assegnare), a differenza di un identificatore (che non può essere modificato)



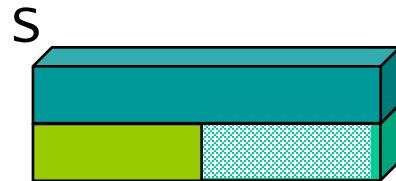
Il puntatore

2. Il puntatore è un'informazione manipolabile (si può calcolare, modificare, assegnare), a differenza di un identificatore (che non può essere modificato)

Novità: il puntatore è (anche) un dato (che punta a un dato)!



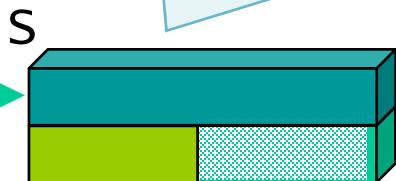
Operatori: riferimento



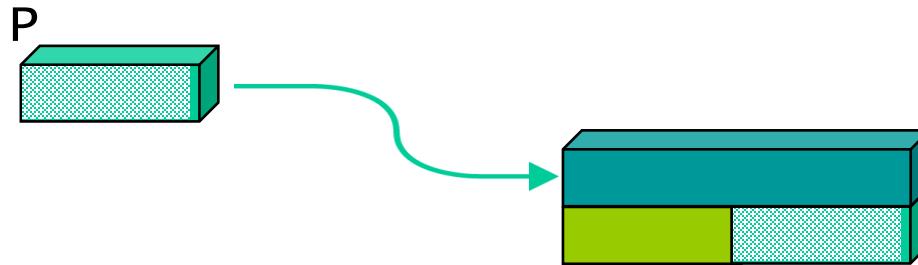
Puntatore a S



Data S (variabile con nome)



Operatori: dereferenziazione



Variabile Puntatore

P



Oggetto/dato (senza nome)
puntato da P

*P



* e &

- I simboli * e & sono utilizzati, in **definizioni** e **uso** dei puntatori, per indicare (in forma prefissa)
 - *... : dato puntato da ...
 - &... : puntatore a ...
- Gli operatori dereferenziazione * e riferimento & sono duali.

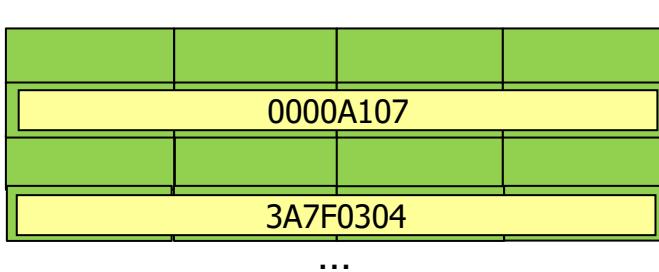
Esempio

- Variabile intera n = 41223 (=0x0000A107) all'indirizzo 0x3A7F0304
- Variabile puntatore a intero p (già dichiarata) all'indirizzo 0x3A7F030C
- Memoria 4 GB, byte-addressable, celle da 1 byte, parole da 4 byte

Esempio

```
p = &n;
```

0xA7F0304
0xA7F0308
0xA7F030C



```
printf("n: %d\n", n);
printf("n: %d\n", *p);
```

sono equivalenti

```
scanf("%d", &n);
scanf("%d", p);
```

sono equivalenti

Dichiarazione

- La dichiarazione di una variabile puntatore richiede il riferimento a un tipo base (quello del dato puntato)

```
int *px;  
char *p0, *p1;  
struct studente *pstud;  
FILE *fp;
```

La dichiarazione

`int *px;`

può essere letta in due modi:

a) `*px` (dato puntato da `px`) sarà (!) di tipo intero.

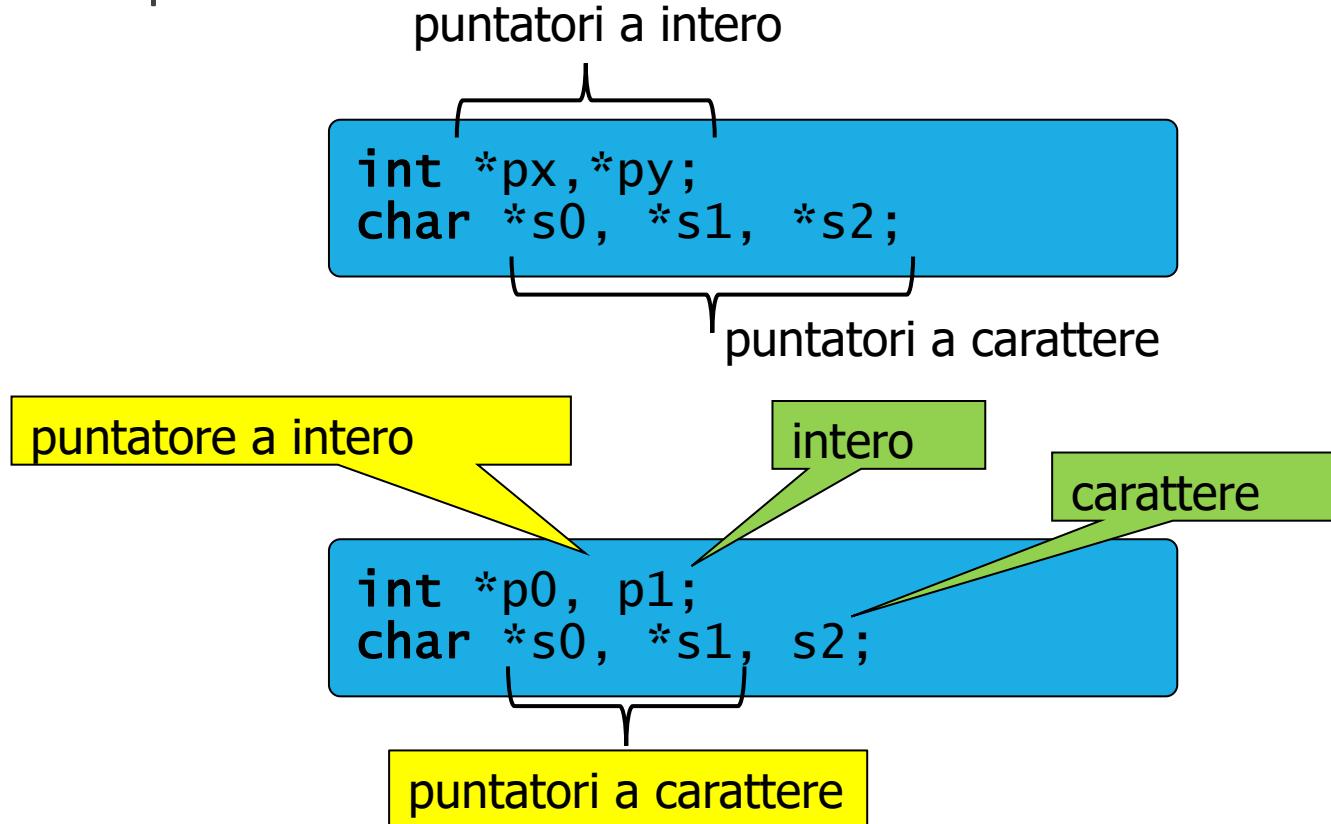
NOTA: la variabile `px`, al momento della definizione, NON contiene ancora un dato (un puntatore). NON esiste ancora un dato puntato, ma ci sarà dopo la prima assegnazione!

b) `int *` (tipo puntatore a intero) è il tipo della variabile `px`

Dichiarazione fattorizzata

- la dichiarazione di più variabili puntatore (stesso tipo base) nella stessa istruzione segue la strategia (a):
$$<\text{tipo base}> *<\text{id_1}>, *<\text{id_2}>..., *<\text{id_n}>;$$
- si scrive una sola volta il tipo base, mentre si premette un asterisco per ogni variabile dichiarata.

Esempio



Dichiarazione con inizializzazione

- Si può assegnare un valore a una variabile puntatore contestualmente alla dichiarazione
 - Esempi:

```
int x=0;
int *p = &x;
char *s = NULL;
```

Dichiarazione con inizializzazione

- Si può assegnare un valore a una variabile puntatore contestualmente alla dichiarazione
 - Esempi:

```
int x=0;
int *p = &x;
char *s = NULL;
```

- oppure (dichiarazioni equivalenti):

```
int x, *p = &x;
char *s = NULL;
```

La costante NULL

- Il valore effettivamente assegnato ad una variabile puntatore è un indirizzo in memoria
- Esiste una costante utilizzabile come “puntatore nullo” (lo “zero” dei tipi puntatori). Tale costante corrisponde al valore intero 0
- La costante simbolica **NULL** (definita in `<stdio.h>`) può essere utilizzata per rappresentare tale costante

Il tipo void *

- Un puntatore generico può essere definito in C facendo riferimento al tipo **void ***
- Un puntatore generico (**void ***) può essere convertito (e assegnato) in modo legale da/a un puntatore di altro tipo (es. **int ***)

```
int *px;
char *s0;
void *generic;
...
generic = px;
...
s0 = generic;
```

Assegnazione

- Finora si sono solo viste DICHIARAZIONI (di variabile puntatore)
- E le assegnazioni? COSA SI ASSEGNA?
 - Un indirizzo di memoria a una variabile puntatore?
 - Un valore a una variabile puntata da un puntatore?
- Due tipologie di assegnazione:
 - **puntatore come dato:** si assegna a una variabile puntatore il risultato di un'espressione che calcola un puntatore/indirizzo (del tipo corretto)
 - **puntatore come riferimento:** si assegna al dato (variabile) puntato (da un puntatore) un valore compatibile con il tipo di dato

Puntatore come dato: esempi

```
p = &x;  
s = p;  
pnome = &(stud.nome);  
p_i = &dati[i];
```

Puntatore come riferimento: esempi

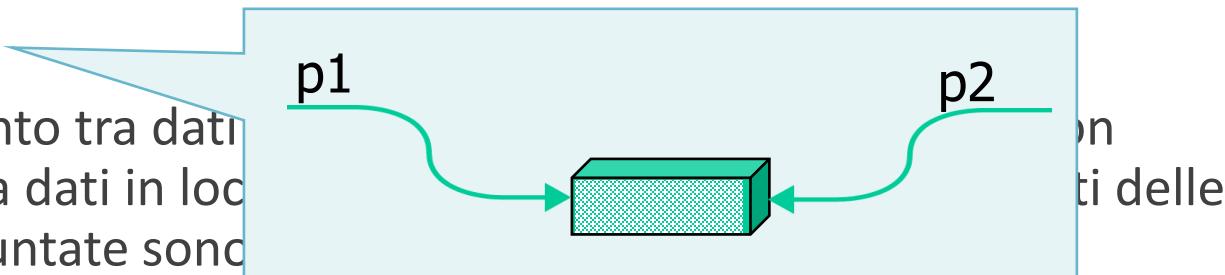
```
*p = 3*(x+2);  
*s = *p;  
*p_i = *p_i+1;
```

Oператорi relazionali == e !=

- Un confronto tra due puntatori ritorna valore vero se i due puntatori fanno riferimento allo stesso dato (stesso indirizzo in memoria)
 - $p1==p2$
- Un confronto tra dati puntati ritorna valore vero se (pur con puntatori a dati in locazioni diverse di memoria) i contenuti delle variabili puntate sono uguali
 - $*p1==*p2$

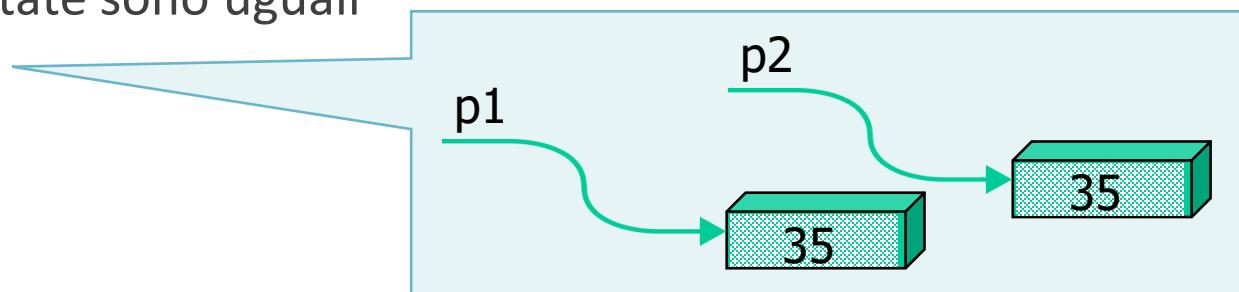
Operatori relazionali == e !=

- Un confronto tra due puntatori ritorna valore vero se i due puntatori fanno riferimento allo stesso dato (stesso indirizzo in memoria)
 - $p1 == p2$
- Un confronto tra dati puntati da due puntatori a dati in loco variabili puntate sono uguali
 - $*p1 == *p2$



Operatori relazionali == e !=

- Un confronto tra due puntatori ritorna valore vero se i due puntatori fanno riferimento allo stesso dato (stesso indirizzo in memoria)
 - $p1 == p2$
- Un confronto tra dati puntati ritorna valore vero se (pur con puntatori a dati in locazioni diverse di memoria) i contenuti delle variabili puntate sono uguali
 - $*p1 == *p2$



Aritmetica dei puntatori

- Una variabile puntatore contiene un indirizzo
- l'indirizzo è un intero e sugli interi sono definite
 - somma e sottrazione + -
 - incremento e decremento di 1 ++ --

Data l'istruzione `p=p+i; o p++;` l'effettivo incremento non è `i` o `1`, bensì:

- per `p=p+i i*(sizeof(*p))`
- per `p++ sizeof(*p)`

`i` e `1` non rappresentano indirizzi contigui, bensì dati del tipo puntato.

Aritmetica dei puntatori

- Una variabile puntatore contiene un indirizzo
- l'indirizzo è un intero e sugli indirizzi si possono eseguire:
 - somma e sottrazione + -
 - incremento e decremento di 1 ++ --

Quanti byte occupa un dato?
Operatore **sizeof()**

Data l'istruzione `p=p+i; o p++;` l'effettivo incremento non è `i` o `1`, bensì:

- per `p=p+i i*(sizeof(*p))`
- per `p++ sizeof(*p)`

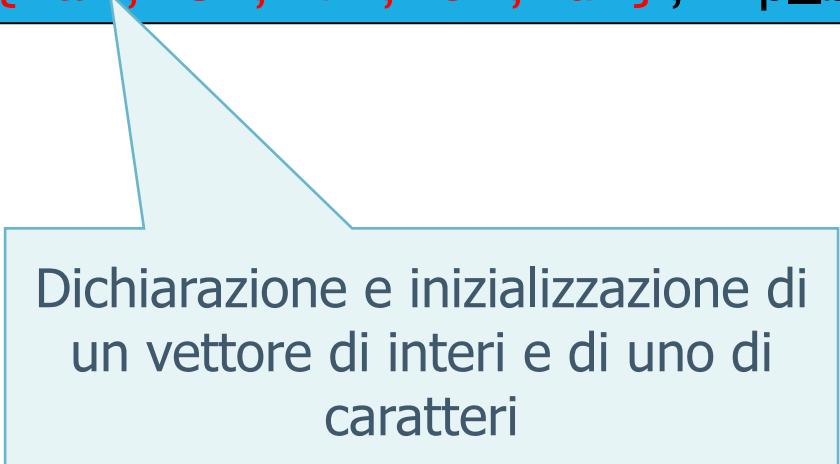
`i` e `1` non rappresentano indirizzi contigui, bensì dati del tipo puntato.

Esempio:

```
int a[3]={1,9,2}, *p_a=&a[0];
char b[5]={'a','e','i','o','u'}, *p_b=&b[0];
```

Esempio:

```
int a[3]={1,9,2}, *p_a=&a[0];  
char b[5]={'a','e','i','o','u'}, *p_b=&b[0];
```



Dichiarazione e inizializzazione di
un vettore di interi e di uno di
caratteri

Esempio:

```
int a[3]={1,9,2}, *p_a=&a[0];  
char b[5]={'a','e','i','o','u'}, *p_b=&b[0];
```

Dichiarazione e inizializzazione di
2 puntatori alla prima cella

Le istruzioni:

```
printf("a[0]=*p_a=%d,p_a=%p\n",a[0],p_a);  
printf("a[1]=*(p_a+1)=%d,p_a+1=%p\n",a[1],p_a+1);  
printf("b[0]=*p_b=%c,p_b=%p\n",b[0],p_b);  
printf("b[3]=*(p_b+3)=%c,p_b+3=%p\n",b[3],p_b+3);
```

visualizzeranno:

```
a[0]=*p_a=1,p_a=0028FEF8  
a[1]=*(p_a+1)=9,p_a+1=0028FEFC  
b[0]=*p_b=a,p_b=0028FEF3  
b[3]=*(p_b+3)=o,p_b+3=0028FEF6
```

- Incrementare (decrementare) di 1 un puntatore equivale a calcolare il puntatore al dato successivo (precedente) in memoria (supposto contiguo) dello stesso tipo
 - Esempio:

```
int x[100], *p = &x[50], *q, *r;  
...  
q = p+1; /* equivale a q=&x[51] */  
r = p-1; /* equivale a r=&x[49] */  
q++;      /* ora q punta a x[52] */
```

- Sommare (sottrarre) un valore intero i a un puntatore corrisponde a incrementare (decrementare) i volte di 1 il puntatore
 - Esempio:

```
int x[100], *p = &x[50], *q, *r;  
...  
q = p+10; /* equivale a q=&x[60] */  
r = p-10; /* equivale a r=&x[40] */  
r -= 5;   /* ora r punta a x[35] */
```

Il passaggio dei parametri

- Il linguaggio C prevede unicamente passaggio di parametri a funzioni per valore (“by value”)
 - Il valore del parametro **attuale**, calcolato alla chiamata della funzione, viene copiato nel parametro **formale**
- Non è previsto passaggio per riferimento (“by reference”), ma lo si realizza, in pratica, mediante
 - Passaggio per valore di puntatore a dato (“by pointer”)
 - E la funzione deve **usare il puntatore per accedere al dato**

Esempio: swap di 2 interi (**ERRATO!**)

Tentativo di fare una funzione che scambia i contenuti di due variabili

```
void swapInt (int x, int y) {  
    int tmp =x;  
    x=y; y=tmp;  
}  
...  
void main (void) {  
    int a, b;  
    ...  
    swapInt(a,b);  
    ...  
}
```

Lo scambio ha effetto solo nella funzione, non nel main

Esempio: swap di 2 interi (**CORRETTO!**)

Funzione che scambia i contenuti di due variabili (mediante puntatori)

```
void swapInt (int *px, int *py) {  
    int tmp = *px;  
    *px=*py; *py=tmp;  
}  
...  
void main (void) {  
    int a, b;  
    ...  
    swapInt(&a,&b);  
    ...  
}
```

Il main passa i puntatori

Esempio: swap di 2 interi (**CORRETTO!**)

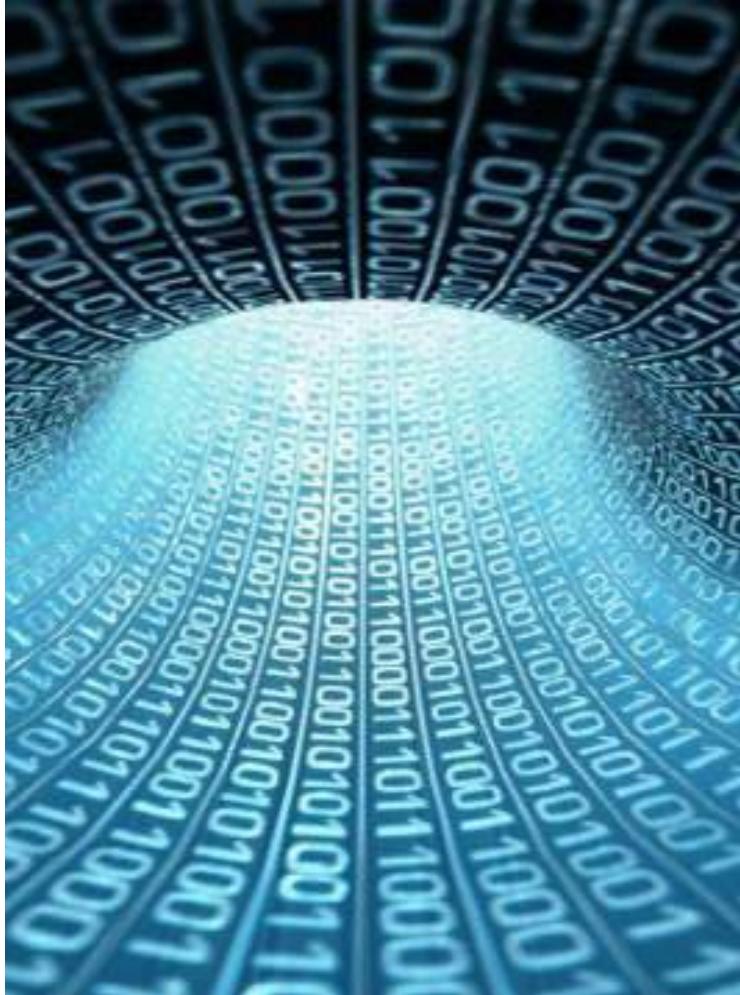
Funzione che scambia i contenuti di due variabili (mediante puntatori)

```
void swapInt (int *px, int *py) {  
    int tmp = *px;  
    *px=*py; *py=tmp;  
}  
...  
void main (void) {  
    int a, b;  
    ...  
    swapInt(&a,&b);  
    ...  
}
```

La funzione
scambia I DATI
PUNTATI

Capitolo 2: La Dualità Puntatore Vettore

PUNTATORI E STRUTTURE DATI DINAMICHE:
ALLOCAZIONE DELLA MEMORIA E
MODULARITÀ IN LINGUAGGIO C



Vettori e puntatori

Vettori e puntatori sono **duali** e consentono l'accesso ai dati in 2 forme:

- **vettoriale** con indici e []
- con **puntatori** mediante &, * e aritmetica dei puntatori

REGOLA: Il **nome** della variabile che identifica il vettore corrisponde formalmente al **puntatore** al primo elemento del vettore stesso:

$$\langle \text{nome vettore} \rangle \Leftrightarrow \&\langle \text{nome vettore} \rangle[0]$$

Vettori e puntatori

Vettori e puntatori sono **duali** e consentono l'accesso ai dati in 2 forme:

- **vettoriale** con indici e []
- con **puntatori** mediante &, * e aritmetica dei puntatori

REGOLA: Il **nome** della variabile che identifica il vettore corrisponde formalmente al **puntatore** al primo elemento del vettore stesso:

$$<\text{nome vettore}> \Leftrightarrow \&<\text{nome vettore}>[0]$$

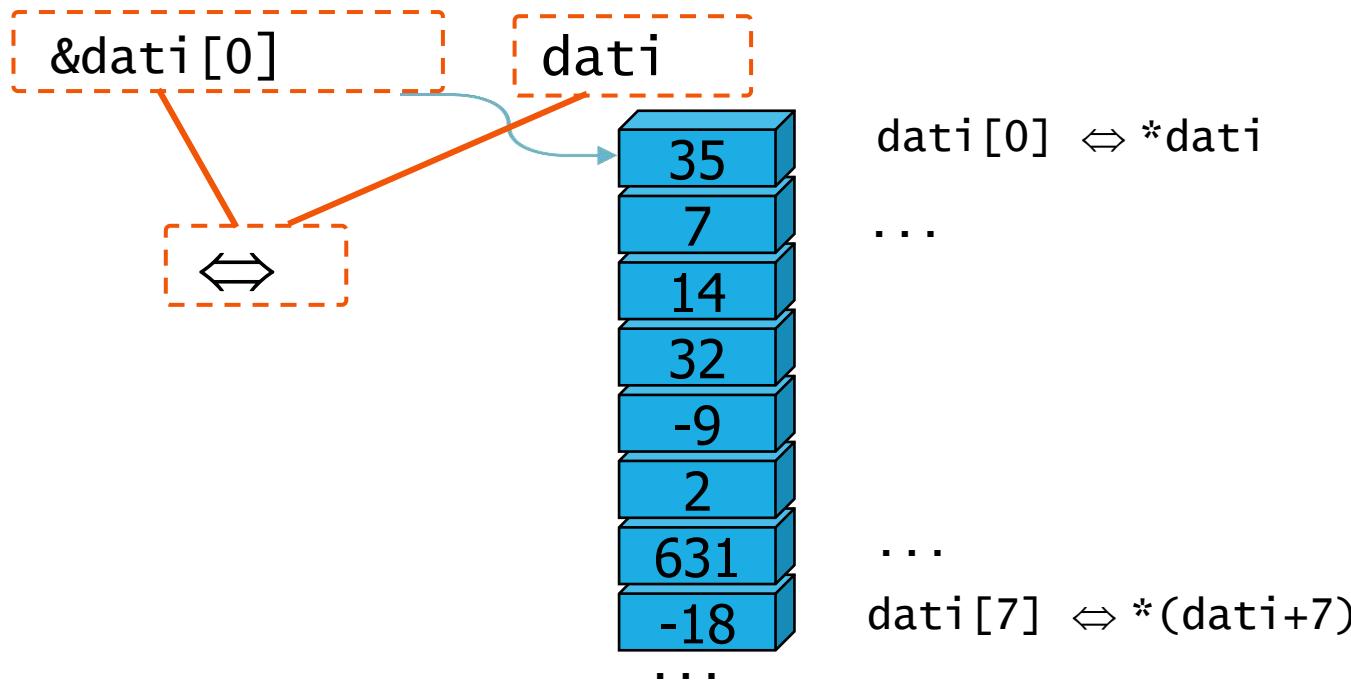
Basterebbe questo. Tutto il resto è una conseguenza!

Vettori e puntatori

Esempio: data una variabile di tipo vettore

```
int dati[100];
```

- $\text{dati} \Leftrightarrow \&\text{dati}[0]$
- $*\text{dati} \Leftrightarrow \text{dati}[0]$
- $*(\text{dati}+i) \Leftrightarrow \text{dati}[i]$



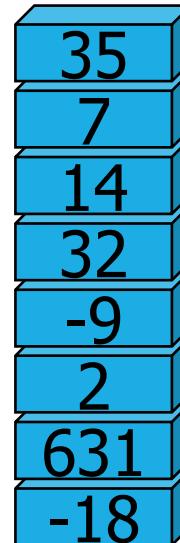
Esempio: lettura di un vettore di 100 interi

1. come vettore, con notazione vettoriale []
 - scorrendo le caselle mediante un indice
2. con puntatore a intero `int *p`, inizializzato a `&dati[0]`
 - aritmetica dei puntatori (somma tra puntatore e indice intero)
3. con puntatore a intero `int *p`, inizializzato a `&dati[0]`
 - scansione dei dati direttamente mediante puntatore (aggiornato ad ogni iterazione).

Modo 1:

```
int dati[100];
...
for (i=0;i<100;i++)
    scanf("%d",&dati[i]);
```

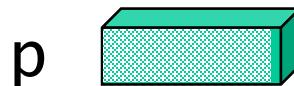
dati



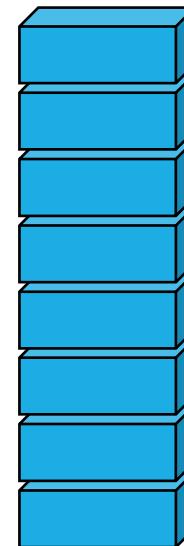
dati[0]
...
dati[7]

Modo 2:

```
int dati[100], *p;  
...
```



dati



dati[0]

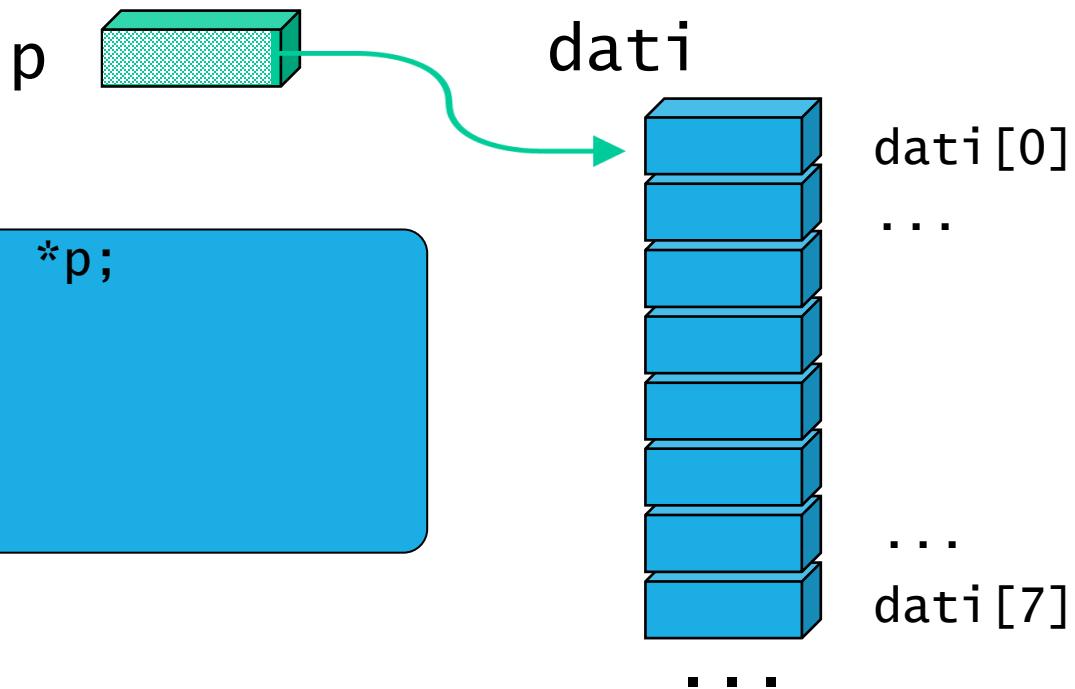
...

dati[7]

...

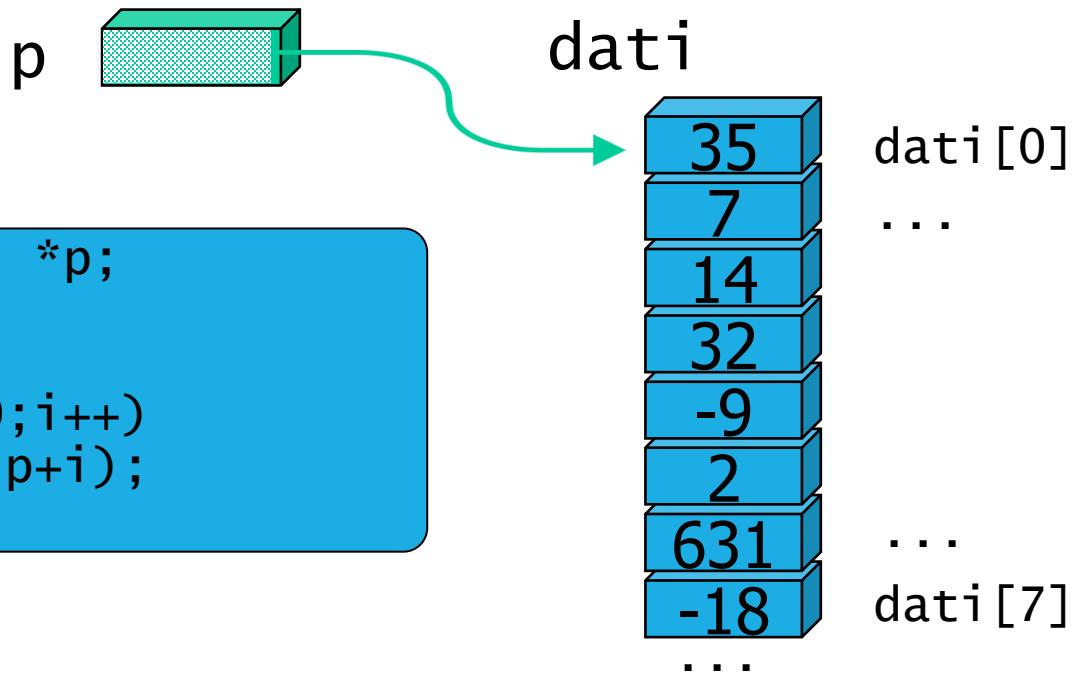
Modo 2:

```
int dati[100], *p;  
...  
p = &dati[0];
```



Modo 2:

```
int dati[100], *p;  
...  
p = &dati[0];  
for (i=0;i<100;i++)  
    scanf("%d", p+i);
```



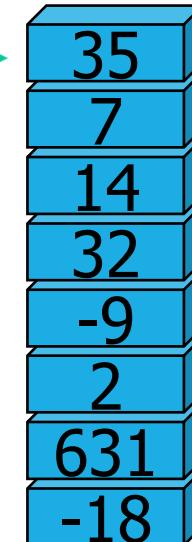
Modo 2:

```
int dati[100], *p;  
...  
p = &dati[0];  
for (i=0;i<100;i++)  
    scanf("%d", p+i);
```

p

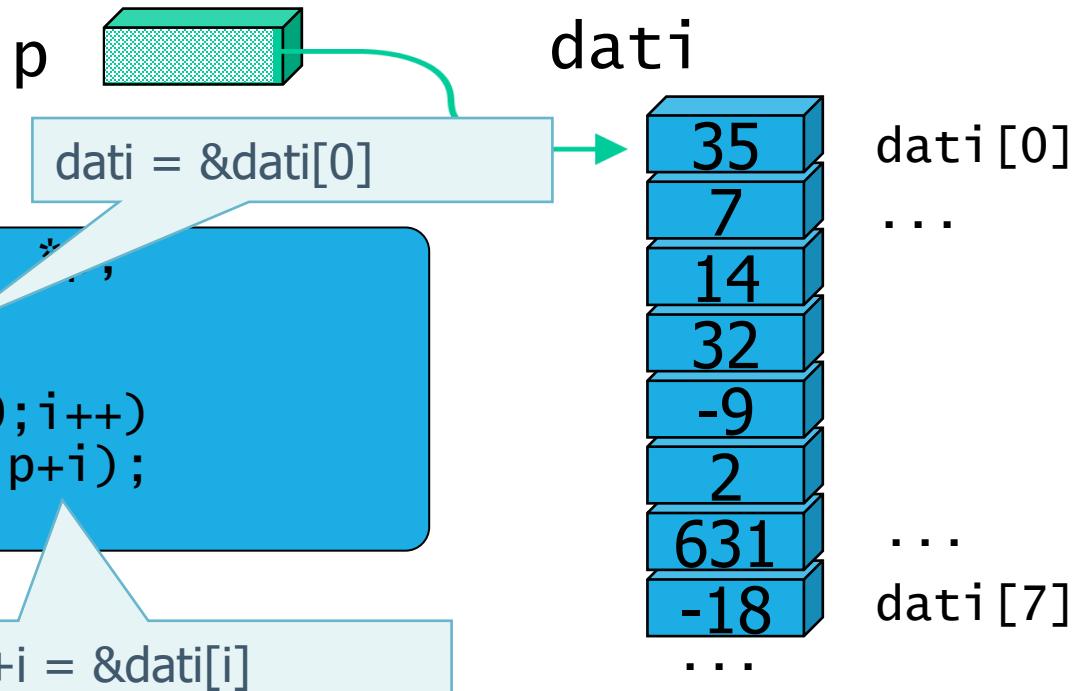
dati = &dati[0] (sono equivalenti)

dati



dati[0]
...
dati[7]

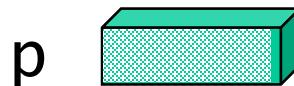
Modo 2:



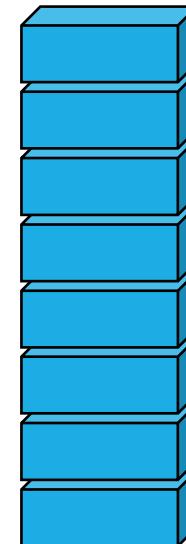
Modo 3:

```
int dati[100], *p;
```

...



dati



dati[0]

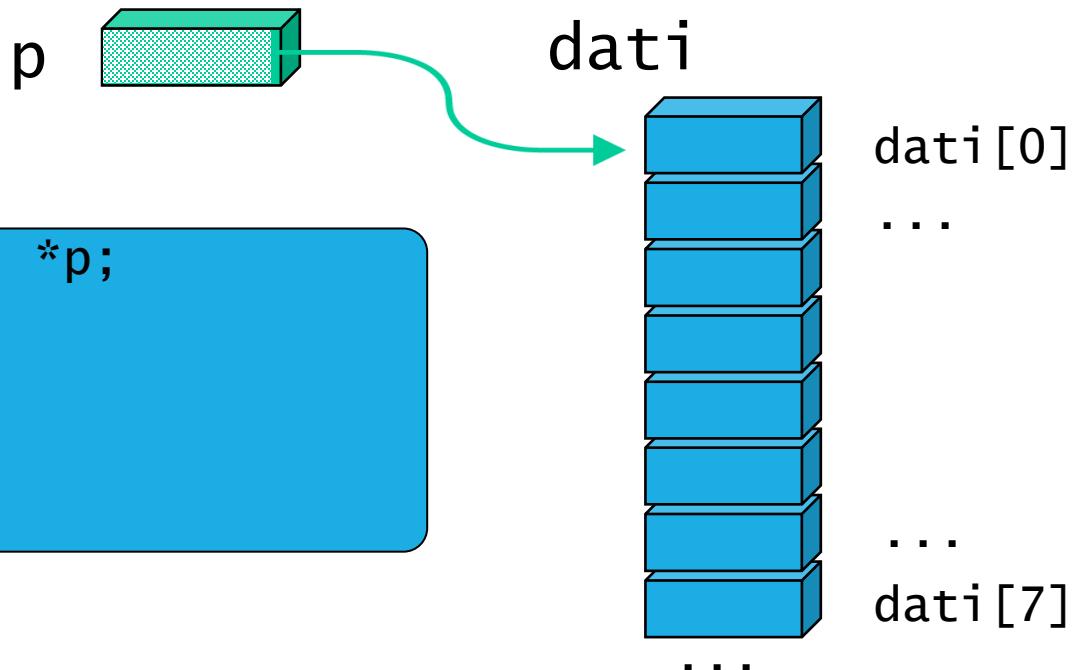
...

dati[7]

...

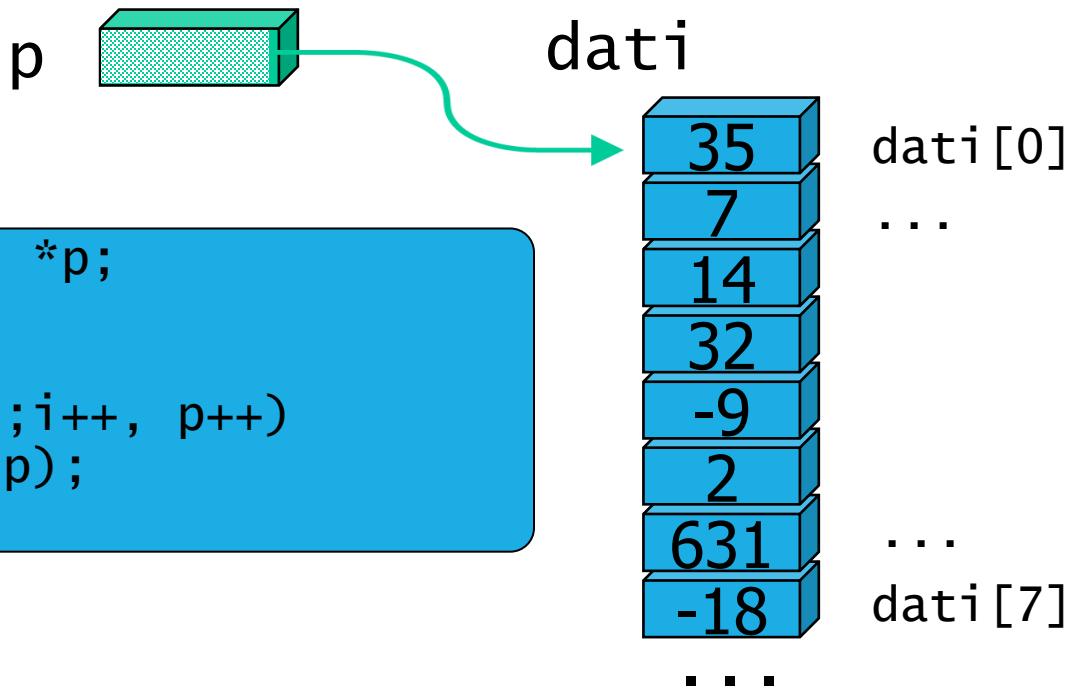
Modo 3:

```
int dati[100], *p;  
...  
p = &dati[0];
```



Modo 3:

```
int dati[100], *p;  
...  
p = &dati[0];  
for (i=0;i<100;i++, p++)  
    scanf("%d", p);
```



Sono lecite notazioni miste:

- **puntatore** a intero `int *v`, inizializzato a `dati` (`=&dati[0]`) e usato **con notazione vettoriale**

```
int *v = dati;
for (i=0; i<100; i++)
    scanf("%d", &v[i]);
```

- **vettore** di interi `dati` utilizzato come **puntatore**

```
for (i=0; i<100; i++)
    scanf("%d", dati+i);
```

Limite alla dualità

- il nome del vettore corrisponde ad una **costante** puntatore, non ad una variabile, quindi non può essere incrementato per scandire il vettore

```
for (i=0; i<100; i++, dati++)
    scanf("%d", dati);
```

Limite alla dualità

- il nome del vettore corrisponde ad una **costante** puntatore, non ad una variabile, quindi **non può essere incrementato** per scandire il vettore

```
for (i=0; i<100; i++, dati++)  
scanf("%d", dati);
```

Puntatori e sottovettori

Per identificare un sottovettore compreso tra indici l e r di un vettore dato:

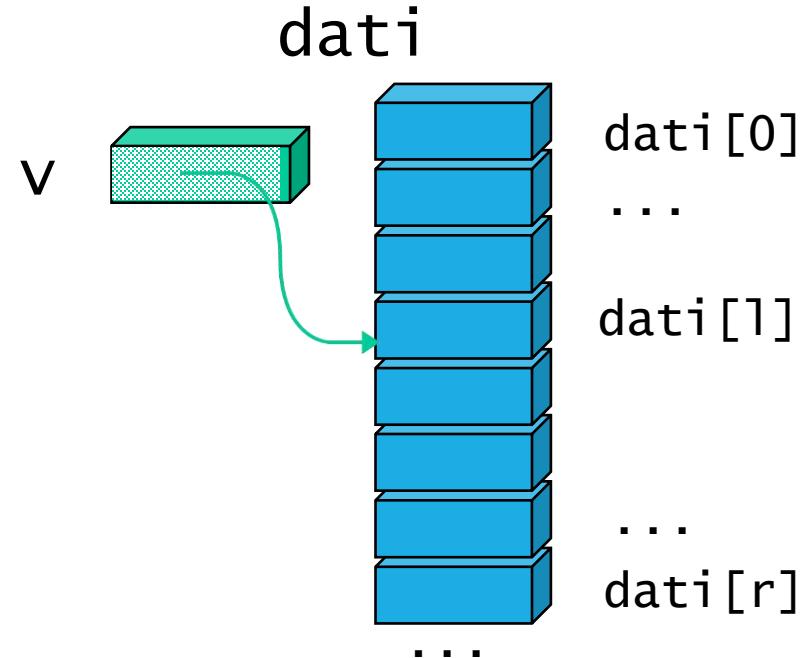
- si limita tra l e r l'indice i (identificazione implicita del sottovettore):

```
for (i=l; i<=r; i++)
    scanf("%d", &dati[i]);
for (i=l; i<=r; i++)
    printf("%d", dati[i]);
```

- si identifica esplicitamente un sottovettore tramite un puntatore alla sua casella di indice 1:

```
int dati[100], *v, i;  
v = &dati[1];  
n = r - 1 + 1;  
for (i=0; i<n; i++)  
    scanf("%d", &v[i]);  
for (i=0; i<n; i++)  
    printf("%d", v[i]);
```

Modo 1

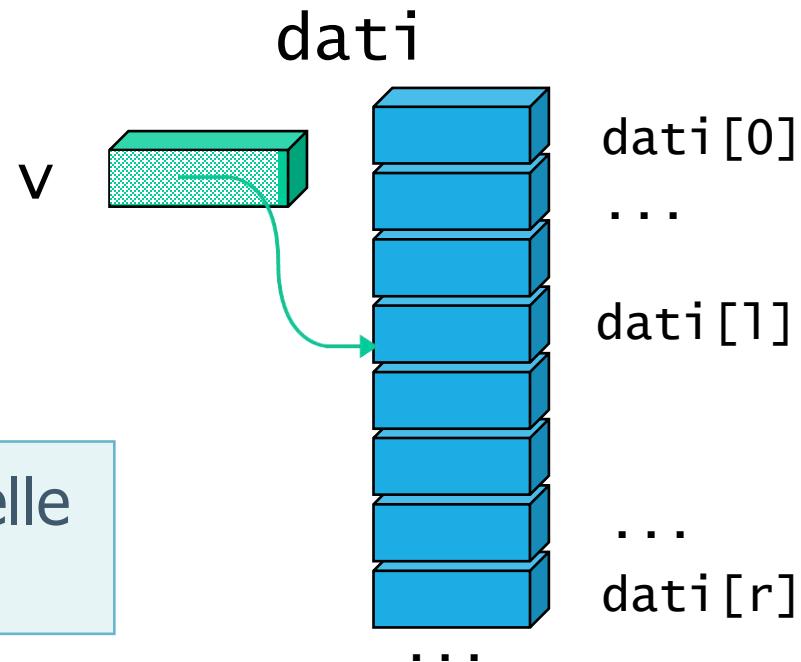


- si identifica esplicitamente un sottovettore tramite un puntatore alla sua casella di indice 1:

```
int dati[100], *v, i;  
v = &dati[1];  
n = r - 1 + 1;  
for (i=0; i<n; i++)  
    scanf("%d", &v[i]);  
for (i=0; i<n; i++)  
    printf("%d", v[i]);
```

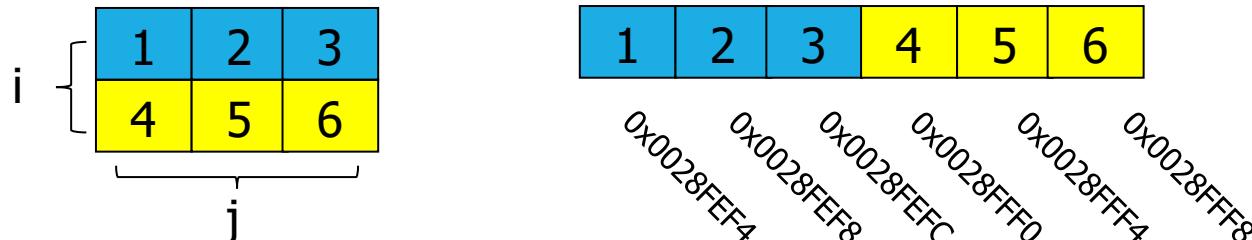
Modo 1

n è il numero di caselle
del sottovettore



Decomposizione di matrici

- Le matrici (bidimensionali e multidimensionali) si possono decomporre per righe (o sotto-matrici)
- Le matrici sono memorizzate con la tecnica **row-major**
 - matrice bidimensionale come vettore di righe
 - casella di una riga adiacenti e righe in sequenza



Esempio

prodotto scalare di una matrice M di NR righe per NC colonne per un vettore di NC elementi.

Il risultato è un vettore di NR elementi:

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} \times \begin{array}{|c|} \hline 5 \\ \hline 6 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 17 & 39 \\ \hline \end{array}$$

Modo 1

matrice bidimensionale con accesso a riga mediante iterazione su colonne:

```
float M[NR][NC], V[NC], Prod[NR];
int r, c;
...
for (r=0; r<NR; r++) {
    Prod[r] = 0.0;
    for (c=0; c<NC; c++)
        Prod[r] = Prod[r] + M[r][c]*V[c];
}
```

Modo 2

righe della matrice identificate grazie ad un puntatore **riga** (possibile grazie al row-major):

```
float M[NR][NC], V[NC], Prod[NR], *riga;  
int r, c;  
...  
for (r=0; r<NR; r++) {  
    Prod[r] = 0.0;  
    riga = M[r];  
    for (c=0; c<NC; c++)  
        Prod[r] = Prod[r] + riga[c]*V[c];  
}
```

Modo 2

righe della matrice identificate grazie ad un puntatore **riga** (possibile grazie al row-major):

```
float M[NR][NC], v[NC], Prod[NR], *riga;  
int r, c;  
...  
for (r=0; r<NR; r++) {  
    Prod[r] = 0.0;  
    riga = M[r];  
    for (c=0; c<NC; c++)  
        Prod[r] = Prod[r] + riga[c]*v[c];  
}
```

Equivale a
 $\text{riga} = \&(\text{M}[r][0]);$

Vettori e matrici come parametri

Vettori e matrici passati come parametri non vengono generati all'interno della funzione:

- Passando il nome di un vettore (come parametro attuale) a una funzione, si passa il puntatore al primo elemento del vettore
- si **passa solo il puntatore alla prima casella** (non la dimensione: se la si vuole, va passata come parametro aggiuntivo e indipendente)

La dualità puntatore \Leftrightarrow vettore è:

- totale per vettori (monodimensionali)
- parziale per matrici (vettori multidimensionali)

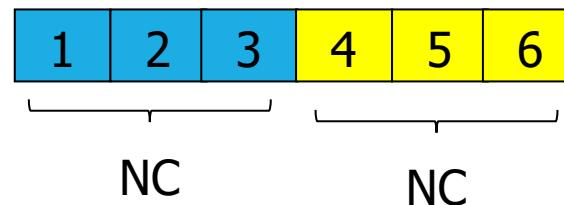
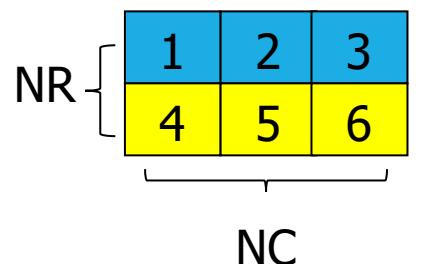
Vettori e matrici come parametri

- Per accedere all'i-esimo elemento di un vettore **vett[N]**, una funzione necessita仅仅 dell'indirizzo del primo elemento:

vett[i] equivale a $*(\text{vett} + i)$

- Per accedere all'elemento [i,j] di una matrice **mat[NR][NC]**, a una funzione serve l'indirizzo del primo elemento E il numero di colonne NC della matrice:

mat[i][j] equivale a $*(\text{mat} + \text{NC} * i + j)$



Vettori e matrici come parametri

Sono ridondanti le seguenti informazioni:

- dimensione di un vettore
- prima dimensione di una matrice

Esempi: prototipi di (compatibili a) funzioni di libreria

```
size_t strlen(const char s[]);  
int strcmp(const char s1[], const char s2[]);  
char *strcpy (char dest[], const char src[]);
```

Parametri formali

Notazione a vettore o a puntatore sono interscambiabili nei parametri formali

Esempi: prototipi di (compatibili a) funzioni di libreria:

```
size_t strlen(const char s[]);
int strcmp(const char s1[], const char s2[]);
char *strcpy (char dest[], const char src[]);
```

Esempi: prototipi reali di funzioni di libreria:

```
size_t strlen(const char *s);
int strcmp(const char *s1, const char *s2);
char *strcpy (char *dest, const char *src);
```

Esempio: argomenti al `main`

il vettore di puntatori a carattere `argv` è dichiarato come:

- vettore adimensionato di puntatori

```
int main(int argc, char *argv[])
```

- puntatore a puntatore (al primo elemento di unvettore di puntatori)

```
int main(int argc, char **argv)
```

Dimensione (effettiva) come parametro

Sovente si passano come parametri:

- il vettore adimensionato
- la sua dimensione effettiva

per realizzare funzioni che si adattano ad essa (la dimensione):

```
int leggi(int v[], int maxDim);
int main (void) {
    int v1[DIM1], v2[DIM2];
    int n1, n2;
    n1 = leggi(v1,DIM1);
    n2 = leggi(v2,DIM2);
    ...
}
```

```
int leggi(int v[], int maxDim) {
    int i, fine=0;
    for (i=0; !fine && i<maxDim; i++) {
        printf("v[%d] (0 per terminare): ", i);
        scanf("%d",&v[i]);
        if (v[i]==0) {
            fine = 1;
            i--; // trascura lo 0
        }
    }
    return i;
}
```

Corrispondenza parametri formali-attuali

PARAMETRO FORMALE VETTORE - ATTUALE PUNTATORE

PARAMETRO FORMALE PUNTATORE – ATTUALE VETTORE

parametro formale vettore - attuale puntatore

- Consente di generare un vettore (per una funzione) da un sotto-vettore, oppure da un puntatore (a memoria contigua)

Esempio: ordinamento di vettore per gruppi

```
void ordinaInt(int v[], int n);
...
int dati[20];
...
for (i=0;i<20;i+=4)
    ordinaInt(&dati[i],4);
```

parametro formale vettore - attuale puntatore

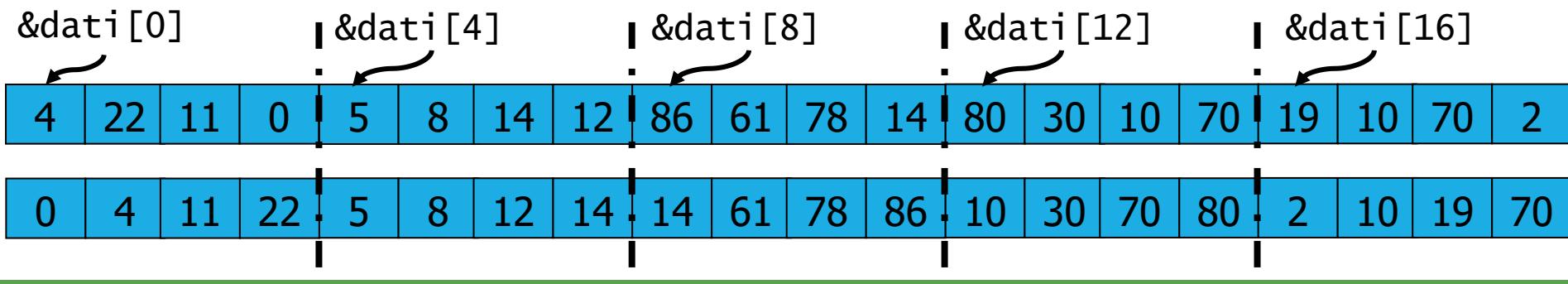
Esempio: ordinamento di vettore per gruppi

```
void ordinaInt(int v[], int n);
```

```
...
int dati[20];
```

```
...
for (i=0;i<20;i+=4)
    ordinaInt(&dati[i],4);
```

Ordinamento applicato a sotto-vettori di
4 elementi



parametro formale puntatore – attuale vettore

- Ad un parametro formale puntatore può corrispondere un parametro attuale vettore
 - Il puntatore, a sua volta, può essere trattato internamente come vettore

```
int leggi(int *v, int maxDim);  
...  
int sim, dati[100];  
...  
dim = leggi(dati, 100);
```

Puntatori e stringhe

LE STRINGHE MANIPOLATE COME VETTORI O MEDIANTE PUNTATORI

Puntatori e stringhe

- In C le stringhe non sono un tipo
- Una stringa è (formalmente) un vettore di caratteri (terminato da '\0')
- Le stringhe possono essere manipolate:
 - come vettori
 - con i puntatori e la loro aritmetica

Strlen (versione 0)

Funzione equivalente a `strlen` con stringa come vettore:

- Conta i caratteri cercando l'indice del '\0'

```
int strlen0(char s[]){
    int cnt=0;
    while (s[cnt]!='\0')
        cnt++;
    return cnt;
}
```

Strlen (versione 0)

Funzione **equivalente** a `strlen` con stringa come vettore:

- Conta i caratteri cercando l'indice del '\0'

```
int strlen0(char s[]){
    int cnt=0;
    while(s[cnt] != '\0')
        cnt++;
    return cnt;
}
```

ATTENZIONE: non è la funzione di libreria `strlen`, ma una implementazione equivalente

Strlen (versione 0)

Funzione equivalente a `strlen` con stringa come vettore:

- Conta i caratteri cercando l'indice del '\0'

```
int strlen0(char s[]){
    int cnt=0;
    while (s[cnt]!='\0')
        cnt++;
    return cnt;
}
```

`cnt` funge sia da indice che
da contatore

Strlen (versione 1)

Funzione equivalente a `strlen` con stringa come vettore:

- Scorre la stringa mediante un puntatore p
- Conta mediante contatore intero cnt

```
int strlen1(char s[]){
    int cnt=0;
    char *p=&s[0];
    while (*p != '\0') {
        cnt++;
        p++;
    }
    return cnt;
}
```

cnt funge da contatore,
lo scorrimento avviene
mediante puntatore p

Strlen (versione 2)

Funzione equivalente a `strlen` con stringa come puntatore:

- Scorre la stringa direttamente mediante il parametro puntatore `s`
- Conta mediante contatore intero `cnt`

```
int strlen2(char *s){  
    int cnt=0;  
    while (*s++ != '\0')  
        cnt++;  
    return cnt;  
}
```

`cnt` funge da contatore,
lo scorrimento avviene
mediante puntatore `s`

Strlen (versione 3)

Funzione equivalente a `strlen` con stringa come puntatore:

- Scorre la stringa mediante puntatore p
- Non conta ma usa aritmetica dei puntatori (differenza)

```
int strlen3(char *s){  
    char *p = s;  
    while (*p != '\0')  
        p++;  
    return p-s;  
}
```

non serve un contatore esplicito:
p scorre la stringa.

Altri esempi nella versione completa

- `strcmp`
- `strncmp`
- `strstr`
- Input mediante `sscanf` e formato `%n`

Vettori di puntatori

MATRICI REALIZZATE COME VETTORI DI PUNTATORI (A SOTTO-MATRICI)

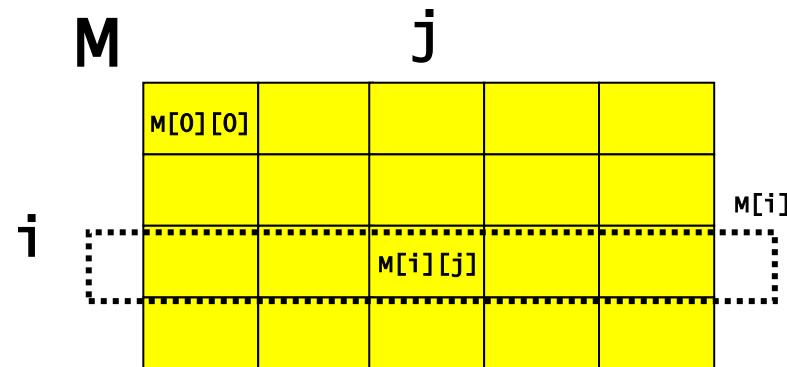
Vettori di puntatori

- Essendo il puntatore un dato
 - possono esistere vettori di puntatori
- Siccome un puntatore può corrispondere ad un vettore
 - Allora un vettore di puntatori può corrispondere a un vettore di vettori (una matrice)
- **ATTENZIONE!**
 - C'è dualità ma le matrici realizzate come vettori di puntatori sono diverse dalle matrici dichiarate come tali (con più livelli di parentesi quadre)

Matrice come vettore di righe

Esempio: matrice come vettore di righe (NO PUNTATORI!)

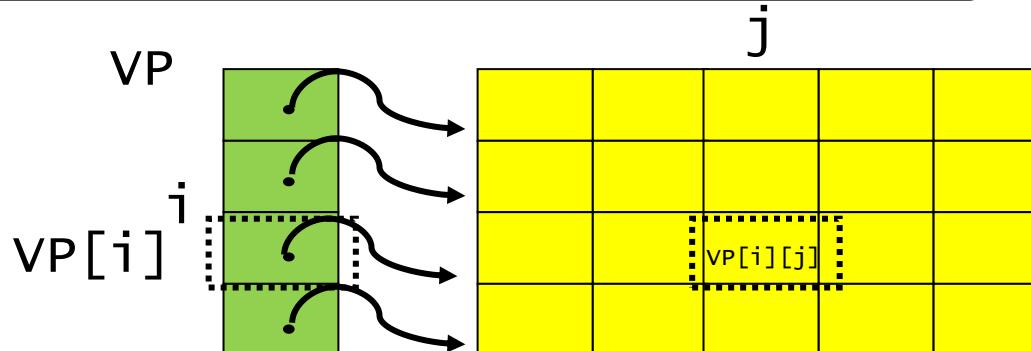
```
#define NR 4  
#define NC 5  
float M[NR][NC];  
...
```



Matrice come vettore di puntatori (a righe)

Matrice come vettore di puntatori a (alle caselle iniziali di) vettori

```
#define NR 4
#define NC 5
float R0[NC],R1[NC],R2[NC],R3[NC];
float *VP[NR] = {R0,R1,R2,R3};
...
```

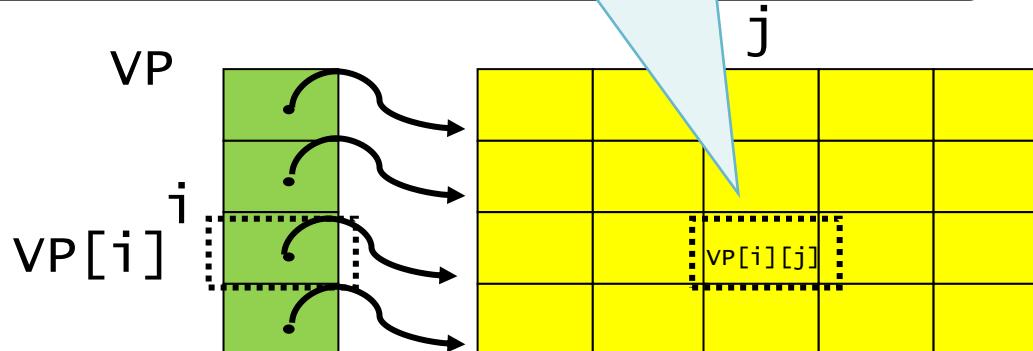


Matrice come vettore di puntatori (a righe)

Matrice come vettore di pu

```
#define NR 4  
#define NC 5  
float R0[NC],R1[NC],R2[NC],R3[NC];  
float *VP[NR] = {R0,R1,R2,R3};  
...
```

Notazione matriciale per
 $VP[i][j] \Leftrightarrow (VP[i])[j]$
(nonostante VP sia un vettore)



Vettori di vettori a dimensione variabile

Grazie ai vettori di puntatori (vettori di vettori) con notazione matriciale si possono realizzare matrici con righe di dimensione variabile.

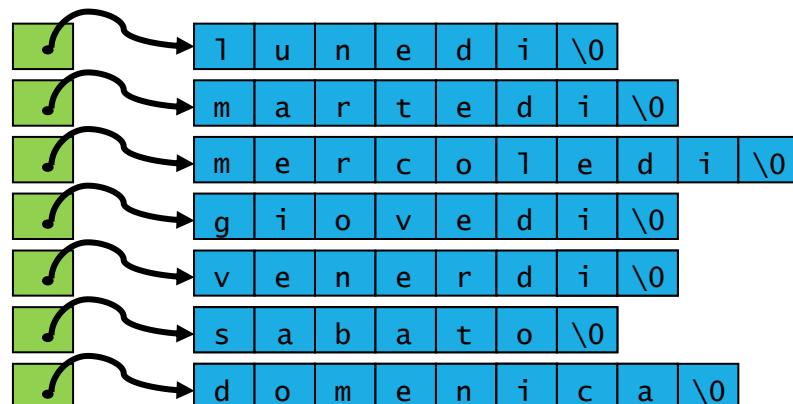
- opportunità sfruttata sovente nei vettori di stringhe, accessibili anche come matrici di caratteri

Esempio: vettore con i nomi di giorni della settimana e stampa dell'i-esimo carattere del nome se esiste.

Esempio: vettore con i nomi di giorni della settimana e stampa dell'i-esimo carattere del nome se esiste:

- soluzione 1: matrice di caratteri
- soluzione 2: vettore di stringhe

l	u	n	e	d	i	\0			
m	a	r	t	e	d	i	\0		
m	e	r	c	o	l	e	d	i	\0
g	i	o	v	e	d	i	\0		
v	e	n	e	r	d	i	\0		
s	a	b	a	t	o	\0			
d	o	m	e	n	i	c	a	\0	



Matrice di caratteri

```
void main (void) {
    int i,g;
    char giorni[7][10]={"lunedì","martedì",
                        "mercoledì","giovedì",
                        "venerdì","sabato","domenica"};
    printf("quale carattere (1-6)? ");
    scanf("%d",&i);
    for (g=0; g<7; g++)
        if (i<strlen(giorni[g]))
            printf("%c ", giorni[g][i-1]);
        else printf("_ ");
    printf("\n");
}
```

Matrice di caratteri

```
void main (void) {
    int i,g;
    char giorni[7][10]={"lunedì","martedì",
                        "mercoledì","giovedì",
                        "venerdì","sabato","domenica"};
    printf("quale carattere (1-6)? ");
    scanf("%d",&i);
    for (g=0; g<7; g++)
        if (i<strlen(giorni[g]))
            printf("%c ", giorni[g][i-1]);
        else printf("_ ");
    printf("\n");
}
```

Inizializzazione con costanti stringa

Matrice di caratteri

```
void main (void) {
    int i,g;
    char giorni[7][10]={"lunedì","martedì"
                        "mercoledì","giovedì",
                        "venerdì","sabato" "domenica"};
    printf("quale carattere (1-6)? ");
    scanf("%d",&i);
    for (g=0; g<7; g++)
        if (i<strlen(giorni[g]))
            printf("%c ", giorni[g][i-1]);
        else printf("_ ");
    printf("\n");
}
```

Riga generica identificabile
con un solo indice

Matrice di caratteri

```
void main (void) {  
    int i,g;  
    char giorni[7][10]={"lunedì","martedì",  
                        "mercoledì","giovedì",  
                        "venerdì","sabato","domenica"};  
  
    printf("quale carattere (1-6)? ");  
    scanf("%d",&i);  
    for (g=0; g<7; g++)  
        if (i<strlen(giorni[g]))  
            printf("%c ", giorni[g][i-1]);  
        else printf("_ ");  
  
    printf("\n");  
}
```

Singolo carattere
identificabile con due indici

Vettore di puntatori a stringa

```
void main (void) {
    int i,g;
    char *giorni[7]={"lunedì","martedì",
                      "mercoledì","giovedì",
                      "venerdì","sabato","domenica"};
    printf("quale carattere (1-6)? ");
    scanf("%d",&i);
    for (g=0; g<7; g++)
        if (i<strlen(giorni[g]))
            printf("%c ", giorni[g][i-1]);
        else printf("_ ");
    printf("\n");
}
```

Vettore di puntatori a stringa

```
void main (void) {
    int i,g;
    char *giorni[7]={"lunedì","martedì",
                     "mercoledì","giovedì",
                     "venerdì","sabato","domenica"};
    printf("quale carattere (1-6)? ");
    scanf("%d",&i);
    for (g=0; g<7; g++)
        if (i<strlen(giorni[g]))
            printf("%c ", giorni[g][i-1]);
        else printf("_ ");
    printf("\n");
}
```

Vettore di puntatori a char

Vettore di puntatori a stringa

```
void main (void) {
    int i,g;
    char *giorni[7]={"lunedì","martedì",
                      "mercoledì","giovedì",
                      "venerdì","sabato","domenica"};
    printf("quale carattere (1-6)? ");
    scanf("%d",&i);
    for (g=0; g<7; g++)
        if (i<strlen(giorni[g]))
            printf("%c ", giorni[g][i-1]);
        else printf("_ ");
    printf("\n");
}
```

Inizializzazione con puntatori
a stringhe (costanti)

Vettore di puntatori a stringa

```
void main (void) {
    int i,g;
    char *giorni[7]={"lunedì","martedì",
                      "mercoledì","giovedì",
                      "venerdì","sabato","domenica"};
    printf("quale carattere (1-6)? ");
    scanf("%d",&i);
    for (g=0; g<7; g++)
        if (i<strlen(giorni[g]))
            printf("%c ", giorni[g][i-1]);
        else printf("_ ");
    printf("\n");
}
```

giorni [g]: stringa
di ordine g

Vettore di puntatori a stringa

```
void main (void) {
    int i,g;
    char *giorni[7]={"lunedì","martedì",
                      "mercoledì","giovedì",
                      "venerdì"};
    printf("quale carattere? ");
    scanf("%d",&i);
    for (g=0; g<7; g++)
        if (i<strlen(giorni[g]))
            printf("%c ", giorni[g][i-1]);
        else printf("_ ");
    printf("\n");
}
```

giorni [g][i-1]: i-esimo
carattere della stringa di ordine g

Vettore di puntatori a stringa

```
void main (void) {  
    int i,g;  
    char *giorni[7]={ "domenica", "lunedì", "martedì",  
                      "mercoledì", "giovedì", "venerdì", "sabato"};  
  
    printf("quale carattere  
          inserire? ");  
    scanf("%d",&i);  
  
    for (g=0; g<7; g++)  
        if (i<strlen(giorni[g]))  
            printf("%c ", -giorni[g][i-1]);  
        else printf("_ ");  
  
    printf("\n");  
}
```

giorni[g][i-1]: vettore di stringhe utilizzato come matrice di caratteri

Vettore di stringhe

- Un vettore di stringhe può essere realizzato come:
 - matrice di caratteri: vettore bidimensionale (righe, colonne). Le righe hanno tutte la stessa lunghezza (vanno sovrardimensionate sulla stringa più lunga)
 - vettore di puntatori a stringhe: ogni elemento del vettore punta a una stringa distinta. Le stringhe possono avere lunghezze diverse
- Con entrambi i metodi si può utilizzare la notazione matriciale

Esempio: ordinamento di stringhe

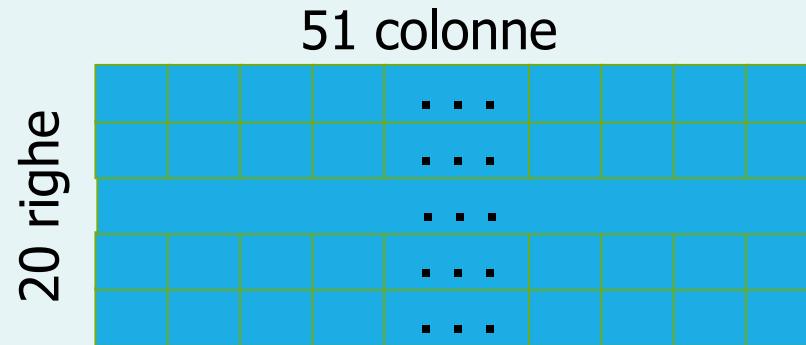
- Leggere da tastiera delle stringhe:
 - al massimo 20
 - ognuna al massimo di 50 caratteri
 - la somma delle lunghezze delle stringhe è ≤ 500
 - l'input termina con una stringa vuota
- Ordinare le stringhe in ordine crescente (secondo `strcmp`)
- Visualizzarle (secondo l'ordine precedente) su video

Matrice di caratteri

```
void main (void){  
    int i,ns;  
    char m[20][51]; // 51 colonne per i \0  
    printf("scrivi stringhe:\n");  
    for (ns=0; ns<20; ns++) {  
        gets(m[ns]);  
        if (strlen(m[ns])==0) break;  
    }  
    ordinaMatrice(m,ns);  
    printf("stringhe ordinate:\n");  
    for (i=0; i<ns; i++)  
        printf("%s\n", m[i]);  
}
```

Matrice di caratteri

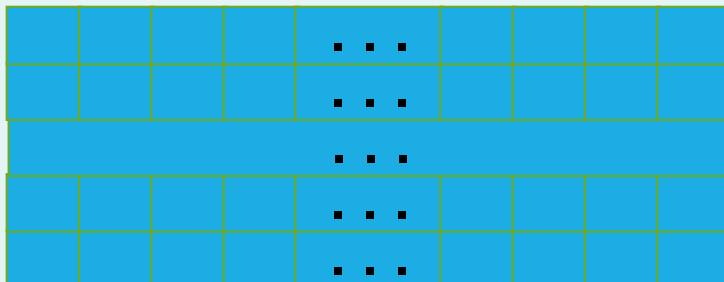
```
void main (void){  
    int i,ns;  
    char m[20][51]; // 51 colonne per i \0  
    printf("scrivi stringhe\n");  
    for (ns=0; ns<20; ns++) {  
        gets(m[ns]);  
        if (strlen(m[ns])==0)  
    }  
    ordinaMatrice(m,ns);  
    printf("stringhe ordinate\n");  
    for (i=0; i<ns; i++)  
        printf("%s\n", m[i]);  
}
```



Matrice di caratteri

```
void main (void){  
    int i,ns;  
    char m[20][51]; // 51 colonne per i \0  
    printf("scrivi stringhe:\n");  
    for (ns=0; ns<20; ns++) {  
        gets(m[ns]);  
        if (strlen(m[ns])==0) break;  
    }  
    ordinaMatrice(m,ns);  
    printf("stringhe ordinate:\n");  
    for (i=0; i<ns; i++)  
        printf("%s\n", m[i]);  
}
```

$m[ns] = \&(m[ns][0])$

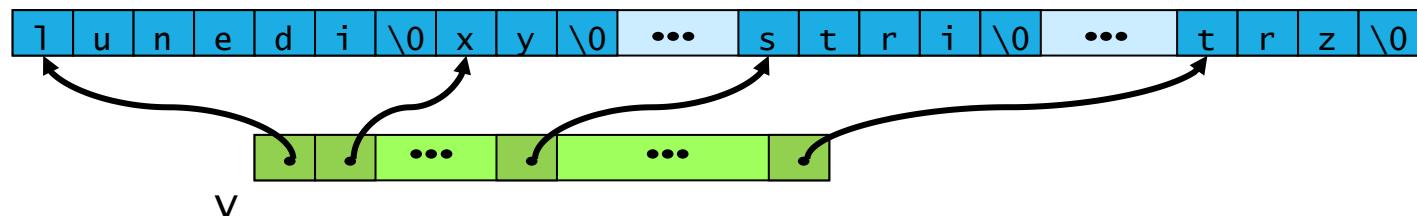


Vettore di puntatori

Doppio livello di memorizzazione

- vettore **buf** di 520 elementi che contiene le stringhe (terminatore incluso) una dopo l'altra
- vettore **v** di 20 puntatori al primo carattere di ogni stringa
 - $v[0]$ coincide con **buf** (o $\&buf[0]$), gli altri puntatori si calcolano in base alla lunghezza della stringa

buf

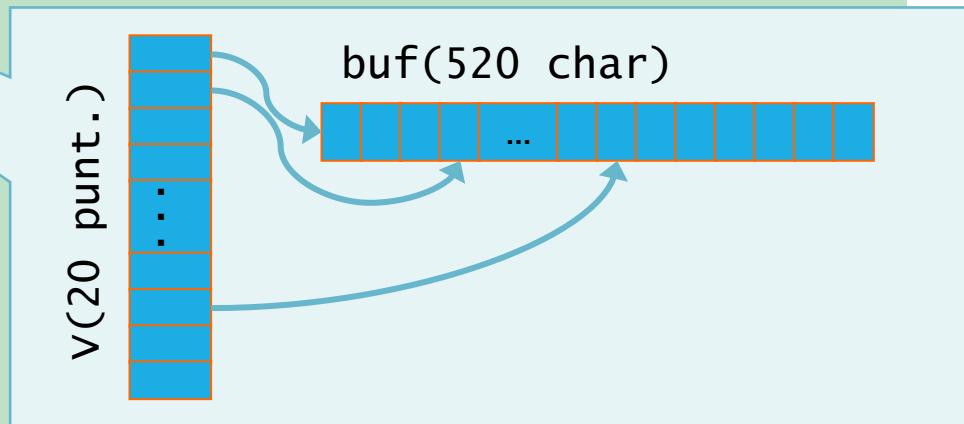


Vettore di puntatori

```
void main (void) {
    int i,ns;
    char *v[20], buf[520];
    printf("scrivi stringhe:\n");
    for (ns=i=0; ns<20; ns++) {
        v[ns]=buf+i; gets(v[ns]);
        if (strlen(v[ns])==0) break;
        i = i+strlen(v[ns])+1;
    }
    ordinaVettore(v,ns);
    printf("stringhe ordinate:\n");
    for (i=0; i<ns; i++)
        printf("%s\n", v[i]);
}
```

Vettore di puntatori

```
void main (void) {
    int i,ns;
    char *v[20], buf[520];
    printf("scrivi stringhe:\n");
    for (ns=i=0; ns<20; i++) {
        v[ns]=buf+i; gets(v[ns]);
        if (strlen(v[ns])==0) break;
        i = i+strlen(v[ns])+1;
    }
    ordinaVettore(v,ns);
    printf("stringhe ordinate:\n");
    for (i=0; i<ns; i++)
        printf("%s\n", v[i]);
}
```



Vettore di puntatori

```
void main (void) {  
    int i,ns;  
    char *v[20], buf[520];  
    printf("scrivi stringhe:\n");  
    for (ns=i=0; ns<20; ns++) {  
        v[ns]=buf+i; gets(v[ns]);  
        if (strlen(v[ns])==0) break;  
        i = i+strlen(v[ns])+1;  
    }  
    ordinaVettore(v,ns);  
    printf("stringhe ordinate:\n");  
    for (i=0; i<ns; i++)  
        printf("%s\n", v[i]);  
}
```

buf+i = &buf[i]

Vettore di puntatori

```
void main (void) {  
    int i,ns;  
    char *v[20], buf[520];  
    printf("scrivi stringhe:\n");  
    for (ns=i=0; ns<20; ns++) {  
        v[ns]=buf+i; gets(v[ns]);  
        if (strlen(v[ns])==0) break;  
        i = i+strlen(v[ns])+1;  
    }  
    ordinaVettore(v,ns);  
    printf("stringhe ordinate:\n");  
    for (i=0; i<ns; i++)  
        printf("%s\n", v[i]);  
}
```

Avanza in buf saltando stringa corrente più terminatore ('\0')

Confronto tra le soluzioni:

- Matrice di caratteri
 - 20 righe: massimo numero di stringhe
 - 51 colonne: massima lunghezza di stringa
 - $20 \times 51 = 1020$ caratteri: dimensione matrice
- Vettore di puntatori a stringhe
 - 20 puntatori: dimensione vettore di puntatori
 - 520 caratteri: dimensione di caratteri contenente le stringhe (500 caratteri per le stringhe + 20 terminatori)

Confronto tra le soluzioni:

- Matrice di caratteri
 - 20 righe: massimo numero di stringhe
 - 51 colonne: massima lunghezza di stringa
 - $20 \times 51 = 1020$ caratteri: dimensione matrice
- Vettore di puntatori a stringhe
 - 20 puntatori: dimensione vettore di puntatori
 - 520 caratteri: dimensione di caratteri contenente le stringhe (500 caratteri per le stringhe + 20 terminatori)



20 puntatori + 520 caratteri < 1020 caratteri !

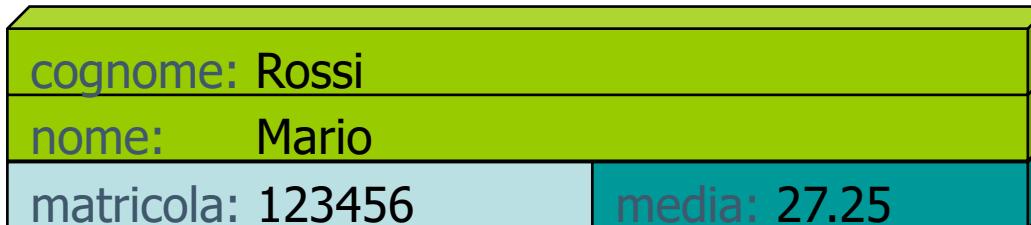
Nella versione completa

- Ordinamento di vettore di stringhe
 - Come matrice di caratteri
 - Come vettore di puntatori

Struct, puntatori e vettori

- Più informazioni eterogenee possono essere unite come parti (campi) di uno stesso dato dato (aggregato)

studente



Richiami sui tipi **struct**

- Il dato aggregato in C è detto **struct**. In altri linguaggi si parla di **record**
- Una **struct** (struttura) è un dato costituito da campi:
 - i campi sono di tipi (base) noti (eventualmente altre **struct** o puntatori)
 - ogni campo all'interno di una **struct** è accessibile mediante un identificatore (anziché un indice, come nei vettori)

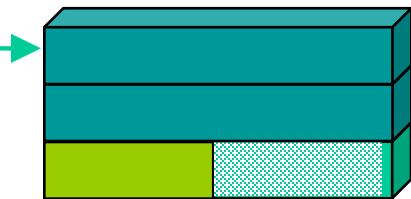
Puntatore a **struct**

- Per accedere a una struttura (tipo **struct**), utilizzando puntatori, si utilizzano le stesse regole viste per gli altri tipi
- Si noti che un puntatore può:
 - puntare a una struttura intera
 - puntare a un campo di struttura
 - essere un campo di una struttura

Puntatore a struttura (intera)

```
struct studente{  
    char cognome[MAX], nome[MAX];  
    int matricola;  
    float media;  
};  
...  
struct studente *p;  
...  
p = ...;
```

p



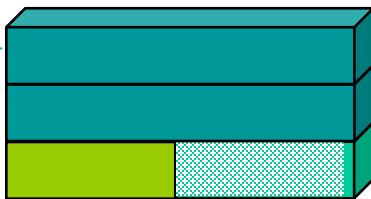
Puntatore a struttura (intera)

Variabile
puntatore

p

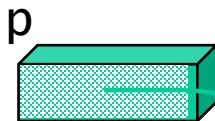


```
struct studente{  
    char cognome[MAX], nome[MAX];  
    int matricola;  
    float media;  
};  
...  
struct studente *p;  
...  
p = ...;
```



Puntatore a struttura (intera)

```
struct studente{  
    char cognome[MAX], nome[MAX];  
    int matricola;  
    float media;  
};  
...  
struct studente *p;  
...  
p = ...;
```



$*p$

Struttura puntata da p

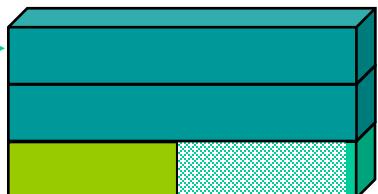
Puntatore a struttura (intera)

Variabile
puntatore

p



```
struct studente{  
    char cognome[MAX] , nome[MAX];  
    int matricola;  
    float media;  
};  
...  
struct studente *p;  
...  
p = ...;
```



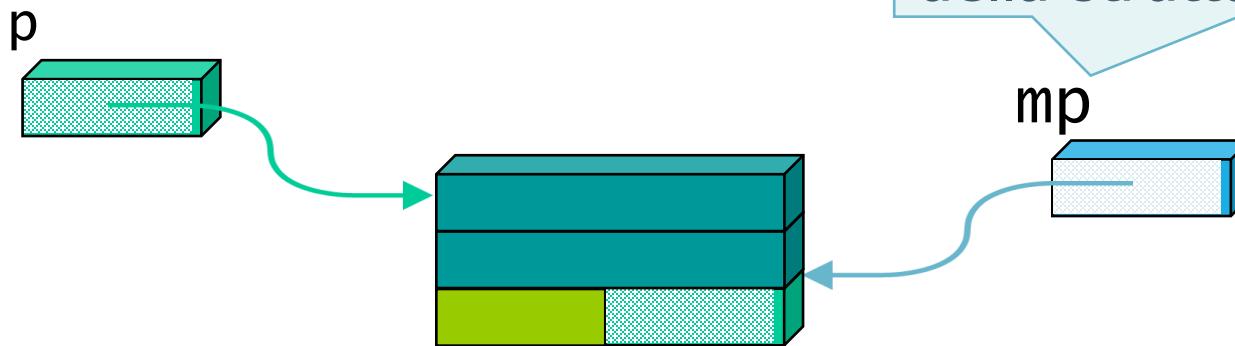
`(*p).media`

Campo media della
struttura puntata da p

Puntatore a campo di struttura

```
...
struct studente *p;
float *mp;
...
p = ...;
mp = &(*p).media;
```

Puntatore a campo media
della struttura puntata da p



Puntatore come campo di struttura

```
struct esame {  
    int scritto, orale  
};  
struct studente {  
    char cognome[MAX], nome[MAX];  
    int matricola;  
    struct esame *es;  
};
```

p



Attenzione: NON è una struttura interna:
struct esame es;
ma un puntatore:
struct esame *es;



Puntatore come campo di struttura

```
struct esame {  
    int scritto, orale  
};  
struct studente {  
    char cognome[MAX],  
    int matricola;  
    struct esame *es;  
};
```

p



La struct esame (puntata)
- è FUORI da struct studente
- deve esistere autonomamente
- NON viene creata in modo
AUTOMATICO



Puntatore come campo di struttura

```
struct esame {  
    int scritto, orale;  
};  
struct studente {  
    char cognome[MAX], nome[MAX];  
    int matricola;  
    struct esame *es;  
};
```

p



$(\ast p).es$

$\ast(\ast p).es$

$(\ast(\ast p).es).scritto$
 $(\ast(\ast p).es).orale$

Accesso a struttura puntata

- Il C dispone di una ***notazione alternativa*** (compatta) per rappresentare i campi di una struttura puntata

- Anzichè

`(*p).media`

`(*(*p).esame).scritto`

- Si può scrivere

`p->media`

`p->esame->scritto`

Accesso a struttura puntata

- Il C dispone di una **notazione alternativa** (compatta) per rappresentare i campi di una struttura puntata

- Anzichè

- $(*p).media$

- $(*(*p).esame).scritto$

- Si **può** scrivere

- $p->media$

- $p->esame->scritto$

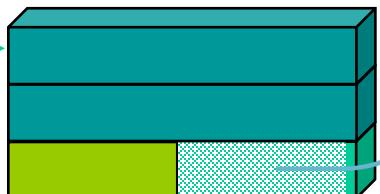
Si può va letto "**si deve**":

- la notazione con le parentesi e l'asterisco è difficile da leggere
- la maggior parte dei programmatori (tutti) usano la notazione $->$

Puntatore come campo di struttura

```
struct esame {  
    int scritto, orale;  
};  
struct studente {  
    char cognome[MAX], nome[MAX];  
    int matricola;  
    struct esame *es;  
};
```

p



$p \rightarrow \text{es}$

$\ast(p \rightarrow \text{es})$



$p \rightarrow \text{es} \rightarrow \text{scritto}$
 $p \rightarrow \text{es} \rightarrow \text{orale}$

Struct ricorsive

- Si dice **ricorsiva** una struct che include tra i suoi campi uno o più puntatori a strutture dello stesso tipo
- Servono per realizzare liste, alberi, grafi
- ATTENZIONE: definite solo per completezza (non si usano ora)

Esempio: soluzione 1

```
struct studente {  
    char cognome[MAX] , nome[MAX];  
    int matricola;  
    struct studente *link;  
};
```

Struct ricorsive

- Si dice **ricorsiva** una struct che include tra i suoi campi uno o più puntatori a strutture dello stesso tipo
- Servono per realizzare liste, alberi, grafi
- ATTENZIONE: definite solo per completezza (non si usano ora)

Esempio: soluzione 1

```
struct studente {  
    char cognome[MAX], nome[MAX];  
    int matricola;  
    struct studente *link;  
};
```

campo link di tipo puntatore
a struct studente

Struct ricorsive

- Si dice **ricorsiva** una struct che include tra i suoi campi uno o più puntatori a strutture dello stesso tipo
- Servono per realizzare liste, alberi, grafi
- ATTENZIONE: definite solo per completezza (non si usano ora)

Esempio: soluzione 1

```
struct studente {  
    char cognome[MAX] , nome[MAX];  
    int matricola;  
    struct studente *link;  
};
```

NON punterà a se stessa ma a un'altra struct dello stesso tipo

Struct ricorsive

- Si dice **ricorsiva** una struct che include tra i suoi campi uno o più puntatori a strutture dello stesso tipo
- Servono per realizzare liste
- ATTENZIONE: definite solo **ECCEZIONE:** si "usa" struct studente (per definire un puntatore) PRIMA di aver definito struct studente (lo è dopo **};**)

Esempio: soluzione 1

```
struct studente {  
    char cognome[MAX]; nome[MAX];  
    int matricola;  
    struct studente *link;  
};
```

Struct ricorsive

- Si dice **ricorsiva** una struttura se contiene puntatori a strutture dello stesso tipo
- Servono per realizzare liste circolari
- ATTENZIONE: definite solo i campi

Esempio: soluzione 1

ECCEZIONE: si "usa" struct studente (per definire un puntatore) PRIMA di aver definito struct studente (lo è dopo **};**)
Si può fare **solo coi puntatori** (non con altri tipi) perché la DIMENSIONE è nota (32 o 64 bit, dipende dal processore)

```
struct studente {  
    char cognome[MAX] , nome[MAX] ;  
    int matricola;  
    struct studente *link;  
};
```

Vettori di puntatori a **struct**

- Un vettore di **struct** è diverso da un vettore di puntatori a **struct**

v

1	3
12	10
3	34
8	3
-5	45
32	-1
17	6
8	23
72	-4

vettore di
puntatori
a **struct**

vP

	1	3
	12	10
	3	34
	8	3
	-5	45
	32	-1
	17	6
	8	23
	72	-4

vettore di **struct**

Esempio

- Tipo struct

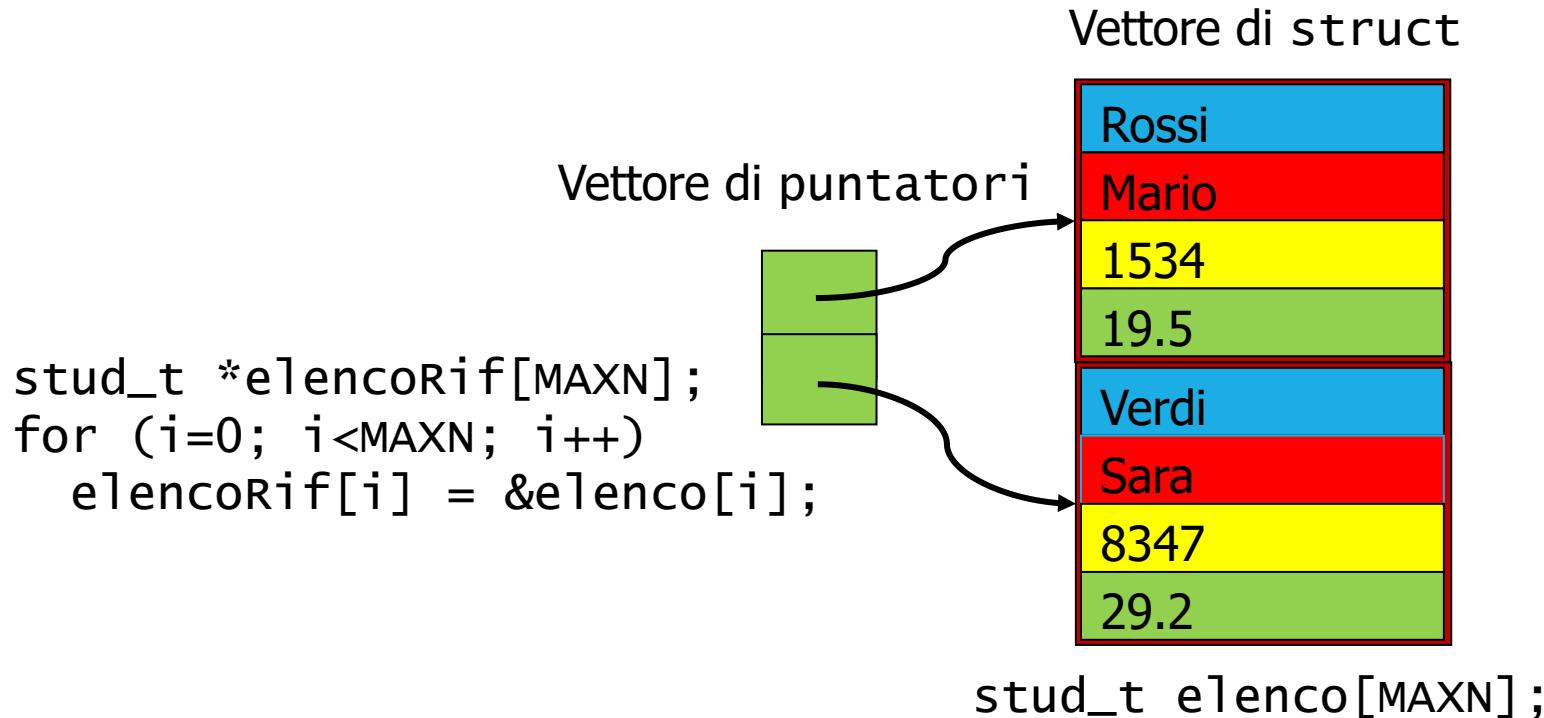
```
typedef struct studente {  
    char cognome[MAXS];  
    char nome[MAXS];  
    int matr;  
    float media;  
} stud_t;
```

Vettore di struct



```
stud_t elenco[MAXN];
```

Vettore di puntatori a **struct**



Esempio

- Scrivere un programma che:
 - acquisisce un elenco di studenti da un file il cui nome è ricevuto come primo argomento al main
 - ordina l'elenco per numeri di matricola crescenti
 - scrive l'elenco su un secondo file, il cui nome è ricevuto come secondo argomento.

Soluzione 1: vettore di **struct** (scambio di valori)

```
/* ... #include e #define */  
  
typedef struct studente {  
    char cognome[MAXS]; char nome[MAXS];  
    int matr; float media;  
} stud_t;  
  
/* ... prototipi */  
  
int main(int argc, char *argv[]) {  
    stud_t elenco[MAXN];  
    int ns = leggiStud(argv[1],elenco,MAXN);  
    ordStudPerMatr(elenco,ns);  
    lavoroSuElencoOrd(elenco,ns);  
    scrivistud(argv[2],elenco,ns);  
    return 0;  
}
```

```
int leggiStud(char *nomeFile, stud_t *el, int nmax) {  
    int n;  
    FILE *fp = fopen(nomeFile,"r");  
    for (n=0; n<nmax; n++) {  
        if (fscanf(fp,"%s%s%d%f", el[n].cognome,  
                   el[n].nome, &el[n].matr,  
                   &el[n].media)==EOF) break;  
    }  
    fclose(fp);  
    return n;  
}
```

Soluzione 1: vettore di struct (scambio di valori)

```
/* ... #include e #define */  
  
typedef struct studente {  
    char cognome[MAXS]; char nome[MAXS];  
    int matr; float media;  
} stud_t;  
  
/* ... prototipi */  
  
int main(int argc, char *argv[]) {  
    stud_t elenco[MAXN];  
  
    int ns = leggiStud(argv[1], elenco, MAXN);  
    ordStudPerMatr(elenco, ns);  
    lavorosuElencoOrdinato(ns);  
    scrivistud(argv[2], elenco);  
  
    return 0;  
}
```

```
int leggiStud(char *nomeFile, stud_t *el, int nmax) {  
  
    dimensione massima del  
    vettore  
  
    FILE *fp = fopen(nomeFile, "r");  
    if (fp == NULL) {  
        printf("Impossibile aprire il file %s\n", nomeFile);  
        exit(1);  
    }  
  
    for (int n=0; n<nmax; n++) {  
        if (fscanf(fp, "%s%s%d%f", el[n].cognome,  
                   el[n].nome, &el[n].matr,  
                   &el[n].media)==EOF) break;  
    }  
  
    fclose(fp);  
  
    return n;  
}
```

dimensione effettivamente utilizzata
del vettore (ricavata dal file)

Soluzione 1: vettore di **struct** (scambio di valori)

```
/* ... #include e #define */  
  
typedef struct studente {  
    char cognome[MAXS]; char nome[MAXS];  
    int matr; float media;  
} stud_t;  
  
/* ... prototipi */  
  
int main(int argc, char *argv[]) {  
    stud_t elenco[MAXN];  
    int ns = leggiStud(argv[1],elenco,MAXN);  
    ordStudPerMatr(elenco,ns);  
    lavoroSuElencoOrd(elenco,ns);  
    scrivistud(argv[2],elenco,ns);  
    return 0;  
}
```

```
int leggiStud(char *nomeFile, stud_t *el, int nmax) {  
    int n;  
    FILE *fp = fopen(nomeFile,"r");  
    for (n=0; n<nmax; n++) {  
        if (fscanf(fp,"%s%s%d%f", el[n].cognome,  
                   el[n].nome, &el[n].matr,  
                   &el[n].media)==EOF) break;  
    }  
    fclose(fp);  
    return n;  
}
```

Soluzione 1: vettore di struct (scambio di valori)

```
void scriviStud(char *nomeFile, stud_t *el,int n) {  
    int i;  
    FILE *fp = fopen(nomeFile,"w");  
    for (i=0; i<n; i++) {  
        fprintf(fp, "%s %s %d %f\n", el[i].cognome,  
                el[i].nome, el[i].matr,  
                el[i].media);  
    }  
    fclose(fp);  
}  
  
// confronto per struct ricevute by value  
int confrMatr(stud_t s1, stud_t s2) {  
    return s1.matricola-s2.matricola;  
}
```

```
// confronto per struct ricevute by reference/pointer  
int confrMatrByRef(stud_t *ps1, stud_t *ps2) {  
    return ps1->matricola-ps2->matricola;  
}  
  
void ordStudPerMatr(stud_t *el, int n) {  
    stud_t temp;  
    int i, j, imin;  
    for (i=0; i<n-1; i++) {  
        imin = i;  
        for (j = i+1; j < n; j++)  
            if (confrMatr(el[j],el[imin])<0)  
                imin = j;  
        temp = el[i]; el[i] = el[imin]; el[imin] = temp;  
    }  
}
```

Soluzione 1: vettore di struct (scambio di valori)

```
void scriviStud(char *nomeFile, stud_t *el,int n) {  
    int i;  
    FILE *fp = fopen(nomeFile,"w");  
    for (i=0; i<n; i++) {  
        fprintf(fp  
        passaggio by value:  
        la funzione riceve una "copia"  
        delle struct da confrontare  
    }  
    fclose(fp);  
}  
  
// confronto per struct ricevute per valore  
int confrMatr(stud_t s1, stud_t s2) {  
    return s1.matricola-s2.matricola;  
}
```

```
// confronto per struct ricevute per riferimento/puntatore  
int confrMatrByRef(stud_t *ps1, stud_t *ps2) {  
    return ps1->matricola-ps2->matricola;  
}  
  
void ordStudPerMatr(stud_t *el, int n) {  
    stud_t temp;  
    int i, j, imin;  
    for (i=0; i<n-1; i++) {  
        imin = i;  
        for (j = i+1, j < n; j++)  
            if (confrMatr(el[j],el[imin])<0)  
                imin = j;  
        temp = el[i]; el[i] = el[imin]; el[imin] = temp;  
    }  
}
```

selection sort

Soluzione 1: vettore di struct (scambio di valori)

```
void scriviStud(char *nomeFile, stud_t *el,int n) {  
    int i;  
    FILE *fp = fopen(nomeFile,"w");  
    for (i=0; i<n; i++)  
        fprintf(fp, "%d %s %s %d", el[i].matricola, el[i].cognome,  
                el[i].Nome, el[i].Anno);  
    fclose(fp);  
}  
  
// confronto per struct ricevute by value  
int confrMatr(stud_t s1, stud_t s2) {  
    return s1.matricola-s2.matricola;  
}
```

ALTERNATIVA:
passaggio by reference/pointer:
la funzione riceve i puntatori
alle struct da confrontare

```
// confronto per struct ricevute by reference/pointer  
int confrMatrByRef(stud_t *ps1, stud_t *ps2) {  
    return ps1->matricola-ps2->matricola;  
}  
  
void ordStudPerMatr(stud_t *el, int n) {  
    stud_t temp;  
    int i, j, imin;  
    for (i=0; i<n-1; i++) {  
        imin = i;  
        for (j = i+1; j < n; j++)  
            if (confrMatrByRef(&el[j],&el[imin])<0)  
                imin = j;  
        temp = el[i]; el[i] = el[imin]; el[imin] = temp;  
    }  
}
```

Soluzione 2: vettore di puntatori a **struct** (scambio di puntatori)

```
/* ... #include e #define */  
/* ... Typedef struct ... */  
/* ... prototipi */  
  
int main(int argc, char *argv[]) {  
    stud_t elenco[MAXN], *elencoRif[MAXN];  
  
    int i, ns = leggiStud(argv[1],elenco,MAXN);  
  
    for (i=0; i<ns; i++)  
        elencoRif[i]=&elenco[i];  
  
    ordRifStudPerMatr(elencoRif,ns);  
  
    lavoroSuElencoRifOrd(elencoRif,ns);  
  
    scriviRifStud(argv[2],elencoRif,ns);  
  
    return 0;  
}  
  
/* ... leggiStud e scrivistud non cambiano */
```

```
int confrMatrByRef(stud_t *ps1, stud_t *ps2) {  
    return ps1->matricola-ps2->matricola;  
}  
  
void ordRifStudPerMatr(stud_t **elR, int n) {  
    stud_t *temp;  
    int i, j, imin;  
    for (i=0; i<n-1; i++) {  
        imin = i;  
        for (j = i+1; j < n; j++)  
            if (confrMatrByRef(elR[j],elR[imin])<0)  
                imin = j;  
        temp=elR[i]; elR[i]=elR[imin]; elR[imin]=temp;  
    }  
}
```

Soluzione 2: vettore di puntatori a **struct** (scambio di puntatori)

```
/* ... #include e #define */  
/* ... Typedef struct ... */  
/* ... prototipi */  
  
int main(int argc, char *argv[]) {  
    stud_t elenco[MAXN], *elencoRif[MAXN];  
  
    int i, ns = leggiStud(argv[1], elenco, MAXN);  
    for (i=0; i<ns; i++)  
        elencoRif[i]=elenco[i];  
  
    ordRifStudPerMatr(elencoRif, ns);  
  
    lavoroSuElencoRifOrd(elencoRif, ns);  
  
    scriviRifStud(argv[2], elencoRif, ns);  
  
    return 0;  
}  
/* ... leggiStud e scriviRifStud */
```

int confrMatrByRef(stud_t *ps1, stud_t *ps2) {
 ;
 int n) {

dimensione massima del
vettore

```
stud_t *temp;  
int i, j, imin;  
for (i=0; i<n-1; i++) {  
    imin = i;  
    for (j = i+1; j < n; j++)  
        if (confrMatrByRef(elR[j], elR[imin])<0)  
            imin = j;  
    temp=elR[i]; elR[i]=elR[imin]; elR[imin]=temp;
```

dimensione effettivamente utilizzata
del vettore (ricavata dal file)

Soluzione 2: vettore di puntatori a **struct** (scambio di puntatori)

```
/* ... #include e #define */  
/* ... Typedef struct ... */  
/* ... prototipi */  
  
int main(int argc, char *argv[]) {  
    stud_t elenco[MAXN], *elencoRif[MAXN];  
    int i, ns = leggiStud(argv[1],elenco,MAXN);  
    for (i=0; i<ns; i++)  
        elencoRif[i]=&elenco[i];  
    ordRifStudPerMatr(elencoRif,ns);  
    lavoroSuElencoRifOrd(elencoRif,ns);  
    scriviRifStud(argv[2],elencoRif,ns);  
    return 0;  
}  
  
/* ... leggiStud e scrivistud non cambiano */
```

Prima carica il vettore di struct

```
int confrMatrByRef(stud_t *ps1, stud_t *ps2) {  
    ;  
}  
int n) {  
    stud_t *temp;  
    int i, j, imin;  
    for (i=0; i<n-1; i++) {  
        imin = i;  
        for (j = i+1; j < n; j++)  
            if (confrMatrByRef(elR[j],elR[imin])<0)  
                imin = j;  
        temp=elR[i]; elR[i]=elR[imin]; elR[imin]=temp;  
    }  
}
```

Soluzione 2: vettore di puntatori a **struct** (scambio di puntatori)

```
/* ... #include e #define */  
/* ... Typedef struct ... */  
/* ... prototipi */  
  
int main(int argc, char *argv[]) {  
    stud_t elenco[MAXN], *elencoRif[MAXN];  
    int i, ns = leggiStud(argv[1], elenco, MAXN);  
    for (i=0; i<ns; i++)  
        elencoRif[i]=&elenco[i];  
    ordRifStudPerMatr(elencoRif, ns);  
    lavoroSuElencoRifOrd(elencoRif, ns);  
    scriviRifStud(argv[2], elencoRif, ns);  
    return 0;  
}  
  
/* ... leggiStud e scrivistud non cambiano */
```

```
int confrMatrByRef(stud_t *ps1, stud_t *ps2) {  
    ;  
    int n) {
```

Prima carica il vettore di
struct

```
    stud_t *temp;  
    int i, j, imin;  
    for (i=0; i<n-1; i++) {  
        imin = i;  
        for (j = i+1; j < n; j++)  
            if (confrMatrByRef(elR[i].elR[imin])<0)  
                imin]=temp;
```

Poi "aggancia" i puntatori alle struct

```
}
```

Soluzione 2: vettore di puntatori a **struct** (scambio di puntatori)

```
/* ... #incl  
/* ... Typed  
/* ... proto  
int main(int a  
stud_t elenco[MAXN], *elencoRif[MAXN];  
int i, ns = leggiStud(argv[1],elenco,MAXN);  
for (i=0; i<ns; i++)  
    elencoRif[i]=&elenco[i];  
ordRifStudPerMatr(elencoRif,ns);  
lavoroSuElencoRifOrd(elencoRif,ns);  
scriviRifStud(argv[2],elencoRif,ns);  
return 0;  
}  
/* ... leggiStud e scrivistud non cambiano */
```

La funzione di ordinamento
riceve SOLO il vettore di
puntatori a struct

```
int confrMatrByRef(stud_t *ps1, stud_t *ps2) {  
    return ps1->matricola-ps2->matricola;  
}  
  
void ordRifStudPerMatr(stud_t **elR, int n) {  
    stud_t *temp;  
    int i, j, imin;  
    for (i=0; i<n-1; i++) {  
        imin = i;  
        for (j = i+1; j < n; j++)  
            if (confrMatrByRef(elR[j],elR[imin])<0)  
                imin = j;  
        temp=elR[i]; elR[i]=elR[imin]; elR[imin]=temp;  
    }  
}
```

Soluzione 2: vettore di puntatori a **struct** (scambio di puntatori)

```
/* ... #include e #define */  
/* ... Typedef struct ... */  
/* ... prototipi */  
int main() {  
    /* ... */  
  
    Confronta struct a partire  
    dai puntatori  
    Scambia puntatori  
  
    elencoRif[i]=&elenco[i];  
  
    ordRifStudPerMatr(elencoRif,ns);  
  
    lavoroSuElencoRifOrd(elencoRif,ns);  
  
    scriviRifStud(argv[2],elencoRif,ns);  
  
    return 0;  
}  
  
/* ... leggiStud e scrivistud non cambiano */
```

```
int confrMatrByRef(stud_t *ps1, stud_t *ps2) {  
    return ps1->matricola-ps2->matricola;  
}  
  
void ordRifStudPerMatr(stud_t **elR, int n) {  
    stud_t *temp;  
    int i, j, imin;  
    for (i=0; i<n-1; i++) {  
        imin = i;  
        for (j = i+1; j < n; j++)  
            if (confrMatrByRef(elR[j],elR[imin])<0)  
                imin = j;  
        temp=elR[i]; elR[i]=elR[imin]; elR[imin]=temp;  
    }  
}
```

Soluzione 2: vettore di puntatori a **struct** (scambio di puntatori)

```
void scriviRifStud(char *nomeFile, stud_t **elR,int n) {  
    int i;  
  
    FILE *fp = fopen(nomeFile,"w");  
  
    for (i=0; i<n; i++) {  
  
        fprintf(fp, "%s %s %d %f\n", elR[i]->cognome,  
                elR[i]->nome, elR[i]->matr,  
                elR[i]->media);  
  
    }  
  
    fclose(fp);  
}
```

Soluzione 2: vettore di puntatori a **struct** (scambio di puntatori)

```
void scriviRifStud(char *nomeFile, stud_t **elR,int n) {  
    int i;  
  
    FILE *fp = fopen(nomeFile,"w");  
  
    for (i=0; i<n; i++) {  
  
        fprintf(fp, "%s %s %d %f\n", elR[i]->cognome,  
                elR[i]->nome, elR[i]->matr,  
                elR[i]->media);  
    }  
  
    fclose(fp);  
}
```

Come scriviStud, ma parte
da vettore di puntatori:
stud_t **elR
equivale a
stud_t *elR[]

Soluzione 2: vettore di puntatori a **struct** (scambio di puntatori)

```
void scriviRifStud(char *nomeFile, stud_t **elR,int n) {  
    int i;  
  
    FILE *fp = fopen(nomeFile,"w");  
  
    for (i=0; i<n; i++) {  
  
        fprintf(fp, "%s %s %d %f\n", elR[i]->cognome,  
                elR[i]->nome, elR[i]->matr,  
                elR[i]->media);  
    }  
  
    fclose(fp);  
}
```

... quindi si usano le frecce
(->) anziché i punti per
accedere ai campi delle
struct

vettore di puntatori a **struct**: vantaggi

```
/* ... Parti omesse */

stud_t el[MAXN],
*elRif0[MAXN], *elRif1[MAXN], *elRif2[MAXN];
int i, ns = leggiStud(argv[1],el,MAXN);
for (i=0; i<n; i++)
    elRif0[i] = elRif1[i] = elRif2[i] = &el[i];
ordRifStudPerMatr(elRif0,ns);
ordRifStudPerCogn(elRif1,ns);
ordRifStudPerMedia(elRif2,ns);

// altri lavori a partire dai tre elenchi
// scritture a partire dai tre elenchi
...
```

vettore di puntatori a **struct**: vantaggi

```
/* ... Parti omesse */

stud_t el[MAXN],
*elRif0[MAXN], *elRif1[MAXN], *elRif2[MAXN];
int i, ns = leggiStud(argv[1],el,MAXN);
for (i=0; i<n; i++)
    elRif0[i] = elRif1[i] = elRif2[i] = &el[i];
ordRifStudPerMatr(elRif0,ns);
ordRifStudPerCogn(elRif1,ns);
ordRifStudPerMedia(elRif2,ns);

// altri lavori a partire dai tre elenchi
// scritture a partire dai tre elenchi
...
```

Più ordinamenti possibili
insieme

vettore di puntatori a **struct**: vantaggi

```
/* ... Parti omesse */
stud_t el[MAXN],
*elRif0[MAXN], *elRif1[MAXN], *elRif2[MAXN];
int i, ns = leggiStud(argv[1],el,MAXN);
for (i=0; i<n; i++)
    elRif0[i] = elRif1[i] = elRif2[i] = &el[i];
ordRifStudPerMatr(elRif0,ns);
ordRifStudPerCogn(elRif1,ns);
ordRifStudPerMedia(elRif2,ns);
// altri lavori a partire dai tre elenchi
// scritture a partire dai tre elenchi
...
```

Un solo vettore di struct:
non REPLICA i dati

vettore di puntatori a **struct**: vantaggi

```
/* ... Parti omesse */

stud_t el[MAXN],
*elRif0[MAXN], *elRif1[MAXN], *elRif2[MAXN];

int i, ns = leggiStud(argv[1],el,MAXN);
for (i=0; i<n; i++)
    elRif0[i] = elRif1[i] = elRif2[i] = &el[i];
ordRifStudPerMatr(elRif0,ns);
ordRifStudPerCogn(elRif1,ns);
ordRifStudPerMedia(elRif2,ns);

// altri lavori a partire dai tre elenchi
// scritture a partire dai tre elenchi
...
```

Tre vettori di puntatori:
REPLICA (solo) i puntatori

vettore di puntatori a **struct**: vantaggi

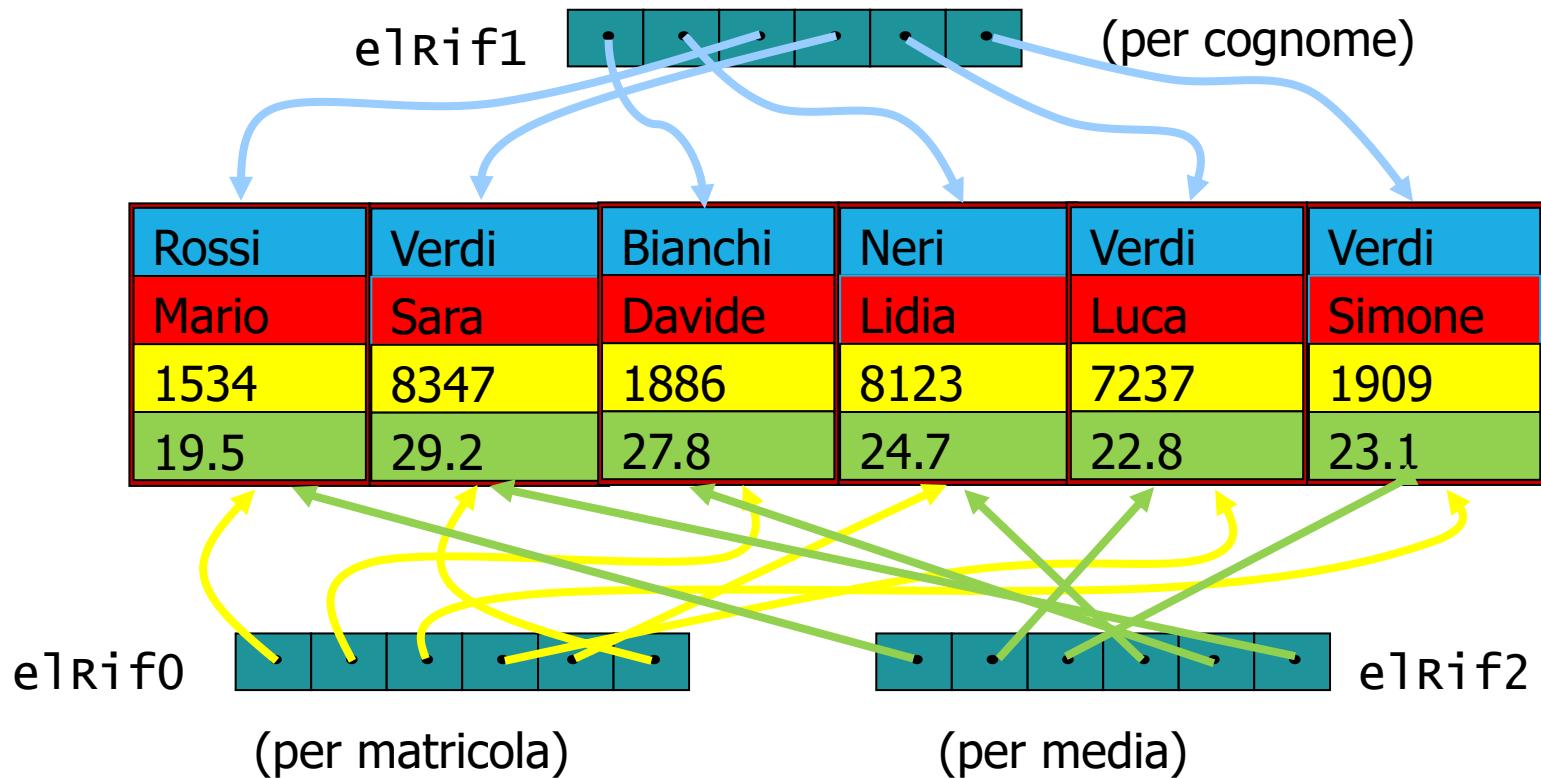
```
/* ... Parti omesse */

stud_t el[MAXN],
*elRif0[MAXN], *elRif1[MAXN], *elRif2[MAXN];
int i, ns = leggiStud(argv[1],el,MAXN);
for (i=0; i<n; i++)
    elRif0[i] = elRif1[i] = elRif2[i] = &el[i];
ordRifStudPerMatr(elRif0,ns);
ordRifStudPerCogn(elRif1,ns);
ordRifStudPerMedia(elRif2,ns);

// altri lavori a partire dai tre elenchi
// scritture a partire dai tre elenchi
...
```

Tre diverse funzioni di ordinamento, applicate ai tre vettori di puntatori

In concreto ...



Puntatori e indici

- Ai dati di un vettore si accede mediante il loro **indice**
- L'indice spazio locale di memoria del vettore equivale al puntatore nello spazio globale di memoria
- **Si può emulare in un vettore il comportamento dei puntatori usando gli indici.**

In concreto ...

e1Ind1

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

(per cognome)

Rossi	Verdi	Bianchi	Neri	Verdi	Verdi
Mario	Sara	Davide	Lidia	Luca	Simone
1534	8347	1886	8123	7237	1909
19.5	29.2	27.8	24.7	22.8	23.1

e1Ind0

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

(per matricola)

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

e1Ind2

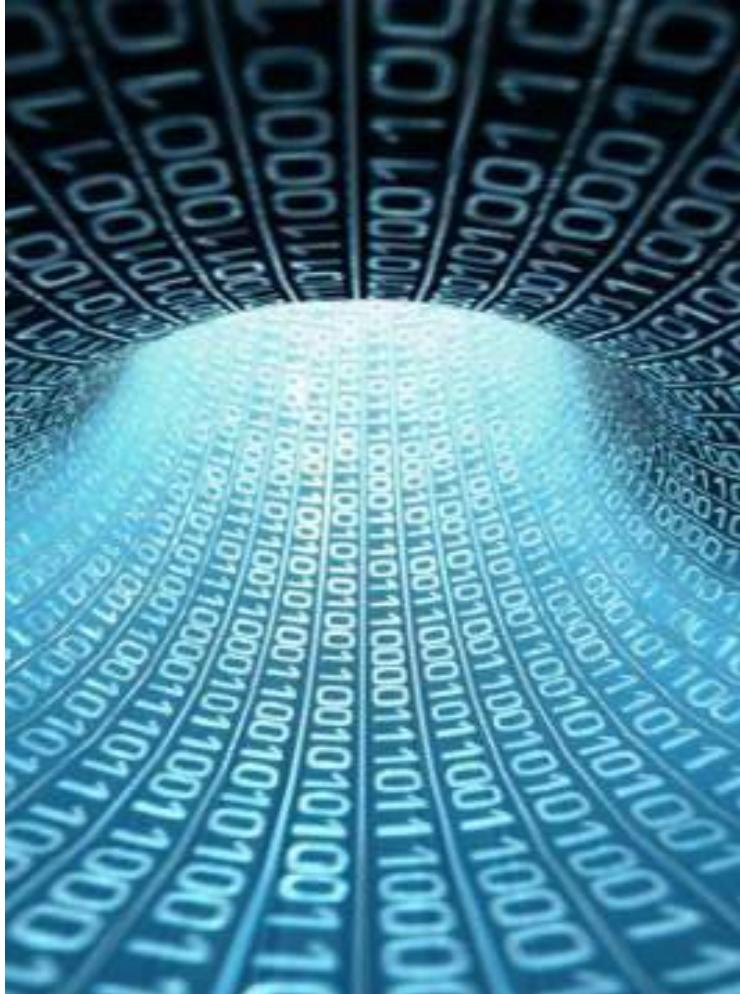
(per media)

Nella versione completa

- Ulteriore esempio con vettore di puntatori
 - Filtro su elenco di dati
- Uso avanzato di puntatori
 - Puntatore a funzione
 - Puntatore generico/opaco (`void *`)

Capitolo 3: L'allocazione dinamica della memoria

PUNTATORI E STRUTTURE DATI DINAMICHE:
ALLOCAZIONE DELLA MEMORIA E
MODULARITÀ IN LINGUAGGIO C



Allocazione

REGOLE E FUNZIONI PER ALLOCARE/DE-ALLOCARE

Allocare = collocare in memoria

- Allocare una variabile = associarvi una porzione di memoria (in cui collocare i dati)
- L'allocazione è:
 - implicita, automatica e statica se gestita dal sistema
 - esplicita se sotto controllo del programmatore
 - dinamica:
 - avviene in fase di esecuzione
 - permette di cambiare la dimensione della struttura dati
 - permette di realizzare “contenitori” cui si aggiungono o tolgo elementi.

Da codice sorgente a eseguibile

Le variabili sono soggette a precise regole di:

- **esistenza**
- **memoria**
- **visibilità.**

Le variabili si distinguono in:

- **globali**
- **locali.**

Variabili **globali**:

- definite al di fuori da funzioni (main incluso)
- permanenti
- visibili dovunque nel file a partire dalla loro definizione
- definite in generale nell'intestazione del file.

Vantaggi:

- accessibili a tutte le funzioni
- non necessario passarle come parametri
- utilizzo semplice ed efficiente

Svantaggi:

- minore modularità, leggibilità, affidabilità

Variabili **locali**:

- variabili definite all'interno delle funzioni (main incluso)
- parametri alle funzioni
- temporanee (iniziano ad esistere quando è chiamata la funzione e cessano quando se ne esce)
- visibili solo nella funzione in cui sono dichiarate.

Compilatore:

- programma che esegue un'analisi
 - lessicale
 - sintattica
 - semantica
- del codice sorgente
- e genera un codice oggetto (in linguaggio macchina)

Il codice oggetto contiene riferimenti a funzioni di libreria

Linker:

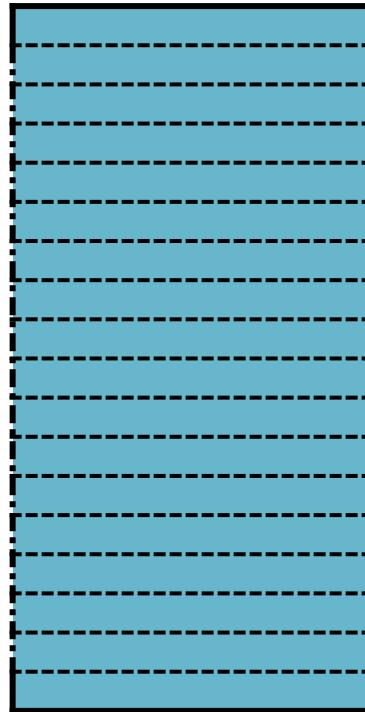
- programma che risolve:
 - i riferimenti a funzioni di libreria
 - I riferimenti reciproci tra più file oggetto.
- e genera un codice eseguibile

Loader:

- modulo del sistema operativo che carica in memoria centrale:
 - Il codice eseguibile (istruzioni)
 - I dati su cui opera

Programma in memoria

```
#define MAX 100
#define MAXR 20
struct stud { ...};
...
struct stud dati[MAX];
int main(void) {
    char nomefile[MAXR];
    FILE *fp;
    ...
}
void ordinaStud(
    struct stud el[], int n){
    int i, j, max;
    ...
};
```



RAM (1 GB)

Programma in memoria

```
#define MAX 100
#define MAXR 20
struct stud { ...};
...
struct stud dati[MAX];
int main(void) {
    char nomefile[MAXR];
    FILE *fp;
    ...
}
void ordinaStud(
    struct stud el[], int n){
    int i, j, max;
    ...
};
```



RAM (1 GB)

Programma in memoria

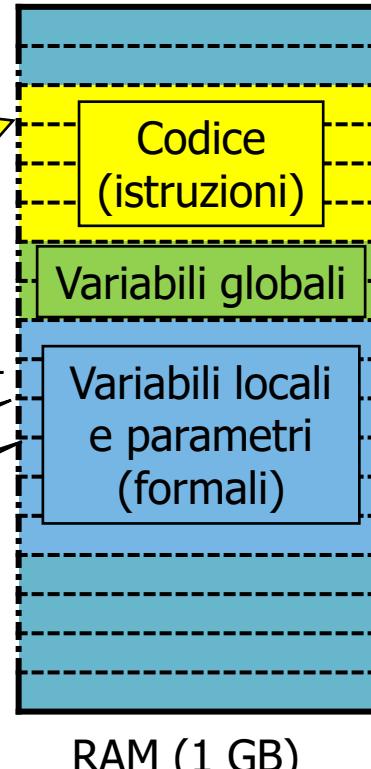
```
#define MAX 100
#define MAXR 20
struct stud { ...};

...
struct stud dati[MAX];
int main(void) {
    char nomefile[MAXR];
    FILE *fp;
    ...
}
void ordinaStud(
    struct stud el[], int n){
    int i, j, max;
    ...
};
```



Programma in memoria

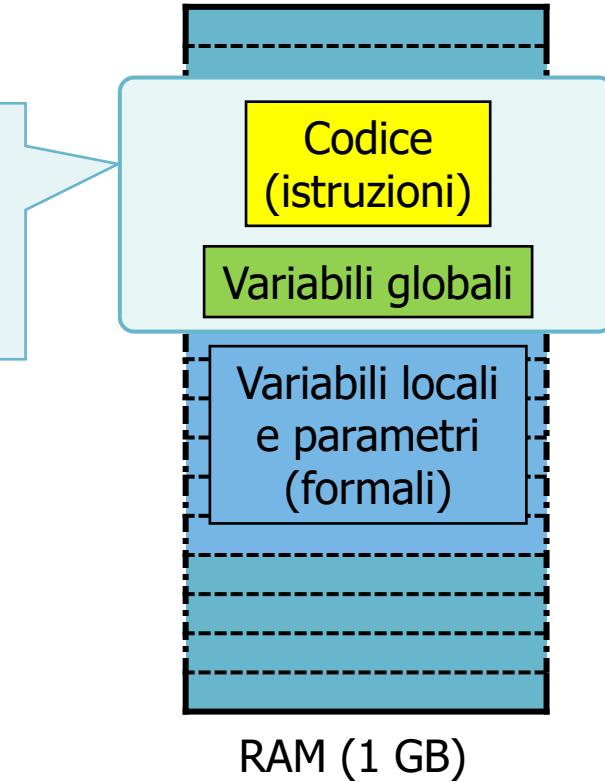
```
#define MAX 100
#define MAXR 20
struct stud { ...};
...
struct stud dati[MAX];
int main(void) {
    char nomefile[MAXR];
    FILE *fp;
    ...
}
void ordinaStud(
    struct stud el[]+
) {
    int i, j, max;
    ...
};
```



Programma in memoria

```
#define MAX 100
#define MAXR 20
struct stud {
    ...
};
int main()
{
    char c;
    FILE *fp;
    ...
}
void ordinaStud(
    struct stud el[], int n){
    int i, j, max;
    ...
};
```

- in memoria (virtualmente) durante tutta l'esecuzione del programma
- indirizzi bassi

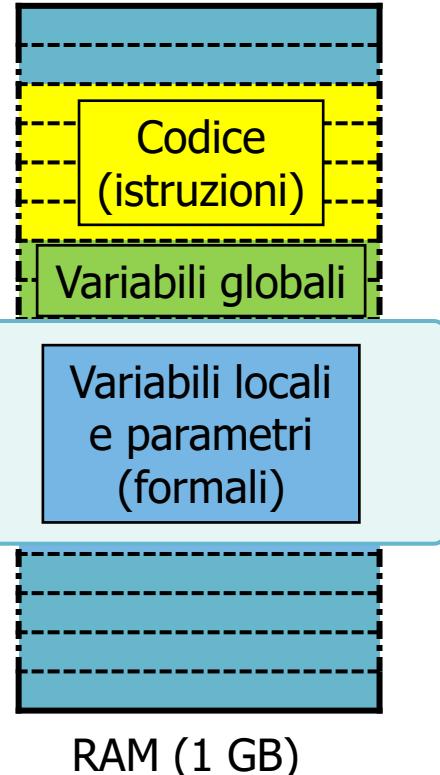


Programma in memoria

```
#define MAX 100
#define MAXR 20
struct stud { ...};
...
struct stud dati[MAX];
int main(void) {
    char nomefile[MAXR];
    FILE *f;
    ...
}
void
str
int
...
};

};
```

- in memoria (virtualmente) durante l'esecuzione della relativa funzione: allocate e de-allocate automaticamente
- stack frame nello stack



Programma in memoria

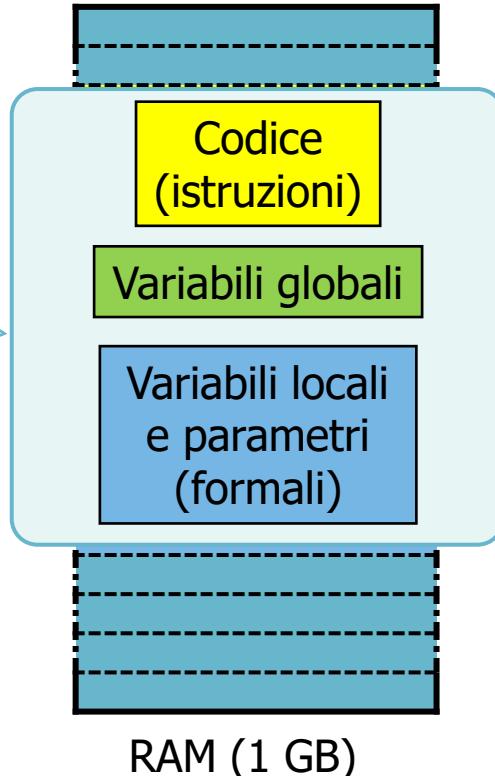
```
#define MAX 100  
#define MAXR 20  
struct stud { ...};  
...
```

Si calcola la quantità di memoria da allocare
è determinata (IMPLICITAMENTE)
dal programmatore:

- istruzioni
- tipo e numero delle variabili

\ - dimensione dei vettori

```
struct stud el[],int n){  
    int i, j, max;  
    ...  
};
```



Regole di allocazione automatica

- dimensioni:

- variabili globali e locali hanno dimensione nota
 - vettori e matrici devono avere dimensione calcolabile
 - i vettori come parametri formali sono puntatori

- variabili globali:

- allocate all'avvio del programma
 - restano in vita per tutto il programma
 - ricordano i valori assegnati da funzioni
 - l'attributo **static** limita la loro visibilità al file in cui compaiono

Regole di allocazione automatica (2)

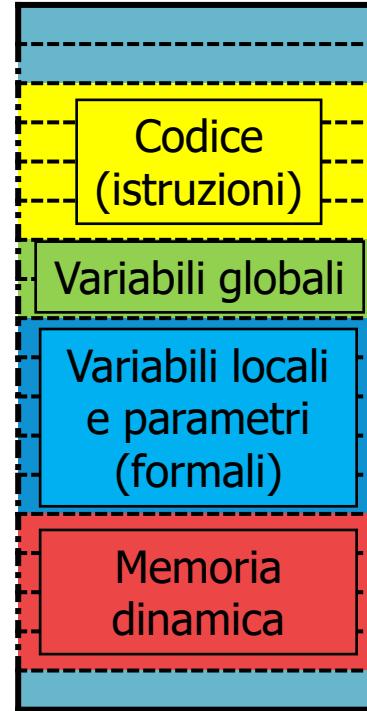
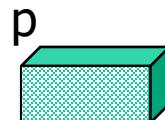
- variabili locali:
 - raggruppate con i parametri formali in uno stack frame
 - allocate nello stack ad ogni chiamata della funzione
 - deallocate automaticamente all'uscita dalla funzione
 - non ricordano i valori precedenti
- variabili locali con attributo **static**:
 - visibilità limitata alla funzione
 - allocate assieme alle variabili globali
 - ricordano i valori (della chiamata precedente)

Allocazione/rilascio esplicativi

- Osservazione: manca un modo per poter decidere, durante l'esecuzione di un programma:
 - la creazione/distruzione un dato
 - il dimensionamento di un vettore o matrice
- Soluzione: istruzioni per allocare e de-allocare dati (memoria) **in modo esplicito**:
 - in funzione di dati forniti da chi esegue il programma
 - allocazioni e de-allocazioni sono (ovviamente) previste e gestite dall'autore del programma
 - la componente del sistema operativo che si occupa di allocazione/deallocazione è **l'allocatore di memoria dinamica**
 - la memoria dinamica si trova in un'area detta **heap**
 - alla memoria dinamica si accede **solo mediante puntatore**

Allocazione e rilascio esplicativi: malloc e free

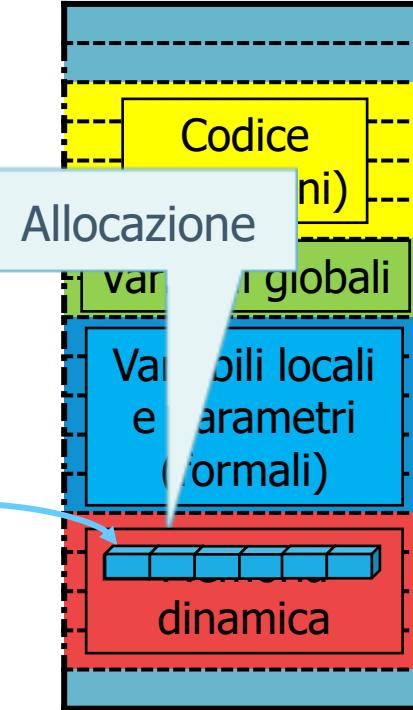
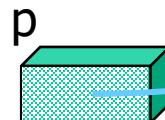
```
int main(void){  
    int *p = malloc(...);  
    ...  
    /*  
       p usato come vettore  
    */  
    free(p);  
}
```



RAM (1 GB)

Allocazione e rilascio espliciti: malloc e free

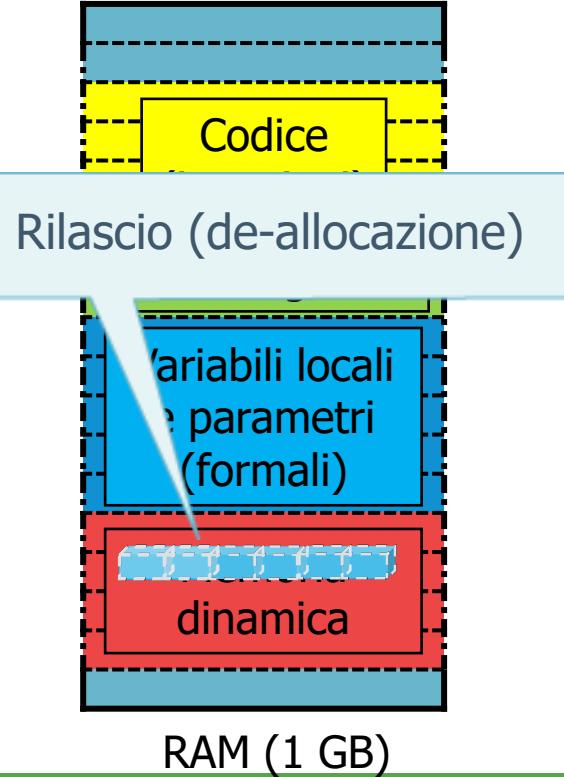
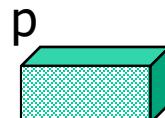
```
int main(void){  
    int *p = malloc(...);  
    ...  
    /*  
     * p usato come vettore  
     */  
    free(p);  
}
```



RAM (1 GB)

Allocazione e rilascio espliciti: malloc e free

```
int main(void){  
    int *p = malloc(...);  
    ...  
    /*  
     * p usato come vettore  
     */  
    free(p);  
}
```



Tipologie di creazione/utilizzo di strutture dati

- dimensione
 - fissa determinata in fase di esecuzione
 - modificabile (aumentabile o diminuibile) mediante riallocazione
 - «contenitore» di singoli dati allocati a pezzi (aggiunta o eliminazione di elementi): es. liste concatenate, alberi, code, tabelle di simboli (tutte viste nel corso).
- Fasi di una struttura dati dinamica:
 - creazione (allocazione) esplicita
 - utilizzo con possibilità di:
 - riallocazione
 - Inserimenti
 - cancellazioni
 - distruzione (de-allocazione) esplicita.

La funzione malloc

- La memoria in C viene allocata dinamicamente tramite la funzione malloc:

```
void* malloc (size_t size);
```

- size è il numero (intero) di byte da allocare
- il valore di ritorno è un puntatore:
 - contiene l'indirizzo iniziale della memoria allocata (NULL se non c'è memoria disponibile)
 - è tipo **void ***, tale da poter essere assegnato a qualunque tipo di puntatore

La funzione malloc

- La memoria in C viene allocata dinamicamente tramite la funzione malloc:

```
void* malloc (size_t size);
```

- size è il numero (intero) di byte da allocare
- il valore di ritorno è un puntatore:
 - contiene l'indirizzo iniziale della memoria allocata (NULL se non c'è memoria disponibile)
 - è tipo **void ***, tale da poter essere assegnato a qualsiasi tipo di puntatore

size_t è un intero senza segno
(si può usare come parametro attuale un int)

Le regole

- Per usare `malloc` occorre includere `<stdlib.h>`
- La **dimensione** del dato è **responsabilità del programmatore**
 - Solitamente si ricorre all'operatore `sizeof` per determinare la dimensione (in byte) di un dato:
 - `sizeof(<tipo>)`
 - `sizeof <espressione riconducibile a tipo>`
- Al dato allocato si accede **unicamente tramite puntatore**
 - Il puntatore ritornato è **opaco**, tocca al programmatore passare al tipo desiderato mediante assegnazione a **opportuna variabile puntatore**

Forma generale di chiamata a malloc:

Quattro forme, suddivise da

- tipo di sizeof (basato su un tipo o su una variabile/espressione)
- dalla presenza di cast esplicito o no

- cast implicito

```
p = malloc(sizeof (<tipo>));
```

```
p = malloc(sizeof <espr>);
```

- cast esplicito (non obbligatorio, ma permette controllo di errore se p non è del tipo corretto):

```
p = (<tipo> *) malloc(sizeof (<tipo>));
```

```
p = (<tipo> *) malloc(sizeof <espr>);
```

Esempi (1)

Data una **struct stud** ed una variabile **s** puntatore a **struct stud**

```
#define MAX 20
struct stud {char cognome[MAX];int matr; };
struct stud *s;
```

per calcolare la dimensione **struct** ci sono 2 modi:

sizeof(struct stud)

sizeof(*s)

dato puntato da **s**

Esempi (2)

Allocazione di una **struct stud** puntata dalla variabile (puntatore) **s**:

- Con cast implicito:

```
s = malloc(sizeof(struct stud));
```

```
s = malloc(sizeof(*s));
```

- Con cast esplicito:

```
s = (struct stud *)malloc(sizeof(struct stud));
```

```
s = (struct stud *)malloc(sizeof(*s));
```

Errori comuni

- dimensione richiesta inferiore alla necessaria:
 - a causa di uso del tipo errato in `sizeof`

```
double *pd;  
struct stud *ps, *v;  
int n;  
  
...  
pd = malloc (sizeof (int));
```

dovrebbe essere
`sizeof (double)`

Un cast esplicito avrebbe reso più semplice
l'identificazione dell'errore:

```
pd = (double *)malloc (sizeof (double));
```

Errori comuni

- dimensione richiesta inferiore alla necessaria:
 - a causa di uso del tipo errato in `sizeof`

```
double *pd;  
struct stud *ps, *v;  
int n;
```

...

```
pd = malloc
```

dovrebbe essere
`sizeof (double)`

Un cast esplicito

l'identificazione dell'err

L'errore SI VEDE!

(da una parte double, dall'altra int

```
pd = (double *)malloc (sizeof (int));
```

Errori comuni (2)

sizeof(struct stud)

2. uso del tipo puntatore a dato al posto del tipo del dato puntato

```
ps=(struct stud *)malloc(sizeof(struct stud *));
```

3. dimensione n (serve per fabbricare un vettore) omessa o errata

```
v = malloc (sizeof *v);
```

n * sizeof (*v)

4. omissione di sizeof

```
v = (struct stud *)malloc (n);
```

n*sizeof(struct stud)

Errori comuni

5. assegnazione di puntatore incompatibile con il dato:

```
struct stud *
```

```
struct stud **ps;  
...  
ps = malloc(n*sizeof (struct stud));
```

il cast esplicito permette al compilatore
di segnalare l'errore:

```
struct stud **ps;  
...  
ps=(struct stud *)malloc(n*sizeof (struct stud));
```

Errori comuni

5. assegnazione di puntatore incompatibile con il dato:

```
struct stud **ps;
```

```
...
```

```
ps = malloc(n*sizeof (struct stud));
```

struct stud *

```
struct stud **ps;
```

```
...
```

```
ps=(struct stud **)malloc(n*sizeof (struct stud *));
```

Versione corretta: si voleva un vettore
di puntatori

Errori comuni

5. assegnazione di puntatore incompatibile con il dato: secondo esempio

```
struct stud *ps;  
int *pi;  
...
```

```
pi = malloc (sizeof (struct stud));
```

pi e sizeof(struct stud) non sono compatibili

```
struct stud *ps;  
int *pi;  
...
```

```
pi=(struct stud *)malloc(sizeof (struct stud));
```

il cast esplicito permette al compilatore di segnalare l'errore:

pi deve puntare a intero

Errori comuni

5. assegnazione di puntatore incompatibile con il dato: terzo esempio

pi e sizeof(*ps) non sono compatibili: non lo dice il compilatore
Si deve vedere «a occhio»

```
struct stud *ps;  
int *pi;  
...  
pi = malloc (sizeof *ps);
```

Conseguenze degli errori sui puntatori

Errori individuati dal compilatore: occorre correggerli (il programma non compila correttamente)

Errori NON individuati dal compilatore

- succede spesso, perché la `malloc` riceve solo un NUMERO e ritorna un INDIRIZZO, quindi non conosce le «intenzioni» del programmatore
- Due possibilità
 - La dimensione allocata è SUPERIORE a quella necessaria. Non succede NULLA, se non lo SPRECO di memoria (allocata e non usata)
 - La dimensione allocata è inferiore al necessario
 - NON capita di solito NULLA nell'allocazione
 - DOPO, mentre si accede ai dati, SI RISCHIA DI USARE (mediante puntatore) MEMORIA NON ALLOCATA oppure ALLOCATA per un altro DATO

Cosa succede DOPO, se si accede a memoria non allocata oppure allocata ad altri dati?

- **crash** del programma per accesso a indirizzo non ammesso (esito auspicabile)
- accesso ad indirizzo legale, ma al di fuori della struttura dati allocata (errore subdolo) : si **SPORCA un altro dato**

Errori comuni

5. assegnazione di puntatore incompatibile con il dato: terzo esempio

Errore probabilmente non distruttivo: struct stud è più grande di un intero

```
struct stud *ps;  
int *pi;  
...  
pi = malloc (sizeof *ps);
```

Errori comuni

5. assegnazione di puntatore incompatibile con il dato: terzo esempio

Errore con conseguenze: un intero è probabilmente più piccolo di struct stud

```
struct stud *ps;  
int *pi;  
...  
ps = malloc (sizeof *pi);
```

Errori comuni (2)

- uso del tipo puntatore a dato al posto del tipo del dato puntato

Si rischia di allocare sempre meno del necessario

```
ps=(struct stud *)malloc(sizeof(struct stud *));
```

sizeof(struct stud)

- dimensione n (serve per fabbricare un vettore) omessa o errata

```
v = malloc (sizeof *v);
```

n * sizeof (*v)

- omissione di sizeof

```
v = (struct stud *)malloc (n);
```

n*sizeof(struct stud)

Memoria dinamica insufficiente

Succede poco, indica che non c'è più memoria allocabile nell'heap, per la dimensione richiesta (provare ad allocare $n * \text{sizeof(int)}$, con n molto grande):

- `malloc` ritorna `NULL`
- opportuno testare, segnalando o uscendo con `exit` o `return`

```
int *p;  
...  
p = malloc(sizeof(int));  
if (p == NULL)  
    printf("Errore di allocazione\n");  
else  
    ...
```

La funzione `calloc`

- Equivale a:

`malloc(n*size);`

con memoria ritornata azzerata

```
void* calloc (size_t n, size_t size);
```

La `calloc` ha costo (in tempo) $O(n)$, a causa dell'azzeramento, mentre `malloc` è $O(1)$. Tuttavia, molto sovente, l'azzeramento è opportuno lo necessario (andrebbe fatto comunque)

La funzione `free`

- Tutta la memoria allocata dinamicamente con `malloc/calloc` viene restituita tramite la funzione `free` (`<stdlib.h>`)

```
void free (void* p);
```

- `p` punta alla memoria (precedentemente allocata) da liberare
- Viene di solito chiamata quando è terminato il lavoro sulla struttura dinamica, affinchè la memoria possa essere riutilizzata
- ATTENZIONE: l'allocatore mantiene internamente una tabella di ciò che ha allocato:
 - Si può solo chiamare `free` per un indirizzo precedentemente ritornato da `malloc/calloc` (o `realloc`)
 - NON si può liberare un PEZZO della memoria ottenuta (allocata) con `malloc/calloc` (o `realloc`)

Uso di `free` consigliato, ma non obbligatorio

- al termine dell'esecuzione di un programma la memoria viene comunque liberata (in molti casi questo è sufficiente)
- Ma è possibile che SIA OPPORTUNO LIBERARE per poter OTTENERE nuova memoria DURANTE l'esecuzione: es. programma che ripete iterativamente un lavoro che richiede allocazione
- Attenzione ai **memory leak** (dimenticare di de-allocare):
 - la mancata de-allocazione di una porzione di memoria
 - Non si può riutilizzare la memoria per un nuovo dato da allocare. Effetto: aumenta la probabilità (con programmi che allocano molto) di `malloc/calloc` che ritornano NULL.

Esempio di memory leak

```
int *vett = malloc(10 * sizeof(int));  
...  
// uso di vett, SENZA liberazione  
vett = malloc(25 * sizeof(int));
```

ora la porzione di memoria allocata dalla prima `malloc` non è più indirizzabile né utilizzabile per ulteriori allocazioni (è ancora allocata, ma non puntata e non usata)

La funzione `realloc`

- In C la dimensione della memoria allocata può essere modificata aggiungendo o togliendo una porzione in fondo tramite `realloc`:

```
void* realloc (void* p, size_t size);
```

- `p` punta a memoria precedentemente allocata
- `size` è la nuova dimensione richiesta (maggiore o minore)
- il valore di ritorno è un puntatore

- Uso tipico:

```
p = malloc (oldSize);
...
p = realloc (p, newSize);
// si lavora sulla struttura dati espansa o ristretta
...
```

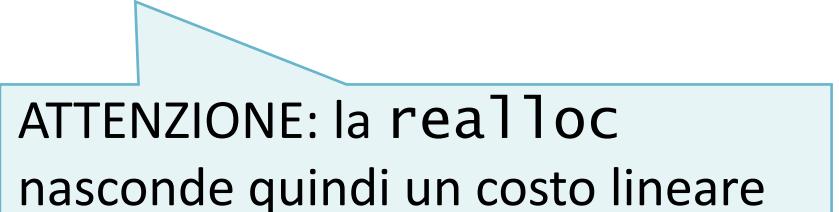
`newSize` è la nuova dimensione, diversa da `oldSize`

Cosa succede?

- La riduzione della dimensione è sempre possibile
- L'aumento della dimensione può:
 - essere impossibile (non c'è memoria extra disponibile) e si ritorna NULL
 - essere possibile: esiste memoria disponibile contigua, alla FINE del blocco già allocato (quindi espandibile). Si ritorna il puntatore p invariato
 - essere possibile, ma altrove (non c'è spazio sufficiente alla fine del blocco):
 - Si alloca un nuovo intervallo in memoria (di dimensione newSize)
 - si ricopia con costo lineare nella dimensione ($O(n)$) il contenuto della vecchia porzione di memoria nella nuova
 - si ritorna un puntatore p aggiornato.

Cosa succede?

- La riduzione della dimensione è sempre possibile
- L'aumento della dimensione può:
 - essere impossibile (non c'è memoria extra disponibile) e si ritorna NULL
 - essere possibile: esiste memoria disponibile contigua, alla FINE del blocco già allocato (quindi espandibile). Si ritorna il puntatore p invariato
 - essere possibile, ma altrove (non c'è spazio sufficiente alla fine del blocco):
 - Si alloca un nuovo intervallo in memoria (di dimensione newSize)
 - si ricopia con costo lineare nella dimensione ($O(n)$) il contenuto della vecchia porzione di memoria nella nuova
 - si ritorna un puntatore p aggiornato.



ATTENZIONE: la `realloc` nasconde quindi un costo lineare

Implementazione ad alto livello della `realloc`

```
// non è la vera implementazione - serve solo per capirla
void* realloc (void* p, size_t size) {
    size_t oldSize = cercaDimensione (p, ...); // cerca in tabella la dimensione precedente
    if /*si può espandere o ridurre*/ {
        cambiaDimensione (p, ...); // togli o aggiungi un pezzo in fondo
        return p;
    }
    else {
        void *newp = malloc(size); // nuova allocazione
        copiaMemoria(newp,p,min(size,oldSize));
        free(p);
        return newp;
    }
}
```

Implementazione ad alto livello della `realloc`

```
// non è la vera implementazione - serve solo per capirla
void* realloc (void* p, size_t size) {
    size_t oldSize = cercaDimensione (p, ...); // cerca in tabella la dimensione precedente
    if /*si può espandere o ridurre*/ {
        cambiaDimensione (p, ...); // togli o aggiungi un pezzo in fondo
        return p;
    }
    else {
        void *newp = malloc(size); // nuova allocazione
        copiaMemoria(newp,p,min(size,oldSize));
        free(p);
        return newp;
    }
}
```



copiaMemoria ha complessità
 $O(\min(\text{size}, \text{oldSize}))$

Strutture dati dinamiche

VETTORI E MATRICI ALLOCATI DINAMICAMENTE

Strutture dati dinamiche

La dimensione delle strutture dati dinamiche:

- è nota solo in fase di esecuzione
- può variare nel tempo.

Possono anche contenere dati aggregati in quantità non note a priori e variabili nel tempo (liste, Cap. 4).

Vettori dinamici

- La dimensione è nota solo in fase di esecuzione del programma
- Può variare per riallocazione
- Si evita il sovradimensionamento del vettore nonché i suoi limiti:
 - è necessario conoscere la dimensione massima (costante)
 - data la dimensione massima e una parte iniziale del vettore effettivamente utilizzata, quella restante è sprecata.
- Soluzione:
 - uso di puntatore, sfruttando la dualità puntatore-vettore, con entrambi le notazioni
 - allocazione mediante `malloc/calloc`
 - rilascio mediante `free`
 - Il resto è identico al vettore sovradimensionato in modo statico.

Esempio

- Acquisire da tastiera una sequenza di numeri reali e memorizzarli in un vettore
- Stamparli successivamente in ordine inverso a quello di acquisizione
- La quantità di dati non è nota al programmatore, né sovradimensionabile, ma è acquisita come primo dato da tastiera

invertiOrdine.c

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *v; int N, i;
    printf("N? "); scanf("%d", &N);
    v = (float *) malloc (N*(sizeof (float)));
    if (v==NULL) exit(1);
    printf("Inserisci %d elementi\n", N);
    for (i=0; i<N; i++) {
        printf("El. %d: ", i);
        scanf("%f", &v[i]);
    }
}
```

```
printf("Dati in ordine inverso\n");
for (i=N-1; i>=0; i--)
    printf("El. %d: %f\n", i, v[i]);
free(v);
return 0;
};
```

invertiOrdine.c

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *v; int N, i;
    printf("N? "); scanf("%d", &N);
    v = (float *) malloc (N*(sizeof (float)));
    if (v==NULL) exit(1);
    printf("Inserisci %d elementi \n", N);
    for (i=0; i<N; i++) {
        printf("El. %d: ", i);
        scanf("%f", &v[i]);
    }
}
```

```
printf("Dati in ordine inverso\n");
for (i=N-1; i>=0; i--)
    printf("El. %d: %f\n", i, v[i]);
free(v);
return 0;
};
```

allocazione vettore dinamico

Controllo allocazione riuscita

invertiOrdine.c

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    float *v; int N, i;
    printf("N? "); scanf("%d", &N);
    v = (float *) malloc (N*(sizeof (float)));
    if (v==NULL) exit(1);
    printf("Inserisci %d elementi\n", N);
    for (i=0; i<N; i++) {
        printf("El. %d: ", i);
        scanf("%f", &v[i]);
    }
}
```

input

```
printf("Dati in ordine inverso\n");
for (i=N-1; i>=0; i--)
    printf("El. %d: %f\n", i, v[i]);
free(v);
return 0;
};
```

output

invertiOrdine.c

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *v; int N, i;
    printf("N? "); scanf("%d", &N);
    v = (float *) malloc (N*(sizeof (float)));
    if (v==NULL) exit(1);
    printf("Inserisci %d elementi\n", N);
    for (i=0; i<N; i++) {
        printf("El. %d: ", i);
        scanf("%f", &v[i]);
    }
}
```

```
printf("Dati in ordine inverso\n");
for (i=N-1; i>=0; i--)
    printf("El. %d: %f\n", i, v[i]);
free(v);
return 0;
};
```

de-allocazione

La dimensione del vettore dinamico

- ATTENZIONE: bisogna conoscere il numero di dati per creare e usare il vettore dinamico!
- Se il numero di dati (ignoto) fosse segnalato da un terminatore (es. input del valore 0):
 - 2 letture da file (quindi non va bene da tastiera): la prima per calcolare il numero di dati, la seconda per memorizzarli
 - uso di `realloc`, tenendo presente il suo costo lineare nascosto

Esempio (modificato)

- Acquisire da tastiera una sequenza di numeri reali e memorizzarli in un vettore
- Stamparli successivamente in ordine inverso a quello di acquisizione
- La quantità di numeri non è nota al programmatore, né sovradimensionabile, ~~ma è acquisita come primo dato da tastiera~~

Esempio (modificato)

- Acquisire da tastiera una sequenza di numeri reali e memorizzarli in un vettore
- Stamparli successivamente in ordine inverso a quello di acquisizione
- La quantità di numeri non è nota al programmatore, né sovradimensionabile. I dati terminano con un dato non valido.

Ri-allocazione: soluzione A

Il vettore viene ri-allocato ad ogni iterazione:

- vettore dinamico di dimensione iniziale $N=1$
- riallocazione ad ogni nuovo dato con N incrementato di 1
- **$O(N^2)$** : *pur trattandosi di un caso peggiore, difficilmente realizzabile, può accadere (quasi ogni volta che si alloca, non si riesce ad «allargare», e si deve «spostare» il vettore).*

Ri-allocazione: soluzione A

Il vettore viene ri-allocato ad ogni iterazione:

- vettore dinamico di dimensione iniziale $N=1$
- riallocazione ad ogni nuovo dato con N incrementato di 1
- $O(N^2)$: *pur trattandosi di un caso peggiore, difficilmente realizzabile, può accadere (quasi ogni volta che si allarga, non si riesce ad «allargare», e si deve «spostare» il vettore).*

SCONSIGLIATA !

inverteOrdine.c (con ri-allocazione A)

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *v; int N=1, i;
    v = malloc(N*(sizeof (float)));
    printf("Inserisci elementi\n");
    printf("Elemento 0: ");
    while (scanf("%f", &d)>0) {
        if (i==N) {
            // attivato sempre eccetto con i==0
            N = N+1;
            v = realloc(v,N*sizeof(float));
            // controllo di errore omesso
        }
        v[i++] = d;
        printf("Elemento %d: ", i) ;
    }
    printf("Dati in ordine inverso\n");
    for (i=N-1; i>=0; i--)
        printf("El. %d: %f\n", i, v[i]);
    free(v);
    return 0;
};
```

inverteOrdine.c (con ri-allocazione A)

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    float *v; int N=1, i;
    v = malloc(N*(sizeof (float)));
    printf("Inserisci elementi\n");
    printf("Elemento 0: ");
    while (scanf("%f", &d)>0) {
        if (i==N) {
            // attivato sempre eccetto con i==0
            N = N+1;
            v = realloc(v,N*sizeof(float));
            // controllo di errore omesso
        }
        ++] = d;
        printf("Elemento %d: ", i) ;
    }
    printf("Dati in ordine inverso\n");
    for (i=N-1; i>=0; i--)
        printf("El. %d: %f\n", i, v[i]);
    free(v);
    return 0;
};
```

allocazione iniziale

Riallocazione
(con N incrementato)

Ri-allocazione: soluzione B

Il vettore viene ri-allocato un numero logaritmico di volte:

- vettore dinamico di dimensione iniziale $N=1$
- controllo se vettore pieno
- riallocazione se pieno con N raddoppiato (sovradimensionamento)
- **$O(N \log N)$** : compromesso memoria (sovra-dimensionata, al peggio quasi doppia) tempo (linearitmico anziché quadratico)

CONSIGLIATA !

inverteOrdine.c (con ri-allocazione B)

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *v; int N, MAXN=1, i=0;
    v = malloc (MAXN*(sizeof (float)));
    printf("Inserisci elementi\n");
    printf("Elemento 0: ");
    while (scanf("%f", &d)>0) {
        if (i==MAXN) {
            // numero logaritmico di attivazioni
            MAXN = MAXN*2;
            v = realloc(v,MAXN*sizeof(float));
            // controllo di errore omesso
        }
        v[i++] = d;
        printf("Elemento %d: ", i) ;
    }
    N = i; // compreso tra MAXN/2 e MAXN
    printf("Dati in ordine inverso\n");
    for (i=N-1; i>=0; i--)
        printf("El. %d: %f\n", i, v[i]);
    free(v);
    return 0;
};
```

inverteOrdine.c (con ri-allocazione B)

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *v; int N, MAXN=1, i=0;
    v = malloc (MAXN*(sizeof (float)));
    printf("Inserisci elementi\n");
    printf("Elemento 0: ");
    while (scanf("%f", &d)>0) {
        if (i==MAXN) {
            // numero logaritmico di attivazioni
            MAXN = MAXN*2;
            v = realloc(v,MAXN*sizeof(float));
            // controllo di errore omesso
        }
        allocation iniziale
        controllo
        se pieno
        riallocazione
        con raddoppio
    }
    v[i++] = d;
    printf("Elemento %d: ", i) ;
    i++;
    / compreso tra MAXN/2 e MAXN
    printf("Dati in ordine inverso\n");
    for (i=N-1; i>=0; i--)
        printf("El. %d: %f\n", i, v[i]);
    free(v);
    return 0;
}
```

Matrici dinamiche

Due possibilità:

- Soluzione 1 (meno flessibile):
 - vettore **MONODIMENSIONALE** dinamico di **nr x nc** elementi
 - organizzazione **manuale** di righe e colonne su vettore: l'elemento (i, j) si trova in posizione $nc*i + j$.
- Soluzione 2:
 - vettore **BIDIMENSIONALE** dinamico di **nr** puntatori a righe
 - iterazione sulle **nr** righe (**nc colonne**) per allocare un vettore di tipo desiderato di **nc (nr)** elementi
 - Il resto è identico alla matrice sovradimensionata in modo statico.

Esempio

- Acquisire da tastiera una sequenza di numeri reali e memorizzarli in una matrice
- Stampare successivamente la matrice trasposta (righe e colonne scambiate di ruolo)
- Le dimensioni della matrice (righe e colonne) non sono note al programmatore, né sovradimensionabili, ma sono acquisite come primo dato da tastiera

Con vettore dinamico monodimensionale

```
...  
  
float *v;  
  
int nr,nc,i,j;  
  
printf("nr nc: "); scanf("%d%d", &nr, &nc);  
v = (float *) malloc(nr*nc*(sizeof (float)));  
if (v==NULL) exit(1);  
for (i=0; i<nr; i++) {  
    printf("Inserisci riga %d\n", i);  
    for (j=0; j<nc; j++)  
        scanf("%f", &v[nc*i+j]);  
}
```

```
printf("Matrice trasposta\n");  
for (j=0; j<nc; j++) {  
    for (i=0; i<nr; i++)  
        printf("%6.2f", v[nc*i+j]);  
    printf("\n");  
}  
free(v);  
...
```

Con vettore dinamico monodimensionale

```
...
float *v;
int nr,nc,i,j;

printf("nr nc: "); scanf("%d%d", &nr, &nc);
v = (float *) malloc(nr*nc*(sizeof (float)));
if (v==NULL) exit(1);
for (i=0; i<nr; i++) {
    printf("Inserisci riga %d\n", i);
    for (j=0; j<nc; j++)
        scanf("%f", &v[nc*i+j]);
}
```

allocazione

```
printf("Matrice trasposta\n");
for (j=0; j<nc; j++) {
    for (i=0; i<nr; i++)
        printf("%6.2f", v[nc*i+j]);
    printf("\n");
}
free(v);
```

controllo
di errore

Gestione manuale: [nc*i+j]

Con matrice dinamica bidimensionale

```
...  
  
float **m;  
  
int nr,nc,i,j;  
  
printf("nr nc: "); scanf("%d%d", &nr, &nc);  
m = (float **) malloc(nr*(sizeof (float *)));  
if (m==NULL) exit(1);  
for (i=0; i<nr; i++) {  
    printf("Inserisci riga %d\n", i);  
    m[i] = (float *) malloc(nc*sizeof (float));  
    if (m[i]==NULL) exit(1);  
}
```

```
    for (j=0; j<nc; j++)  
        scanf("%f", &m[i][j]);  
    }  
    printf("Matrice trasposta\n");  
    for (j=0; j<nc; j++) {  
        for (i=0; i<nr; i++)  
            printf("%6.2f", m[i][j]);  
        printf("\n");  
    }  
    for (i=0; i<nr; i++)  
        free(m[i]);  
    free(m);  
    ...
```

Con matrice dinamica bidimensionale

```
...
float **m;
int nr,nc,i,j;

printf("nr nc: ");
scanf("%d%d", &nr, &nc);

m = (float **) malloc(nr*(sizeof (float *)));
if (m==NULL) exit(1);
for (i=0; i<nr; i++) {
    printf("Inserisci riga %d\n", i);
    m[i] = (float *) malloc(nc*sizeof (float));
    if (m[i]==NULL) exit(1);
    for (j=0; j<nc; j++)
        m[i][j] = (float) rand()/(float) RAND_MAX;
}
for (i=0; i<nr; i++)
    free(m[i]);
free(m);
```

vettore di vettori di float

float **: vettore di puntatori

allocazione di vettore di
nr puntatori a riga

for (i=0; i<nr; i++)

printf("%6.2f", m[i][j]);

printf("\n");

}

for (i=0; i<nr; i++)

free(m[i]);

vettore di float: uno per riga

Con matrice dinamica bidimensionale

```
...
float **m;
int nr,nc,i,j;

printf("nr nc: ");
scanf("%d%d", &nr, &nc);

m = (float **) malloc(nr*(sizeof (float *)));
if (m==Prima le singole righe
for (i= (una alla volta)
printf("Inserisci riga %u: ", i);
m[i] = (float *) malloc(nc*(sizeof (float)));
m[i][0] = 1;
for (j=1; j<nc; j++)
    m[i][j] = 0;
Poi il vettore dei puntatori alle righe
if (m==liberazione memoria
dinamica (niente di automatico)
for (j=0; j<nc; j++)
    printf("%f", &m[i][j]);
Matrice trasposta\n");
for (j=0; j<nc; j++) {
    for (i=0; i<nr; i++)
        printf("%6.2f", m[i][j]);
    printf("\n");
}
for (i=0; i<nr; i++)
    free(m[i]);
free(m);
...
```

Vettori e matrici creati da funzioni

- Un vettore o matrice dinamici sono accessibili a partire da un puntatore
- Il puntatore è un dato: può quindi essere passato e/o ritornato da funzioni, come pure copiato tra variabili
- Le variabili puntatore esistono fintanto che è in essere la funzione dove sono dichiarate e sono visibili in essa
- Matrici e vettori creati in una funzione f (ad esempio un vettore dinamico v) sono
 - a volte usate solamente nella funzione f in cui sono allocate e de-allocate
 - altre volte può essere necessario che siano accessibili da altre funzioni (chiamate da f , oppure che chiamano f)

- Per fare in modo che una funzione g (chiamata da f) usi il vettore v (un puntatore), è sufficiente passare il puntatore per valore:

$g(\dots, v, \dots)$

- Per rendere v accessibile al programma chiamante (funzione h che chiama f):
 - si dichiara il puntatore come variabile globale (**sconsigliato!**)
 - si inserisce il puntatore tra i parametri della funzione e lo si modifica (passaggio by pointer/reference, quindi, in C “by value” di un puntatore a puntatore). Ad esempio, h chiamerà: $f(\dots, &v, \dots)$
 - si ritorna il puntatore come valore di ritorno della funzione.
Ad esempio: $v = f(\dots)$

Esempio

- Si realizzino due funzioni che allocano (`malloc2d`) o liberano (`free2d`) una matrice bidimensionale di elementi di tipo `Item` con `nr` righe e `nc` colonne.
- La funzione di allocazione `malloc2d` ha 2 versioni:
 - Puntatore ritornato come **risultato**: `malloc2dR` dove il puntatore alla matrice è restituito come valore di ritorno della funzione
 - Passaggio by **pointer/reference**: `malloc2dP` dove il puntatore alla matrice è restituito tra i parametri della funzione

Puntatore come valore di ritorno della funzione

```
typedef ... Item;  
Item **malloc2dR(int nr, int nc);  
void free2d(Item **m, int nr);  
...  
void h /* parametri formali */ {  
    Item **matr;  
    int nr, nc;  
    ...  
    matr = malloc2dR(nr, nc);  
    ... /* lavoro su matr */  
    free2d(matr, nr);  
}
```

```
Item **malloc2dR(int nr, int nc) {  
    Item **m;  
    int i;  
    m = malloc (nr*sizeof (Item *));  
    for (i=0; i<nr; i++) {  
        m[i] = malloc (nc*sizeof (Item));  
    }  
    return m;  
}  
void free2d(Item **m, int nr) {  
    int i;  
    for (i=0; i<nr; i++) {  
        free(m[i]);  
    }  
    free(m);  
}
```

Puntatore come valore di ritorno della funzione

```
typedef ... Item;  
Item **malloc2dR(int nr, int nc);  
void free2d(Item **m, int nr);  
...  
void h /* parametri formali */ {  
    Item **matr;  
    int nr, nc;  
    ...  
    matr = malloc2dR(nr, nc);  
    ... /* lavoro su matr */  
    free2d(matr, nr);  
}
```

funzione di tipo puntatore
a vettore di Item

matrice di Item

```
Item **malloc2dR(int nr, int nc) {  
    Item **m;  
    int i;  
    m = malloc(sizeof(Item *));  
    ...  
    for (i=0; i<nr; i++) {  
        m[i] = malloc(sizeof(Item));  
    }  
    return m;  
}  
  
free2d(Item **m, int nr) {  
    int i;  
    for (i=0; i<nr; i++) {  
        free(m[i]);  
    }  
    free(m);  
}
```

Puntatore come valore di ritorno della funzione

```
typedef ... Item;
```

funzione di tipo puntatore
a vettore di Item

```
void h /* parametri formali */ {
```

m variabile locale (doppio
puntatore): puntatore a vettore di
puntatori (righe)

```
... /* lavoro su matr */  
free2d(matr, nr);  
}
```

```
Item **malloc2dR(int nr, int nc) {  
    Item **m;  
    int i;  
    m = malloc (nr*sizeof (Item *));  
    for (i=0; i<nr; i++) {  
        m[i] = malloc (nc*sizeof (Item));  
    }  
    return m;  
}  
void free2d(Item **m, int nr) {  
    int i;  
    for (i=0; i<nr; i++) {  
        free(m[i]);  
    }  
    free(m);  
}
```

m ritornato come risultato

Puntatore come valore di ritorno della funzione

```
typedef ... Item;  
Item **malloc2dR(int nr, int nc);
```

Free più facile:

- riceve puntatore /by value)
- non restituisce risultato

```
int nr, nc;  
...  
matr = malloc2dR(nr, nc);  
... /*  
free2d  
}
```

Prima libera le singole righe,
poi il vettore di puntatori

```
Item **malloc2dR(int nr, int nc) {  
    Item **m;  
    int i;  
    m = malloc (nr*sizeof (Item *));  
    for (i=0; i<nr; i++) {  
        m[i] = malloc (nc*sizeof (Item));  
    }  
    return m;  
}  
void free2d(Item **m, int nr) {  
    int i;  
    for (i=0; i<nr; i++) {  
        free(m[i]);  
    }  
    free(m);  
}
```

Puntatore come parametro by pointer/reference

```
typedef ... Item;  
  
void malloc2dP(Item ***mp, int nr, int nc);  
  
void free2d(Item **m, int nr);  
  
...  
  
void h /* parametri formali */ {  
    Item **matr;  
    int nr, nc;  
  
    ...  
    malloc2dP(&matr, nr, nc);  
    ... /* lavoro su matr */  
    free2d(matr, nr);  
}
```

```
void malloc2dP(Item ***mp, int nr, int nc) {  
    Item **m;  
    int i;  
    m = (Item **)malloc (nr*sizeof(Item *));  
    for (i=0; i<nr; i++)  
        m[i] = (Item *)malloc (nc*sizeof(Item));  
    *mp = m;  
}  
  
void free2d(Item **m, int nr) {  
    int i;  
    for (i=0; i<nr; i++) {  
        free(m[i]);  
    }  
    free(m);  
}
```

Puntatore come parametro by pointer/reference

```
typedef ... Item;  
void malloc2dP(Item ***mp, int nr, int nc);  
void free2d(Item **m, int nr);  
...  
void h /* parametri formali */ {  
    Item **matr;  
    int nr, nc;  
    ...  
    malloc2dP(&matr, nr, nc);  
    ... /* lavoro su matr */  
    free2d(matr, nr);  
}
```

```
void malloc2dP(Item ***mp, int nr, int nc) {  
    Item **m;  
    *mp = m;  
    free2d(Item **m, int nr) {  
        int i;  
        for (i=0; i<nr; i++) {  
            free(m[i]);  
        }  
        free(m);  
    }  
}
```

funzione di tipo void, con
parametro puntatore a
a matrice di Item

matrice di Item

Puntatore come parametro by pointer/reference

```
typedef ... Item;  
void malloc2dP(Item ***mp, int nr, int nc);  
void free2d(Item **m, int nr);  
...  
}
```

mp: puntatore a matrice di Item
puntatore (triplo): puntatore a un
puntatore a un vettore (puntatore a Item)
di righe

```
... /* lavoro su matr */  
free2d(matr, nr);  
}
```

```
void malloc2dP(Item ***mp, int nr, int nc) {  
    Item **m;  
    m = (Item **)malloc (nr*sizeof(Item *));  
    for (i=0; i<nr; i++)  
        m[i] = (Item *)malloc (nc*sizeof(Item));  
    ...  
    free2d(Item **m, int nr) {  
        for (i=0; i<nr; i++) {  
            free(m[i]);  
        }  
        free(m);  
    }  
}
```

Puntatore come parametro by pointer/reference

```
typedef ... Item;  
void malloc2dP(Item ***mp, int nr, int nc);  
  
int nr, nc;  
...  
malloc2dP(&matr, nr, nc);  
/* lavoro su matr */  
free2d(matr, nr);  
}
```

punta a variabile del programma chiamante (puntatore doppio) in cui occorre trasferire il risultato

```
void malloc2dP(Item ***mp, int nr, int nc) {  
    Item **m;  
    *mp = m;  
    m = (Item **)malloc (nr*sizeof(Item *));  
    for (i=0; i<nr; i++)  
        m[i] = (Item *)malloc (nc*sizeof(Item));  
    *mp = m;  
}  
  
void free2d(Item **m, int nr) {  
    int i;  
    for (i=0; i<nr; i++) {  
        free(m[i]);  
    }  
    free(m);  
}
```

Puntatore come parametro by pointer/reference

```
typedef ... Item;  
void malloc2dP(Item ***mp, int nr, int nc);
```

m variabile locale (doppio puntatore):
non obbligatoria ma comoda
«dentro» alla funzione

```
Item *matr,  
int nr, nc;  
  
...  
malloc2dP(&matr, nr, nc);  
... /  
free2d  
}
```

m copiata in *mp (risultato)
al termine della funzione
(equivale a matr = m)

```
void malloc2dP(Item ***mp, int nr, int nc) {  
    Item **m;  
    int i;  
    m = (Item **)malloc (nr*sizeof(Item *));  
    for (i=0; i<nr; i++)  
        m[i] = (Item *)malloc (nc*sizeof(Item));  
    *mp = m;  
  
    void free2d(Item **m, int nr) {  
        int i;  
        for (i=0; i<nr; i++) {  
            free(m[i]);  
        }  
        free(m);  
    }
```

```
void malloc2dP(Item ***mp, int nr, int nc) {  
    int i;  
    *mp = malloc (nr*sizeof (Item *));  
    for (i=0; i<nr; i++) {  
        (*mp)[i] = malloc (nc*sizeof (Item));  
    }  
}
```

VARIANTE (meno leggibile ma più compatta):
si lavora direttamente sulla variabile del programma
chiamante (*mp) senza usare una variabile locale.

```
void malloc2dP(Item ***mp, int nr, int nc) {  
    int i;  
    *mp = malloc (nr*sizeof (Item *));  
    for (i=0; i<nr; i++) {  
        (*mp)[i] = malloc (nc*sizeof (Item));  
    }  
}
```

le parentesi tonde sono necessarie per la
precedenza degli operatori

```
void malloc2dP(Item ***mp, int nr, int nc) {  
    int i;  
    *mp = malloc (nr*sizeof (Item *));  
    for (i=0; i<nr; i++) {  
        (*mp)[i] = malloc (nc*sizeof (Item));  
    }  
}
```

le parentesi tonde sono necessarie per la
precedenza degli operatori

Molti programmati seguono questa strategia (a volte perché non prendono in considerazione la variabile locale):
AUMENTA LA PROBABILITA' DI ERRORE (proprio perché si dimenticano parentesi): `*mp[i]` è SBAGLIATO!!!

Vettori a dimensione variabile

Se serve «solo» dimensionare vettori e matrici in fase di esecuzione, anche in C esistono i

vettori a lunghezza variabile
(variable length arrays)

- Si dichiara un vettore/matrice come variabile locale usando, come dimensioni, variabili o espressioni anziché costanti.
- Allocazione e deallocazione sono automatiche e implicite (è comodo).

L'uso di vettori a lunghezza variabile è scoraggiato in quanto:

- con essi si realizza un sottoinsieme di ciò che si può fare con l'allocazione dinamica
- presentano svantaggi:
 - non si può controllare se l'allocazione è andata a buon fine (mentre invece il puntatore ritornato da `malloc/calloc/realloc` si può confrontare con `NULL`): l'effetto è un crash, quando lo stack è troppo piccolo.
 - Il vettore è cancellato all'uscita dalla funzione, ma il programmatore può pensare che esista ancora e continua a farvi riferimento.

Esempio

```
void inverti(int N) {  
    int i;  
    float v[N];  
    printf("Inserisci %d elementi\n", N);  
    for (i=0; i<N; i++) {  
        printf("Elemento %d: ", i) ;  
        scanf("%f", &v[i]) ;  
    }  
    printf("Dati in ordine inverso\n");  
    for (i=N-1; i>=0; i--)  
        printf("Elemento %d: %f\n", i, v[i]);  
}
```

Esempio con ERRORE !!! (ritornare v)

```
float *inverte(int N) {  
    int i;  
    float v[N];  
    printf("Inserisci %d elementi\n", N);  
    for (i=0; i<N; i++) {  
        printf("Elemento %d: ", i) ;  
        scanf("%f", &v[i]) ;  
    }  
    printf("Dati in ordine inv.  
    for (i=N-1; i>=0; i-)  
        printf("Elemento %d: %f\n", i, v[i]);  
    return v;  
}
```

V è nello stack.
Quando la funzione termina viene deallocated automaticamente.
Il programma chiamante riceve un puntatore a memoria appena rilasciata

Esempio corretto (con malloc)

```
float *inverti(int N) {  
    int i;  
    float *v = (float *)malloc(N*sizeof(float));  
    printf("Inserisci %d elementi\n", N);  
    for (i=0; i<N; i++) {  
        printf("Elemento %d: ", i) ;  
        scanf("%f", &v[i]) ;  
    }  
    printf("Dati in ordine inverso:\n");  
    for (i=N-1; i>=0; i--) {  
        printf("Elemento %d: %f\n", i, v[i]);  
    }  
    return v;  
}
```

V è nello heap.

Quando la funzione termina NON viene deallocated automaticamente.

Il programma chiamante riceve un puntatore a memoria ancora allocata
(andrà liberata con free)

Capitolo 4: Le liste

PUNTATORI E STRUTTURE DATI DINAMICHE:
ALLOCAZIONE DELLA MEMORIA E
MODULARITÀ IN LINGUAGGIO C



Sequenze Lineari

DEFINIZIONI E POSSIBILI REALIZZAZIONI

Sequenza lineare

- Detta anche enumerazione o **lista**
- Insieme finito di elementi di tipo generico **Item** disposti consecutivamente, in cui a ogni elemento è associato univocamente un indice

$$a_0, a_1, \dots, a_i, \dots, a_{n-1}$$

- Sulle coppie di elementi è definita una relazione predecessore/successore:

$$a_{i+1} = \text{succ}(a_i)$$

$$a_i = \text{pred}(a_{i+1})$$

$$\not\exists \text{ succ}(a_{n-1})$$

$$\not\exists \text{ pred}(a_0)$$

Sequenza lineare

- La sequenza è:
 - Ordinata con criterio posizionale (primo dato, secondo, i-esimo)
- In base a una chiave (parte di dato), la sequenza è:
 - Ordinata
 - non ordinata
- L'accesso avviene:
 - in base ad una chiave (ricerca)
 - in base alla posizione nella sequenza
 - diretto, costo $O(1)$
 - sequenziale, costo $O(n)$

Sequenza lineare

- La sequenza è:
 - Ordinata con criterio posizionale (primo dato, secondo, i-esimo)
- In base a una chiave (parte di dato), la sequenza è:
 - Ordinata
 - non ordinata
- L'accesso avviene:
 - in base ad una chiave (ricerca)
 - in base alla posizione nella sequenza
 - diretto, costo $O(1)$
 - sequenziale, costo $O(n)$

Esempio:
[-3,4,21,43]

Sequenza lineare

- La sequenza è:
 - Ordinata con criterio posizionale (primo dato, secondo, i-esimo)
- In base a una chiave (parte di dato), la sequenza è:
 - Ordinata
 - non ordinata
- L'accesso avviene:
 - in base ad una chiave (ricerca)
 - in base alla posizione nella sequenza
 - diretto, costo $O(1)$
 - sequenziale, costo $O(n)$

Esempio:
[21,4,-2,43]

Sequenza lineare

- La sequenza è:
 - Ordinata con criterio posizionale (primo dato, secondo, i-esimo)
- In base a una chiave (parte di dato), la sequenza è:
 - Ordinata
 - non ordinata
- L'accesso avviene:
 - in base ad una chiave (ricerca)
 - in base alla posizione nella sequenza
 - diretto, costo $O(1)$
 - sequenziale, costo $O(n)$

Esempio:
cerca il 4

Sequenza lineare

- La sequenza è:
 - Ordinata con criterio posizionale (primo dato, secondo, i-esimo)
- In base a una chiave (parte di dato), la sequenza è:
 - Ordinata
 - non ordinata
- L'accesso avviene:
 - in base ad una chiave (ricerca)
 - in base alla posizione nella sequenza
 - diretto, costo $O(1)$
 - sequenziale, costo $O(n)$

Esempio:
cerca il terzo

Vettore (sequenza lineare in vettore)

Modalità di memorizzazione:

- dati **contigui** in memoria

-3	4	21	43
----	---	----	----

Accesso diretto:

- dato l'indice i , si accede all'elemento a_i senza dover scorrere la sequenza lineare
- il costo dell'accesso non dipende dalla posizione dell'elemento nella sequenza lineare, quindi è $O(1)$

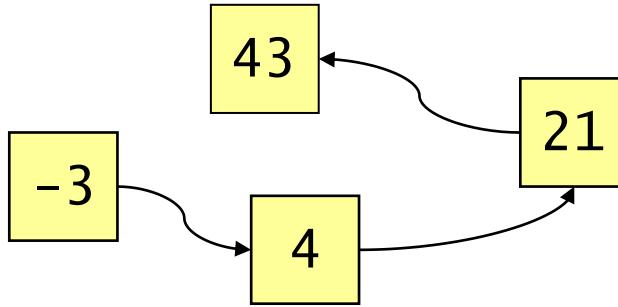
Lista concatenata (linked)

Modalità di memorizzazione:

- dati **non contigui** in memoria

Accesso sequenziale:

- dato l'indice i , si accede all'elemento a_i , scorrendo la sequenza lineare a partire da uno dei suoi 2 estremi, solitamente quello di SX
- il costo dell'accesso dipende dalla posizione dell'elemento nella sequenza lineare, quindi è $O(n)$ nel caso peggiore



Operazioni sulle sequenze lineari (liste)

- **ricerca** di un elemento con campo **chiave di ricerca** uguale a chiave data
- **inserzione** di un elemento:
 - **in testa** alla lista non ordinata
 - **in coda** alla lista non ordinata
 - **nella posizione** tale da garantire l'invarianza della proprietà di ordinamento per una lista ordinata
- **cancellazione** di un elemento:
 - che si trova **in testa** alla lista non ordinata
 - che si trova in una posizione **arbitraria** della lista non ordinata
 - che ha **un campo** con contenuto uguale a quello di una chiave di cancellazione (richiede solitamente una ricerca preventiva dell'elemento da cancellare)
 - con o senza restituzione dell'elemento cancellato (estrazione).

Operazioni sulle sequenze lineari (*liste*)

- **ricerca** di un elemento con campo **chiave di ricerca** uguale a chiave data
- **inserzione** di un elemento:
 - **in testa** alla *lista* non ordinata
 - **in coda** alla *lista* non ordinata
 - **nella posizione** tale da garantire l'invarianza della proprietà di ordinamento per una *lista* ordinata
- **cancellazione** di un elemento:
 - che si trova **in testa** alla *lista* non ordinata
 - che si trova in una posizione **arbitraria** della *lista* non ordinata
 - che ha **un campo** con contenuto uguale a quello di una chiave di cancellazione (richiede solitamente una ricerca preventiva dell'elemento da cancellare)
 - con o senza restituzione dell'elemento cancellato (estrazione).

Liste (per brevità)
NON necessariamente
«concatenate»...

Operazioni sulle sequenze lineari (*liste*)

- **ricerca** di un elemento con campo **chiave di ricerca** uguale a chiave data
- **inserzione** di un elemento:
 - **in testa** alla *lista* non ordinata
 - **in coda** alla *lista* non ordinata
 - **nella posizione tale da garantire l'invianza della proprietà di ordinamento per una *lista ordinata***
- **cancellazione** di un elemento:
 - che si trova **in testa** alla *lista* non ordinata
 - che si trova in una posizione **arbitraria** della *lista* non ordinata
 - che ha **un campo** con contenuto uguale a quello di una chiave di cancellazione (richiede solitamente una ricerca preventiva dell'elemento da cancellare)
 - con o senza restituzione dell'elemento cancellato (estrazione).

ATTENZIONE:
si intende ordinamento
in base a una chiave

Lista realizzata mediante vettore

Le liste possono essere realizzate mediante vettori (allocazione contigua):

- se è noto o stimabile il numero massimo di elementi, oppure sfruttando la ri-allocazione
- sfruttando la contiguità fisica degli elementi (elemento all'indice i successore di quello all'indice $i-1$ e predecessore di quello all'indice $i+1$)
- disaccoppiando contiguità fisica e relazione predecessore/successore mediante indici (lista concatenate mediante indici)

Liste realizzate mediante concatenazione

Le liste possono essere realizzate mediante strutture ricorsive allocate individualmente:

- se non è noto o stimabile il numero massimo di elementi
- se la relazione è da predecessore a successore si hanno **liste concatenate semplici**
- se è in entrambi i versi si hanno **liste concatenate doppie.**

Liste Concatenate

REALIZZATE MEDIANTE STRUCT RICORSIVE

Le liste concatenate

Strutture dati dinamiche come sequenze di nodi.

In C ogni nodo è una **struct** con:

- un numero arbitrario (fisso, una volta definite la struct) di dati, generalmente racchiusi in un campo `val` di tipo `Item` (si tratta di una convenzione, non di una regola)
- uno o due riferimenti (“link”) che puntano al nodo successivo e/o precedente

Le liste concatenate

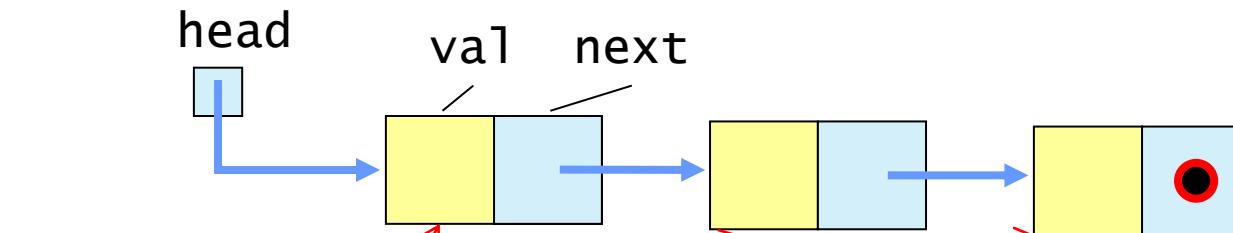
Strutture dati dinamiche come sequenze di nodi.

In C ogni nodo è una **struct** con:

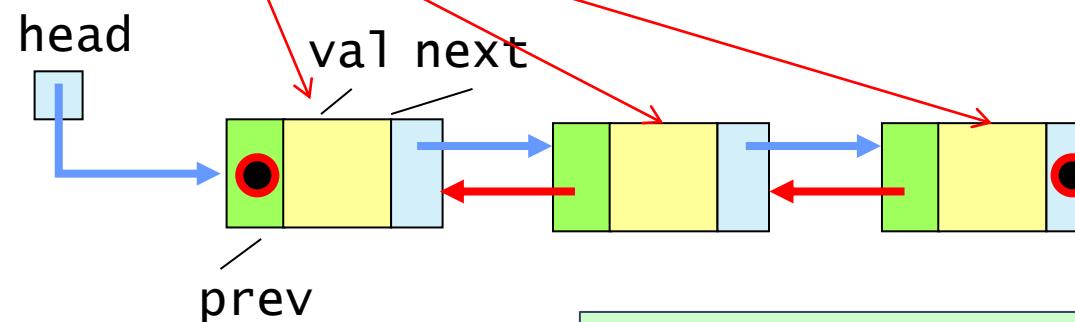
- un numero arbitrario (fissato una volta definite la struct) di dati, generalmente racchiusi in un campo `val` di tipo `Item` (si tratta di una convenzione, non di un'angola)
- uno o due riferimenti (“link”) che puntano al nodo successivo e/o precedente

Di qui in avanti solo
liste, omettendo
concatenate

Lista concatenata semplice



struct node



Lista concatenata doppia

Definizione dei dati

```
typedef ... Item;  
typedef ... Key;
```

dato in lista

```
typedef int Item;
```

chiave del dato

```
typedef struct {  
    char nome[40];  
    data_t nascita;  
} Item;
```

```
typedef int Key;
```

```
typedef char *Key;
```

Definizione delle funzioni (per gestire chiave)

Otttenere chiave
dal dato

```
Key KEYget(Item d);  
int KEYeq(Key k1, Key k2);  
int KEYless(Key k1, Key k2);  
int KEYgreater(Key k1, Key k2);
```

Confrontare
chiavi

Diversi modi per definire i nodi

1. senza `typedef`, definendo il puntatore `next` mentre si definisce il tipo `struct node`

```
struct node {  
    Item val;  
    struct node *next;  
};
```

Diversi modi per definire i nodi

2. con `typedef`, definendo sia un alias `node_t` per `struct node`, sia un alias `link` per il puntatore a oggetto di tipo `struct node`

```
typedef struct node {  
    Item val;  
    struct node *next;  
} node_t, *link;
```

Diversi modi per definire i nodi

3. con separazione tra `typedef` e dichiarazione della `struct node`, dichiarando un alias `link` per il puntatore a oggetto di tipo `struct node`. Nella dichiarazione di tipo `struct node` si usa il tipo `link` appena definito

```
typedef struct node *link;

struct node {
    Item val;
    link next;
};
```

Diversi modi per definire i nodi

4. con separazione tra `typedef` e dichiarazione della `struct node`, dichiarando un alias `node_t` per `struct node`. Nella dichiarazione di `struct node` si dichiara `next` come puntatore a oggetto di tipo `node_t`

```
typedef struct node node_t;

struct node {
    Item val;
    node_t *next;
};
```

Diversi modi per definire i nodi

5. con separazione tra `typedef` e dichiarazione della `struct node`, dichiarando un alias `link` per un puntatore a `struct node` e un alias `node_t` per `struct node`. Nella dichiarazione di `struct node` si usa `link`

```
typedef struct node *link, node_t;

struct node {
    Item val;
    link next;
};
```

Operazioni Atomiche

ALLOCAZIONE, INSERIMENTO, CANCELLAZIONE, ATTRAVERSAMENTO

Allocazione di un nodo

Con la terza modalità:

- si dichiara un puntatore x a un nodo come:

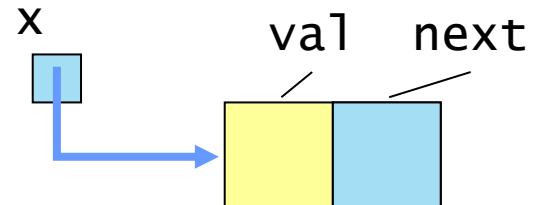
```
link x;
```

- si alloca il nodo come:

```
x = malloc(sizeof *x);
```

o

```
x = malloc(sizeof(struct node));
```



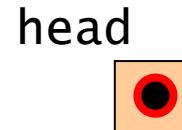
Operazioni atomiche su liste

Creazione mediante generazione del puntatore alla testa:

```
link head = NULL;
```

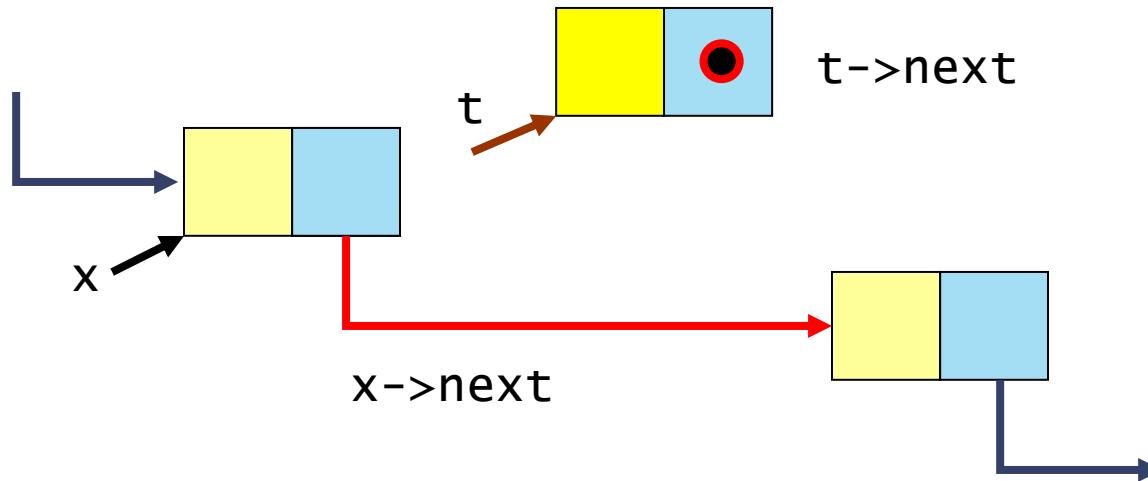
Test se la lista è vuota

```
if (head == NULL)
```

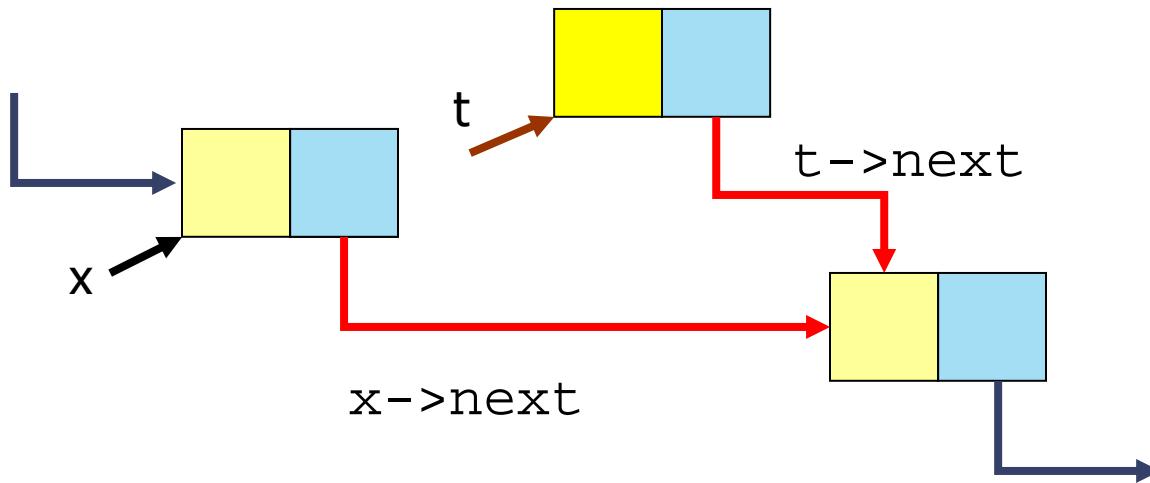


Inserimento

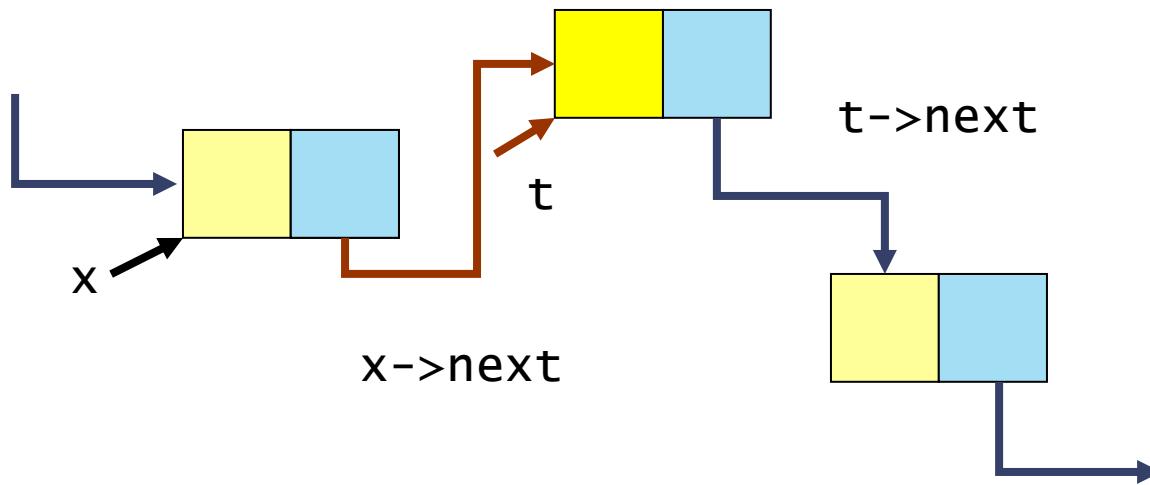
Inserimento del nodo puntato da t dopo il nodo
puntato da x in lista esistente:



```
t->next = x->next;
```

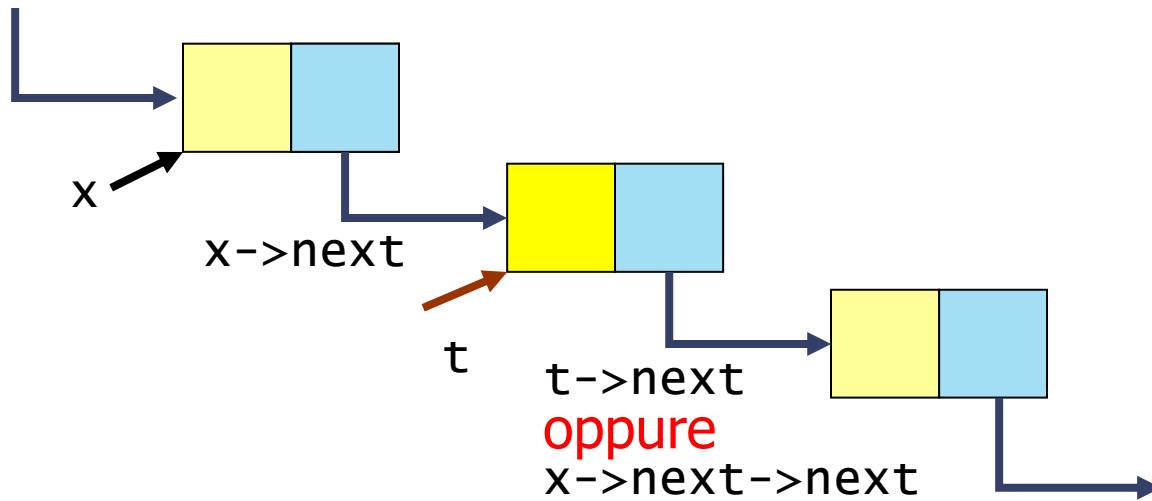


```
x->next = t;
```



Cancellazione

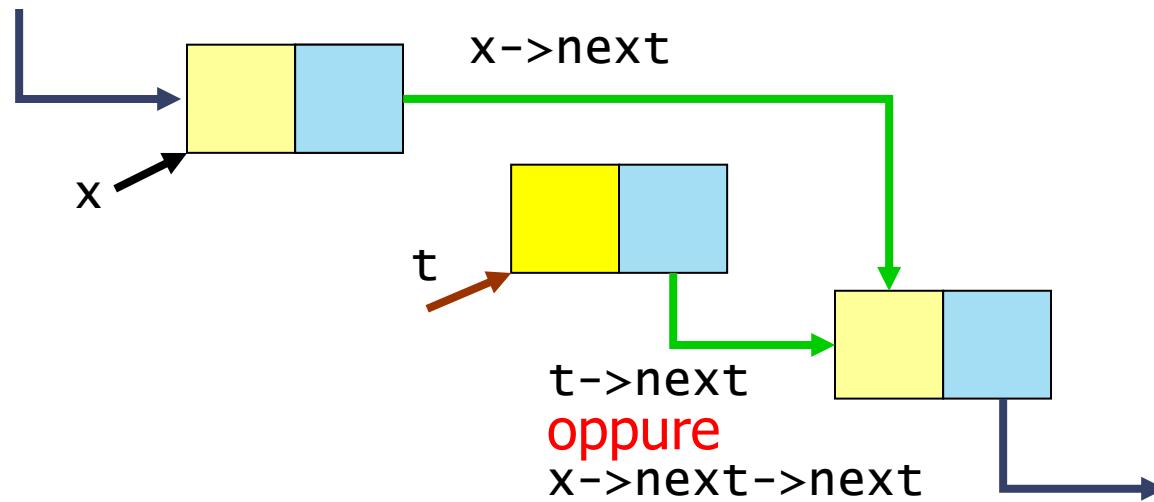
Cancellazione del nodo puntato da t , successore del nodo puntato da x :



```
x->next = x->next->next;
```

oppure

```
x->next = t->next;
```

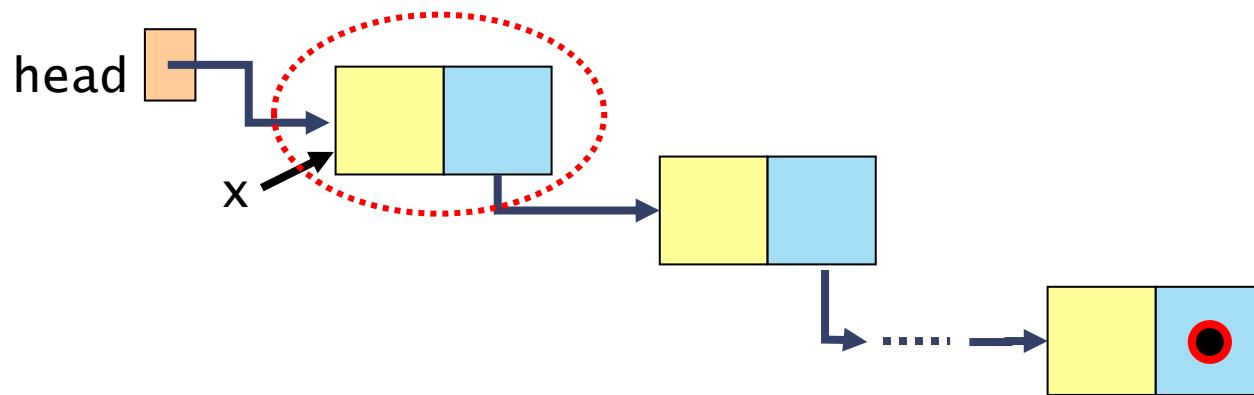


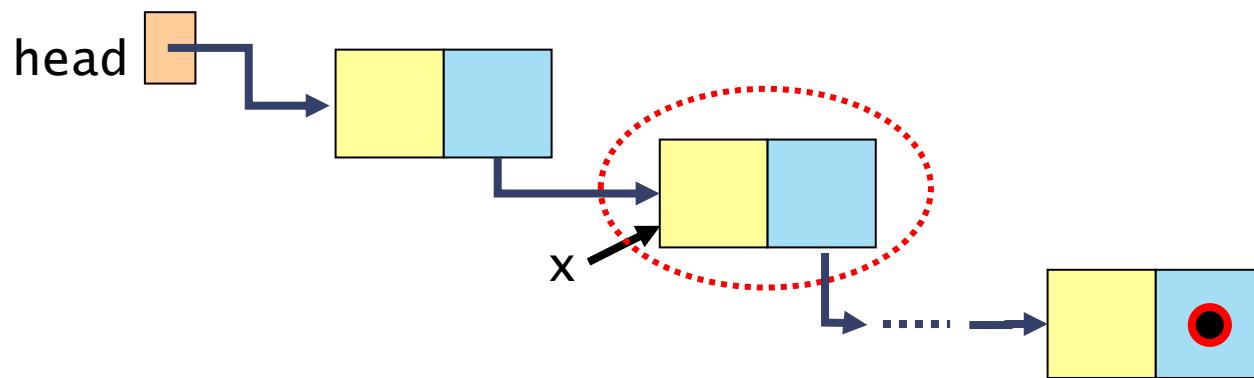
Attraversamento (I)

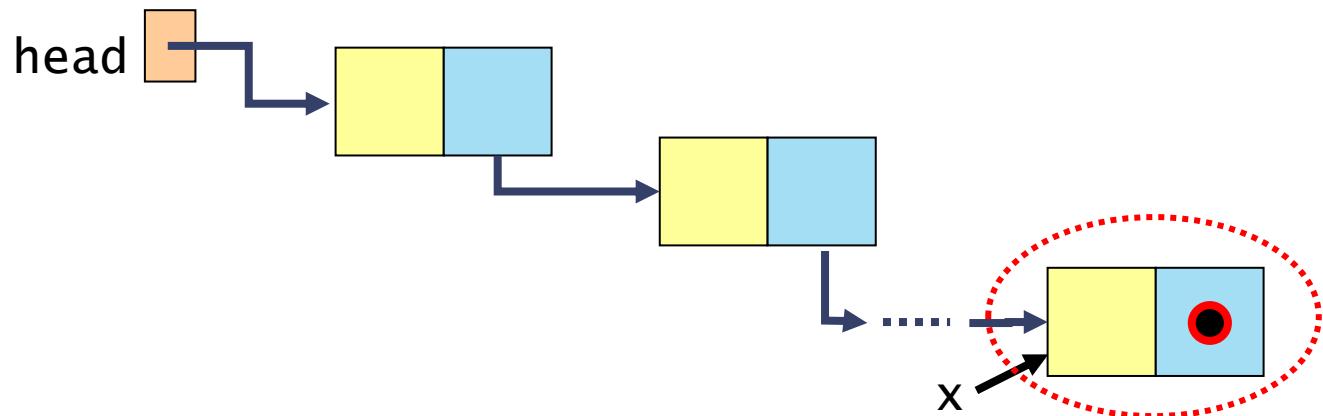
Con elaborazioni ***semplici*** basta il puntatore al nodo corrente x :

non si modifica la lista: es. ricerca,
visualizzazione, conteggio, ...

```
link x, head;  
...  
for (x=head; x!=NULL; x=x->next) {...}
```





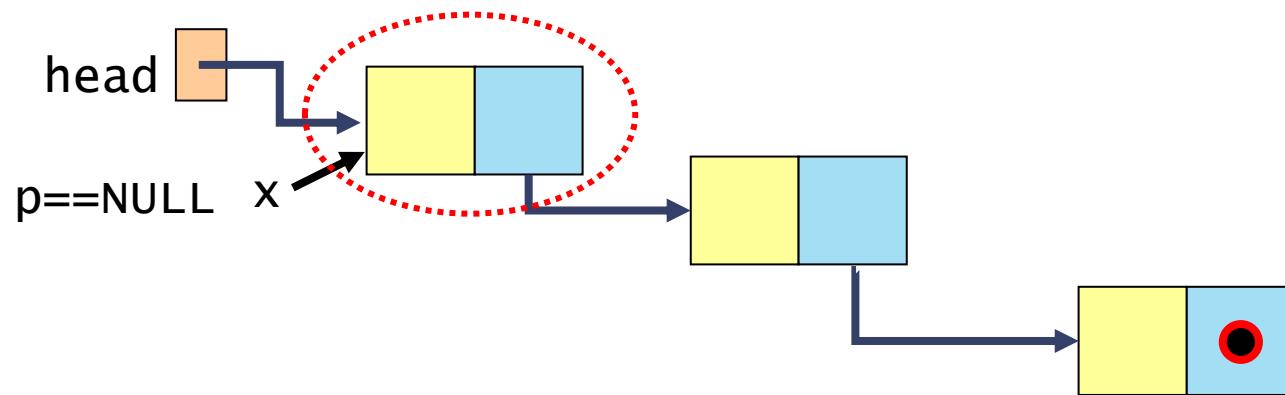


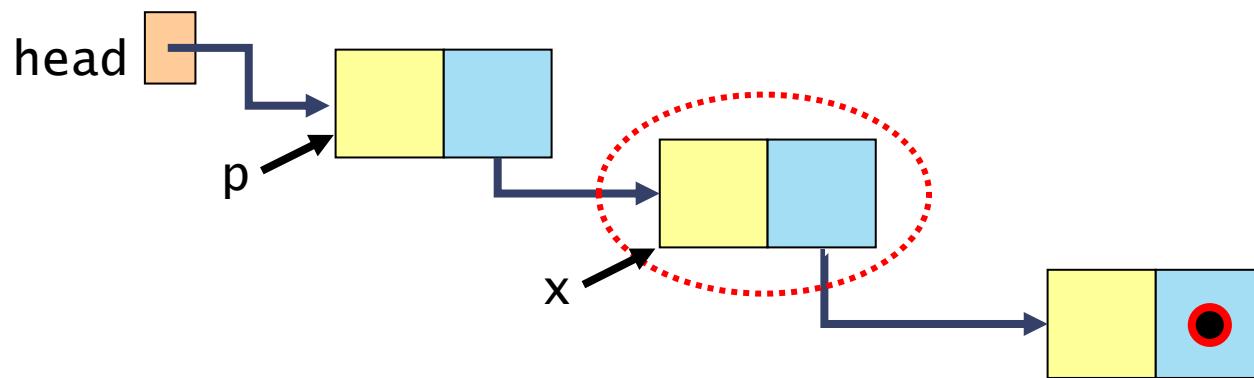
Attraversamento (II)

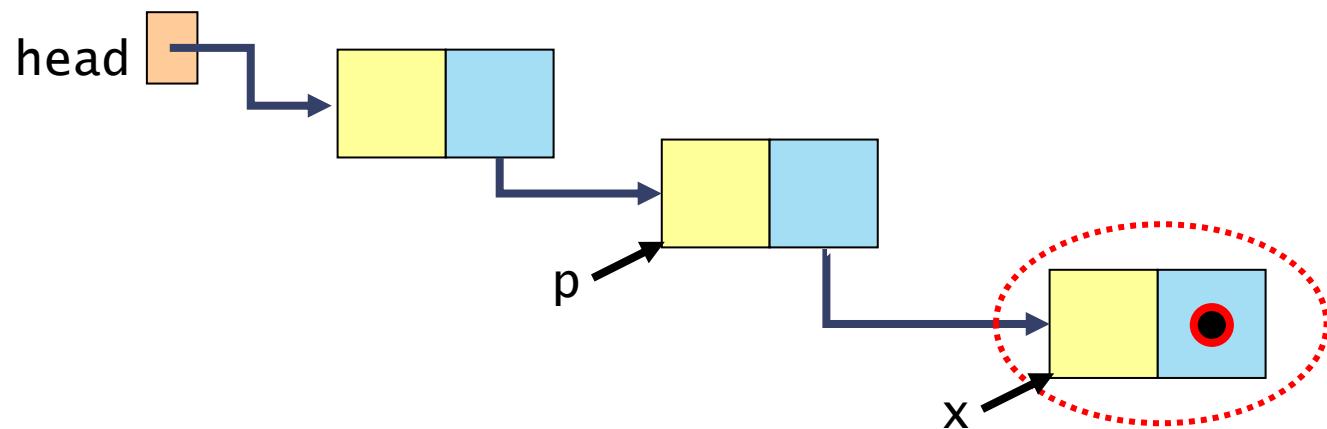
Con elaborazioni **complesse** serve il puntatore al nodo corrente x e al suo predecessore p :

si modifica la lista: es.
inserzione, cancellazione ...

```
link x, p, head;  
...  
p = NULL;  
for (x=head; x!=NULL; p = x, x=x->next) {...}
```





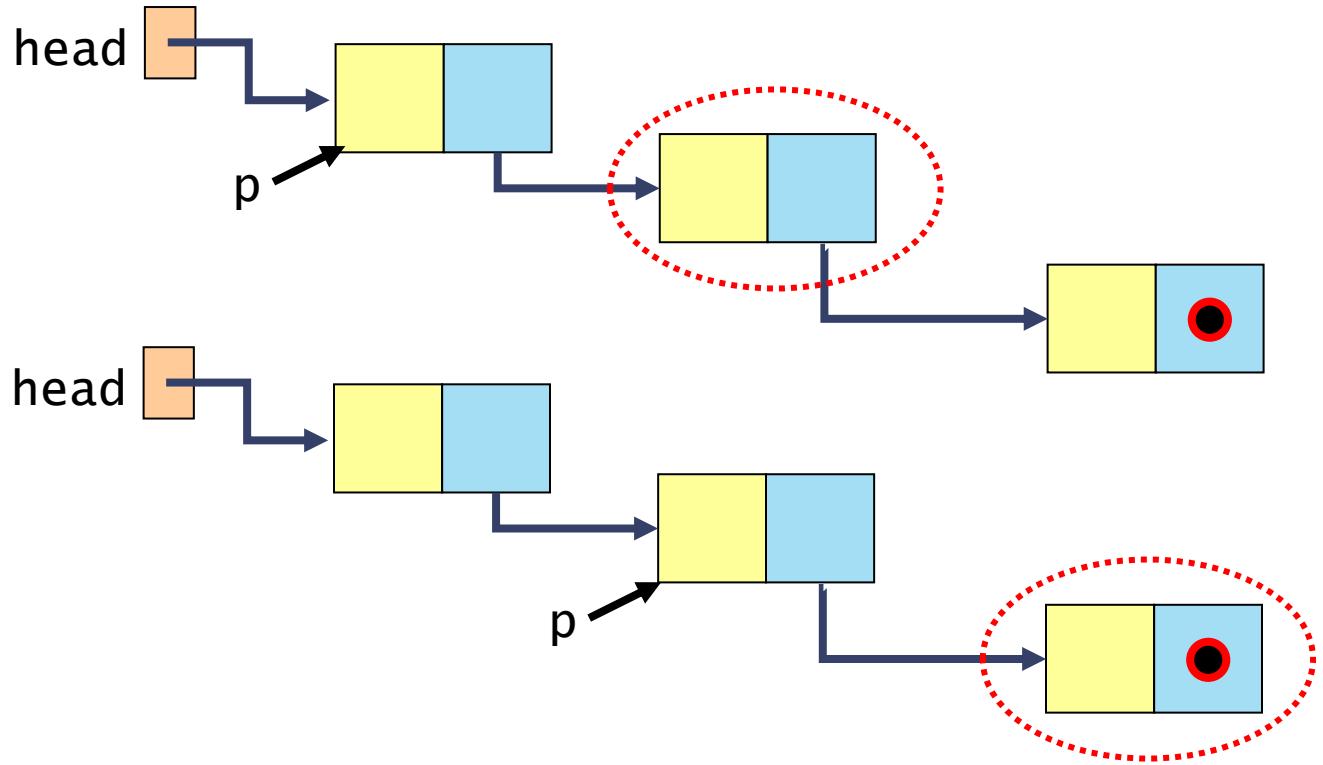


Attraversamento (II bis)

Il puntatore al nodo corrente x è inutile se:

- si tratta il nodo in testa fuori dall'iterazione
- si comincia l'iterazione dal secondo elemento, inizializzando p con $head$
- per terminare si testa che la lista non sia vuota e che il prossimo elemento non sia l'ultimo
- si scorre la lista aggiornando p (puntatore al predecessore)

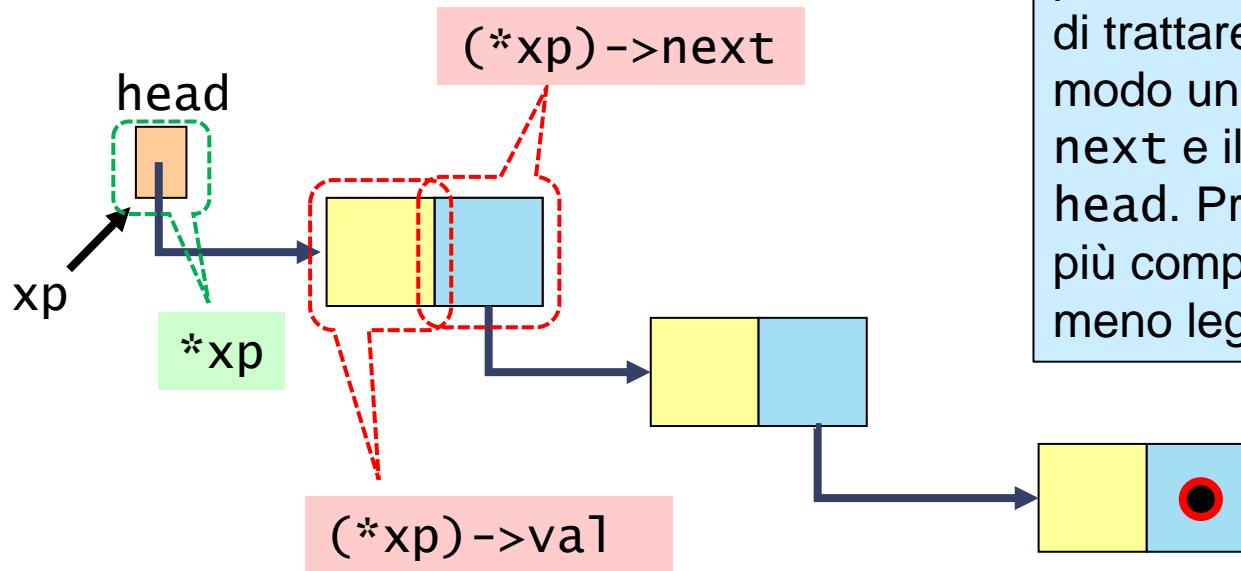
```
link p, head;  
...  
/* gestione separata nodo in testa */  
for (p=head;p!=NULL && p->next!=NULL;p=p->next){...}
```



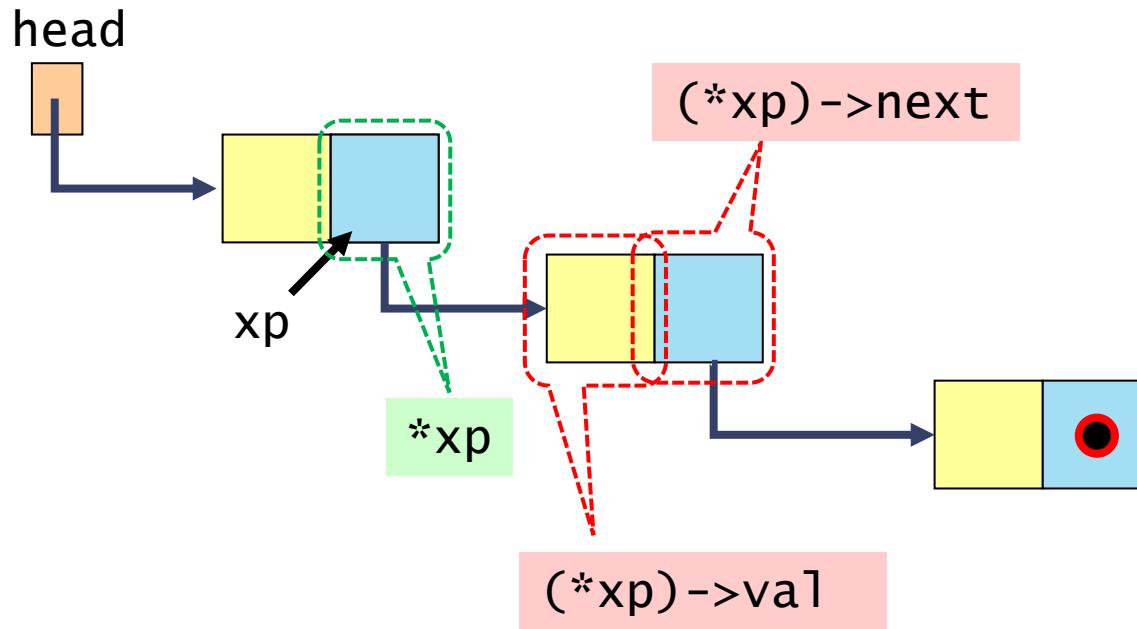
Attraversamento (III)

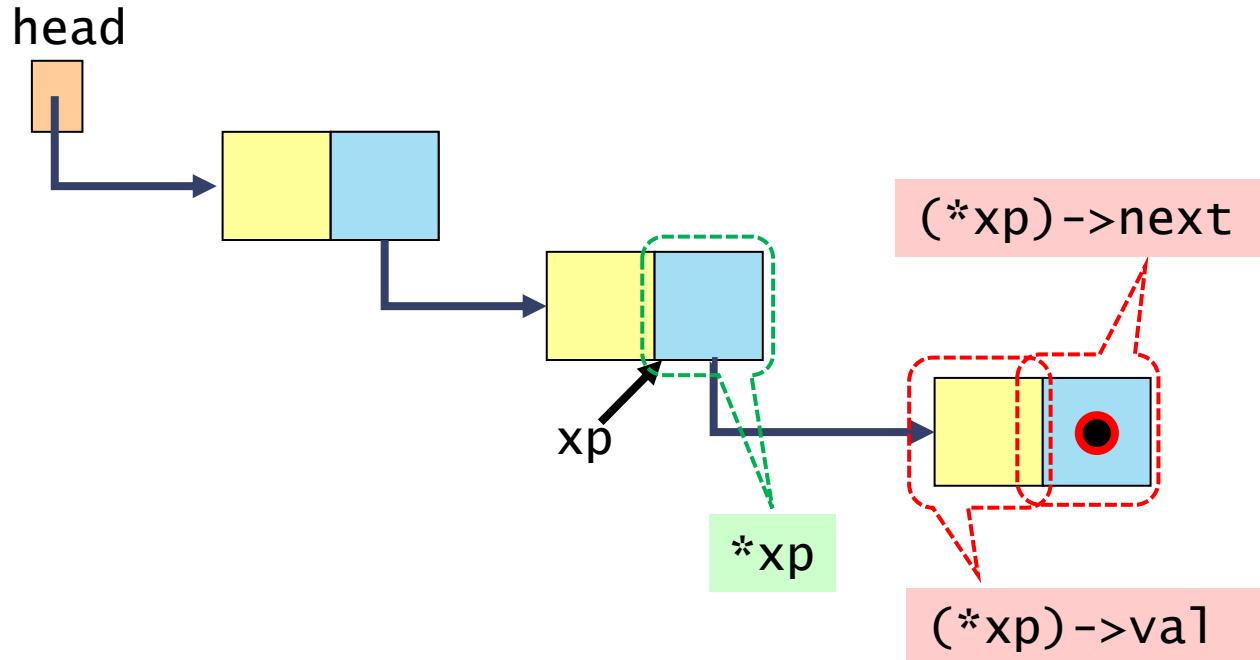
Con puntatore a puntatore a nodo `xp` per accedere al campo puntatore a successore della struct:

```
link *xp, head;  
...  
for (xp=&head; *xp!=NULL; xp=&((*xp)->next)) {  
    ...  
}
```



Il puntatore a puntatore permette di trattare allo stesso modo un campo **next** e il puntatore **head**. Programmi più compatti ma meno leggibili.





Attraversamento (IV)

Ricorsivo: è immediato anche l'attraversamento all'indietro senza bisogno di puntatore al predecessore:

```
void listTravR(link h) {  
    if (h == NULL) return;  
    ITEMdisplay(h->val);  
    listTravR(h->next);  
}
```

```
void listRevTravR(link h) {  
    if (h == NULL) return;  
    listRevTravR(h->next);  
    ITEMdisplay(h->val);  
}
```

Operazioni su liste

LISTE NON ORDINATE E LISTE ORDINATE

Operazioni sulle liste

- Creazione nodo
- Liste non ordinate:
 - inserimento in testa
 - inserimento in coda
 - ricerca di chiave
 - cancellazione dalla testa
 - estrazione dalla testa
 - cancellazione di nodo con chiave data.
- Liste ordinate:
 - Inserimento
 - ricerca di chiave
 - cancellazione di nodo con chiave data.

Creazione nodo

```
link newNode(Item val, link next) {  
    link x = malloc(sizeof *x);  
    if (x==NULL)  
        return NULL;  
    else {  
        x->val = val;  
        x->next = next;  
    }  
    return x;  
}
```

Inserzione in testa

Lista non ordinata

Soluzione 1: parametri di ingresso: `val` e puntatore alla testa `h`. Valore di ritorno: nuovo puntatore alla testa, che il `main` assegnerà a `head`:

```
link listInsHead (link h, item val) {  
    h = newNode(val,h);  
    return h;  
}  
/* main */  
.  
link head = NULL;  
item d;  
.  
head = listInsHead(head, d);
```

Inserzione in testa

Lista non ordinata

Soluzione 2: parametri di ingresso: val e puntatore al puntatore alla testa hp. La funzione modifica direttamente il puntatore alla testa *hp. In chiamata si passa l'indirizzo del puntatore alla testa &head:

```
void listInsHeadP(link *hp, Item val) {  
    *hp = newNode(val, *hp);  
}  
/* main */  
...  
link head = NULL;  
Item d;  
...  
listInsHead(&head, d);
```

Inserzione in coda

Lista non ordinata

Serve il puntatore all'ultimo nodo:

- ricavato con un attraversamento di costo $O(n)$
- mantenuto con costo $O(1)$ (MEGLIO!!!)

Soluzione 1 ($O(n)$):

- lista vuota: inserzione in testa con modifica di **head**
- lista non vuota: ciclo di attraversamento per raggiungere l'ultimo nodo, creazione di un nuovo nodo e aggancio come successore dell'ultimo nodo, **head** rimane invariato.

```
link listInstail(link h, Item val) {  
    link x;  
    if (h==NULL)  
        return newNode(val, NULL);  
    for (x=h; x->next!=NULL; x=x->next);  
    x->next = newNode(val, NULL);  
    return h;  
}  
/* main */  
...  
link head=NULL;  
Item d;  
...  
head = listInstail(head, d);
```

cerca ultimo nodo

crea nuovo nodo e
aggancialo
all'ultimo

Inserzione in coda

Lista non
ordinata

Soluzione 2 ($O(n)$):

parametri di ingresso: `val` e puntatore al puntatore alla testa `hp`

- `x = *hp` identifica la testa della lista
- lista vuota: inserzione in testa con modifica di `*hp`
- lista non vuota: ciclo di attraversamento per raggiungere l'ultimo nodo, creazione di un nuovo nodo e aggancio come successore dell'ultimo nodo, `*hp` rimane invariato.

```
void listInstailP(link *hp, Item val) {  
    link x=*hp;  
    if (x==NULL)  
        *hp = newNode(val, NULL);  
    else {  
        for (; x->next!=NULL; x=x->next);  
        x->next = newNode(val, NULL);  
    }  
}  
/*main */  
...  
link head=NULL;  
Item d;  
...  
listInstailP(&head, d);
```

cerca ultimo nodo

crea nuovo nodo e
aggancialo
all'ultimo

Inserzione in coda

Lista non
ordinata

Soluzione 3 ($O(n)$):

- attraversamento con variabile `xp` di tipo puntatore a puntatore a nodo che punta al campo puntatore a successore della `struct`
- unificazione dei casi di inserimento in lista vuota e non vuota.

cerca ultimo nodo
(comprende il caso di lista vuota)

```
void listInsTailP(link **hp, Item val) {  
    link *xp = hp;  
    while (*xp != NULL)  
        xp = &((*xp)->next);  
    *xp = newNode(val, NULL);  
}  
/* main */  
...  
link head=NULL;  
Item d;  
...  
listInsTailP(&head, d);
```

Inserzione in coda

Lista non
ordinata

Soluzione 4 (**O(1)**):

- uso di 2 variabili di tipo puntatore a puntatore a nodo `hp` e `tp` per accedere a primo e ultimo nodo
- `*hp` identifica la testa della lista, `*tp` la coda
- lista vuota: inserzione in testa con modifica di `*hp` e di `*tp`
- lista non coda vuota: creazione di un nuovo nodo e aggancio come successore dell'ultimo nodo, `*hp` rimane invariato, `*tp` viene aggiornato.

```
void listInstFast(link *hp,link *tp,Item val) {  
    if (*hp==NULL)  
        *hp = *tp = newNode(val, NULL);  
    else {  
        (*tp)->next = newNode(val, NULL);  
        *tp = (*tp)->next;  
    }  
}  
/* main */  
...  
link head=NULL, tail=NULL;  
Item d;  
...  
listInstTailFast(&head, &tail, d);
```

Ricerca di una chiave

Lista non ordinata

- Non essendo modificata la lista, basta un solo puntatore per l'attraversamento.
- Se la chiave c'è, si ritorna il dato che la contiene
- Se la chiave non c'è, si ritorna il dato nullo tramite chiamata alla funzione ITEMsetvoid.

```
Item listSearch(link h, Key k) {
    link x;
    for (x=h; x!=NULL; x=x->next)
        if (KEYeq(KEYget(x->val), k))
            return x->val;
    return ITEMsetvoid();
}
/* main */
...
link head=NULL;
Item d; Key k;
...
d = listSearch(head, k);
```

Cancellazione dalla testa

Lista non
ordinata

- Se la lista non è vuota, aggiorna la testa della lista con il puntatore al secondo dato che diventa il primo
- Ricorda il primo dato per poi liberarlo con `free`
- Il `main` assegna a `head` il nuovo puntatore alla testa.

```
link listDelHead(link h) {  
    link t = h;  
    if (h == NULL)  
        return NULL;  
    h = h->next;  
    free(t);  
    return h;  
}  
/* main */  
...  
link head = NULL;  
...  
head = listDelHead(head);
```

Estrazione dalla testa

Lista non ordinata

- Per aggiornare la testa della lista si deve usare il puntatore al puntatore alla testa **hp** poiché il valore di ritorno della funzione è il dato
- Se la lista è vuota, si ritorna il dato nullo tramite chiamata alla funzione **ITEMsetvoid**, altrimenti si memorizza il primo dato per poi ritornarlo
- Si ricorda il primo dato per poi liberarlo con **free**

```
Item listExtrHeadP(link *hp) {
    link t = *hp;
    Item tmp;
    if (t == NULL)
        return ITEMsetvoid();
    tmp = t->val;
    *hp = t->next;
    free(t);
    return tmp;
}
/* main */
...
link head = NULL;
Item d;
...
d = listExtrHeadP(&head);
```

Cancellazione di nodo con chiave data

Lista non ordinata

A seguito della cancellazione, il puntatore alla testa della lista può essere:

- NULL perché la lista era vuota
- il puntatore al secondo dato , se la chiave si trovava nel primo
- invariato se la lista non è vuota, la chiave non è il primo dato o non c'è in lista. Un ciclo di attraversamento con 2 puntatori identifica il nodo da cancellare.

```
link listDelKey(link h, key k) {
    link x, p;
    if (h == NULL)
        return NULL;
    for (x=h, p=NULL; x!=NULL; p=x, x=x->next) {
        if (KEYeq(KEYget(x->val),k)) {
            if (x==h)
                h = x->next;
            else
                p->next = x->next;
            free(x);
            break;
        }
    }
    return h;
}
```

Cancellazione di nodo con chiave data

Lista non ordinata

Versione ricorsiva:

- terminazione: si punta al nodo vuoto
- se il nodo corrente non contiene la chiave, si ricorre sulla lista che ha come testa il nodo successore
- se il nodo corrente contiene la chiave, si salva il puntatore al suo successore, si cancella il nodo corrente e si ritorna il puntatore al successore che nell'istanza ricorsiva chiamante viene assegnato come successore del nodo corrente realizzando il bypass.

```
link listDelKeyR(link x, Key k) {
    link t;
    if (x == NULL)
        return NULL;
    if (KEYeq(KEYget(x->val), k)) {
        t = x->next;
        free(x);
        return t;
    }
    x->next = listDelKeyR(x->next, k);
    return x;
}
```

Lista non ordinata

Estrazione di nodo con chiave data

L'estrazione può alterare il puntatore alla testa nel caso la chiave di ricerca sia nel primo dato.

La funzione deve:

- ritornare:
 - il dato nullo tramite chiamata alla funzione `ITEMsetvoid` se la lista è vuota o la chiave non è presente
 - il dato se la chiave è presente
- aggiornare il puntatore alla testa della lista se si estrae il primo dato.

Si propone la tecnica del puntatore a puntatore `xp`, inizializzato al puntatore al puntatore alla testa della lista `hp` (non è l'unica possibile).

Nel ciclo di attraversamento si verifica se si trova la chiave, in caso affermativo se ne salva il puntatore e il dato, si avanza nella lista ed infine si libera il nodo estratto.

```
Item listExtrKeyP(link *hp, Key k) {
    link *xp, t;
    Item i = ITEMsetvoid();
    for (xp=hp; (*xp)!=NULL; xp=&((*xp)->next)) {
        if (KEYeq(KEYget((*xp)->val),k)){
            t = *xp;
            *xp = (*xp)->next;
            i = t->val;
            free(t);
            break;
        }
    }
    return i;
}
```

Liste ordinate

- Dati di tipo **Item** ordinati in base a chiave
- Inserimento ($O(N)$) con ricerca della posizione
- Cancellazione ($O(N)$) con ricerca, *può decidere “non trovato” senza percorrere tutta la lista*

Inserzione

Lista
ordinata

Richiede:

- aggiornamento del puntatore alla testa per inserzione in lista vuota
o inserzione di dato con chiave minima (massima)
- ricerca della posizione in cui inserire, cioè identificazione nodo
predecessore con tecnica del doppio puntatore.

inserimento in testa

```
link SortListIns(link h, item val) {  
    link x, p;  
    Key k = KEYget(val);  
    if (h==NULL || KEYgreater(KEYget(h->val), k))  
        return newNode(val, h);  
    for (x=h->next, p=h;  
         x!=NULL && KEYgreater(k, KEYget(x->val));  
         p=x, x=x->next);  
    p->next = newNode(val, x);  
    return h;  
}
```

attraversamento
per ricerca
posizione

Ricerca

Lista
ordinata

Essendo l'accesso ai dati della lista lineare, anche se sono ordinati, non si usa la ricerca dicotomica.

La ricerca è identica a quella in lista non ordinata con eventuale interruzione anticipata.

```
Item SortListSearch(link h, Key k) {  
    link x;  
    for (x=h;  
         x!=NULL && KEYgeq(k, KEYget(x->val));  
         x=x->next)  
        if (KEYeq(KEYget(x->val), k))  
            return x->val;  
    return ITEMsetvoid();  
}
```

Cancellazione di nodo con chiave data

Lista
ordinata

Si aggiunge una condizione di interruzione anticipata al ciclo di attraversamento.

```
link SortListDel(link h, key k) {  
    link x, p;  
    if (h == NULL) return NULL;  
    for (x=h, p=NULL; x!=NULL && KEYgeq(k,KEYget(x->val));  
         p=x, x=x->next) {  
        if (KEYeq(KEYget(x->val),k)){  
            if (x==h) h = x->next;  
            else  
                p->next = x->next;  
            free(x);  
            break;  
        }  
    }  
    return h;  
}
```

Liste concatenate particolari

- Uso di nodi finti per semplificare i test di lista vuota
- Adiacenza logica di nodo in testa e in coda per ottenere una lista circolare
- Attraversamento in entrambe le direzioni con operazioni tipo cancellazione semplificate: liste concatenate doppie.

Liste con nodi finti (sentinelle)

- Nodo con dato fittizio (in testa e/o coda), usato per rimuovere casi speciali:
 - lista vuota
 - inserimento/cancellazione del primo o ultimo nodo

Lista con nodo fittizio in testa

inizializza	<code>h = malloc(sizeof *h); h->next = NULL;</code>
inserisci t dopo x	<code>t->next = x->next; x->next = t;</code>
cancella dopo x	<code>t = x->next; x->next = t->next;</code>
ciclo di attraversamento	<code>for (t = h->next; t != NULL; t = t->next)</code>
testa se lista vuota	<code>if (h->next == NULL)</code>

Lista con nodi finti in testa e coda

inizializza	<code>h = malloc(sizeof *h); z = malloc(sizeof *z); h->next = z; z->next = z;</code>
inserisci t dopo x	<code>t->next = x->next; x->next = t;</code>
cancella dopo x	<code>x->next = x->next->next;</code>
ciclo di attraversamento	<code>for (t = h->next; t != z; t = t->next)</code>
testa se lista vuota	<code>if (h->next == z)</code>

Lista circolare

- L'ultimo nodo punta al primo
- Utilizzata per gestire casi di servizi a “rotazione”

prima inserzione	<code>h->next = h;</code>
inserisci t dopo x	<code>t->next = x->next;</code> <code>x->next = t;</code>
cancella dopo x	<code>x->next = x->next->next;</code>
ciclo di attraversamento	<code>t = h;</code> <code>do { ... t = t->next; }</code> <code>while (t != h)</code>
testa singolo elemento	<code>if (h->next == h)</code>

Lista concatenata doppia

- Un puntatore in più (al nodo precedente)
- Facilita cancellazione (senza ricerca) dato il puntatore al nodo da cancellare

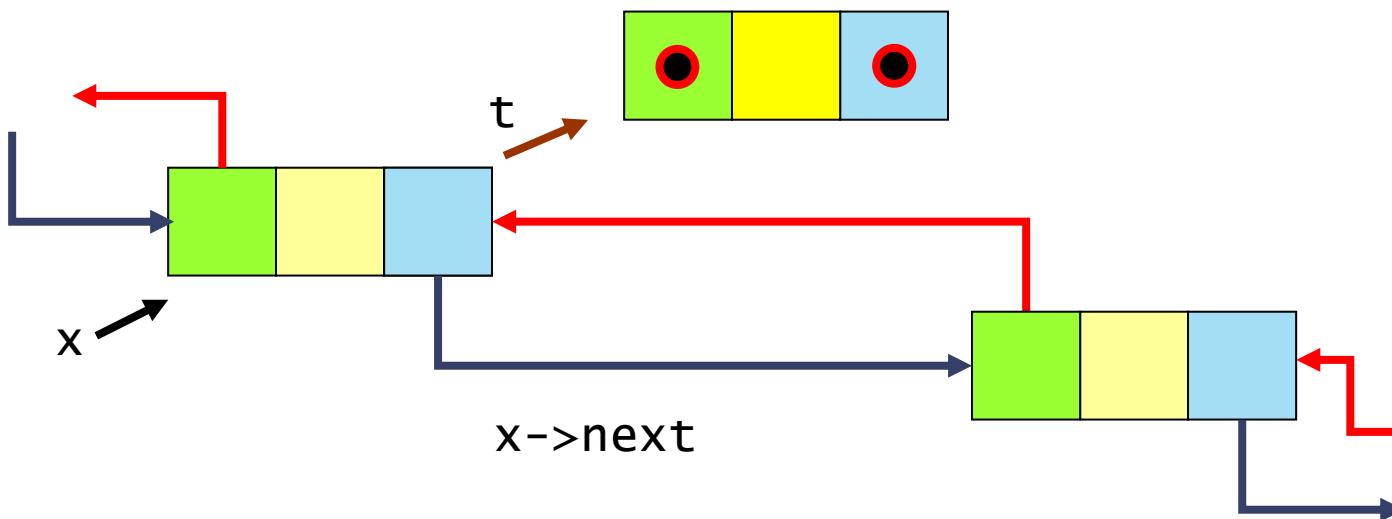
```
typedef struct node *link, node_t;

struct node {
    Item val;
    link next;
    link prev;
};
```

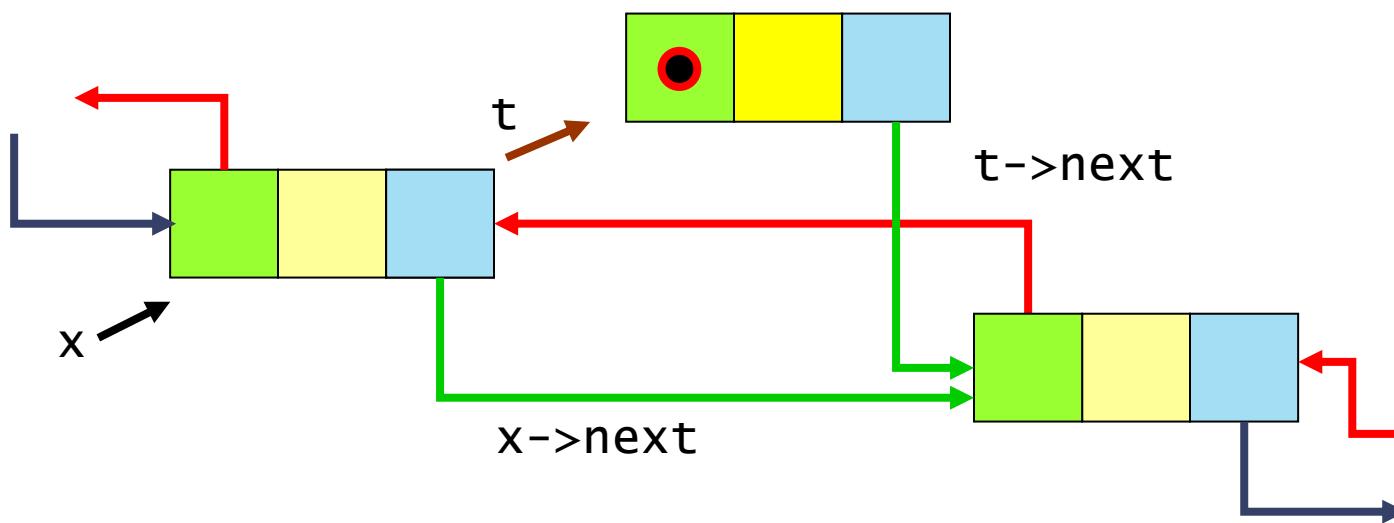
Inserimento di nodo **t** dopo nodo **x**

```
link x, t;
```

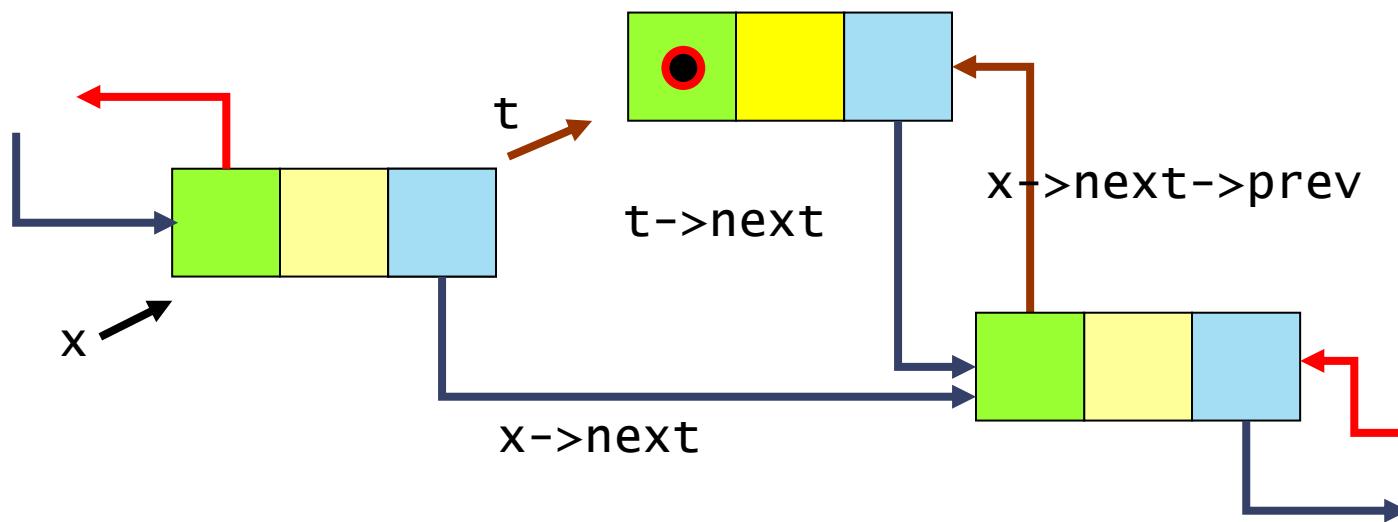
...



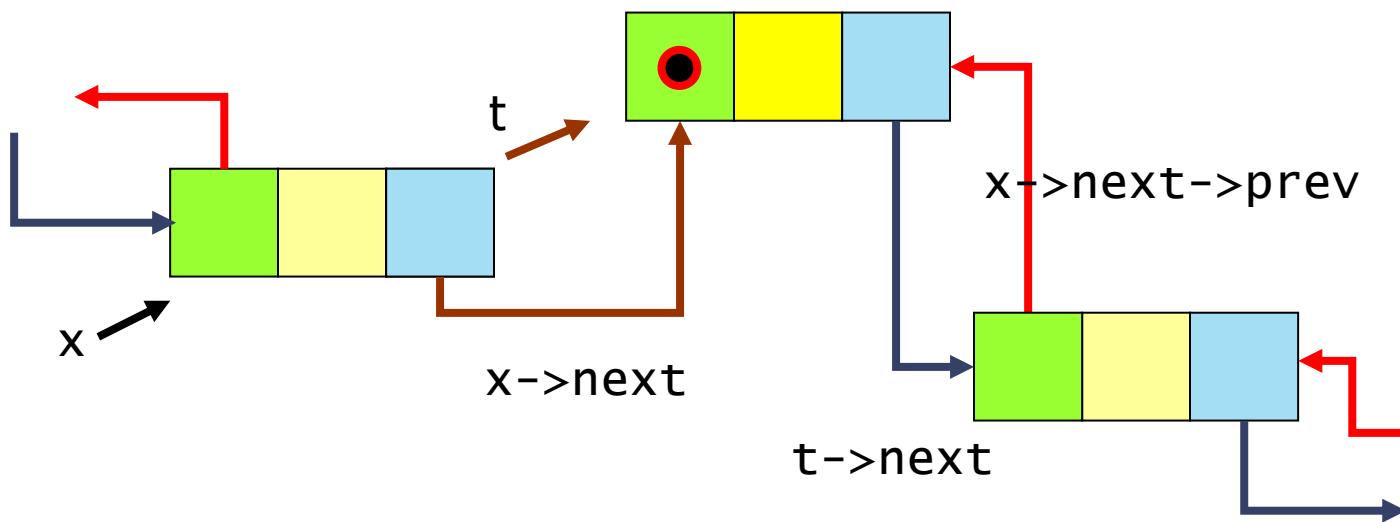
```
t->next = x->next;
```



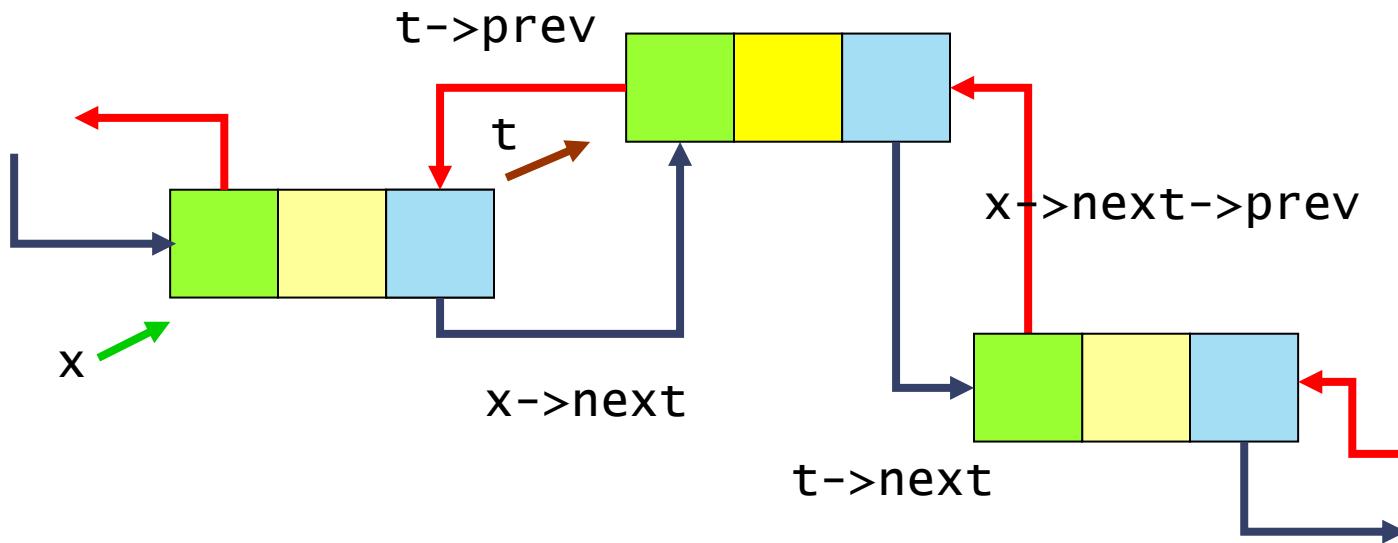
$x \rightarrow \text{next} \rightarrow \text{prev} = t;$



```
x->next = t;
```



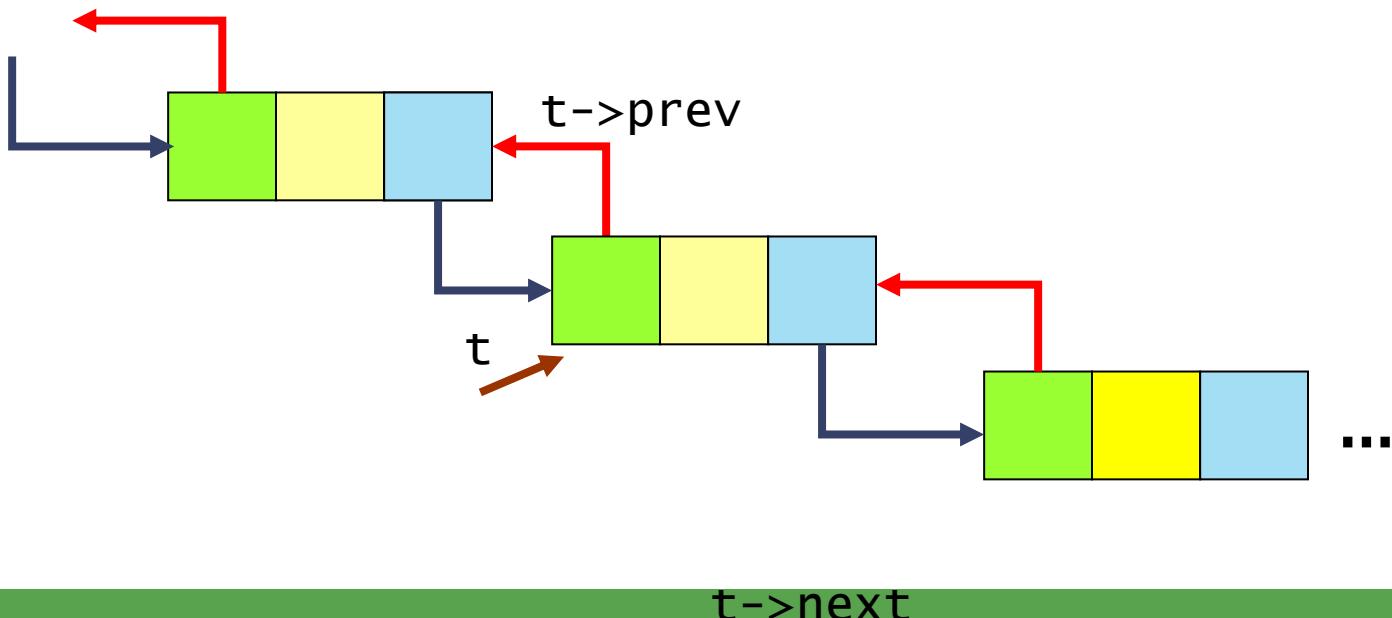
$t \rightarrow \text{prev} = x;$



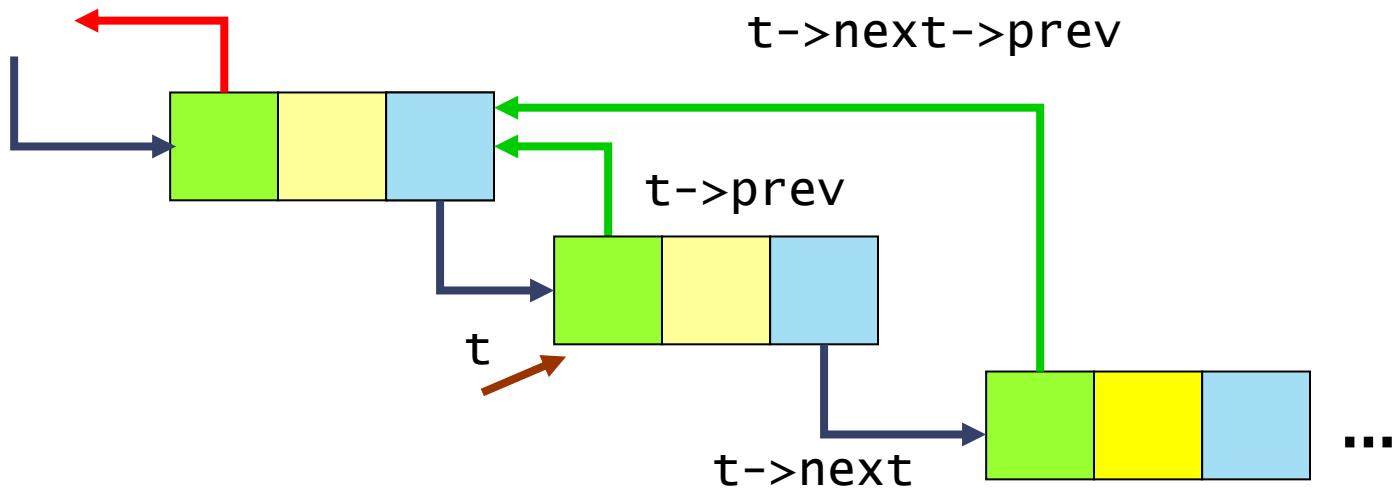
Cancellazione del nodo t

```
link t;
```

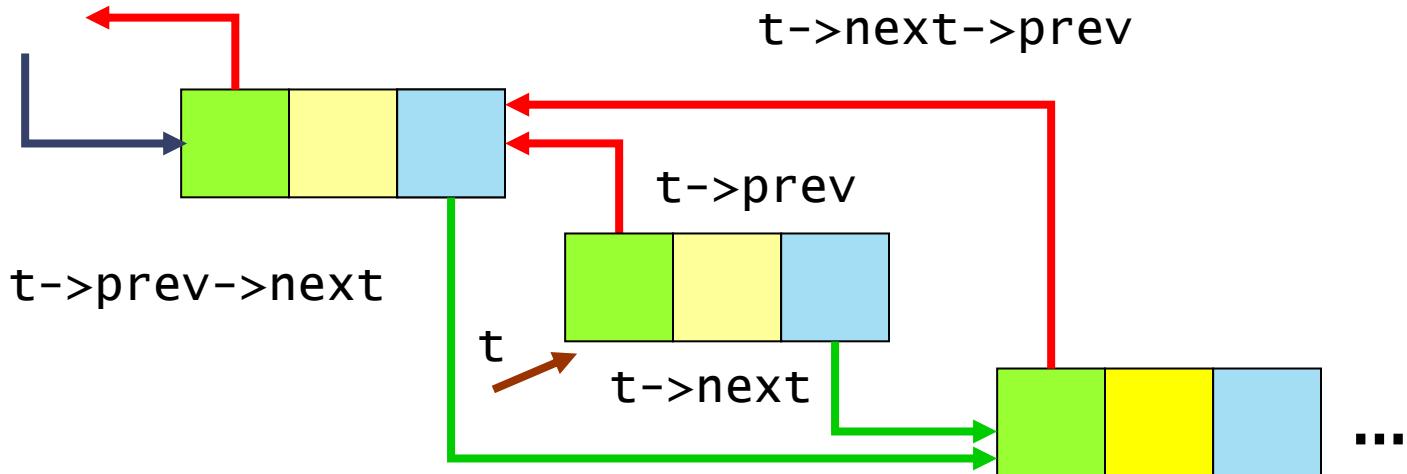
...



```
t->next->prev = t->prev;
```



```
t->prev->next = t->next;
```



Applicazioni

PROBLEMI SEMPLICI SU LISTE

Problemi semplici su liste

- Inversione di lista
- Insertion sort su lista
- Elenchi di canzoni

Inversione di lista

Data una lista, invertirla

Versione con funzioni su liste:

- due liste, vecchia e nuova
 - si estrae in testa dalla lista vecchia,
 - si inserisce in testa nella lista nuova

Algoritmo: finché esiste una porzione non vuota di lista y da invertire (iterazione):

- estrai nodo da testa della lista y
- inserisci nodo in testa alla lista r invertita

Inversione di lista: versione con funzioni

Finché esiste una porzione non vuota di lista y da invertire (iterazione):

- estraì nodo da testa della lista y
- inserisci nodo in testa alla lista r invertita

```
link listReverseF(link x) {
    link y = x, r = NULL;
    Item tmp;
    while (y != NULL) {
        tmp = listExtrHeadP(&y);
        r = listInsHead(r, tmp);
    }
    return r;
}
```

Inversione di lista: versione con funzioni

Finché esiste una porzione non vuota di lista

- estraì nodo da testa della lista y
- inserisci nodo in testa alla lista r invertita

ATTENZIONE: si distrugge una lista, se ne crea un'altra.

NON SI RICICLANO I NODI!

Si estraе un Item, si inserisce un Item

```
link listReverseF(link x) {
    link y = x, r = NULL;
    Item tmp;
    while (y != NULL) {
        tmp = listExtrHeadP(&y);
        r = listInsHead(r, tmp);
    }
    return r;
}
```

Inversione di lista

Data una lista, invertirla:

Versione con operazioni direttamente sulla lista: si «girano» i puntatori (ma concettualmente resta «estrai in testa, inserisci in testa»)

- x: puntatore alla testa della lista
- r: puntatore alla testa della lista già invertita (ultimo nodo già sistemato). Inizialmente r=NULL
- y: puntatore alla porzione di lista da invertire (primo nodo ancora da sistemare). Inizialmente y=x
- t: puntatore al nodo successivo al primo nodo ancora da sistemare (puntato da y)

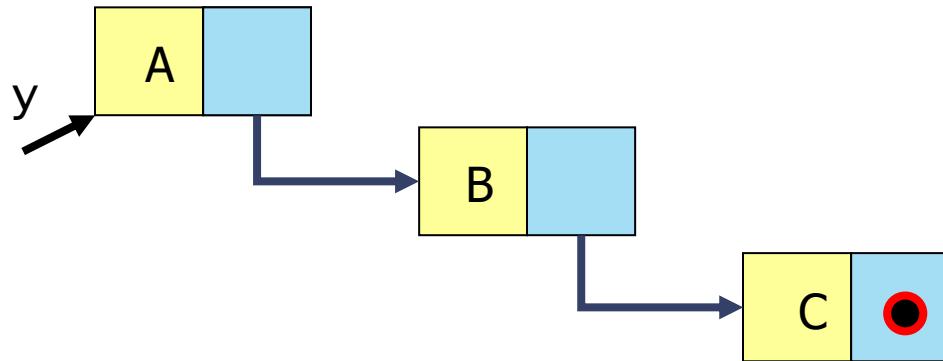
Inversione di lista: versione integrata

Finché esiste una porzione non vuota di lista y da invertire (iterazione):

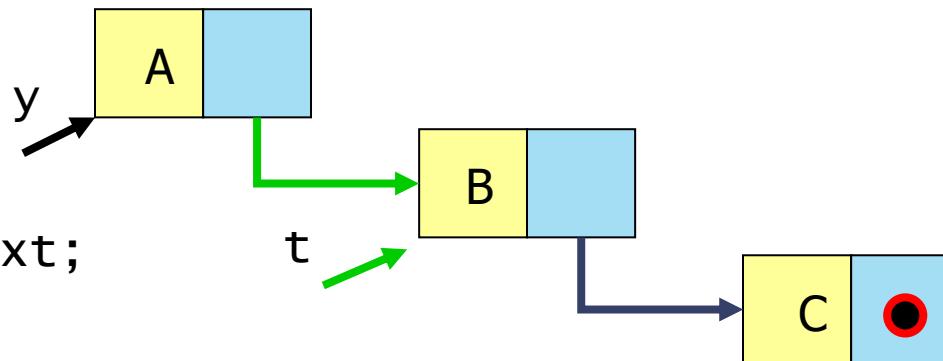
- inserire il nodo puntato da y in testa alla lista puntata da r
- aggiornare la testa della lista invertita r con y
- aggiornare y con il suo successore

```
link listReverseF(link x) {  
    link t, y = x, r = NULL;  
    while (y != NULL) {  
        t = y->next;  
        y->next = r;  
        r = y;  
        y = t;  
    }  
    return r;  
}
```

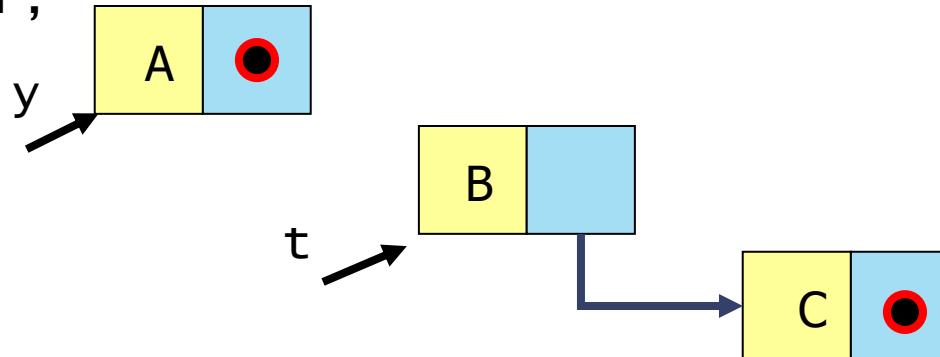
$y=x$
 $r=NULL$



$t = y->next;$

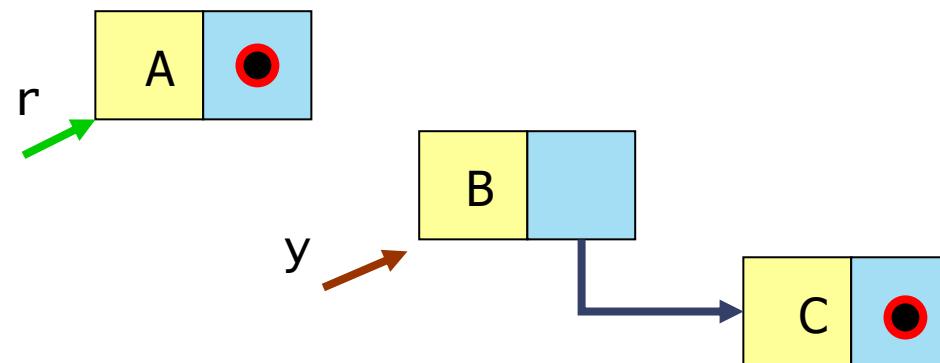


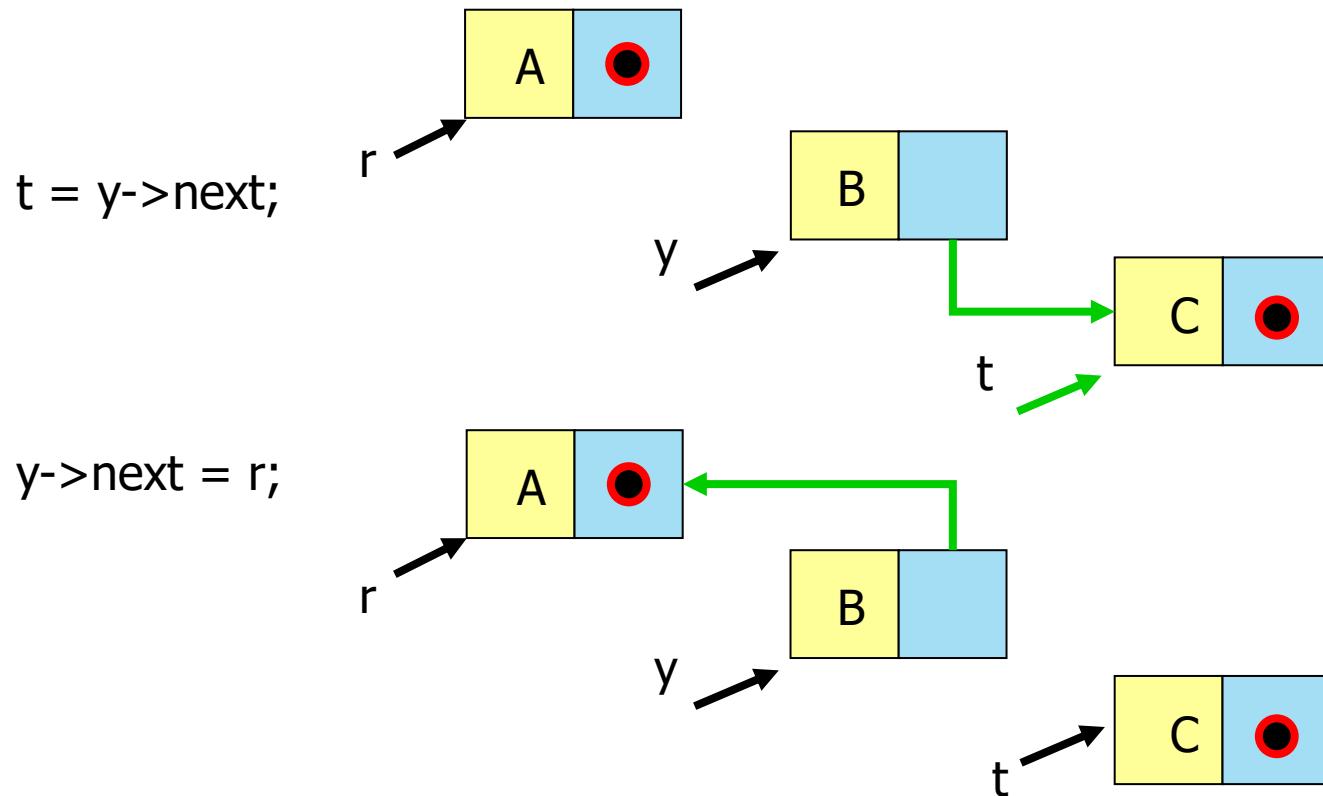
```
y->next = r;
```



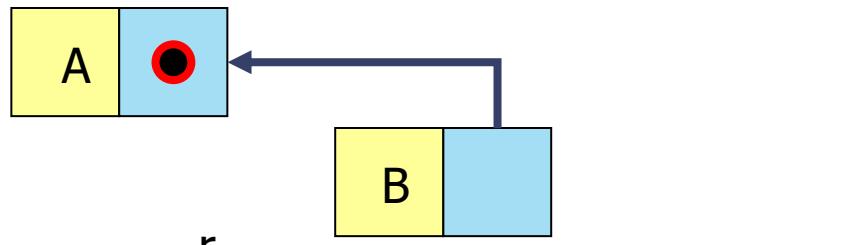
```
r = y;
```

```
y = t;
```

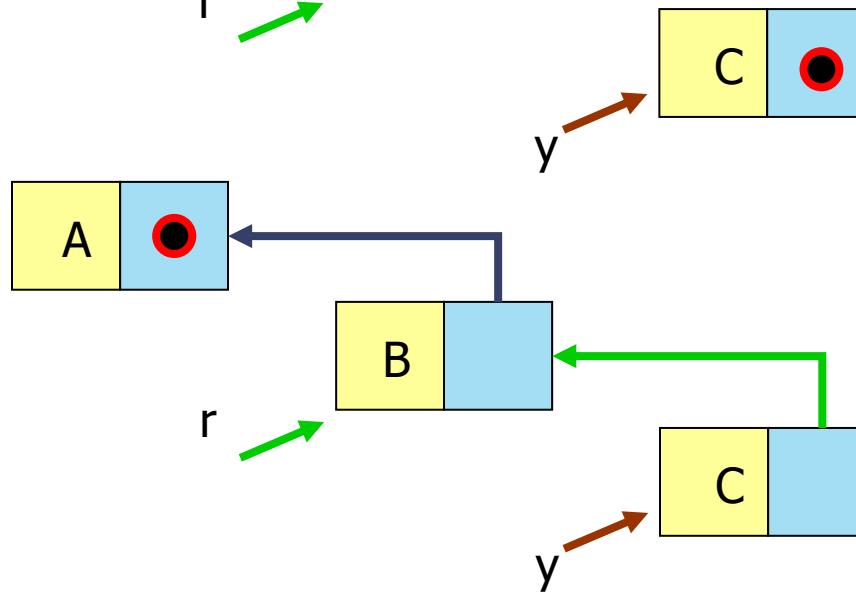




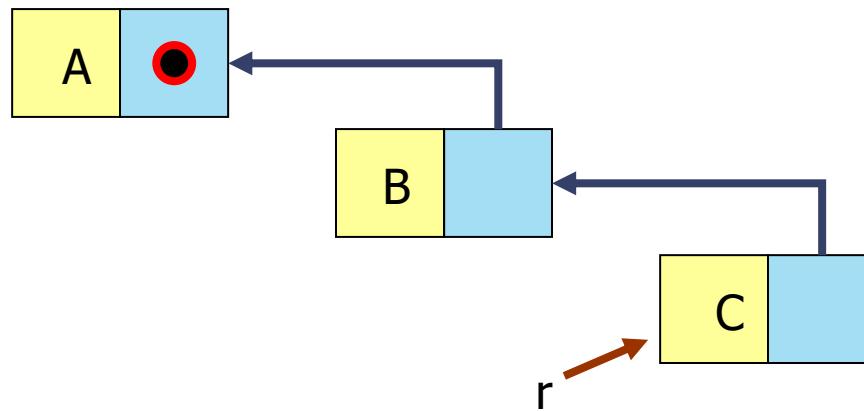
$r = y;$
 $y = t;$



$t = y->\text{next};$
 $y->\text{next} = r;$



$r = y;$
 $y = t;$



Insertion sort su lista

Solo ordinamenti quadratici, essendo impossibile l'accesso diretto tipico dei vettori

Lista contenente N numeri casuali con h puntatore alla testa

Versione con funzioni su liste:

- due liste, vecchia e nuova
 - si estrae in testa dalla lista vecchia (non ordinata),
 - si inserisce in ordine nella lista nuova

Algoritmo: finché esiste una porzione non vuota di lista y da ordinare (iterazione):

- estrai nodo da testa della lista y
- inserisci nodo in ordine nella lista r (ordinata)

Insertion sort su lista: versione con funzioni

Finché esiste una porzione non vuota di lista y da ordinare (iterazione):

- estraì nodo da testa della lista y
- inserisci nodo in ordine nella lista r (ordinata)

```
link listSortF(link h) {  
    link y = h, r = NULL;  
    Item tmp;  
    while (y != NULL) {  
        tmp = listExtrHeadP(&y);  
        r = SortListIns(r, tmp);  
    }  
    return r;  
}
```

Insertion sort su lista: versione con funzioni

Finché esiste una porzione non vuota di lista y da ordinare (iterazione):

- estraio nodo da testa della lista y
- inserisci nodo in ordine nella lista r (ordinata)

Unica differenza rispetto a inversione di lista: inserimento in ordine invece che in testa

```
link listSortF(link h) {  
    link y = h, r = NULL;  
    Item tmp;  
    while (y != NULL) {  
        tmp = listExtrHeadP(&y);  
        r = SortListIns(r, tmp);  
    }  
    return r;  
}
```

Insertion sort su lista: versione integrata

Solo ordinamenti quadratici, essendo impossibile l'accesso diretto tipico dei vettori

Lista contenente N numeri casuali con h puntatore alla testa

Versione con operazioni direttamente sulla lista: si «riutilizzano» i nodi esistenti (ma concettualmente resta «estrai in testa, inserisci in ordine»)

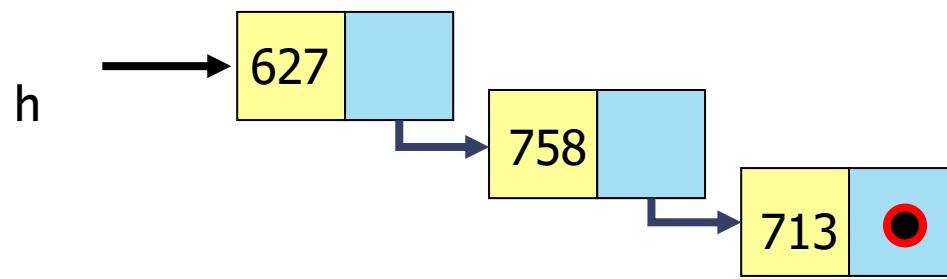
- Algoritmo:
 - inizialmente: nodo di testa puntato da h è primo nodo della lista ordinata ($h->next=NULL;$)
 - attraversamento della lista restante a partire dal secondo nodo puntato da t ($t=h->next;$), mantenimento del puntatore u al nodo successore e scalamento di t e u ad ogni iterazione
 - nel corpo dell'iterazione si verifica se il nodo contiene la chiave minima e lo si inserisce in testa o lo si inserisce in ordine.

```
for(t=h->next, h->next=NULL; t!=NULL; t=u){  
    u = t->next;  
    if (h->val > t->val) {  
        t->next = h;  
        h = t;  
    }  
    else {  
        for (x=h; x->next != NULL; x=x->next)  
            if (x->next->val > t->val)  
                break;  
        t->next = x->next;  
        x->next = t;  
    }  
}
```

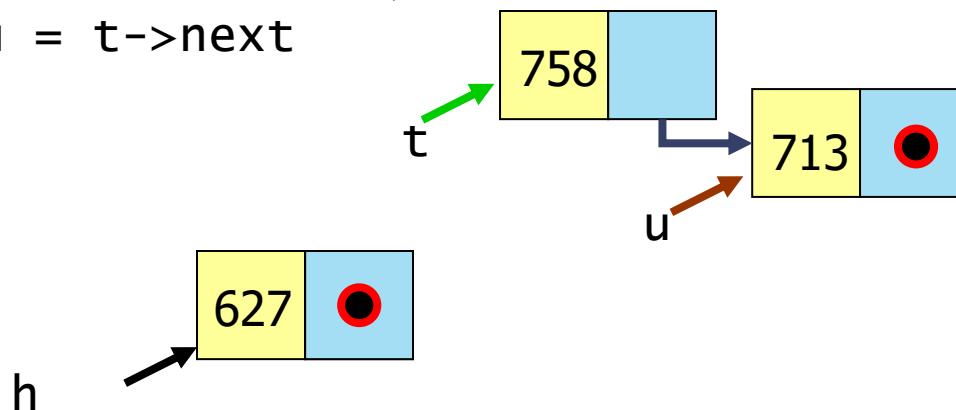
Caso particolare:
inserimento in testa

attraversamento per
ricerca posizione

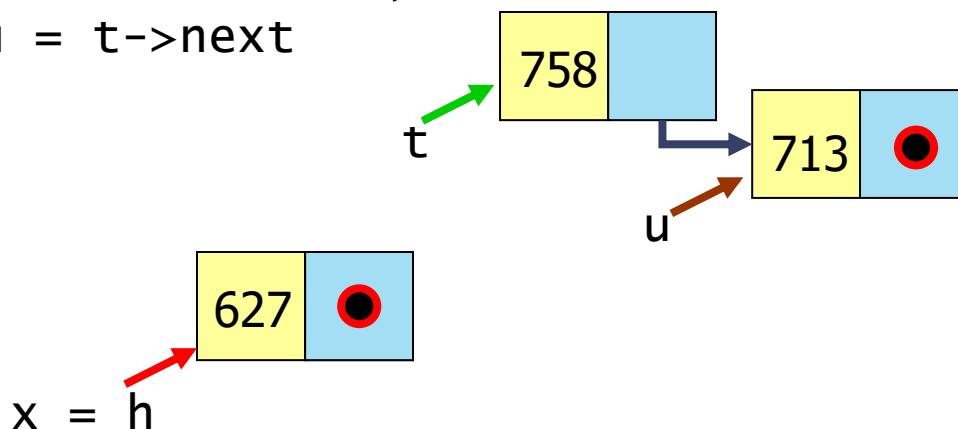
inserimento



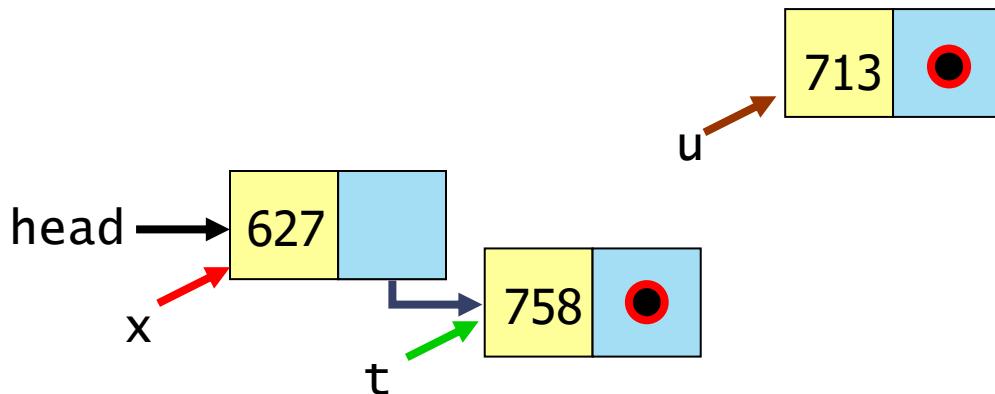
```
t = h->next;  
h->next = NULL;  
u = t->next
```



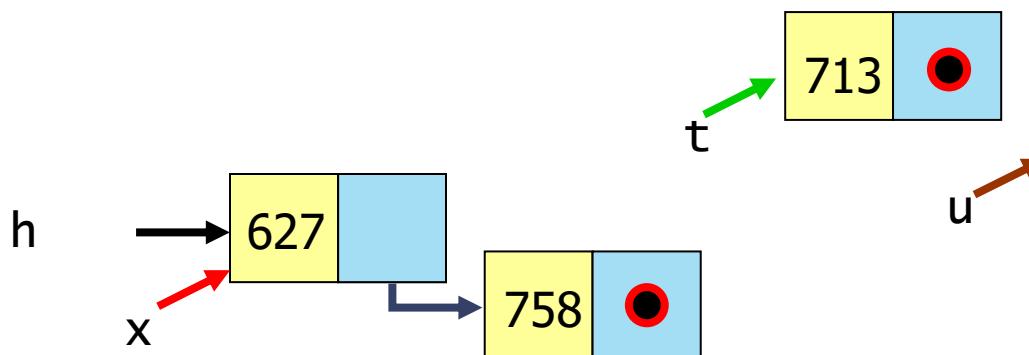
```
t = h->next;  
h->next = NULL;  
u = t->next
```



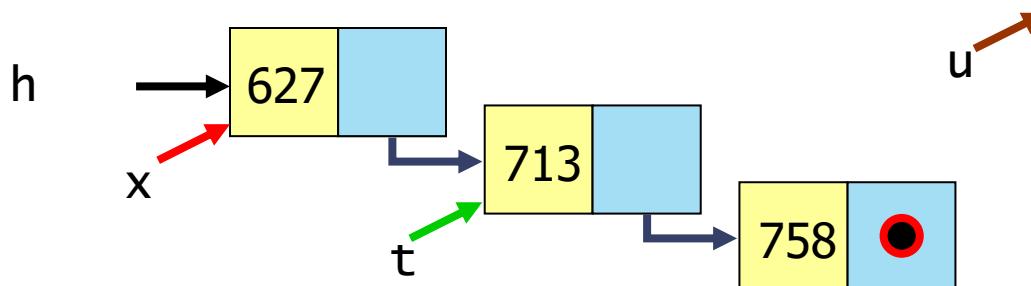
```
t->next = x->next;  
x->next = t;
```



```
t = u;  
u = u->next;
```



```
t->next = x->next;  
x->next = t;
```



Liste (concatenate) con indici

- Possibile realizzare liste con vettori
 - Dati NON ordinati in base alla posizione in lista, ma sparsi (ordinati in base alla cronologia di inserimento o ad altro criterio)
- Siccome i dati sono contenuti in vettori, essi sono individuabili mediante i loro **indici**
- Gli indici fungono da riferimenti, come i puntatori, ma nello spazio dei vettori e non in quello degli indirizzi di sistema
 - Si tratta quindi di vettori di struct (campo valore e campo indice del successore)
 - Un indice di valore -1 gioca il ruolo di NULL

Liste (non concatenate) con vettori

- Possibile realizzare liste con vettori
 - Dati ordinati in base alla posizione in lista
- Limiti:
 - allocazione del vettore della dimensione corretta o riallocazione
 - scarsa dinamicità in relazione alle cancellazioni.

Esempio: Elenchi di canzoni

Dati:

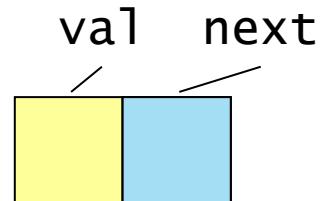
- un elenco di cantanti (stringhe), memorizzato in un vettore
- un elenco di canzoni (stringhe per i titoli, interi che indicano l'indice del cantante nel vettore dei cantanti)

creare per ogni cantante l'elenco delle sue canzoni.

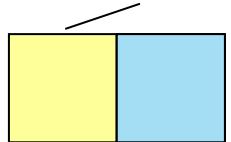
Soluzione 1: con liste concatenate

- Ogni cantante, oltre al nome, ha un puntatore alla testa della lista delle sue canzoni
- La lista delle canzoni è fatta di nodi che contengono la stringa del titolo
- Il tipo **Item** è un puntatore a carattere e si introduce per uniformità (`typedef char *Item`)

```
typedef struct nodo_s {  
    Item val;  
    struct nodo_s *next;  
} nodo_t, *link;
```

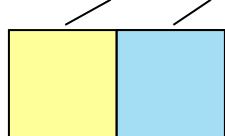


titolo id_cant



```
typedef struct canz_s{  
    char *titolo;  
    int id_cant;  
} canz_t;
```

nome h



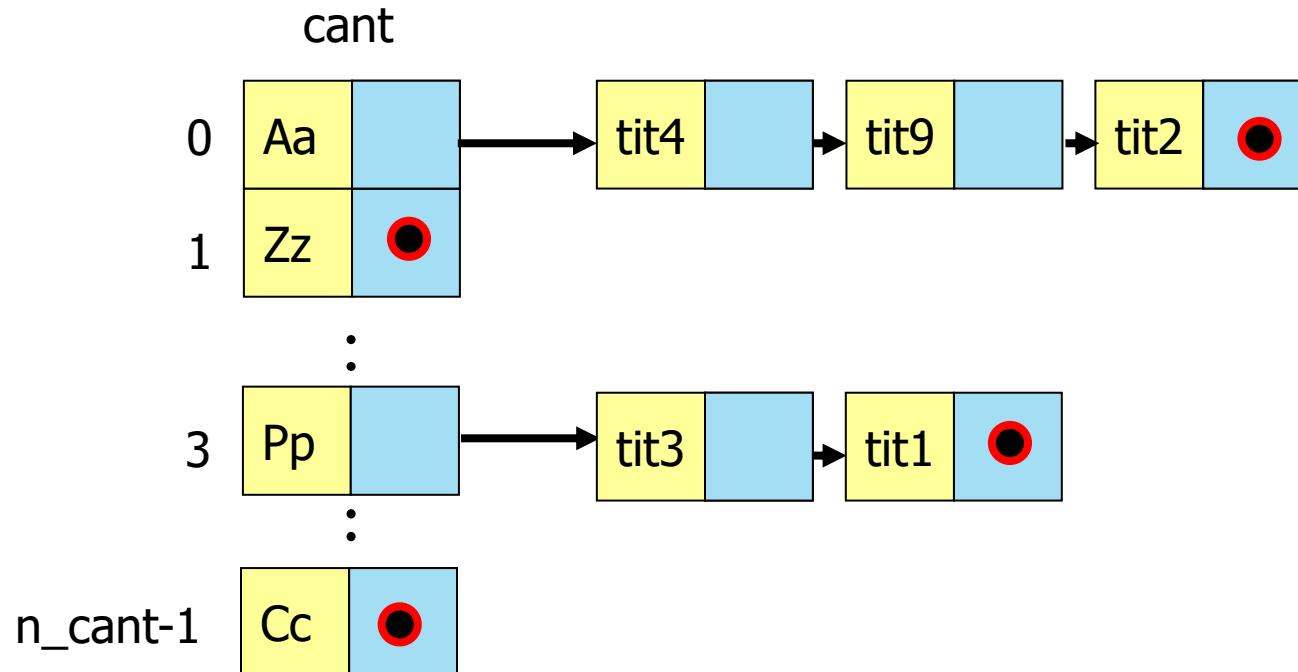
```
typedef struct cant_s{  
    char *nome;  
    link h;  
} cant_t;
```

Inizialmente

	cant
0	Aa
1	Zz
	:
3	Pp
	:
n_cant-1	Cc

	canz
0	tit2
1	tit1
	:
7	tit9
	:
11	tit3
	:
n_canz-1	tit4

Alla fine



```
void genEl(cant_t *cant, int n_cant,
           canz_t *canz, int n_canz) {
    int i, id_c;
    Item v;
    for (i=0; i<n_cant; i++)
        cant[i].h = NULL;
    for (i=0; i<n_canz; i++) {
        id_c = canz[i].id_cant;
        v = strdup(canz[i].titolo);
        cant[id_c].h = newNode(v,cant[id_c].h);
    }
}
```

inizializzazione a NULL
puntatori a teste delle liste

inserimento in testa del nuovo nodo

Equivale a:

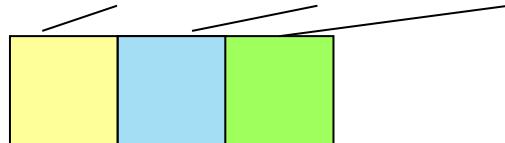
```
v = malloc((strlen(canz[i].titolo)+1)*sizeof(char));
strcpy(v, canz[i].titolo);
```

```
{
    in
    Iter
    for (i=0; i<n_cant; i++)
        canz[i].h = NULL;
    for (i=0; i<n_canz; i++) {
        id_c = canz[i].id_cant;
        v = strdup(canz[i].titolo);
        cant[id_c].h = newNode(v, cant[id_c].h);
    }
}
```

Soluzione 2: con liste concatenate

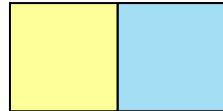
- Ogni cantante, oltre al nome, ha un puntatore alla testa della lista delle sue canzoni
- La lista delle canzoni di un cantante è creata concatenando i nodi del vettore canz che hanno quel cantante come autore
- Non si creano ex novo nodi per la lista, non serve il tipo **Item**

`titolo id_cant next`



```
typedef struct canz_s{  
    char *titolo;  
    int id_cant;  
    struct canz_s *next;  
} canz_t, *link;
```

`nome h`



```
typedef struct cant_s{  
    char *nome;  
    link h;  
} cant_t;
```

Inizialmente

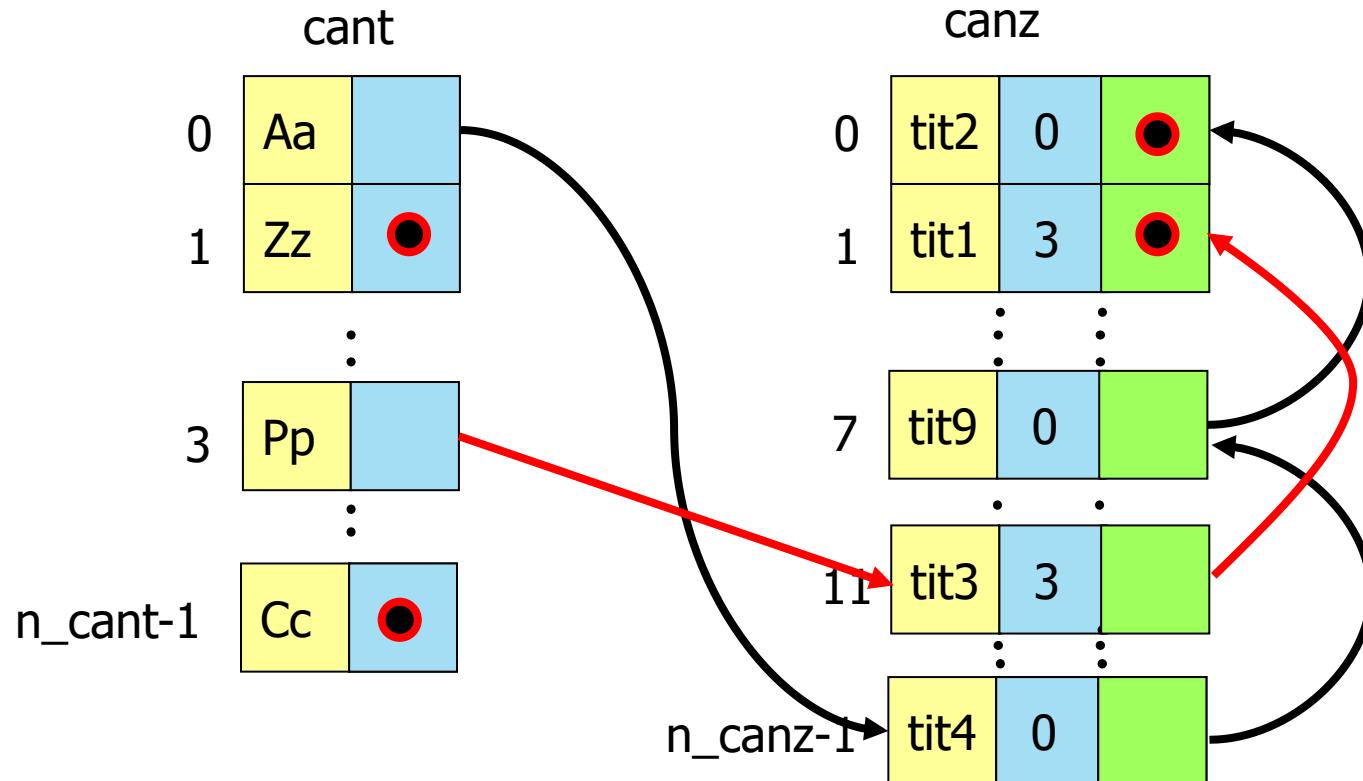
cant

0	Aa	
1	Zz	
	:	
3	Pp	
	:	
n_cant-1	Cc	

canz

0	tit2	0	
1	tit1	3	
	:		
7	tit9	0	
	:		
11	tit3	3	
	:		
n_canz-1	tit4	0	

Alla fine



inizializzazione a NULL
puntatori a teste delle liste

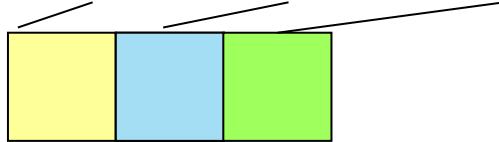
```
void genEl(cant_t *cant, int n_cant,  
          canz_t *canz, int n_canz) {  
    int i, id_c;  
    for (i=0; i<n_cant; i++)  
        cant[i].h = NULL;  
    for (i=0; i<n_canz; i++) {  
        id_c = canz[i].id_cant;  
        canz[i].next = cant[id_c].h;  
        cant[id_c].h = &canz[i];  
    }  
}
```

inserimento in testa del nuovo nodo

Soluzione 3: con indici

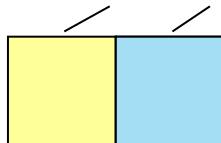
- Ogni cantante, oltre al nome, ha un indice del dato in testa alla lista delle sue canzoni
- Ogni canzone, oltre a titolo e indice del cantante, contiene l'indice della canzone successiva nella lista di quel cantante
- La lista delle canzoni di un cantante è creata concatenando mediante gli indici i nodi del vettore canz che hanno quel cantante come autore
- Non si creano ex novo nodi per la lista, non serve il tipo **Item**

`titolo id_cant id_next`



```
typedef struct canz_s{  
    char *titolo;  
    int id_cant;  
    int id_next;  
} canz_t;
```

`nome h`



```
typedef struct cant_s{  
    char *nome;  
    int id_testa;  
} cant_t;
```

Inizialmente

cant

0	Aa	-1
1	Zz	-1
	:	
3	Pp	-1
	:	
n_cant-1	Cc	-1

canz

0	tit2	0	-1
1	tit1	3	-1
	:	:	
7	tit9	0	-1
	:	:	
11	tit3	3	-1
	:	:	
n_canz-1	tit4	0	-1

Alla fine

cant	
0	Aa ncanz-1
1	Zz -1
⋮	⋮
3	Pp 11
⋮	⋮
n_cant-1	Cc -1

canz	
0	tit2 0 -1
1	tit1 3 -1
⋮	⋮ ⋮
7	tit9 0 0
⋮	⋮ ⋮
11	tit3 3 1
⋮	⋮ ⋮
n_canz-1	tit4 0 7

```
void genEl(cant_t *cant, int n_cant,
           canz_t *canz, int n_canz) {
    int i, id_c;

    for (i=0; i<n_cant; i++)
        cant[i].id_testa = -1;
    for (i=0; i<n_canz; i++) {
        id_c = canz[i].id_cant;
        canz[i].id_next = cant[id_c].id_testa;
        cant[id_c].id_testa = i;
    }
}
```

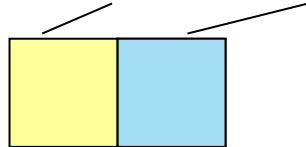
inizializzazione a -1
indici a teste delle liste

inserimento in testa

Soluzione 4: con vettori (lista non concatenata)

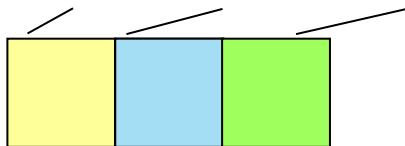
- Ogni cantante, oltre al nome, ha un vettore dinamico delle sue canzoni (puntatore) e il numero delle sue canzoni
- Ogni canzone contiene titolo e indice del cantante
- Il numero di canzoni di ogni cantante è calcolato con un'iterazione sulle canzoni
- La lista delle canzoni è realizzata tramite vettore di indici
- Non si creano ex novo nodi per la lista, non serve il tipo **Item**

`titolo id_cant`



```
typedef struct canz_s{  
    char *titolo;  
    int id_cant;  
} canz_t;
```

`nome lista n_canz`



```
typedef struct cant_s{  
    char *nome;  
    int *lista;  
    int n_canz;  
} cant_t;
```

Inizialmente

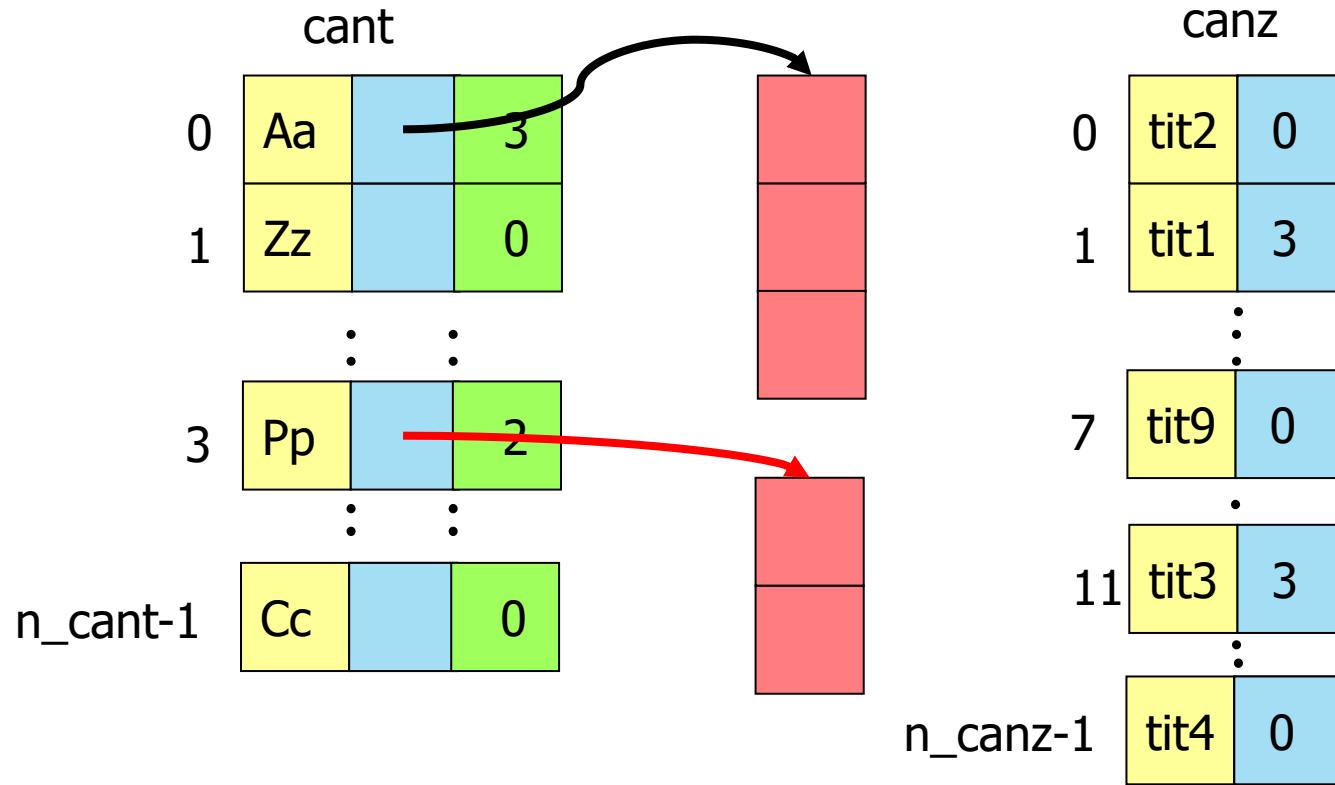
cant

0	Aa		0
1	Zz		0
⋮ ⋮			
3	Pp		0
⋮ ⋮			
n_cant-1	Cc		0

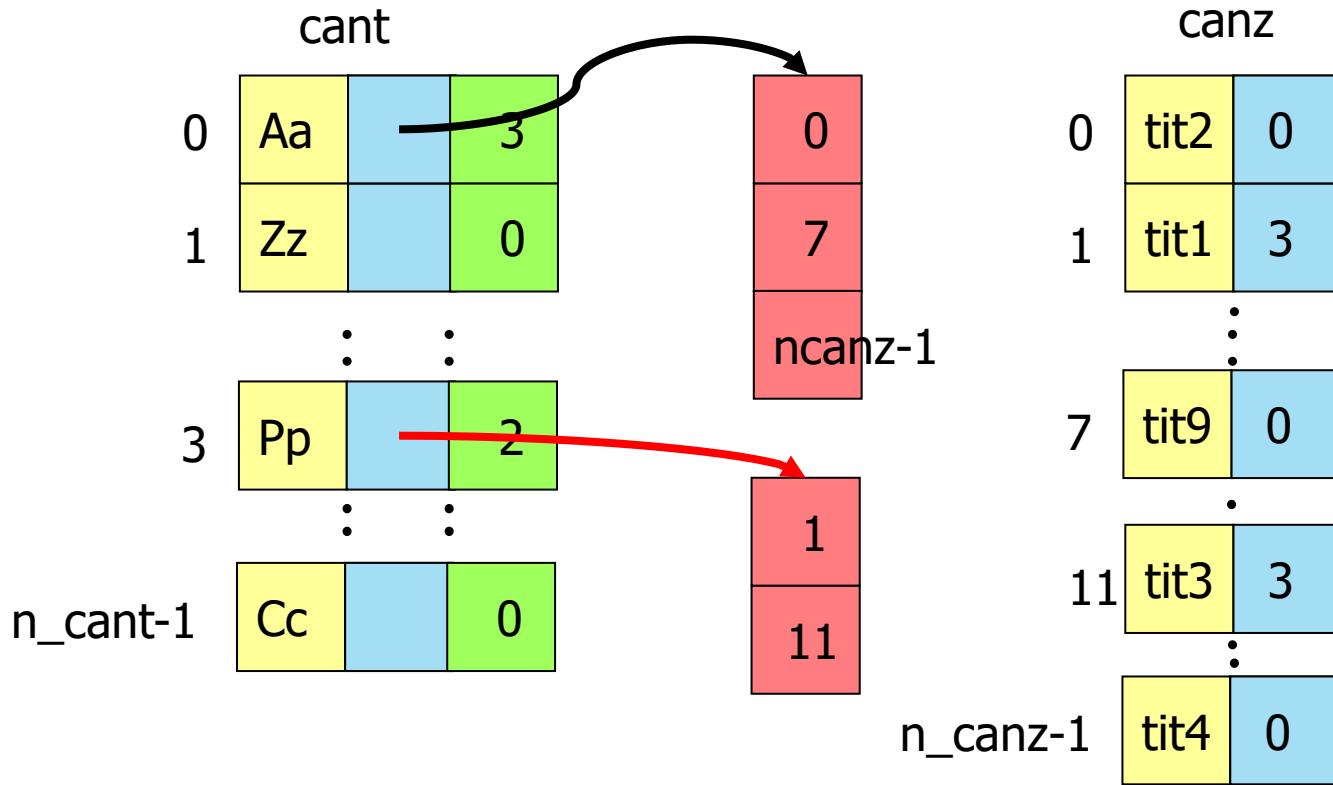
canz

0	tit2	0
1	tit1	3
⋮ ⋮		
7	tit9	0
⋮ ⋮		
11	tit3	3
⋮ ⋮		
n_canz-1	tit4	0

Allocazione vettori



Alla fine



```
void genEl(cant_t *cant, int n_cant,
           canz_t *canz, int n_canz) {
    int i, id_c;
    for (i=0; i<n_cant; i++)
        cant[i].n_canz = 0;
    for (i=0; i<n_canz; i++)
        cant[canz[i].id_cant].n_canz++;
    for (i=0; i<n_cant; i++) {
        if (cant[i].n_canz==0) cant[i].lista=NULL;
        else {
            cant[i].lista = malloc(cant[i].n_canz*sizeof(int));
            if (cant[i].lista == NULL) exit(-1);
            cant[i].n_canz = 0;
        }
    }
    for (i=0; i<n_canz; i++) {
        id_c = canz[i].id_cant;
        cant[id_c].lista[cant[id_c].n_canz] = i;
        cant[id_c].n_canz++;
    }
}
```

inizializzazione a 0 numero di canzoni

conteggio numero di canzoni di ogni cantante

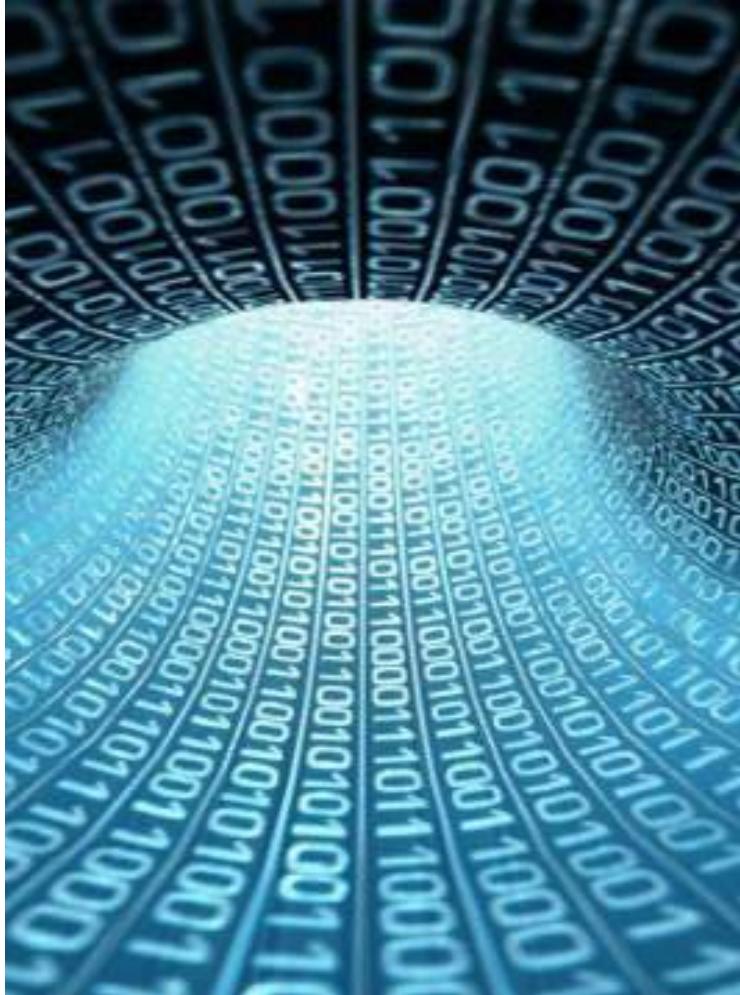
creazione lista canzoni di ogni cantante

re-inizializzazione contatore delle canzoni per successivo inserimento

inserimento in fondo

Capitolo 5: Strutture composte e modularità

PUNTATORI E STRUTTURE DATI DINAMICHE:
ALLOCAZIONE DELLA MEMORIA E
MODULARITÀ IN LINGUAGGIO C



Modularità

MODULARITÀ DI STRUTTURE DATI, IN PARTICOLARE QUELLE DINAMICHE

Modularità

Un programma si dice **modulare** quando:

- il problema viene risolto per scomposizione in sottoproblemi
- la scomposizione è visibile:
 - nell'algoritmo
 - nella struttura dati.

Un programma scomposto in funzioni presenta una forma preliminare di modularità.

Una struttura dati è resa modulare

- identificandone le parti
- associando a ciascuna le funzioni che vi operano.

Modulo come tipo di dato

Un **dato modulare** è un tipo di dato con le relative funzioni.

Esempio:

- acquisizione ripetuta di segmenti tramite i loro estremi (punti sul piano cartesiano con coordinate intere)
- calcolo della loro lunghezza
- terminazione acquisizione: segmenti a lunghezza 0

Soluzione 1: non modulare

- esiste il tipo di dato `punto_t`, ma non vi sono funzioni che operino su di esso
- il `main` accede direttamente alle coordinate dei punti estremi in lettura e per il calcolo della lunghezza
- il tipo `punto_t` serve solo a migliorare la leggibilità

Fa tutto il main

```
typedef struct {  
    int X, Y;  
} punto_t;  
  
int main(void) {  
    punto_t A, B;  
    int fine=0;  
    float l;  
    while (!fine) {  
        printf("coordinate primo estremo: ");  
        scanf("%d%d", &A.X, &A.Y);  
        printf("coordinate secondo estremo: ");  
        scanf("%d%d", &B.X, &B.Y);  
        l=sqrt((B.X-A.X)*(B.X-A.X)+  
                (B.Y-A.Y)*(B.Y-A.Y));  
        printf("Segmento (%d,%d)-(%d,%d) l: %f\n",  
               A.X,A.Y,B.X,B.Y,l);  
        fine = l==0;  
    }  
    return 0;  
}
```

Soluzione 2: modulare

- esiste il tipo di dato `punto_t`, cui sono associate 3 funzioni:
 - `puntoScan`
 - `puntoPrint`
 - `puntoDist`
- il `main` coordina le chiamate alle funzioni

Due moduli:

- `punto`: definizione di `punto_t` e funzioni
- `main`: utilizzatore (client).

Modulo di gestione “punto”

```
typedef struct { int X, Y; } punto_t;

void puntoScan(FILE *fp, punto_t *pp) {
    fscanf(fp, "%d%d", &pp->X, &pp->Y);
}

void puntoPrint(FILE *fp, punto_t p) {
    fprintf(fp, "(%d,%d)", p.X, p.Y);
}

float puntoDist(punto_t p0, punto_t p1) {
    int d2 = (p1.X-p0.X)*(p1.X-p0.X) +
             (p1.Y-p0.Y)*(p1.Y-p0.Y);
    return ((float) sqrt((double)d2));
}
```

funzione di lettura

funzione di scrittura

funzione di elaborazione

Il main chiama funzioni di gestione punto

```
typedef struct { int X, Y; } punto_t;

void puntoScan(FILE *fp, punto_t *pp) {
    fscanf(fp, "%d%d", &pp->X, &pp->Y);
}

void puntoPrint(FILE *fp, punto_t p) {
    fprintf(fp, "(%d,%d)", p.X, p.Y);
}

float puntoDist(punto_t p0, punto_t p1) {
    int d2 = (p1.X-p0.X)*(p1.X-p0.X) +
              (p1.Y-p0.Y)*(p1.Y-p0.Y);
    return ((float) sqrt((double)d2));
}
```

```
int main(void) {
    punto_t A, B;
    int fine=0; float l;
    while (!fine) {
        printf("primo estremo: "); puntoScan(stdin, &A);
        printf("secondo estremo: "); puntoScan(stdin, &B);
        l = puntoDist(A,B);
        printf("il segmento "); puntoPrint(stdout,A);
        printf("-"); puntoPrint(stdout,B);
        printf(" ha lunghezza: %f\n", l);
        fine = lunghezza==0;
    }
    return 0;
}
```

Modularità e allocazione dinamica

Alternativa all'allocazione/deallocazione automatica: **allocazione dinamica** dei dati.

In riferimento all'esempio precedente (in cui A e B sono struct):

- A e B sono puntatori a **struct**
- le **struct** i cui puntatori sono assegnati ad A e B sono allocate dinamicamente ed esplicitamente
- le funzioni ricevono e ritornano puntatori a **struct** e non **struct**.

Dati dinamici:

variabili e parametri formali: puntatori al tipo **punto_t**

```
typedef struct { int X, Y; } punto_t;  
  
punto_t *puntoCrea(void) {  
    punto_t *pp = (punto_t *) malloc(sizeof(punto_t));  
    return pp; }  
  
void puntoLibera(punto_t *pp) {  
    free(pp); }  
  
void puntoScan(FILE *fp, punto_t *pp) {  
    fscanf(fp, "%d%d", &pp->X, &pp->Y); }  
  
void puntoPrint(FILE *fp, punto_t *pp) {  
    fprintf(fp, "(%d,%d)", pp->X, pp->Y); }  
  
float puntoDist(punto_t *pp0, punto_t *pp1) {  
    int d2 = (pp1->X-pp0->X)*(pp1->X-pp0->X) +  
             (pp1->Y-pp0->Y)*(pp1->Y-pp0->Y);  
  
    return ((float) sqrt((double)d2)); }
```

funzione di creazione

funzione di distruzione

funzione di lettura

funzione di scrittura

funzione di elaborazione

Dati dinamici: variabili e parametri formali: puntatori al tipo **punto_t**

```
typedef struct { int X, Y; } punto_t;

punto_t *puntoCrea(void) {
    punto_t *pp = (punto_t *) malloc(sizeof(punto_t));
    return pp;
}

void puntoLibera(punto_t *pp) {
    free(pp);
}

void puntoScan(FILE *fp, punto_t *pp) {
    fscanf(fp, "%d%d", &pp->X, &pp->Y);
}

void puntoPrint(FILE *fp, punto_t *pp) {
    fprintf(fp, "(%d,%d)", pp->X, pp->Y);
}

float puntoDist(punto_t *pp0, punto_t *pp1) {
    int d2 = (pp1->X-pp0->X)*(pp1->X-pp0->X) +
              (pp1->Y-pp0->Y)*(pp1->Y-pp0->Y);
    return ((float) sqrt((double)d2));
}
```

```
int main(void) {
    punto_t *A, *B; int fine=0; float lunghezza;
    A = puntoCrea(); B = puntoCrea();
    while (!fine) {
        printf("I estremo: "); puntoScan(stdin, A);
        printf("II estremo: "); puntoScan(stdin, B);
        lunghezza = puntoDist(A,B);
        printf("Segmento "); puntoPrint(stdout,A);
        printf("-"); puntoPrint(stdout,B);
        printf(" ha lunghezza: %f\n", lunghezza);
        file = lunghezza==0;
    }
    puntoLibera(A);
    puntoLibera(B);
    return 0;
}
```

Variante:

tipo **punto_t** puntatore a struct (non si vedono gli *)

```
typedef struct { int X, Y; } *ppunto_t;

ppunto_t puntoCrea(void) {
    ppunto_t pp = (ppunto_t) malloc(sizeof *pp);
    return pp;
}

void puntoLibera(ppunto_t pp) {
    free(pp);
}

void puntoScan(FILE *fp, ppunto_t pp) {
    fscanf(fp, "%d%d", &pp->X, &pp->Y);
}

void puntoPrint(FILE *fp, ppunto_t pp) {
    fprintf(fp, "(%d,%d)", pp->X, pp->Y);
}

float puntoDist(ppunto_t pp0, ppunto_t pp1) {
    int d2 = (pp1->X-pp0->X)*(pp1->X-pp0->X) +
              (pp1->Y-pp0->Y)*(pp1->Y-pp0->Y);
    return ((float) sqrt((double)d2));
}
```

```
int main(void) {
    ppunto_t A, B; int fine=0; float lunghezza;
    A = puntoCrea(); B = puntoCrea();
    while (!fine) {
        printf("I estremo: "); puntoScan(stdin, A);
        printf("II estremo: "); puntoScan(stdin, B);
        lunghezza = puntoDist(A,B);
        printf("Segmento "); puntoPrint(stdout,A);
        printf("-"); puntoPrint(stdout,B);
        printf(" ha lunghezza: %f\n", lunghezza);
        file = lunghezza==0;
    }
    puntoLibera(A);
    puntoLibera(B);
    return 0;
}
```

Variante:

tipo **punto_t** puntatore a struct (non si vedono gli *)

Visto dal main

Non è chiaro se si tratti di
puntatori o no
... A meno di capire cosa fanno
puntoCrea e puntoLibera

```
int main(void) {
    ppunto_t A, B; int fine=0; float lunghezza;
    A = puntoCrea(); B = puntoCrea();
    while (!fine) {
        printf("I estremo: "); puntoScan(stdin, A);
        printf("II estremo: "); puntoScan(stdin, B);
        lunghezza = puntoDist(A,B);
        printf("Segmento "); puntoPrint(stdout,A);
        printf("-"); puntoPrint(stdout,B);
        printf(" ha lunghezza: %f\n", lunghezza);
        file = lunghezza==0;
    }
    puntoLibera(A);
    puntoLibera(B);
    return 0;
}
```

Funzioni di creazione/distruzione

Per evitare memory leak, gestendo in modo modulare strutture dati allocate dinamicamente, è necessario/opportuno che:

- la creazione di un dato sia evidente e gestita con uniformità. Può essere interna al modulo o visibile anche al **client**
- ci sia un modulo **responsabile** di ogni struttura dinamica.
 - In generale deve distruggere chi ha creato.
 - A volte c'è un “trasferimento” (meglio se esplicito/chiaro) di responsabilità

Esempio: estensione dell'esempio sui punti con

- funzione **puntoDup1** che duplica un punto allocandolo al suo interno
- funzione **puntoM** che, dati 2 punti, ne ritorna un terzo (allocato internamente) che coincide con quello tra i 2 più lontano dall'origine

```
punto_t *puntoDupl(punto_t *pp) {  
    punto_t *pp2 = puntoCrea();  
    *pp2 = *pp;  
    return pp2;  
}  
  
punto_t *puntoM(punto_t *pp0, punto_t *pp1) {  
    punto_t origine = {0,0};  
    float d0 = puntoDist(&origine,pp0);  
    float d1 = puntoDist(&origine,pp1);  
    if (d0>d1)  
        return puntoDupl(pp0);  
    else  
        return puntoDupl(pp1);  
}
```

Chiamata alla funzione di creazione

Creazione/distruzione «nascoste»

Il `main` proposto:

- crea e distrugge le variabili A e B
- distrugge la variabile `max`, creata da `puntoM` mediante chiamata a `puntoDup1`, **senza che il `main` ne renda visibile la creazione.**

La soluzione è corretta, ma **debole**.

```
punto_t *puntoDupl(punto_t *pp) {  
    punto_t *pp2 = puntoCrea();  
    *pp2 = *pp;  
    return pp2;  
}
```

```
punto_t *puntoM(punto_t *pp0, punto_t *pp1) {  
    punto_t origine = {0,0};  
    float d0 = puntoDist(&origine,pp0);  
    float d1 = puntoDist(&origine,pp1);  
    if (d0>d1)  
        return puntoDupl(pp0);  
    else  
        return puntoDupl(pp1);  
}
```

il main distrugge
A, B e max

```
int main(void) {  
    punto_t *A, *B, *max;  
  
    A = puntoCrea(); B = puntoCrea();  
  
    /* input dei 2 punti A e B */  
  
    max = puntoM(A,B);  
    printf("Punto piu' lontano: ");  
    puntoPrint(stdout,max);  
  
    puntoLibera(A);  
    puntoLibera(B);  
    puntoLibera(max);  
    return 0;  
}
```

puntoM crea max

Esempio con memory leak

Nel codice seguente:

- il `main` crea le variabili `A` e `B`
- `puntoM` salva in `A` il punto più lontano dall'origine, risparmiando la variabile `max`
- il vecchio valore di `A` è perso, ma la memoria allocata non è liberata
- il `main` libera solo 2 dei 3 dati allocati (esplicitamente o in maniera nascosta)

La soluzione è scorretta in quanto introduce un memory leak.

Memory leak del «vecchio» A

```
punto_t *puntoDupl(punto_t *pp) {  
    punto_t *pp2 = puntoCrea();  
    *pp2 = *pp;  
    return pp2;  
}
```

il main crea A e B

```
punto_t *puntoM(punto_t *pp0, punto_t *pp1) {  
    punto_t origine = {0,0};  
    float d0 = puntoDist(&origine,pp0);  
    float d1 = puntoDist(&origine,pp1);  
    if (d0>d1)  
        return puntoDupl(pp0);  
    else  
        return puntoDupl(pp1);  
}
```

il main distrugge
A (il nuovo) e B

```
int main(void) {  
    punto_t *A, *B, *max;  
  
    A = puntoCrea(); B = puntoCrea();  
  
    /* input dei 2 punti A e B */  
  
    A = puntoM(A,B);  
    printf("Punto piu' lontano: ");  
    puntoPrint(stdout,max);  
  
    puntoLibera(A);  
    puntoLibera(B);  
    return 0;  
}
```

Il main perde il vecchio
A senza Liberarlo,
puntoM ne crea uno nuovo

Composizione e Aggregazione

CONTENITORI PER VALORE E/O RIFERIMENTO

Composizione e aggregazione

Esempi precedenti:

- modulo come tipo di dato e relative funzioni
- casi semplici di dimensione ridotta.

struct in C

- raggruppare dati omogenei o eterogenei assieme.

Composizione e aggregazione:

strategie per raggruppare dati o riferimenti a dati in un unico dato composto tenendo conto delle relazioni gerarchiche di appartenenza e possesso.

Composizione: A contiene B

- **composizione stretta con possesso:**
 - per valore A *include* B
 - Per riferimento A *include un riferimento a* B
 - B è esterno ad A ma viene considerato “proprietà” di A
- **aggregazione senza possesso:**
 - A *include un riferimento a* B
 - B NON è proprietà di A, che fa quindi riferimento a un dato “esterno”

Composizione con possesso

Casi semplici (esempi):

- oggetti composti da più parti: PC composto da CPU, scheda madre, memoria, dispositivi di I/O etc.
- **annidamento**: replica all'interno dello stesso meccanismo di composizione esterno.

Quando il dato contenuto è a sua volta un dato composto (vettore o struct) ci sono 2 strade:

- includere il **dato (valore)**
- includere un **riferimento** al dato.

Se A **possiede** B, ha la responsabilità di crearlo e distruggerlo

Composizione per valore

un dato contiene completamente il dato interno

- caso inequivocabile di composizione con possesso.

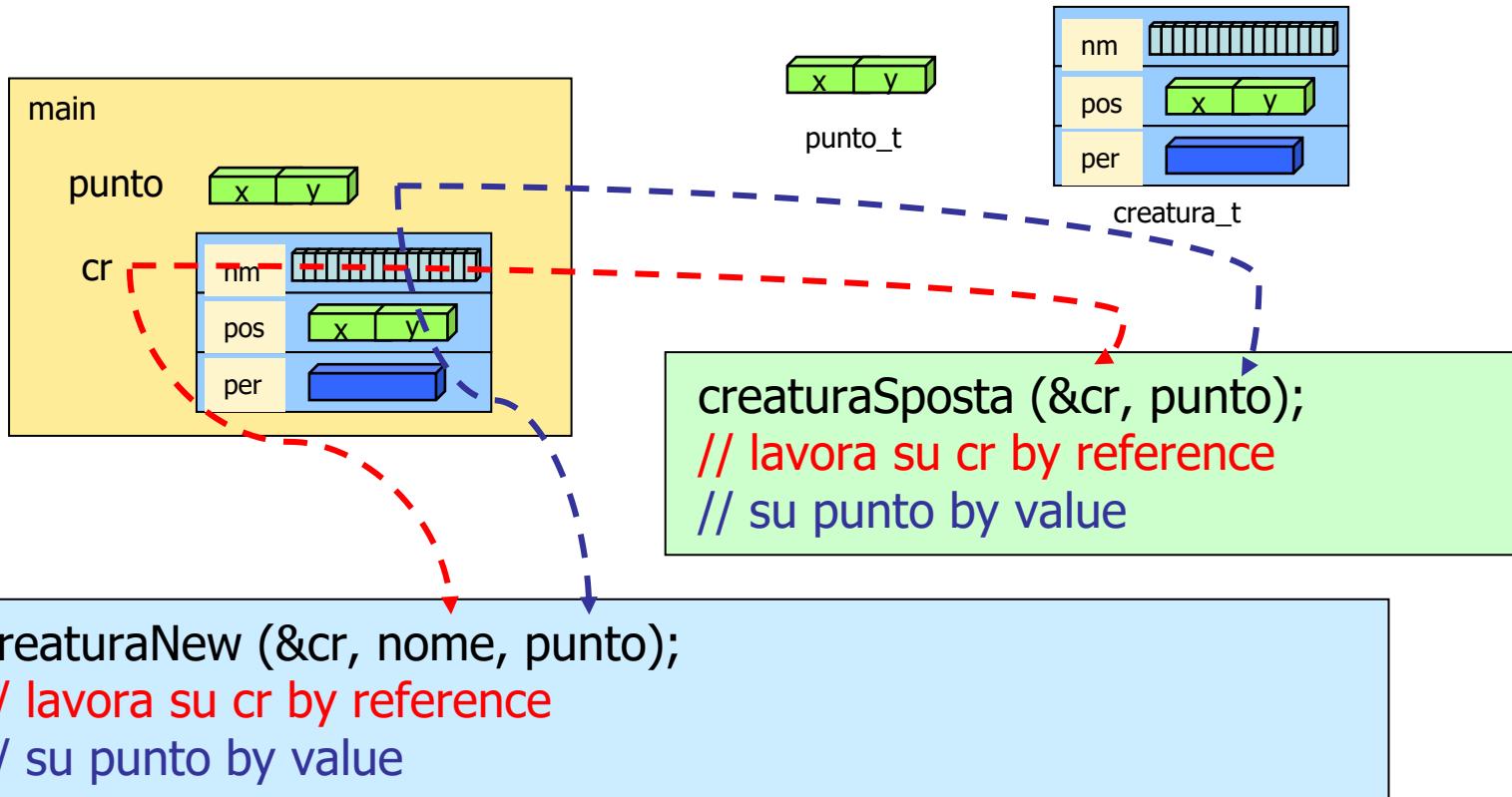
Esempio: simulazione di una creatura che percorre una spezzata (punti su piano cartesiano con coordinate intere non negative):

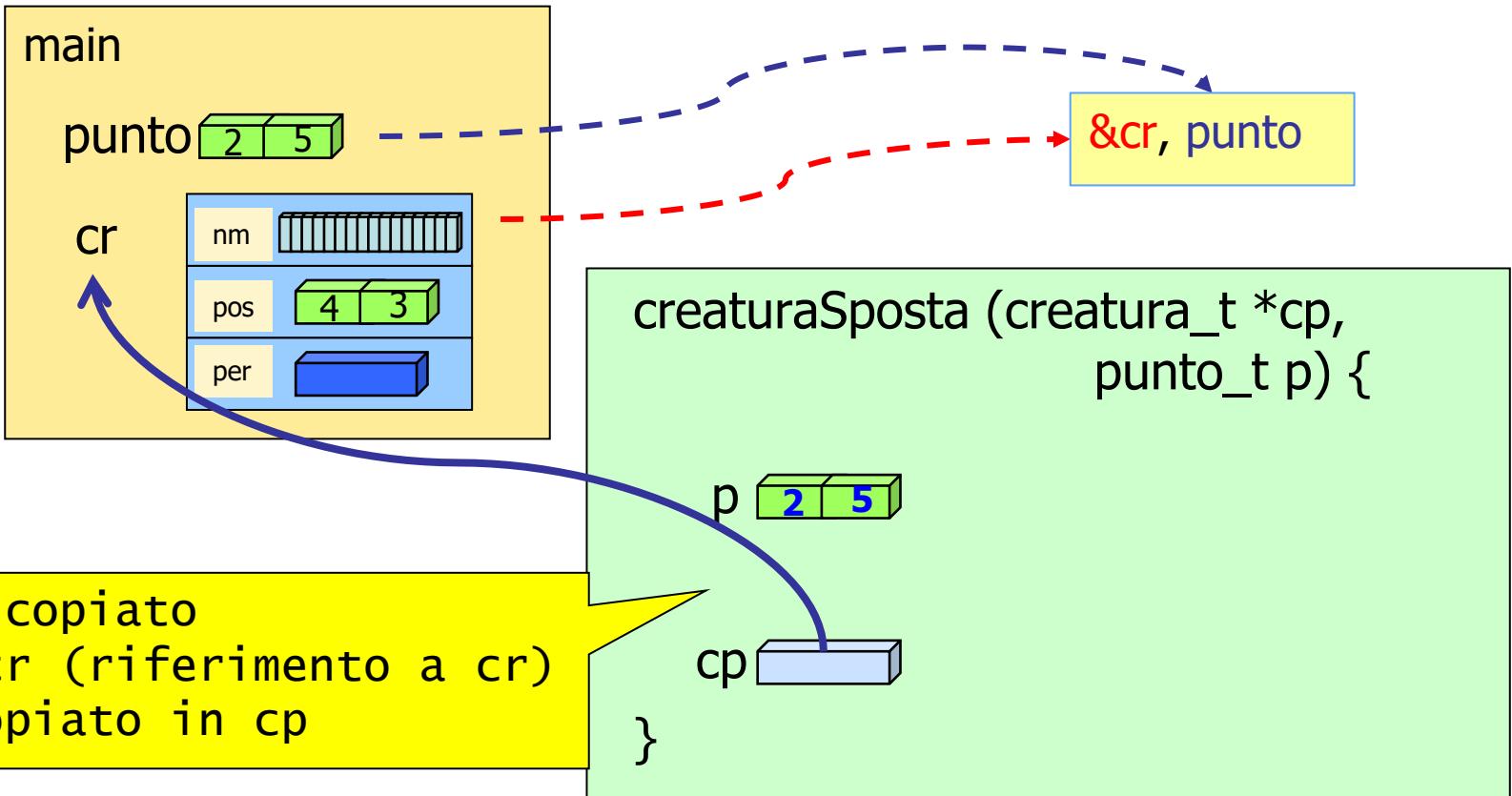
- dato ad alto livello: **creatura**
- dato a basso livello: **punto** (eventualmente ancora composto da 2 coordinate)

Specifiche:

- acquisizione di nome e posizione iniziale della creatura
- acquisizione iterativa delle nuove posizioni
- calcolo del percorso e stampa della lunghezza totale alla fine
- terminazione nel caso di almeno una coordinata negativa.

Composizione per valore

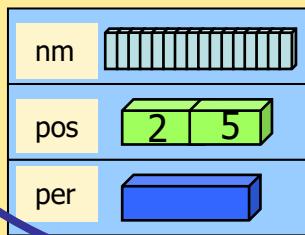




main

punto [2 5]

cr



`&cr, punto`

creaturaSposta (`creature_t *cp,`
`punto_t p) {`

`p [2 5]`

`cp`

`cp->pos = ...`
Aggiorna direttamente `cr`

Definizioni e funzioni di creazione/manipolazione

```
typedef struct {  
    int X, Y;  
} punto_t;  
  
typedef struct {  
    char nome[MAXS];  
    punto_t posizione;  
    float percorsoTotale;  
} creatura_t;  
  
/* funzioni di  
...  
za */
```

il dato posizione è incluso e posseduto dal dato creatura

funzioni di creazione e manipolazione della creatura

```
int puntoFuori(punto_t p) {  
    return (p.X<0 || p.Y<0);  
}  
  
void creaturaNew(creatura_t *cp, char *nome, punto_t punto)  
{  
    strcpy(cp->nome,nome);  
    cp->posizione = punto;  
    cp->percorsoTotale = 0.0;  
}  
  
void creaturaSposta(creatura_t *cp, punto_t p) {  
    cp->percorsoTotale += puntoDist(cp->posizione,p);  
    cp->posizione = p;  
}
```

main

```
int main(void) {
    char nome[MAXS];
    punto_t punto;
    creatura_t cr;
    int fine=0;
    printf("Creatura : "); scanf("%s", nome);
    printf("Inizio: "); puntoScan(stdin,&punto);

    creaturaNew(&cr,nome,punto);
    while (!fine) {
        printf("Nuovo: ");
        puntoScan(stdin,&punto);
        if (puntoFuori(punto))
            fine = 1;
        else {
            creaturaSposta(&cr,punto);
            printf("Ora %s: ",cr.nome);
            puntoPrint(stdout,punto);
            printf("\n");
        }
    }

    printf("%s ha percorso: %f\n", cr.nome,
           cr.percorsoTotale);
    return 0;
}
```

Vantaggi della modularità per composizione:

- ogni tipo di dato è un'entità a se stante, focalizzata su un compito specifico
- ogni componente di un dato è autosufficiente e riutilizzabile
- il tipo di dato di più alto livello coordina il lavoro di quelli di livello inferiore
- modifiche al tipo di dato inferiore sono localizzate, riutilizzabili e invisibili al tipo di dato superiore.

Quando e come realizzare un dato composto

Ogni dato/modulo deve occuparsi di un solo compito:

- immagazzinare e gestire dati
 - Es. il tipo `punto_t` immagazzina e opera sui punti
- coordinare i sotto-dati
 - Es. il tipo `creatura_t` coordina il flusso dei dati e fornisce servizi al `main`

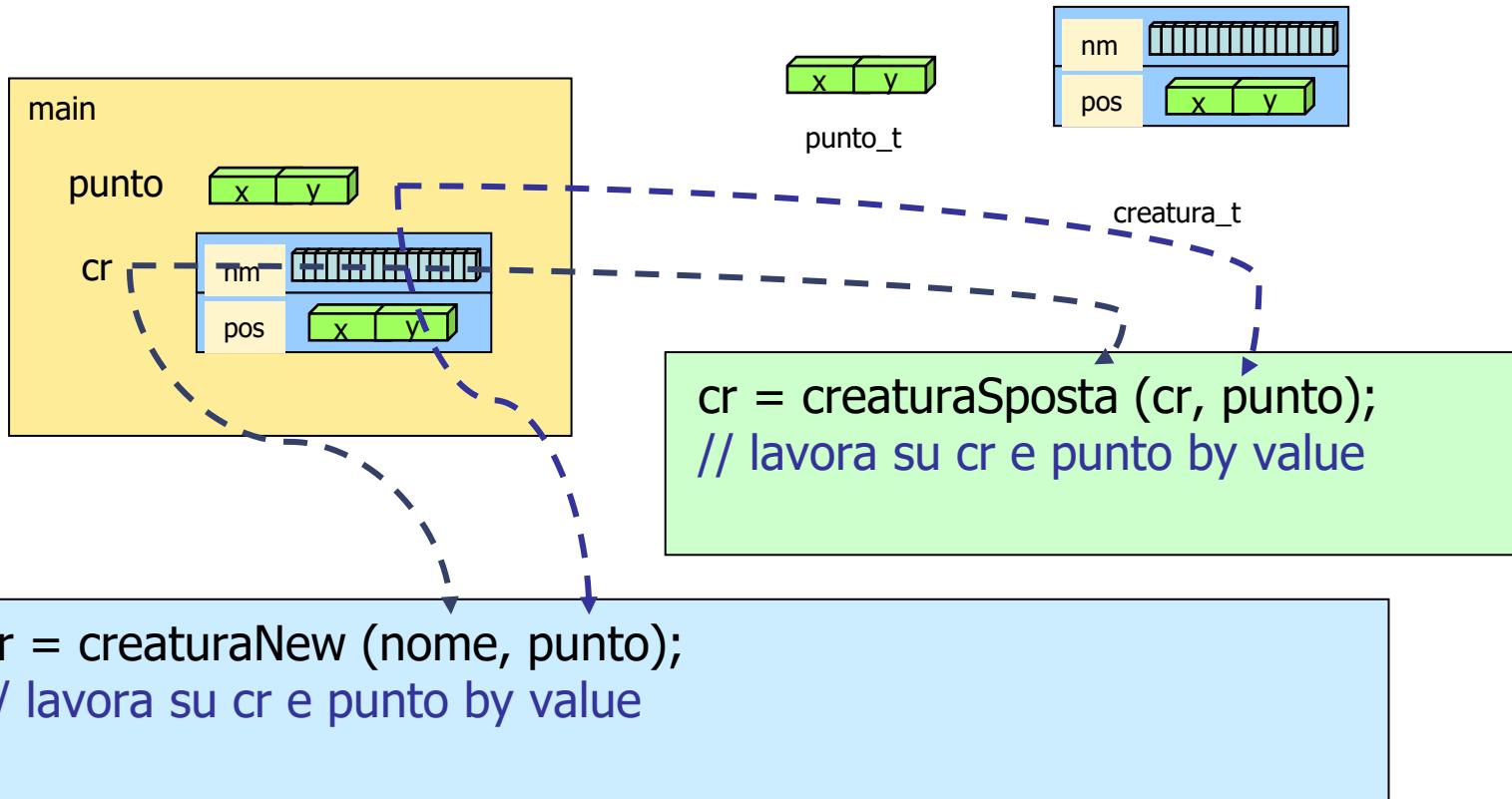
Scelte da fare:

- come scomporre e rappresentare i dati
- come suddividere i dati tra le funzioni (o assegnare le funzioni)
- quali parametri e valori di ritorno utilizzare

Possibili scelte alternative (1)

- lunghezza del percorso calcolata dal `main`, invece che da `creatura_t`:
 - con variabile `distTot` usando la funzione `puntoDist`
 - `creatura_t` perde il campo `percorsoTotale` e questo non viene più calcolato in `creaturaSposta`
- modifiche a `punto_t` e `creatura_t` mediante funzioni che ricevono la `struct` originale e ne ritornano il nuovo valore

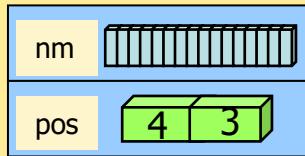
Composizione per valore



main

punto [2 5]

cr

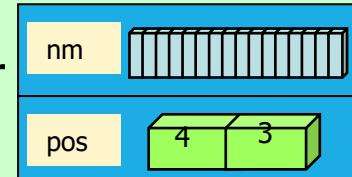


cr, punto

creaturaSposta (creature_t cr,
punto_t p) {

p [2 5]

cr



return cr;

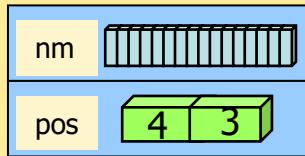
}

P copiato
cr copiato e ritornato

main

punto [2 5]

cr

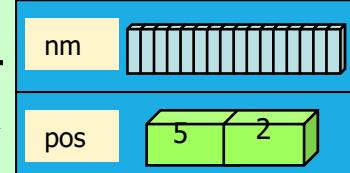


cr, punto

creaturaSposta (creatura_t cr,
punto_t p) {

p [2 5]

cr



return cr;

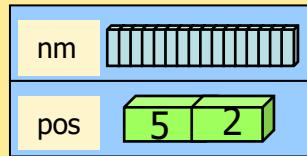
}

cr.pos = ...
Aggiorna variabile
locale cr

main

punto [2 5]

cr



cr, punto

creaturaSposta (creatura_t cr,
punto_t p) {

p [2 5]

cr

The diagram shows the variable 'cr' from the 'creatureSposta' function. It consists of two parts: a top part labeled 'nm' which is a small box containing the number '1', and a bottom part labeled 'pos' which is a green box containing the values '5' and '2' in separate cells. This represents a copy of the original 'cr' variable.

return cr;

}

cr = creatureSposta
aggiorna variabile
del main cr

Possibili scelte alternative (2)

- `puntoScan` non lavora più sul puntatore al punto, ma restituisce il nuovo valore come valore di ritorno
- `creaturaNew` ritorna una `struct` cui sono stati assegnati i valori passati come parametro
- `creaturaSposta` non modifica una `struct` esistente, ma riceve la versione precedente e restituisce il valore aggiornato

Definizioni e funzioni di creazione/manipolazione

```
typedef struct {
    int X, Y;
} punto_t;

typedef struct {
    char nome[MAXS];
    punto_t posizione;
} creatura_t;

/* funzioni di lettura, stampa e calcolo della distanza */
...

punto_t puntoScan(FILE *fp) {
    punto_t p;
    fscanf(fp, "%d %d", &p.X, &p.Y);
    return p;
}
```

```
int puntoFuori(punto_t p) {
    return (p.X<0 || p.Y<0);
}

creatura_t creaturaNew(char *nome, punto_t punto) {
    creatura_t cr;
    strcpy(cr.nome,nome);
    cr.posizione = punto;
    return cr;
}

creatura_t creaturaSposta(creatura_t cr, punto_t p) {
    cr.posizione = p;
    return cr;
}
```

main

```
int main(void) {
    char nome[MAXS]; punto_t punto; creatura_t cr;
    int fine=0; float d, distTot = 0.0;

    printf("Creatura: "); scanf("%s", nome);
    printf("Inizio: "); punto = puntoScan(stdin);
    cr = creaturaNew(nome, punto);

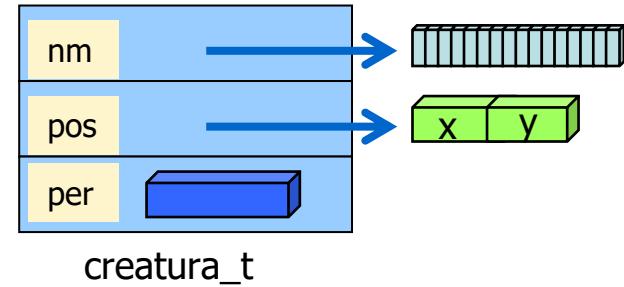
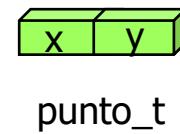
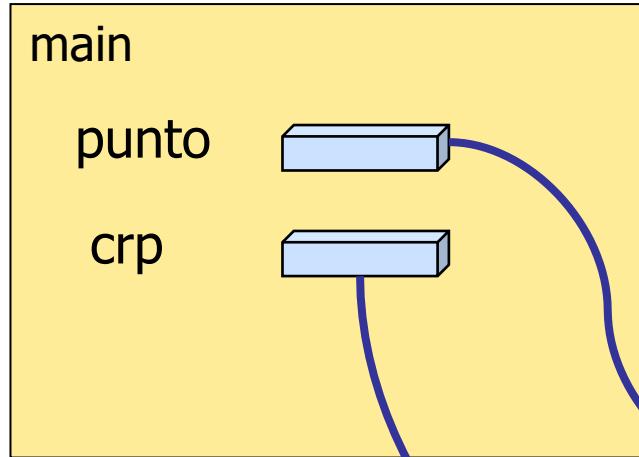
    while (!fine) {
        printf("Nuovo: "); punto = puntoScan(stdin);
        if (puntoFuori(punto))
            fine = 1;
        else {
            distTot += puntoDist(cr.posizione, punto);
            cr = creaturaSposta(cr, punto);
            printf("%s e' nel punto: ", cr.nome);
            puntoPrint(stdout, punto);
            printf("\n");
        }
    }
    printf("%s ha percorso: %f\n",
           cr.nome, distanzaTotale);
    return 0;
}
```

Composizione per riferimento

Un dato contiene un puntatore al dato interno di cui mantiene il completo possesso

Esempio:

- `creatura_t` contiene 2 puntatori
 - a stringa (per il nome)
 - a `punto_t` per il punto



Definizioni e funzioni di creazione/manipolazione

```
typedef struct { int X, Y; } punto_t;  
  
typedef struct {  
    char *nome;  
    punto_t *posizione;  
    float percorsoTotale;  
} creatura_t;  
  
punto_t *puntoCrea(void) {  
    punto_t *pp = (punto_t) malloc(punto_t);  
    return pp;  
}  
  
punto_t *puntoDuplica(punto_t *pp) {  
    punto_t *pp2 = puntoCrea();  
    *pp2 = *pp;  
    return pp2;  
}
```

```
void puntoLibera(punto_t *pp) { free(pp); }  
  
void puntoScan(FILE *fp, punto_t *pp) {  
    scanf("%d %d", &pp->X, &pp->Y);  
}  
  
void puntoPrint(FILE *fp, punto_t *pp) {  
    fprintf(fp, "(%d,%d)", pp->X, pp->Y);  
}  
  
int puntoFuori(punto_t *pp) {  
    return (pp->X<0 || pp->Y<0);  
}  
  
float puntoDist(punto_t *pp0, punto_t *pp1) {  
    int d2 = (pp1->X-pp0->X)*(pp1->X-pp0->X) +  
             (pp1->Y-pp0->Y)*(pp1->Y-pp0->Y);  
    return ((float) sqrt((double)d2));  
}
```

main

```
creatura_t *creaturaNew(char *nome, punto_t *punto) {  
    creatura_t *cp = malloc(sizeof(creatura_t));  
    cp->nome = strdup(nome);  
    cp->posizione = puntoDuplica(punto);  
    cp->percorsoTotale = 0.0; }  
  
void creaturaSposta(creatura_t *cp, punto_t *pp) {  
    puntoLibera(cp->posizione);  
    cp->posizione = puntoDuplica(pp);  
}  
  
int main(void) {  
    char nome[MAXS];  
    punto_t punto; creatura_t *crp;  
    int fine=0; float distanzaTotale = 0.0;  
  
    printf("Creatura: "); scanf("%s", nome);  
    printf("Inizio: "); puntoScan(stdin,&punto);
```

```
    crp = creaturaNew(nome,&punto);  
    while (!fine) {  
        printf("Nuovo: "); puntoScan(stdin,&punto);  
        if (puntoFuori(&punto)) fine = 1;  
        else {  
            creaturaSposta(crp,&punto);  
            printf ("%s e' nel punto: ", crp.nome);  
            puntoPrint(stdout,&punto);  
            printf("\n");  
        }  
    }  
    printf("%s ha percorso: %f\n", crp->nome,  
           crp->percorsoTotale);  
    creaturaLibera(crp);  
    return 0;  
}
```

Aggregazione

Composizione senza possesso.

Esempi:

- elenco dei dipendenti di un'azienda
 - I dipendenti esistono al di là dell'azienda
- volo aereo caratterizzato da compagnia, orario, costo, aeroporti di origine e destinazione
 - Compagnia ed aeroporti esistono al di là del volo.

Composizione vs. Aggregazione

Composizione	Aggregazione
A contiene B	A fa riferimento a B
B tipicamente incluso per valore	B tipicamente esterno, A include puntatore a B
A crea/distrugge B	A non è responsabile di creazione/distruzione di B
B può essere un riferimento, ma A lo possiede (crea/distrugge)	Oltre al puntatore, possibile riferimento a B con indice o nome

Aggregazione con puntatore

- Il dato esterno esiste al di fuori del dato che lo contiene
- Ci si riferisce tramite puntatori.

Esempio: i campi di `creatura_t` sono puntatori a nomi e punti esterni e predeterminati tra i quali l'utente può scegliere (vettori `nomi_a_scelta` e `punti_ammessi`).

Ulteriore variazione:

- *Non c'è obbligo di passaggio per tutti i punti, sullo stesso punto è lecito passare più volte.*

Definizioni e funzioni di creazione/manipolazione

```
typedef struct { int X, Y; } punto_t;  
  
typedef struct {  
    char *nome;  
    punto_t *posizione;  
    float percorsoTotale;  
} creatura_t;  
  
...  
  
void puntoScan(FILE *fp, punto_t *pp) {  
    fscanf(fp, "%d %d", &pp->X, &pp->Y);  
}  
  
void puntoPrint(FILE *fp, punto_t *pp) {  
    fprintf(fp, "(%d,%d)", pp->X, pp->Y);  
}
```

```
float puntoDist(punto_t *pp0, punto_t *pp1) {  
    int d2 = (pp1->X-pp0->X)*(pp1->X-pp0->X) +  
             (pp1->Y-pp0->Y)*(pp1->Y-pp0->Y);  
    return ((float) sqrt((double)d2));  
}  
  
void creaturaNew(creatura_t *cp, char *nome,  
                  punto_t *puntoP) {  
    cp->nome = nome;  
    cp->posizione = puntoP;  
    cp->percorsoTot = 0.0;  
}  
  
void creaturaSposta(creatura_t *cp, punto_t *pP) {  
    cp->percorsoTot+=puntoDist(cp->posizione,pP);  
    cp->posizione = pP;  
}
```

main

```
int main(void) {
    char *nome; creatura_t cr;
    int fine=0, i, np;
    char *nomi_a_scelta[5]={"Spiderman",
                           "Superman","Batman","Ironman","Hulk"};
    punto_t *punti_ammessi; float distTot = 0.0;
    printf("Nome creatura a scelta tra:\n");
    for (i=0; i<5; i++)
        printf("%d) %s\n", i+1, nomi_a_scelta[i]);
    printf("Indice (1..5) nome scelto: ");
    scanf("%d", &i); i--; // i- per riportare i in 0..4
    nome=nomi_a_scelta[i];
    printf("Quanti punti per %s?", nome);
    scanf("%d", &np);
    punti_ammessi = malloc(np*sizeof(punto_t));
```

```
for (i=0; i<np; i++) {
    printf("punto %d) ", i+1);
    puntoScan(stdin,&punti_ammessi[i]);
}
printf("Punti possibili (1..np):\n");
for (i=0; i<np; i++) {
    printf("%d) ", i+1);
    puntoPrint(stdout,&punti_ammessi[i]);
    printf("\n");
}
printf("Inizio: ");
scanf("%d", &i); i--; // i- per riportare i in 0..4
```

main

```
int main(void) {
    char *nome; creatura_t cr;
    int fine=0, i, np;
    char *nomi_a_scelta[5]={"Spiderman",
                           "Superman","Batman","Ironman","Hulk"};
    punto_t *punti_ammessi; float distTot = 0.0;
    printf("Nome creatura a scelta tra:\n");
    for (i=0; i<5; i++)
        printf("%d %s\n", i+1, nomi_a_scelta[i]);
    ...
}
```

```
creaturaNew(&cr,nome,&punti_ammessi[i]);
while (!fine) {
    printf("Nuova posizione: ");
    scanf("%d", &i); i--;
    if (i<0) fine = 1;
    else {
        creaturaSposta(&cr,&punti_ammessi[i]);
        printf("%s e' nel punto: ", cr.nome);
        puntoPrint(stdout,&punti_ammessi[i]);
        printf("\n");
    }
}
printf("%s ha percorso: %f\n", cr.nome,
       cr.percorsoTotale);
return 0;
}
```

Aggregazione con indici

- Il dato esterno esiste al di fuori del dato che lo contiene
- Il dato esterno è contenuto in un vettore
- Ci si riferisce tramite nome del vettore ed indice.

Esempio: `creatura_t` contiene una `struct` posizione i cui campi sono il vettore dei punti ammessi (`punti`) e il suo indice (`indice`).

Definizioni e funzioni di creazione/manipolazione

```
typedef struct {
    int X, Y;
} punto_t;

typedef struct {
    char *nome;
    struct { punto_t *punti; int indice; } posizione;
    float percorsoTot;
} creatura_t;

void puntoScan(FILE *fp, punto_t *pP) {
    fscanf(fp, "%d %d", &pP->X, &pP->Y);
}

void puntoPrint(FILE *fp, punto_t *pP) {
    fprintf(fp, "%d %d", pP->X, pP->Y);
}
```

```
float puntoDist(punto_t *p0P, punto_t *p1P) {
    int d2 = (p1P->X-p0P->X)*(p1P->X-p0P->X) +
              (p1P->Y-p0P->Y)*(p1P->Y-p0P->Y);
    return ((float) sqrt((double)d2));
}

void creaturaNew(creatura_t *cp, char *nome,
                  punto_t *punti, int id) {
    cp->nome = nome; cp->posizione.punti = punti;
    cp->posizione.indice = id; cp->percorsoTot = 0.0;
}

void creaturaSposta(creatura_t *cp, int id) {
    int id0 = cp->posizione.indice;
    cp->percorsoTot += puntoDist(&cp->posizione.punti[id0],
                                  &cp->posizione.punti[id]);
    cp->posizione.indice = id;
}
```

main

```
int main(void) {
    char *nome; punto_t punto; creatura_t cr;
    int fine=0, i, np;
    char *nomi_a_scelta[5]={"Spiderman",
                           "Superman","Batman","Ironman","Hulk"};
    punto_t *punti_ammessi; float distTot = 0.0;
    printf("Nome creatura a scelta tra:\n");
    for (i=0; i<5; i++)
        printf("%d %s\n", i+1, nomi_a_scelta[i]);
    ...
}
```

```
creaturaNew(&cr,nome,punti_ammessi,i);
while (!fine) {
    printf("Nuova posizione: ");
    scanf("%d", &i); i--;
    if (i<0) fine = 1;
    else {
        creaturaSposta(&cr,i);
        printf("%s e' nel punto: ", cr.nome);
        puntoPrint(stdout,&punti_ammessi[i]);
        printf("\n");
    }
}
printf("%s ha percorso: %f\n", cr.nome,
       cr.percorsoTotale);
return 0;
}
```

Strutture dati Contenitore

COLLEZIONI (DINAMICHE) DI DATI OMOGENEI

Le strutture dati contenitore

Tipo **contenitore**: involucro che contiene diversi oggetti:

- omogenei
- che si possono aggiungere o rimuovere.

Le **struct** non sono contenitori, in quanto i loro dati non sono necessariamente omogenei.

I vettori sono contenitori se:

- il contenitore ha capienza massima e il vettore è compatibile con la capienza
- il vettore è allocato/riallocato dinamicamente.

Descrizione

Esempi di tipo **contenitore**:

- vettori, liste, pile, code, tavole di simboli, alberi, grafi

Funzioni che operano su tipi contenitore:

- **creazione** di contenitore vuoto
- **inserimento** di elemento nuovo
- **cancellazione** di elemento
- **conteggio** degli elementi
- **accesso** agli elementi
- **ordinamento** degli elementi
- **distruzione** del contenitore.

La struttura involucro (wrapper)

Un **involucro (wrapper)**

- struttura di più alto livello che racchiude tutti i dati.

Una volta definito un wrapper, esso è la sola informazione necessaria a rappresentare la struttura e ad accedervi.

Esempio:

- **wrapper** per vettore dinamico di interi
int *v caratterizzato da puntatore
al primo dato e dimensione allocate
- Esempio d'uso: ordinamento

```
typedef struct {  
    int *v;  
    int n;  
} ivet_t;
```

```
void ordinavettoreConwrapper(ivet_t *w);
```

Esempio 2

- **wrapper** per lista con puntatore a **head** e **tail**:

```
typedef struct {
    link head; link tail;
} LIST;
```

- Esempio d'uso: inserimento in coda

```
void listwrapInstailFast(LISTA *l, Item val) {
    if (l->head==NULL)
        l->head = l->tail = newNode(val, NULL);
    else {
        l->tail->next = newNode(val, NULL);
        l->tail = l->tail->next;
    }
}
```

Programmi multi-file

MODULARITÀ BASATA SU IMPEMENTAZIONE (.C) E INTERFACCIA (.H)

Programmazione modulare multi-file

Al crescere della complessità dei programmi diventa difficile mantenerli su di un solo file

- la ricompilazione è onerosa
- si impedisce la collaborazione tra più programmatori ciascuno dei quali è indipendente ma coordinato
- non è facile il riuso di funzioni sviluppate separatamente.

Soluzione:

- modularità + **scomposizione su più file**

In pratica

I moduli su più file sono:

- compilati e testati individualmente
- interagiscono in maniera ben definita attraverso **interfacce**
- implementano l'**information hiding**, nascondendo i dettagli interni.

Soluzione adottata:

- file di intestazione (**header**) .h per dichiarare l'interfaccia
- file di **implementazione** .c con l'implementazione di quanto esportato e di quanto non esportato

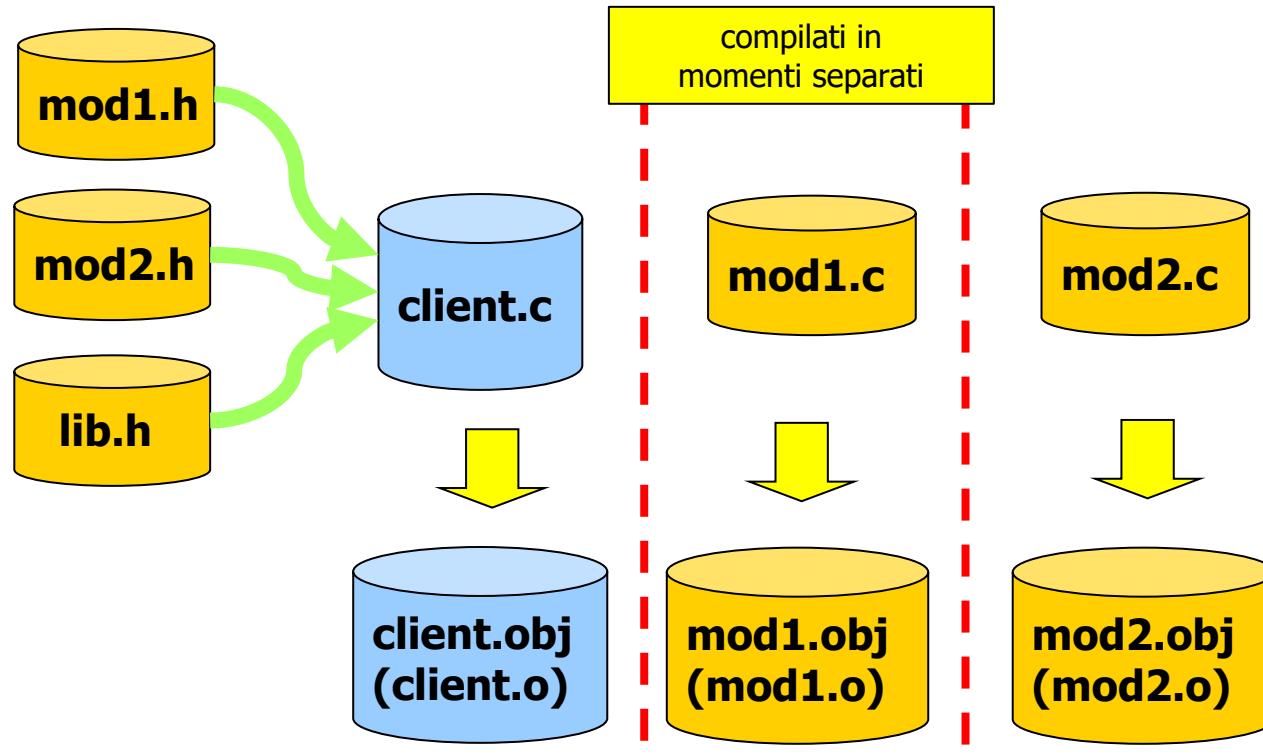
In pratica

Un modulo è utilizzabile da un programma client:

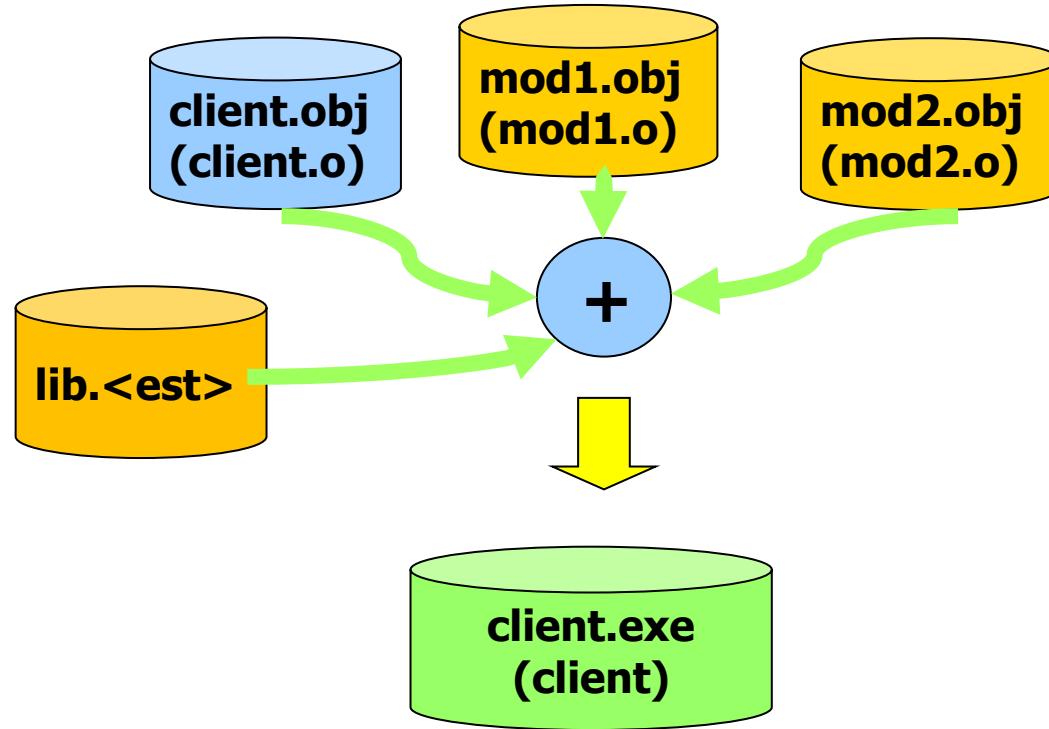
- se il client ne include l'interfaccia con una direttiva `#include <headerfile.h>`
- se l'eseguibile finale contiene sia client che modulo. La compilazione può essere separata, ma il linker combina i file oggetto di client e modulo in un unico eseguibile

Opportuno che il file `.c` del modulo includa il suo `.h` per controllo di coerenza.

Compilazione di più file



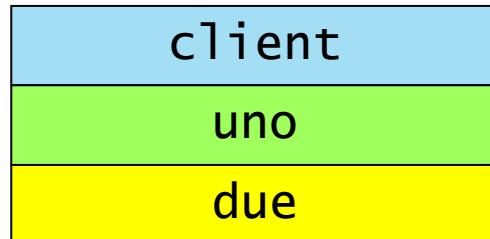
Link di più file



Architettura SW: un client e più moduli

Situazione 1: il client vede un modulo, che vede il secondo modulo

- `client.c`
 - usa il modulo uno, che a sua volta usa il modulo due
 - Non usa DIRETTAMENTE il modulo due
- `client.c` include `uno.h`, `uno.c` include `due.h`



Esempio

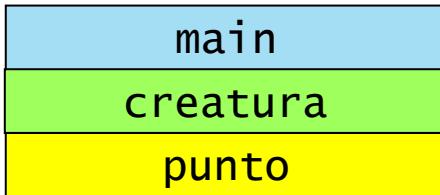
```
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);

/* punto.c */
#include "punto.h"
...
void puntoScan(FILE *fp, punto_t *pP) {
    scanf("%d %d", &pP->X, &pP->Y); }
float puntoDist(punto_t p0,punto_t p1) {
    ...
}
```

```
/* creatura.h */
typedef struct {
    char nome[MAXS];
    punto_t posizione; float percorsoTot;
} creatura_t;
void creaturaSposta(creatura_t *cp);

/* creatura.c */
#include "punto.h"
#include "creatura.h"

void creaturaSposta(creatura_t *cp){
    punto_t p;
    puntoScan(stdin,&punto);
    cp->percorsoTotale +=
        puntoDist(cp->posizione,p);
    cp->posizione = p; }
```

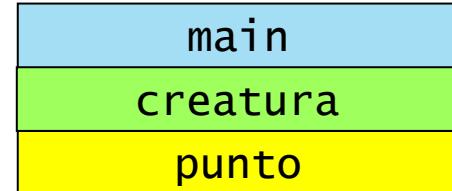


```
/* main.c */
#include "creatura.h"
int main(void) {
    creatura_t cr;
    ...
    creaturaNew(&cr,nome);
    while (!fine) {
        printf("Nuovo: ");
        creaturaSposta(&cr);
    }
    ...
}
```

Esempio

```
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);

/* punto.c */
#include "punto.h"
...
void puntoScan(FILE *fp, punto_t *pP) {
    scanf("%d %d", &pP->X, &pP->Y); }
float puntoDist(punto_t p0,punto_t p1) {
    ...
}
```



```
/* creatura.h */
typedef struct {
    char nome[MAXS];
    punto_t posizione; float percorsoTot;
} creatura_t;
void creaturaSposta(creatura_t *cp);

/* creatura.c */
#include "punto.h"
#include "creatura.h"
void creaturaSposta(creatura_t *cp) {
    punto_t p;
    puntoScan(stdin,&p);
    cp->percorsoTotal += puntoDist(cp->posizione, p);
    cp->posizione = p; }
```

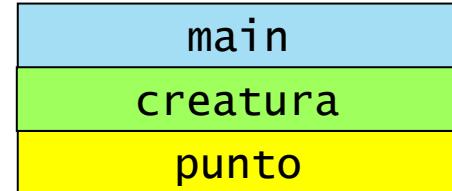
```
/* main.c */
#include "creatura.h"
int main(void) {
    creatura_t cr;
    ...
    creaturaNew(&cr,nome);
    while (!fine) {
        printf("Nuovo: ");
        creaturaSposta(&cr);
    }
    ...
}
```

Semplice ma...
Il main non «vede» punto
Non può avere variabili punto_t
Non puo' chiamare funzioni punto...

Esempio

```
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);

/* punto.c */
#include "punto.h"
...
void puntoScan(FILE *fp, punto_t *pP) {
    scanf("%d %d", &pP->X, &pP->Y); }
float puntoDist(punto_t p0,punto_t p1) {
    ...
}
```



```
/* creatura.h */
typedef struct {
    char nome[MAXS];
    punto_t posizione; float percorsoTot;
} creatura_t;
void creaturaSposta(creatura_t *cp);

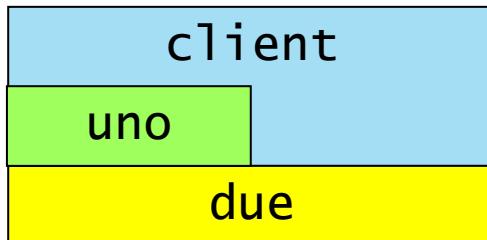
/* creatura.c */
#include "punto.h"
#include "creatura.h"
void creaturaSposta(creatura_t *cp) {
    punto_t p;
    puntoScan(stdin,&punto);
    cp->percorsoTotale +=
        puntoDist(cp->posizione,p);
    cp->posizione = p; }
```

```
/* main.c */
#include "creatura.h"
int main(void) {
    creatura_t cr;
    ...
    creaturaNew(&cr,nome);
    while (!fine) {
        printf("Nuovo: ");
        creaturaSposta(&cr);
    }
    ...
}
```

Vanno «spostati» pezzi
in *creatura.c*:
Es.
Acquisizione punti da *stdin*
(occorre aggiungere funzioni)

Situazione 2: il client vede entrambi i moduli

- `client.c` usa il modulo uno, che a sua volta usa il modulo due
- `client.c` usa DIRETTAMENTE anche il modulo due



- Due Alternative possibili:
 - `client` include `uno.h` e `due.h`
 - `client` include `uno.h` che include `due.h`

Esempio

```
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);
```

```
/* punto.c */
#include "punto.h"
...
void puntoScan(FILE *fp, punto_t *pP) {
    scanf("%d %d", &pP->X, &pP->Y); }
float puntoDist(punto_t p0,punto_t p1) {
    ...
}
```



```
/* creatura.h */
typedef struct {
    char nome[MAXS];
    punto_t posizione; float percorsoTot;
} creatura_t;

void creaturaSposta(creatura_t *cp, punto_t p);

/* creatura.c */
#include "punto.h"
#include "creatura.h"
...
void creaturaSposta(creatura_t *cp, punto_t p){
    cp->percorsoTotale +=
        puntoDist(cp->posizione,p);
    cp->posizione = p;
}
```

```
/* main.c */
#include "punto.h"
#include "creatura.h"

int main(void) {
    punto_t punto; creatura_t cr;
    ...
    creaturaNew(&cr,nome,&punto);
    while (!fine) {
        printf("Nuovo: ");
        puntoScan(stdin,&punto);
        creaturaSposta(&cr,punto);
    }
    ...
}
```

Esempio

```
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);

/* punto.c */
#include "punto.h"
...
void puntoScan(FILE *fp, punto_t *pP) {
    scanf("%d %d", &pP->X, &pP->Y); }
float puntoDist(punto_t p0,punto_t p1) {
    ...
}
```



```
/* creatura.h */
typedef struct {
    char nome[MAXS];
    punto_t posizione; float percorsoTot;
} creatura_t;
void creaturaSposta(creatura_t *cp, punto_t p);

/* creatura.c */
#include "punto.h"
#include "creatura.h"
...
void creaturaSposta(creatura_t *cp, punto_t p){
    cp->percorsoTot += puntoDist(cp->posizione, p);
    cp->posizione = p;
}
```

```
/* main.c */
#include "punto.h"
#include "creatura.h"

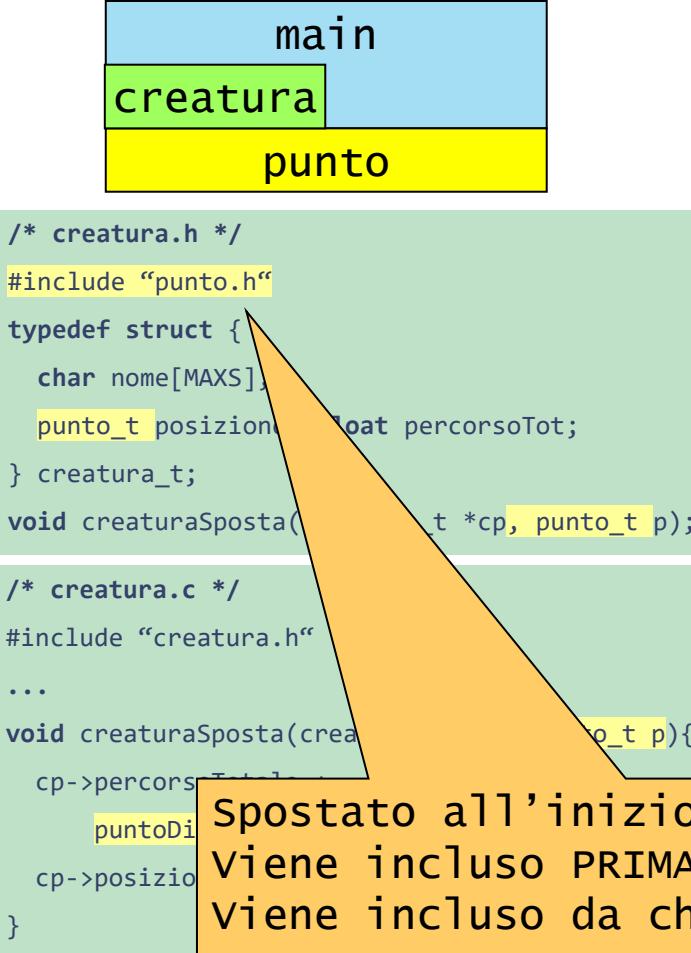
int main(void) {
    punto_t punto; creatura_t cr;
    ...
    creaturaInizializza(&cr,nome,&punto);
    while (...) {
        printf("Nome: %s\n", cr.nome);
        puntoScan(fp,&punto);
        creaturaSposta(&cr,punto);
    }
    ...
}
```

Va incluso PRIMA di creatura.h
Va incluso da creatura.c e main.c

Esempio

```
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);

/* punto.c */
#include "punto.h"
...
void puntoScan(FILE *fp, punto_t *pP) {
    scanf("%d %d", &pP->X, &pP->Y);
}
float puntoDist(punto_t p0,punto_t p1) {
    ...
}
```



```
/* main.c */
#include "creatura.h"

int main(void) {
    punto_t punto; creatura_t cr;
    ...
    creaturaNew(&cr,nome,&punto);
    while (!fine) {
        printf("Nuovo: ");
        puntoScan(stdin,&punto);
        creaturaSposta(&cr,punto);
    }
    ...
}
```

Spostato all'inizio di creatura.h
Viene incluso PRIMA di creatura.h
Viene incluso da chi include creatura.h

Esempio

```
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);

/* punto.c */
#include "punto.h"
...
void puntoScan(FILE *fp, punto_t *pP) {
    scanf("%d %d", &pP->X, &pP->Y); }
float puntoDist(punto_t p0,punto_t p1) {
    ...
}
```



```
/* creatura.h */
#include "punto.h"
typedef struct {
    char nome[MAXS];
    punto_t posizione; float percorsoTot;
} creatura_t;
void creaturaSposta(creatura_t *cp, punto_t p)
```

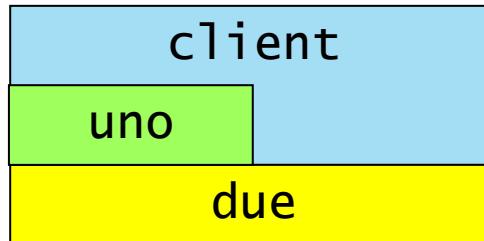
```
/* creatura.c */
#include "creatura.h"
...
void creaturaSposta(creatura_t *cp, punto_t p)
```

```
/* main.c */
#include "punto.h"
#include "creatura.h"
int main(void) {
    punto_t punto; creatura_t cr;
    ...
    cr.nome = "Nuovo";
    creaturaNew(&cr,nome,&punto);
    if(!fine) {
        printf("Nuovo: ");
        puntoScan(stdin,&punto);
        creaturaSposta(&cr,punto);
    }
}
```

Doppia inclusione (diretta + indiretta)
Cosa succede se il main, senza leggere
creatura.h, include punto.h?

Situazione 2

- `client.c` usa il modulo uno, che a sua volta usa il modulo due
- `client.c` usa DIRETTAMENTE anche il modulo due



- Due Alternative possibili:
 - `client` include `uno.h` e `due.h`
 - `client` include `uno.h` che include `due.h`

RISCHIO DI INCLUSIONI MULTIPLE
⇒ COMPILAZIONE CONDIZIONALE

Compilazione condizionale

Direttive `#if` e `#endif`

La compilazione di quanto compare tra le direttive `#if` e `#endif` è condizionata alla condizione che compare come argomento. Utile, ad esempio, per istruzioni che servono solo quando si fa debug.

```
...  
#define DBG 1 //0 per disabilitare  
...  
#if DBG  
// istruzioni da compilare  
// (ed eseguire) in debug  
printf ("serve solo per debug");  
#endif
```

Compilazione condizionale

Direttive `#if` e `#endif`

La compilazione di quanto compare tra le direttive `#if` e `#endif` è condizionata alla condizione che compare come argomento. Utile, ad esempio, per istruzioni che servono solo quando si fa debug.

```
...  
#define DBG 0 //0 per disabilitare  
...  
#if DBG  
// istruzioni da compilare  
// (ed eseguire) in debug  
printf ("serve solo per debug");  
#endif
```

Compilazione condizionale

Direttive `#if` e `#endif`

La compilazione di quanto compare tra le direttive `#if` e `#endif` è condizionata alla condizione che compare come argomento. Utile, ad esempio, per istruzioni che servono solo quando si fa debug.

```
...  
#define DBG 0           Così disabilita:  
...                   è come se fosse commentato  
#if DBG  
// istruzioni da compilare  
// (ed eseguire) in debug  
printf ("serve solo per debug");  
#endif
```

Compilazione condizionale

Direttive `#ifdef` e `#ifndef`

La compilazione è condizionata non dall'argomento, bensì dall'essere definita o meno la macro:

```
...
#define DBG //non interessa il valore
...
#ifndef DBG
// istruzioni da compilare
// (ed eseguire) in debug
printf ("serve solo per debug");
#endif
```

Compilazione condizionale

Direttive `#if` e `#endif`

La compilazione di quanto compare tra le direttive `#if` e `#endif` è condizionata alla condizione che compare come argomento. Utile, ad esempio, per istruzioni che servono solo quando si fa debug.

```
//#define DBG  
...  
#ifdef DBG  
// istruzioni da compilare  
// (ed eseguire) in debug  
printf ("serve solo per debug");  
#endif
```

Così disabilita:
è come se fosse commentato

Compilazione condizionale

Direttive #if e #endif

La compilazione di quanto compare tra le direttive `#if` e `#endif` è condizionata alla condizione che compare come argomento. Utile, ad esempio, per istruzioni che servono solo quando si fa debug.

```
...  
#define DBG  
#undef DBG  
  
...  
#ifdef DBG  
// istruzioni da compilare  
// (ed eseguire) in debug  
printf ("serve solo per debug");  
#endif
```

Oppure così:
è come se fosse commentato

Protezione da inclusione multipla

- Per evitare inclusioni multiple si usa `#ifndef` nel file .h. La macro `_<nomefile>` che funge da argomento gioca il ruolo di una variabile globale:
- Il file può essere incluso più volte in sequenza, ma solo la prima viene «vista»

```
// header1.h
#ifndef _HEADER1
#define _HEADER1
// istruzioni di header1.h
...
#endif
```

Protezione da inclusione multipla

- Il file può essere incluso più volte in sequenza, ma solo la prima viene «vista»

```
// header1.h - prima inclusione
#ifndef _HEADER1
#define _HEADER1
// istruzioni di header1.h
...
#endif
// header1.h - seconda inclusione
#ifndef _HEADER1
#define _HEADER1
// istruzioni di header1.h
...
#endif
```

Protezione da inclusione multipla

- Il file può essere incluso più volte in sequenza, ma solo la prima viene «vista»

```
// header1.h - prima inclusione
#ifndef _HEADER1
#define _HEADER1
// istruzioni di header1.h
...
#endif
// header1.h - seconda inclusione
#ifndef _HEADER1
#define _HEADER1
// istruzioni di header1.h
...
#endif
```

Questa parte è disabilitata:
è come se non ci fosse

Voli e aeroporti

UN ESEMPIO DI MODULARITÀ

Una struttura dati composta: voli

Dati due file contenenti un elenco di aeroporti e un elenco di voli

- costruire una struttura dati contenente le informazioni di aeroporti e voli.

I file (nomi ricevuti come argomenti al main) contengono nella prima riga il numero totale di aeroporti/voli. I formati sono (C indica codice):

<C aeroporto> <nome citta>, <nome aeroporto>

<C aeroporto p> <C aeroporto A> <C volo> <oraP> <oraA>

28

AOI Ancona, Marche

BRI Bari, Palestro

MXP Milano, Malpensa

...

FCO Roma, Fiumicino

TPS Trapani, Birgi

TRN Torino, S. Pertini

42

AOI BGY FR4705 17:45 19:25

AOI BGY FR4887 19:40 21:20

AOI FLR VY1505 19:35 20:50

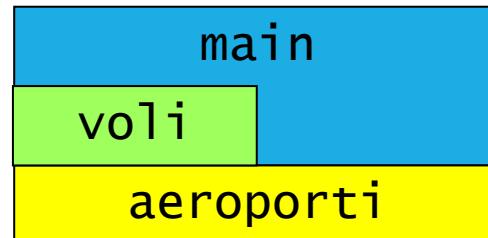
CAG AOI FR8727 10:25 11:50

TRN FCO AZ1430 19:05 20:15

...

Moduli

- Main: client sia di voli che di aeroporti
- Voli: client di aeroporti
- Aeroporti



Strutture dati

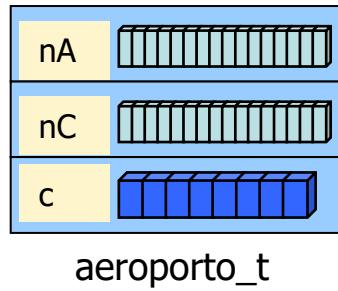
- Basate su wrapper (struct involucro) per
 - Voli
 - Aeroporti
- Dati elementari per volo e aeroporto
 - Composti (A)
 - Aggregati (B)

Composizione (A)

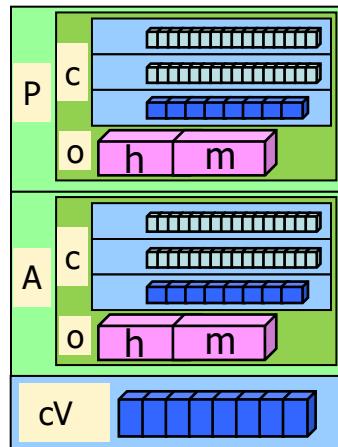
```
typedef struct {  
    char nomeAeroporto[M1];  
    char nomeCitta[M1];  
    char codice[M2];  
} aeroporto_t;
```

```
typedef struct {  
    int h, m;  
} orario_t;
```

```
typedef struct {  
    struct {  
        aeroporto_t citta;  
        orario_t ora;  
    } partenza, arrivo;  
    char codicevolo[M2];  
} volo_t;
```



`aeroporto_t`



`volo_t`

Aggregati o composti per riferimento (B)

```
typedef struct {  
    char *nomeAeroporto;  
    char *nomeCitta;  
    char codice[M2];  
} aeroporto_t;
```

```
typedef struct {  
    int h, m;  
} orario_t;
```

```
typedef struct {  
    struct {  
        aeroporto_t *citta;  
        orario_t ora;  
    } partenza, arrivo;  
    char codicevolo[M2];  
} volo_t;
```

Aggregati o composti per riferimento (B)

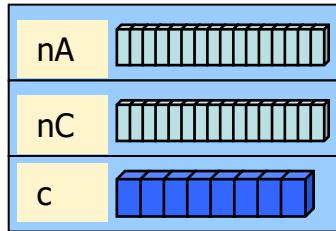
```
typedef struct {  
    char *nomeAeroporto;  
    char *nomeCitta;  
    char codice[M2];  
} aeroporto_t;
```

```
typedef struct {  
    int h, m;  
} orario_t;
```

```
typedef struct {  
    struct {  
        aeroporto_t *citta;  
        orario_t ora;  
    } partenza, arrivo;  
    char codicevolo[M2];  
} volo_t;
```

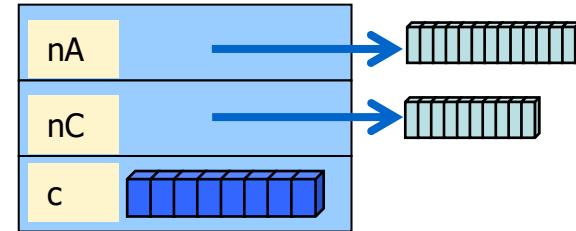
Puntatori a stringhe "esterne"
alla struct
composto o aggregato:
Dipende dal «possesso»

A

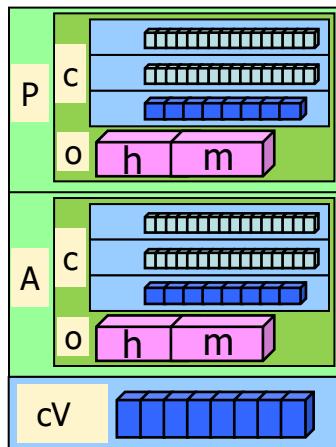


`aeroporto_t`

B



`aeroporto_t`



`volo_t`

Aggregati o composti per riferimento (B)

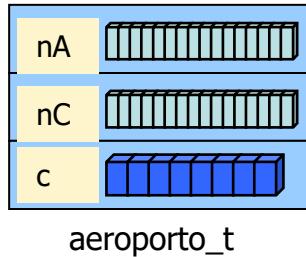
```
typedef struct {  
    char *nomeAeroporto;  
    char *nomeCitta;  
    char codice[M2];  
} aeroporto_t;
```

```
typedef struct {  
    int h, m;  
} orario_t;
```

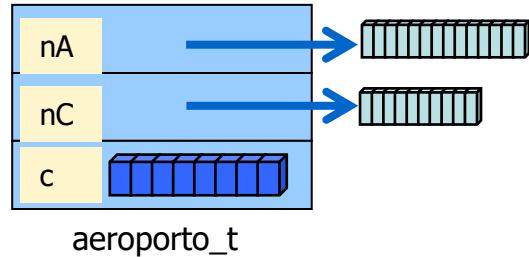
Puntatori a struct "esterna"

```
typedef struct {  
    struct {  
        aeroporto_t *citta;  
        orario_t ora;  
    } partenza, arrivo;  
    char codicevolo[M2];  
} volo_t;
```

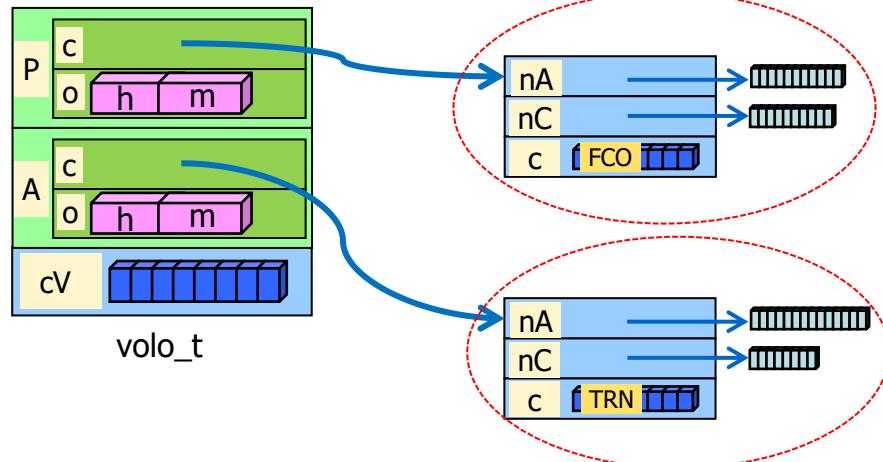
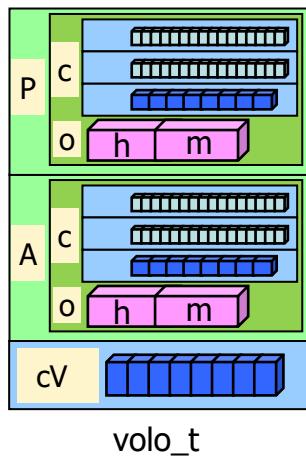
A



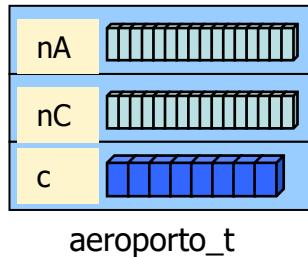
B



Struct allocate
individualmente

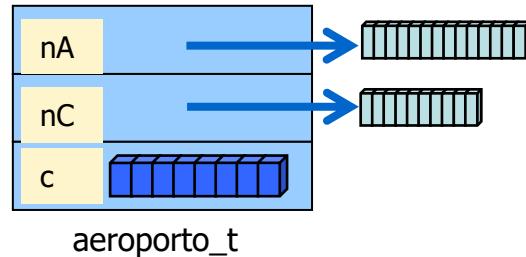


A



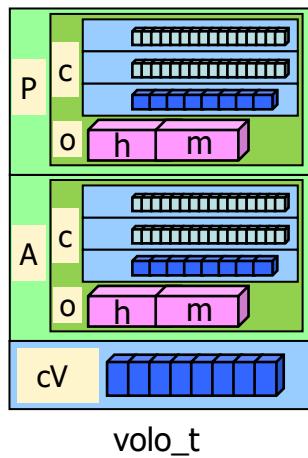
aeroporto_t

B

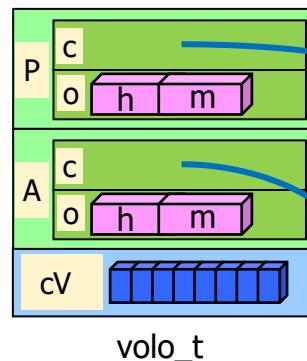


aeroporto_t

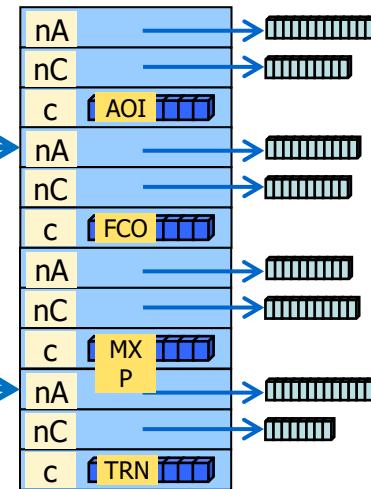
Struct in Vettore
(aggregato)



volo_t



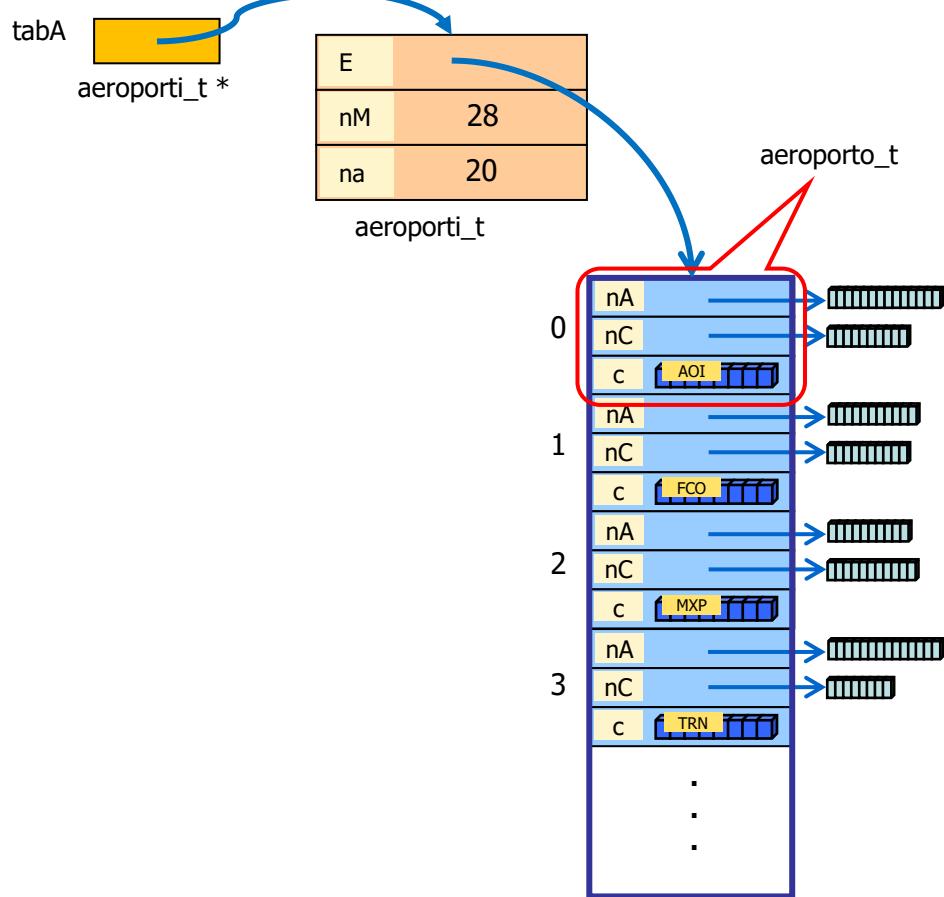
volo_t



Collezioni di aeroporti e voli

Basate su wrapper, struct che racchiude tutte le informazioni su volo/aeroporto

```
typedef struct {
    aeroporto_t *elenco;
    int nmax, na;
} aeroporti_t;
```

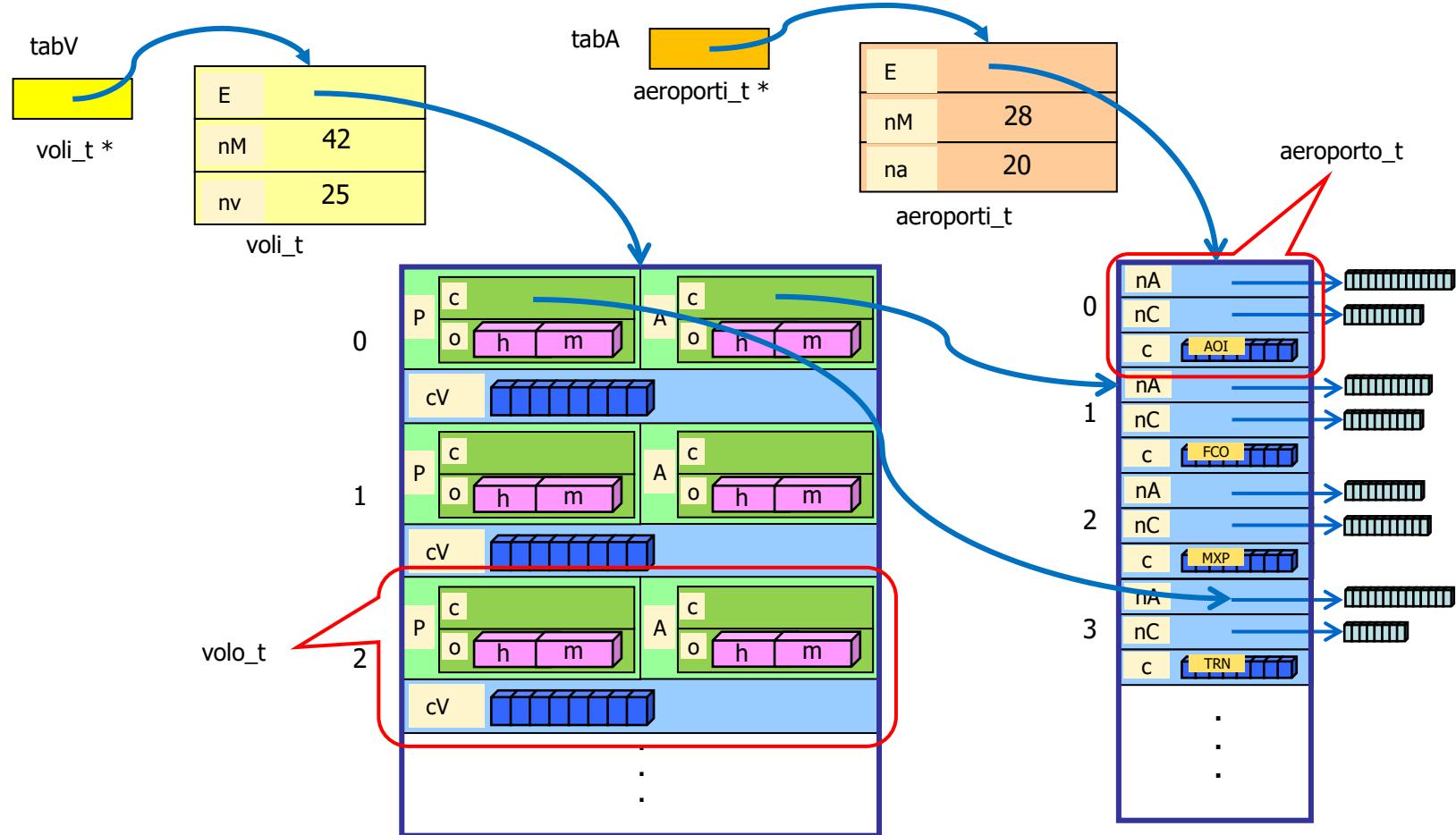


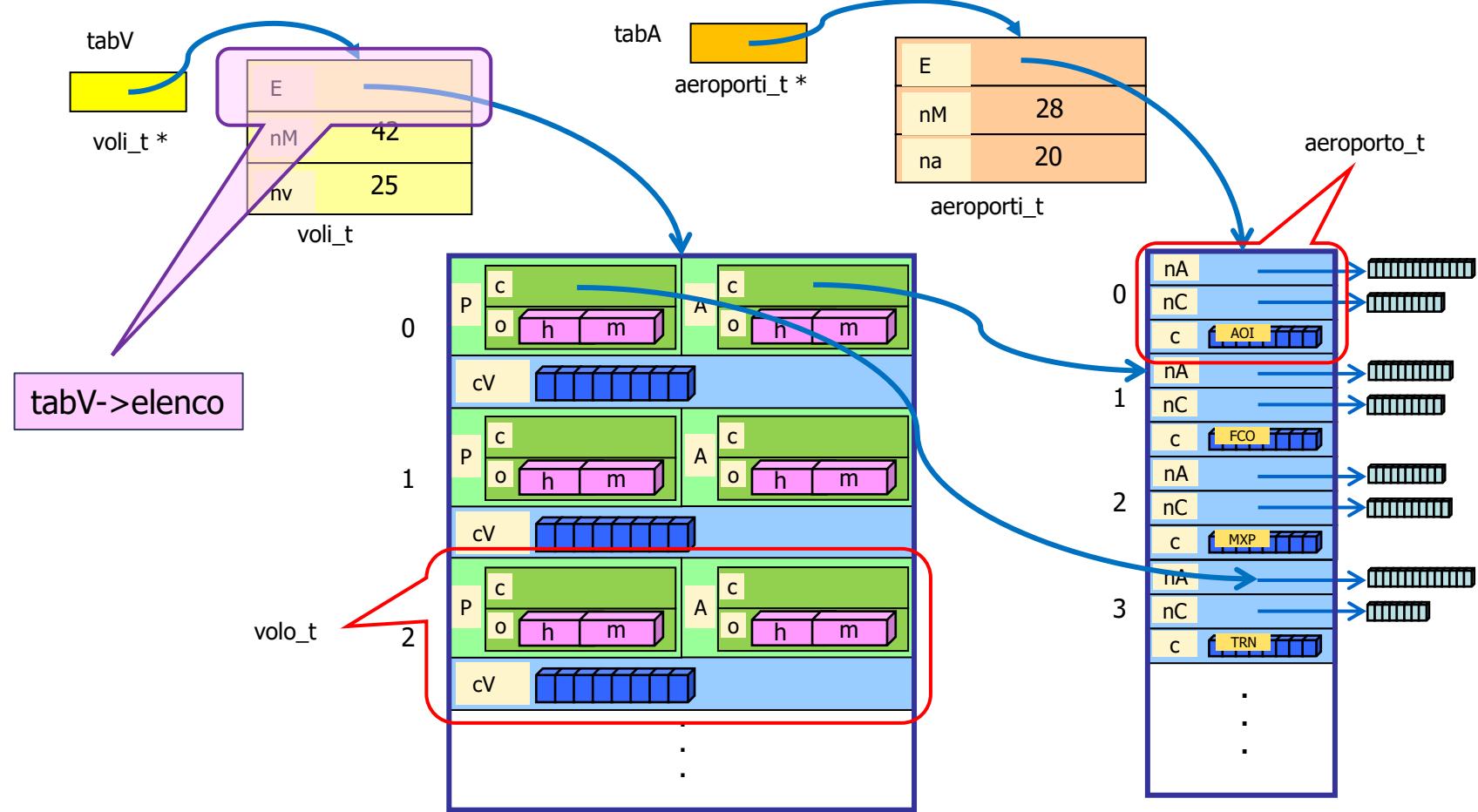
```
typedef struct {  
    char *nomeAeroporto;  
    char *nomeCitta;  
    char codice[M2];  
} aeroporto_t;
```

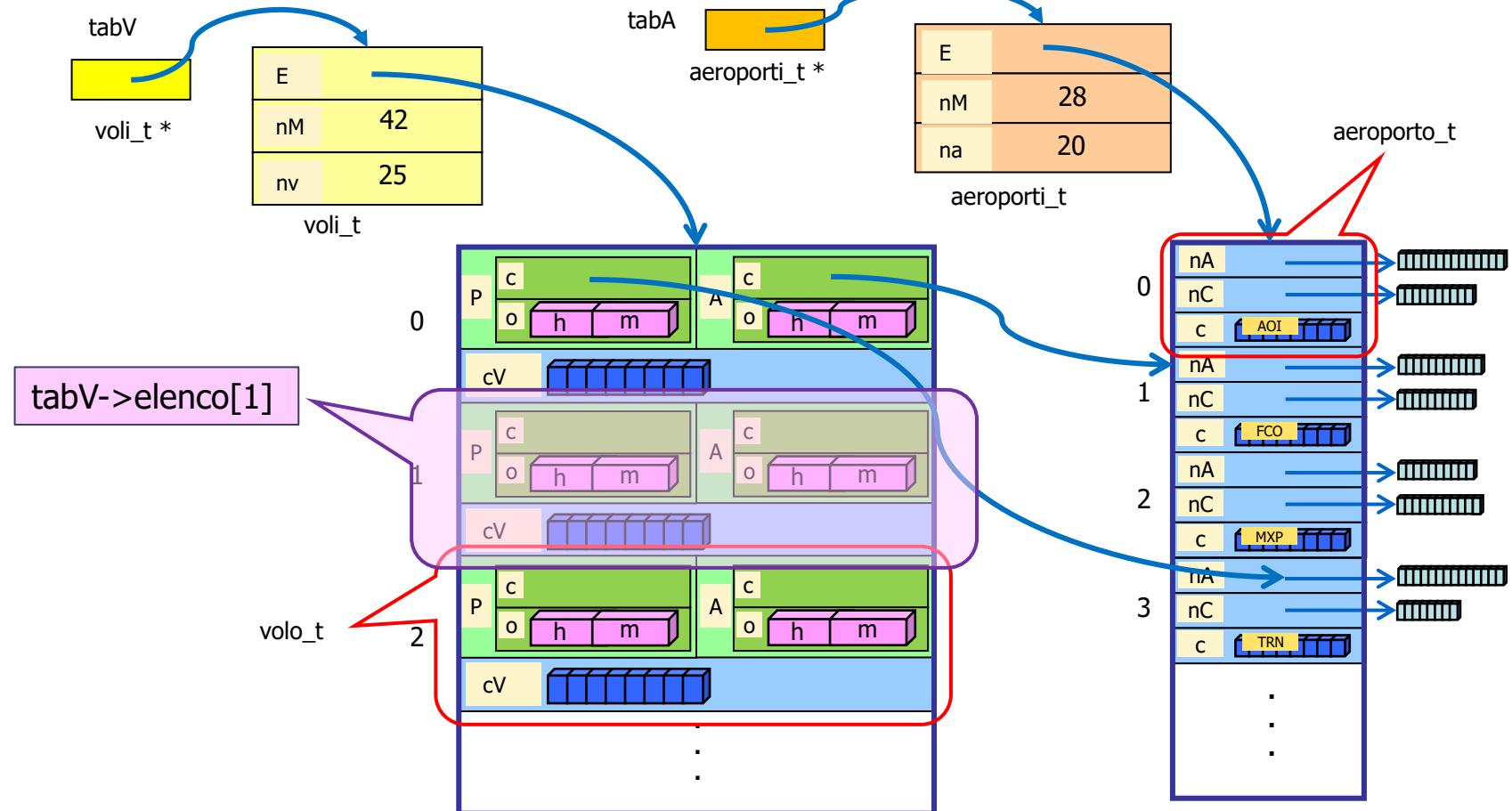
```
typedef struct {  
    aeroporto_t *elenco;  
    int na, nmax;  
} aeroporti_t;
```

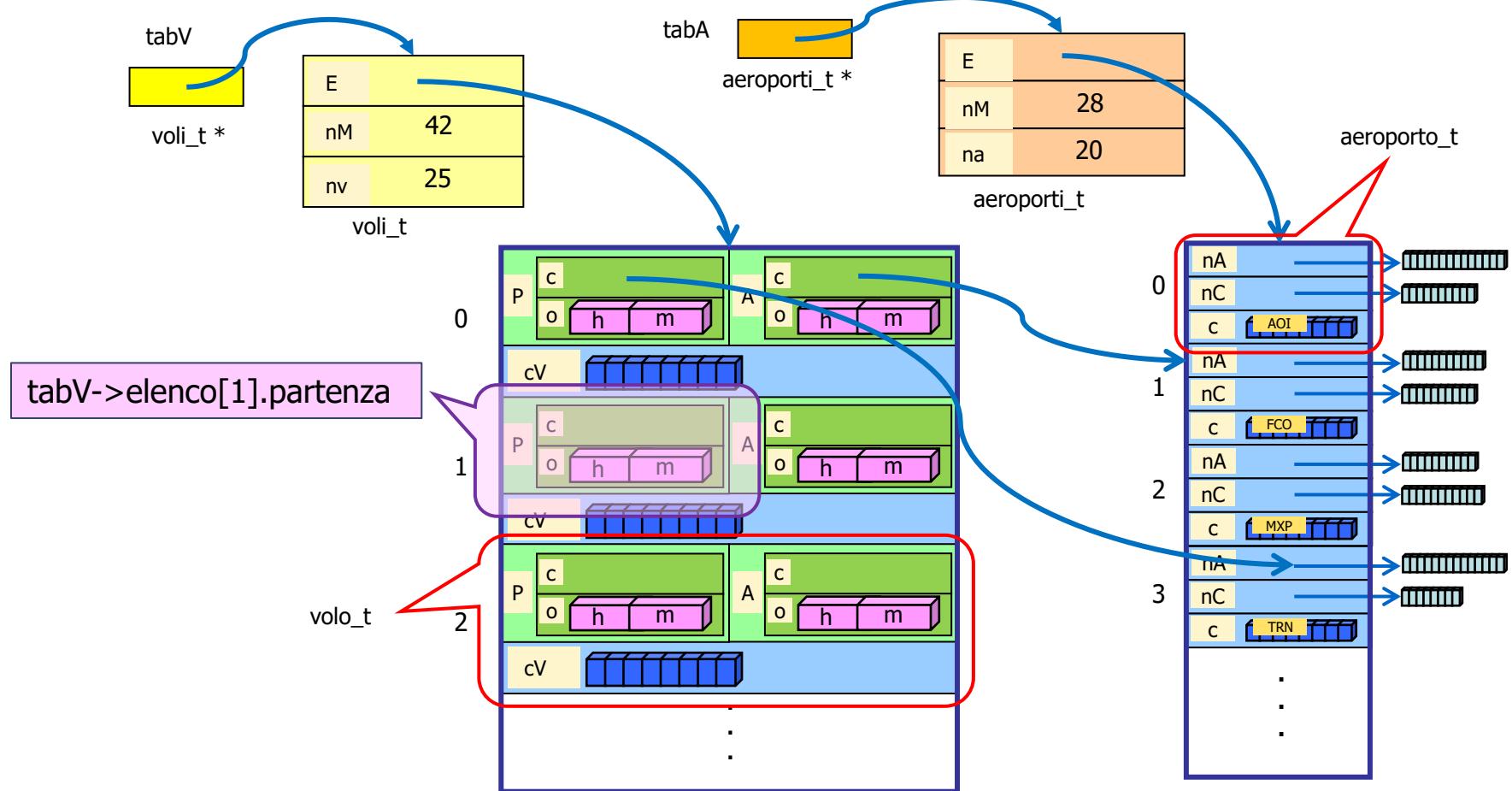
```
typedef struct {  
    struct {  
        aeroporto_t *citta;  
        orario_t ora;  
    } partenza, arrivo;  
} volo_t;
```

```
typedef struct {  
    volo_t *elenco;  
    int nv, nmax;  
} voli_t;
```





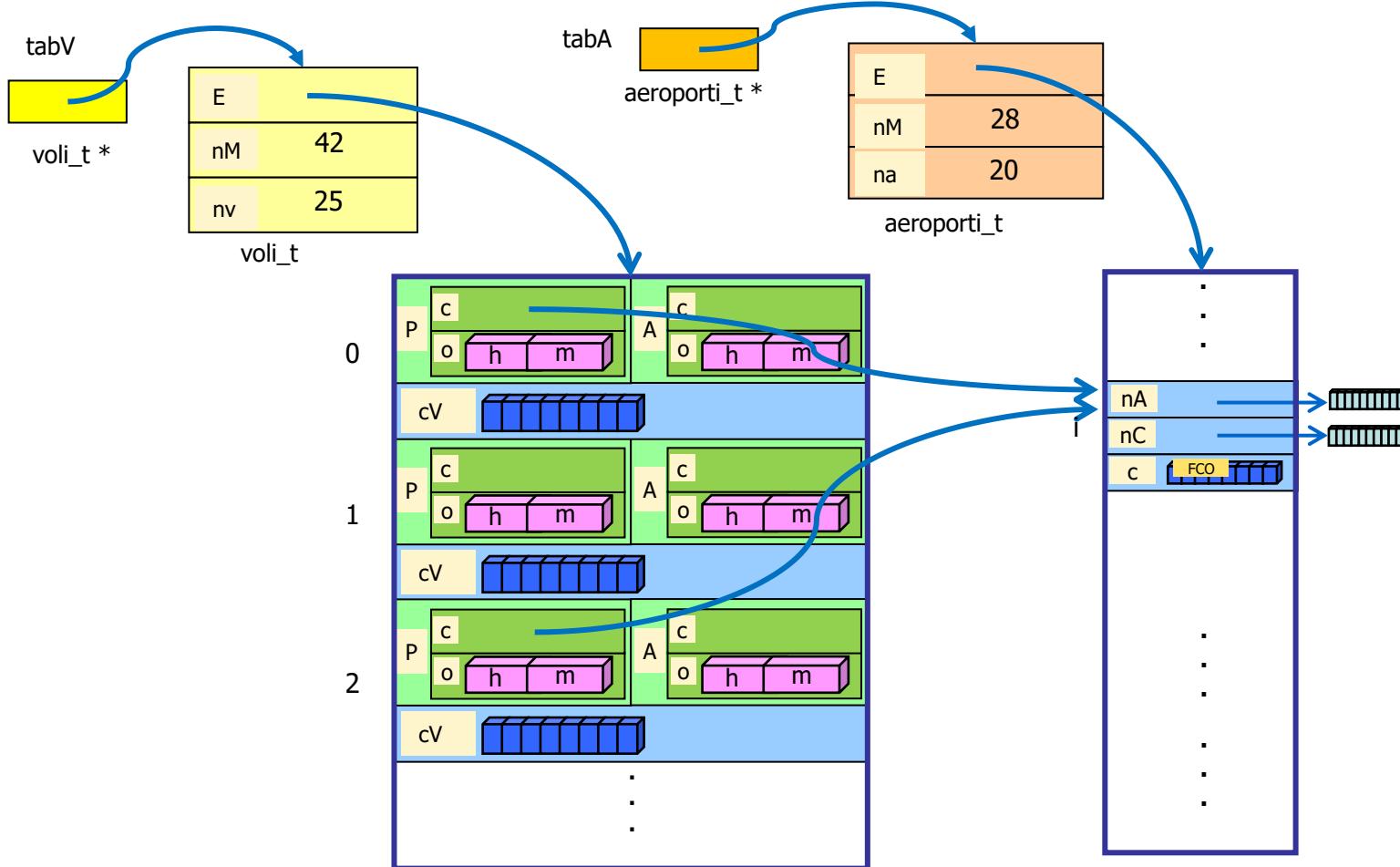




- Vedere le soluzioni proposte:
- voli.c, aeroporti.c, voli.h, aeroporti.h, main.c
- V1: versione base (vettori)
- V2: elenchi voli e aeroporti realizzati con liste
- V3: vettori e riferimenti mediante indici (invece che puntatori)

Voli: versione base

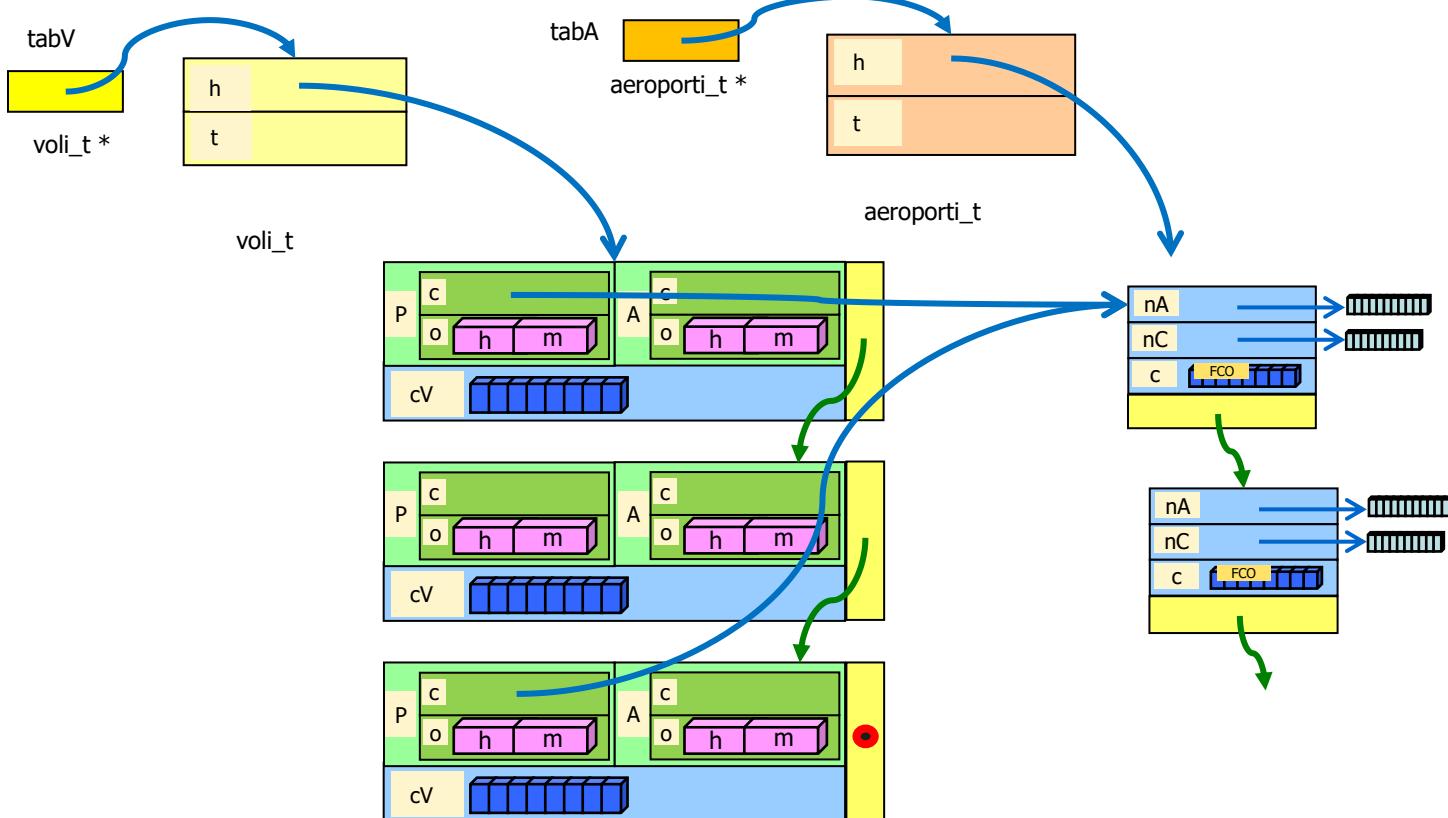
- Modulo aeroporti:
 - aeroporto_t: tipo composto (con riferimenti a nomi)
 - aeroporti_t: wrapper di collezione di aeroporti, realizzata come vettore
- Modulo voli:
 - volo_t: tipo aggregato (i riferimenti ad aeroporti sono esterni)
 - voli_t: wrapper di collezione di voli, realizzata come vettore



Voli: versione con liste

- Modulo aeroporti:
 - aeroporto_t: tipo composto (con riferimenti a nomi)
 - aeroporti_t: wrapper di collezione di aeroporti, realizzata come lista
- Modulo voli:
 - volo_t: tipo aggregato (i riferimenti ad aeroporti sono esterni)
 - voli_t: wrapper di collezione di voli, realizzata come lista

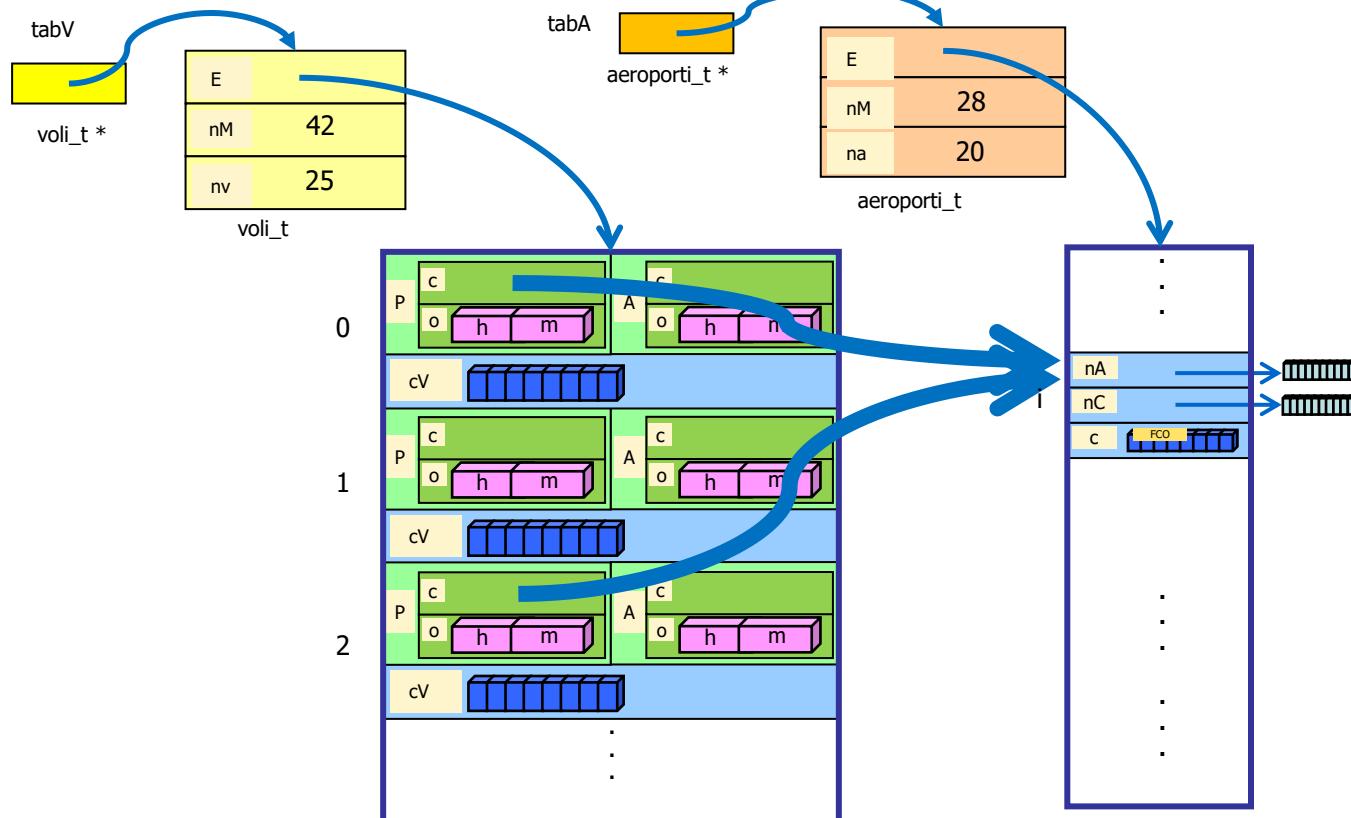
Tabelle aeroporti e voli basate su liste concatenate



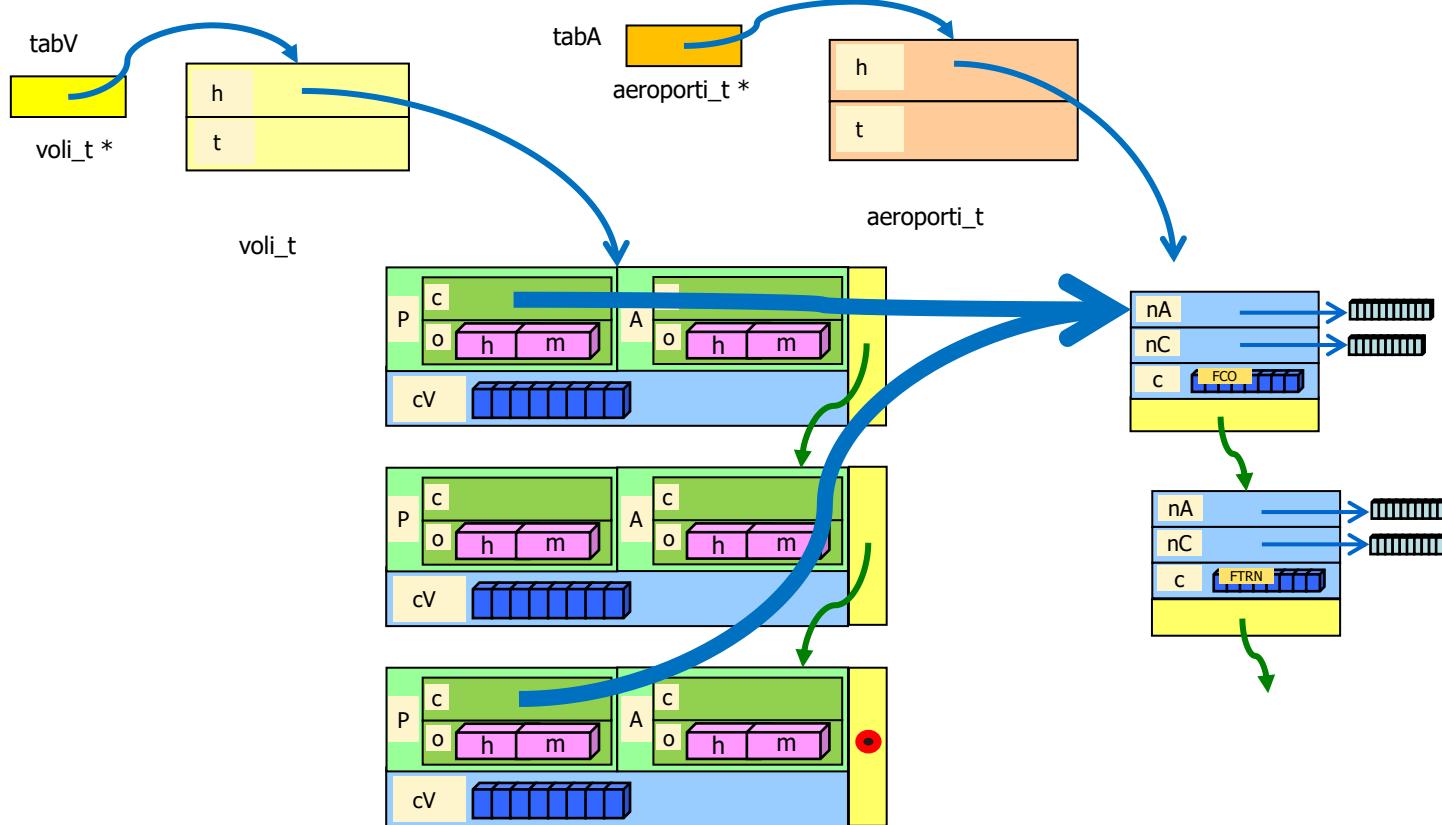
Voli: versione con indici

- Modulo aeroporti:
 - aeroporto_t: tipo composto (con riferimenti a nomi)
 - aeroporti_t: wrapper di collezione di aeroporti, realizzata come vettore
- Modulo voli:
 - volo_t: tipo aggregato (i riferimenti ad aeroporti sono degli indici)
 - voli_t: wrapper di collezione di voli, realizzata come vettore

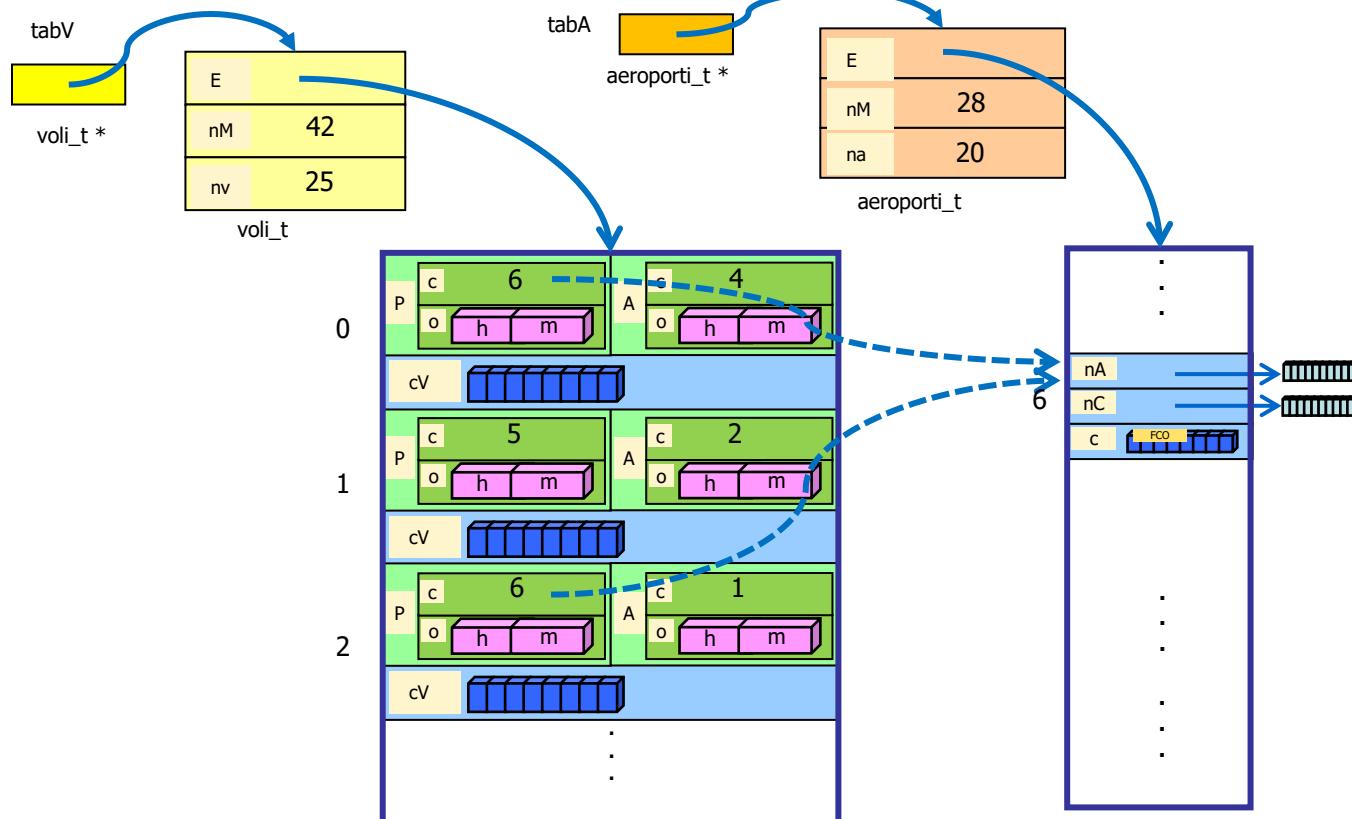
Riferimenti con puntatori (collezioni con vettori)



Riferimenti con puntatori (collezioni con liste)



Riferimenti tra tabelle con indici (necessari vettori)



Riferimenti tra tabelle con indici (necessari vettori)

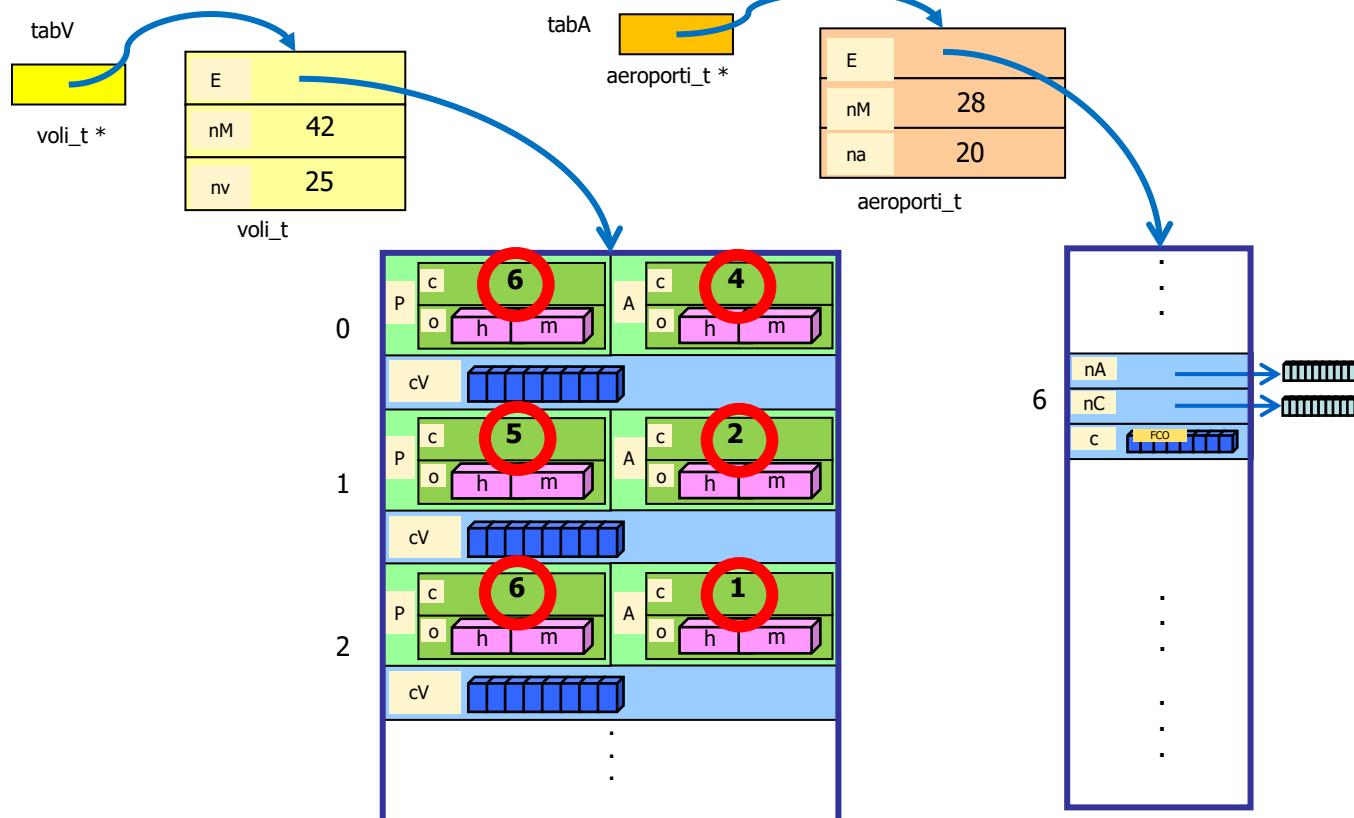
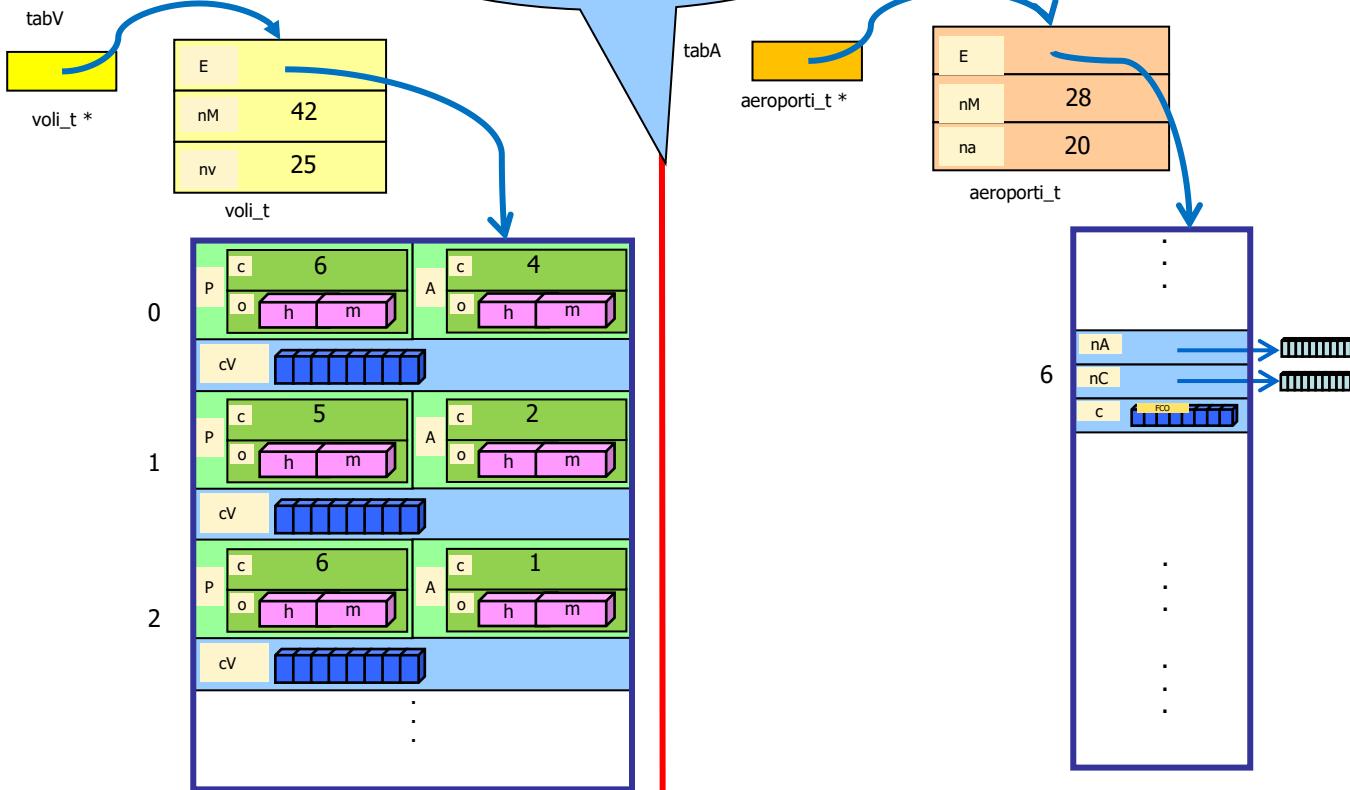
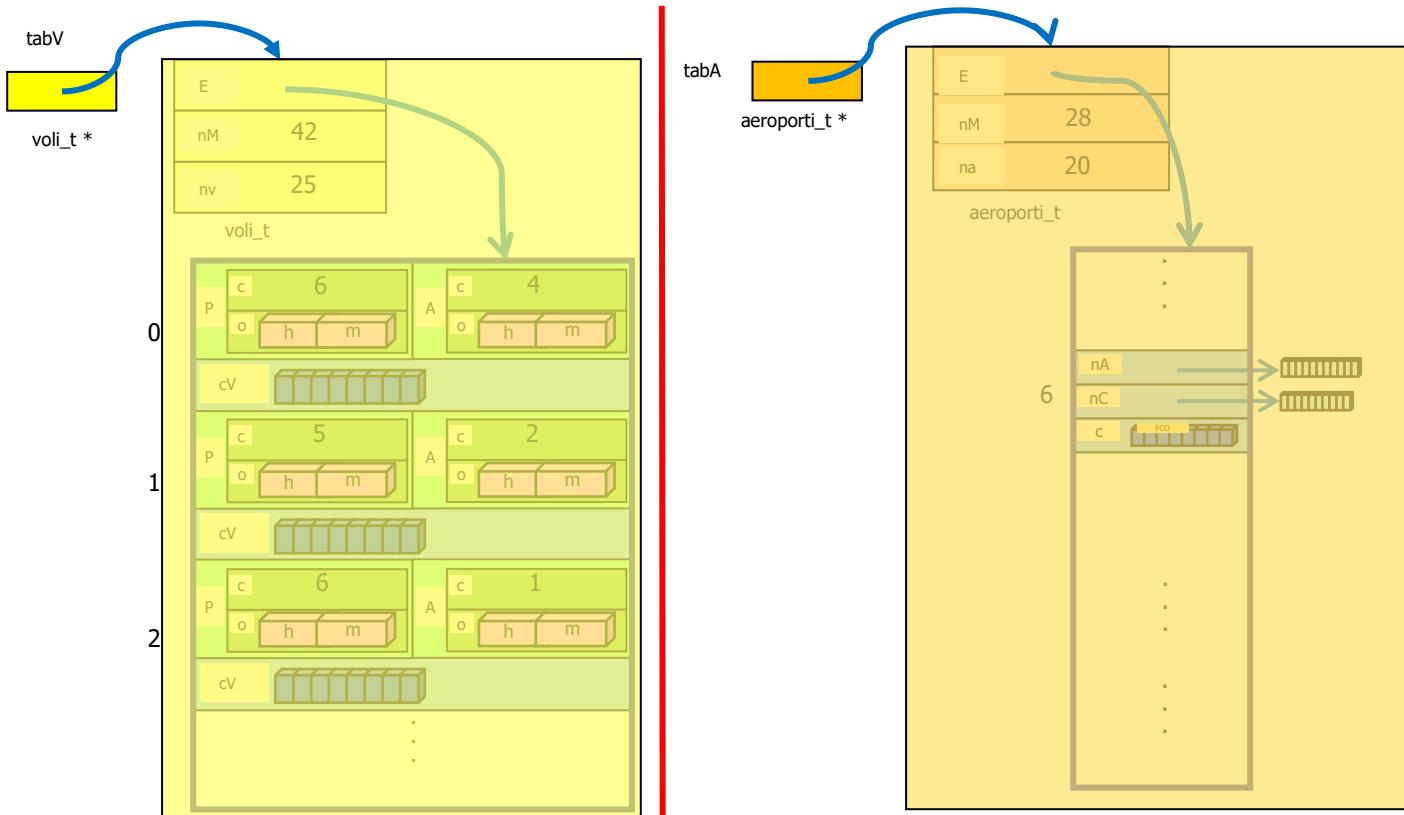


Tabelle completamente separate



Dettagli interni nascosti (puntatori...)





Capitolo 6: I Tipi di Dato Astratto

PUNTATORI E STRUTTURE DATI DINAMICHE:
ALLOCAZIONE DELLA MEMORIA E
MODULARITÀ IN LINGUAGGIO C



ADT per composti

TIPO ITEM, COMPOSTO PER VALORE O RIFERIMENTO

Progettare la struttura dati di un programma

Scegliere una adeguata struttura dati per:

- **codificare** le informazioni del problema proposto (in input, risultati intermedi e finali)
- consentire la manipolazione delle informazioni (le **operazioni**)

Le informazioni in forma utilizzabile da un sistema di elaborazione si dicono **dati** e sono memorizzati in **strutture dati**:

- interne (memoria centrale)
- esterne (memoria di massa)
- statiche (dimensione decisa in stesura del programma e fissa nel tempo)
- dinamiche (dimensione decisa in fase di utilizzo e variabile nel tempo)

Tipi di dato: cosa sono

Definiscono organizzazione e manipolazioni di dati in termini di:

- insieme di valori (es. numeri interi)
- collezione di operazioni sui valori (algoritmi), realizzati da funzioni

Classificazione

- base (standard): forniti dal linguaggio
- **definiti dall'utente**, mediante definizioni di tipo e/o funzioni
- **tipi di dati astratti**: netta separazione tra definizione e implementazione

Tipi di dato standard

Tipi scalari (numeri e caratteri):

- `int` (`signed`, `unsigned`, `long`, `short`)
- `float` (`double`, `long double`)
- `char`

Tipi strutturati (composti/aggregati)

- `array` (vettori/matrici: campi omogenei)
- `struct` (campi eterogenei)

Tipi di dato creati dall'utente

Valori:

- **typedef** permette di introdurre un nuovo nome per un tipo (da ricondurre a un tipo base, scalare o composto/aggregato)

Operazioni

- una **funzione** permette di definire una nuova operazione su argomenti e/o dato ritornato.

ADT: Tipi di Dato Astratto

- **Scopo**
 - livello di astrazione sui dati tale da mascherare completamente l'implementazione rispetto all'utilizzo
- **definizione**
 - tipo di dato (valori + operazioni) accessibile **SOLO** attraverso un'**interfaccia**.
 - utilizzatore = **client**
 - specifica del tipo di dato = **implementazione**

Creazione di ADT

- ❑ Il C non ha un meccanismo semplice ed automatico di creazione di ADT
- ❑ L'ADT è realizzato come modulo con una coppia di file interfaccia/implementazione
- ❑ Enfasi su come nascondere i dettagli dell'implementazione al client.

Quasi ADT

- **Interfaccia**
 - definizione del nuovo tipo con `typedef`
 - raramente si appoggia su tipi base, in generale è un tipo composto, aggregato o contenitore (`struct` wrapper)
 - Prototipi delle funzioni
- **Implementazione**
 - Il client include il file header, quindi vede i dettagli interni del dato e/o del wrapper

Esempio: ADT per numeri complessi

- nuovo tipo **Complex**
 - struct con campi per parte reale e coefficiente dell'immaginario
 - funzione di prodotto tra 2 numeri complessi.

Il client che include **complex.h** vede i dettagli della **struct** (ma non li usa)

complex.h

```
typedef struct {  
    float Re; float Im;  
} Complex;
```

```
Complex prod(Complex c1, Complex c2);
```

controllo di inclusione
multipla d'ora in poi
omesso

main.c

```
#include "complex.h"
```

```
int main (void) {  
    Complex a, b, c;  
    ...  
    c = prod(a,b);  
    ...  
}
```

complex.c

```
#include "complex.h"  
  
Complex prod(Complex c1, Complex c2) {  
    Complex c;  
    c.Re = c1.Re * c2.Re - c1.Im * c2.Im;  
    c.Im = c1.Re * c2.Im + c2.Re * c1.Im;  
  
    return c;  
}
```

ADT di I classe

Per impedire al client di vedere i dettagli della struct:

- il tipo di dato viene dichiarato nel file `.h` di interfaccia come *struttura incompleta*, o come puntatore a **struct** incompleta, non viene quindi definita la **struct** composta, aggregato o wrapper
- la **struct** viene invece completamente definita nel file `.c`
- il client utilizza unicamente puntatori alla struttura incompleta
- Il puntatore è opaco e si dice **handle**.

ADT per numeri complessi

- Interfaccia ad ADT per numeri complessi : `complex.h`:
 - nuovo tipo `Complex` come **puntatore** a `struct complex_s`
 - prototipi della funzione di prodotto tra 2 numeri complessi e delle funzioni di creazione e distruzione
- Implementazione: `complex.c`:
 - definizione completa del tipo `struct complex_s`
 - della funzione di prodotto tra 2 numeri complessi e delle funzioni di creazione e distruzione
- Il client che include `complex.h` NON vede i dettagli della `struct`.

complex.h

```
typedef struct complex_s *Complex;  
  
Complex crea(void);  
void distruggi(Complex c);  
Complex prod(Complex c1, Complex c2);
```

complex.c

```
#include "complex.h"  
  
struct complex_s { float Re; float Im; };  
  
Complex crea(void) {  
    Complex c = malloc(sizeof *c);  
    return c; }  
void distruggi(Complex c) {  
    free(c); }  
Complex prod(Complex c1, Complex c2) {  
    Complex c = crea();  
    c->Re = c1->Re * c2->Re - c1->Im * c2->Im;  
    c->Im = c1->Re * c2->Im + c2->Re * c1->Im;  
    return c;  
}
```

main.c

```
#include "complex.h"  
  
int main (void) {  
    Complex a, b, c;  
    a = crea();  
    b = crea();  
    ...  
    c = prod(a,b);  
    ...  
    distruggi(a);  
    distruggi(b);  
    distruggi(c);  
}
```

Quasi ADT o ADT di I classe?

Per i casi semplici di dati composti o aggregati, che non prevedano
allocazione dinamica, il quasi ADT:

- è un ragionevole compromesso
 - non nasconde completamente i dettagli interni
 - **non richiede allocazione dinamica**
- costituisce una soluzione più semplice e pratica.

L'ADT Item

Tipo di dato generico per dato unico o record che include un campo chiave (composto o aggregato: dipende dai casi).

Esempi:

- numero
- stringa
- dati su una persona
- numero complesso
- punto di piano o spazio
- ...

Vantaggi dell'ADT

- enfasi sull'algoritmo e non sui dati
- prelude al polimorfismo
- Tuttavia NON si tratta di un tipo generico, ma di una soluzione che concentra eventuali modifiche all'interno dell'ADT

Sono accettabili soluzioni

- quasi ADT
 - con tipo visibile al client (che può tuttavia ignorare la visibilità), non necessariamente dinamico
- ADT di 1 classe
 - dato nascosto, ma allocazione dinamica e puntatori

Tipo 1

- Semplice scalare, chiave coincidente

```
typedef int Item;  
typedef int Key;
```

- Nessun problema di proprietà, in quanto non c'è allocazione dinamica

Tipo 2

- Vettore dinamico di caratteri chiave coincidente

```
typedef char *Item;  
typedef char *Key;
```

- Item e chiave sono puntatori a carattere
- La chiave punta al dato

Tipo 3: composto per valore

- **struct** con vettore di caratteri sovradimensionato staticamente e intero (composizione per valore). La chiave è il vettore di caratteri.

```
typedef struct item {  
    char name[MAXC];  
    int num;  
} Item;  
typedef char *key;
```

- Essendo la stringa statica, è interna alla struct
- La chiave punta al (a parte del) dato

Tipo 4: composto per riferimento

- **struct** con vettore di caratteri allocato dinamicamente e intero.
La stringa dinamica è proprietà dell'ADT (composizione per riferimento). La chiave è il vettore di caratteri.

```
typedef struct item {  
    char *name;  
    int num;  
} Item;  
typedef char *Key;
```

- Essendo la stringa dinamica, è esterna alla **struct**
- La chiave punta al (a parte del) dato

Scelte

- quando (la chiave) è un puntatore, la chiave punta al dato o a parte del dato (non si genera un duplicato).
 - Non è l'unica scelta possibile: ci sono programmi e/o funzioni in cui la chiave è esterna al dato (un'aggiunta)
- funzioni di interfaccia indipendenti dallo specifico `Item` per quanto possibile

Versione quasi ADT: definizione di Item e Key

item.h

```
1 typedef int Item;
typedef int Key;
...
```

item.h

```
2 typedef char *Item;
typedef char *Key;
...
```

item.h

```
3 typedef struct item {
    char name[MAXC];
    int num;
} Item;
typedef char *Key;
...
```

item.h

```
4 typedef struct item {
    char *name;
    int num;
} Item;
typedef char *Key;
...
```

Funzioni di interfaccia indipendenti da Item

item.h

```
/* definizione di Item e Key */

int KEYcompare(Key k1, Key k2);
Key KEYscan();

Item ITEMscan();
void ITEMshow(Item val);
int ITEMless(Item val1, Item val2);
int ITEMgreater(Item val1, Item val2);
int ITEMcheckvoid(Item val);
Item ITEMsetvoid();
```

Funzione di interfaccia dipendente da Item

item.h

```
/* caso 1 e 2 */
```

```
Key KEYget(Item val);
```

struct passata per valore

Funzione di interfaccia dipendente da Item:

item.h

```
/* caso 3 e 4 */
```

```
Key KEYget(Item *pval);
```

la chiave è un riferimento a un campo di Item.
Se la struct fosse passata per valore, il
puntatore sarebbe un riferimento alla copia locale
della stringa, deallocato all'uscita della funzione

Implementazione: (1) Semplice scalare, chiave coincidente

item.c

```
Key KEYget(Item val) {
    return (val);
}
int KEYcompare (Key k1, Key k2);
    return (k1-k2);
}
Item ITEMscan() {
    Item val;
    scanf("%d", &val);
    return val;
}
void ITEMshow(Item v) {
    printf("%d", val);
}
int ITEMless(Item val1, Item val2) {
    return (KEYget(val1)<KEYget(val2));
}
```

direttive #include
d'ora in poi omesse

Implementazione:

(2) Vettore dinamico di caratteri chiave coincidente

item.c

```
static char buf[MAXC];  
  
Key KEYget(Item val) {  
    return (val);  
}  
int KEYcompare (Key k1, Key k2) {  
    return (strcmp(k1,k2));  
}  
Item ITEMscan() {  
    scanf("%s",buf);  
    return strdup(buf);  
}  
void ITEMshow(Item val) {  
    printf("%s", val);  
}  
int ITEMless(Item val1, Item val2) {  
    return (strcmp(KEYget(val1),KEYget(val2))<0);  
}
```

vettore statico
sovradimensionato
per acquisire le stringhe

Implementazione:

(3) composizione per valore, chiave campo di struct

item.c

```
Key KEYget(Item *pval) {  
    return (pval->name);  
}  
int KEYcompare (key k1, key k2) {  
    return (strcmp(k1,k2));  
}  
Item ITEMscan() {  
    Item val;  
    scanf("%s %d", val.name, &(val.num));  
    return val;  
}  
void ITEMshow(Item val) {  
    printf("%s %d", val.name, val.num);  
}  
int ITEMless(Item val1, Item val2) {  
    return (strcmp(KEYget(&val1),KEYget(&val2))<0);  
}
```

struct con

- vettore di caratteri
- intero

La chiave è il vettore di caratteri

Implementazione:

(4) composizione per riferimento, chiave campo di struct

item.c

```
static char buf[MAXC];

Key KEYget(Item *pval) {
    return (pval->name);}
int KEYcompare (Key k1, Key k2) {
    return (strcmp(k1,k2));}
Item ITEMscan() {
    Item val;
    scanf("%s %d", buf, &(val.num));
    val.name = strdup(buf);
    return val;
}
void ITEMshow(Item val) {
    printf("%s %d", val.name, val.num);
}
int ITEMless(Item val1, Item val2) {
    return (strcmp(KEYget(&val1),KEYget(&val2))<0);
}
```

struct con

- vettore di caratteri dinamico
- intero

La chiave è il vettore di caratteri

De-allocare o no?

Caso critico: il client legge 2 dati di tipo `Item`, li elabora e poi li distrugge:

```
a = ITEMscan(); b = ITEMscan();
// elabora a e b
ITEMfree(a); ITEMfree(b);
```

La de-allocazione ha senso se c'è stata allocazione, quindi solo nei casi 2 e 4, non nei casi 1 e 3. Due casi: il client

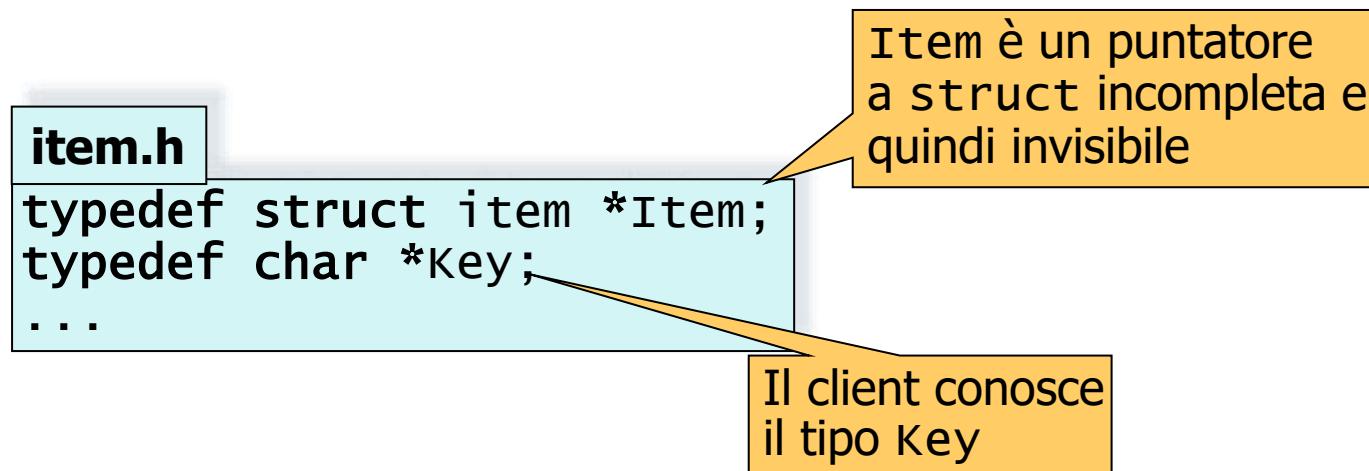
- rinuncia a de-allocare (oppure chiama funzioni di de-allocazione fittizie, che non fanno nulla)
- tratta diversamente i casi con allocazione e senza, diventando dipendente da `Item`.

Conclusione

- accettabili le soluzioni 1 e 3
- accettabile la soluzione 2 anche se disuniforme, purché il client sia responsabile
- sconsigliata la soluzione 4:
 - coi quasi-ADT meglio non usare troppo allocazione dinamica
 - (Oppure) Se si vuole allocazione dinamica, meglio non usare i quasi-ADT

Versione ADT di I classe

- L'ADT di prima classe ha senso per dati «complessi», quindi per le tipologie 3 e 4 di Item basati su struct
- Nelle tipologie 1 e 2 la chiave è talmente semplice che una soluzione a quasi ADT è accettabile



Funzioni di interfaccia indipendenti da Item

item.h

```
Key KEYget(Item val);
Key KEYscan();
int KEYcompare(Key k1, Key k2);

Item ITEMnew();
void ITEMfree(Item val);
Item ITEMscan();
void ITEMshow(Item val);
int ITEMless(Item val1, Item val2);
int ITEMgreater(Item val1, Item val2);
int ITEMcheckvoid(Item val);
Item ITEMsetvoid();
```

Implementazione

item.c

```
3 struct item {  
    char name[MAXC];  
    int num;  
};
```

item.c

```
4 static char buf[MAXC];  
struct item {  
    char *name;  
    int num;  
};
```

Funzioni comuni alle tipologie 3 e 4

item.c

```
Key KEYget(Item val) {
    return (val->name);
}
int KEYcompare (Key k1, Key k2) {
    return (strcmp(k1, k2));
}
void ITEMshow(Item val) {
    printf("%s %d", val->name, val->num);
}
int ITEMless(Item val1, Item val2) {
    return (strcmp(KEYget(val1),KEYget(val2))<0);
}
```

ITEMnew e ITEMfree (tipologia 3)

allocazione/deallocazione della sola struct

Item.c

```
Item ITEMnew(void) {
    Item val=(Item)malloc(sizeof(struct item));
    if (val==NULL)
        return ITEMsetvoid();
    val->name[0] = '\0';
    val->num = 0;
    return val;
}

void ITEMfree(Item val) {
    free(val);
}
```

3

ITEMscan (tipologia 3)

Item.c

```
Item ITEMscan() {
    Item val = ITEMnew();
    if (val != NULL) {
        scanf("%s %d", val->name, &val->num);
    }
    return val;
}
```

3

dato creato (e valori assegnati)
internamente alla ITEMscan

ITEMnew e ITEMfree (tipologia 4)

Item.c

```
4   Item ITEMnew(void) {
        Item val=(Item)malloc(sizeof(struct item));
        if (val==NULL)
            return ITEMsetvoid();
        val->name = NULL;
        val->num = 0;
        return val;
    }
    void ITEMfree(Item val) {
        if (val->name!=NULL) free(val->name);
        free(val);
    }
```

allocazione della struct che include stringa vuota

deallocazione stringa se non vuota

ITEMscan (tipologia 4): versione 1

Item.c

```
4   Item ITEMscan() {
    Item val = ITEMnew();
    if (val != NULL) {
        scanf("%s %d", buf, &val->num);
        val->name = strdup(buf);
    }
    return val;
}
```

dato creato (e valori assegnati)
internamente alla ITEMscan

ITEMscan (tipologia 4): versione 2

Item.c

```
void ITEMscan(Item val) {  
    4   scanf("%s %d", buf, &val->num);  
    val->name = strdup(buf);  
}
```

Item ricevuto per riferimento
e valori assegnati internamente
alla ITEMscan

Conclusione

- ADT di I classe con `ITEMnew` e `ITEMscan` garantisce al client completa responsabilità su allocazione/deallocazione
- Consigliabile per tipologia 4 (campo stringa dinamica).

ADT per collezioni

CONTENITORI DI DATI: LISTE, INSIEMI, CODE GENERALIZZATE

ADT per collezioni di dati

- Suggerita la soluzione ADT di I classe
- Operazioni principali
 - insert: inserisci nuovo oggetto nella collezione
 - delete: cancella un oggetto della collezione
- Altre operazioni
 - inizializzare struttura dati
 - conteggio elementi (o verifica collezione vuota)
 - distruzione struttura dati
 - copia struttura dati

ADT di classe lista (non ordinata)

list.h

```
typedef struct list *LIST;  
  
void listInsHead (LIST l, Item val);  
Item listSearch(LIST l, Key k);  
void listDelKey(LIST l, Key k);
```

list.c

```
typedef struct node *link;  
struct node { Item val; link next; };  
struct list { link head; int N; };  
  
void LISTinsHead (LIST l, Item val) {  
    l->head = newNode(val,l->head);  
    l->N++;  
}  
//implementazione delle altre funzioni
```

Vantaggi dell'ADT di 1 classe

L' ADT di 1 classe:

1. nasconde al client i dettagli
2. permette al client di istanziare più variabili del tipo dell'ADT

Un quasi ADT viola una delle 2 regole precedenti o entrambe.

I quasi ADT visti sinora (per composti/aggregati) violavano la prima regola.

Quasi-ADT (non ADT) per collezioni

Per gli ADT collezioni di dati può bastare avere a disposizione un solo contenitore, facendone una variabile globale dell'implementazione.

Scompare il tipo di dato per la collezione (non c'è un'istruzione `typedef`).

Sarebbe meglio chiamarlo **non ADT**, ma si mantiene il nome storico di «quasi ADT».

Lista non ordinata come quasi ADT in violazione della regola 2

list.h

```
void listInsHead (Item val);  
Item listSearch(Key k);  
void listDelKey(Key k);
```

definizione di nodo e di
puntatore a nodo

list.c

manca typedef
e non c'è il parametro LIST 1

```
typedef struct node *link;  
struct node { Item val; link next; } ;  
  
static link head=NULL;  
static int N=0;  
  
void LISTinsHead (Item val) {  
    head = newNode(val,head);  
    N++;  
}  
//implementazione delle altre funzioni
```

variabili globali per
puntatore alla
testa e cardinalità

ADT di classe Set (insieme)

Set.h

```
typedef struct set *SET;

SET SETinit(int maxN);
void SETfree(SET s);
void SETfill(SET s, Item val);
int SETsearch(SET s, Key k);
SET SETunion(SET s1, SET s2);
SET SETintersection(SET s1, SET s2);
int SETsize(SET s);
int SETempty(SET s);
void SETdisplay(SET s);
```

Implementazioni possibili

- vettore
 - non ordinato
 - ordinato
- lista
 - non ordinata
 - Ordinata
- Perche NON si adotta una soluzione stile union-find (vettore con accesso diretto e corrispondenza dato-indice)?
 - Nella union-find un elemento appartiene a UN SOLO INSIEME
 - Il caso generale è diverso. Un elemento può appartenere a più insiemi, che supportano operazioni di unione, intersezione, differenza, ecc.

Vantaggi/svantaggi

- ❑ la dimensione della lista virtualmente può crescere all'infinito, mentre il vettore va dimensionato
- ❑ complessità della funzione **SETsearch** di appartenenza:
 - vettore ordinato \Rightarrow ricerca dicotomica $\Rightarrow O(\log N)$
 - vettore non ordinato \Rightarrow ricerca lineare $\Rightarrow O(N)$
 - lista (ordinata/non ordinata) \Rightarrow ricerca lineare $\Rightarrow O(N)$
- ❑ complessità delle funzioni **SETunion** e **SETintersection**:
 - vettore/lista ordinato $\Rightarrow O(N)$
 - vettore/lista non ordinato $\Rightarrow O(N^2)$

Implementazione con vettore ordinato

Set.c

```
struct set { Item *v; int N; };

SET SETinit(int maxN) {
    SET s = malloc(sizeof *s);
    s->v = malloc(maxN*sizeof(Item));
    s->N=0;
    return s;
}
void SETfree(SET s) {
    free(s->v);
    free(s);
}
```

dimensione massima

wrapper

ricerca dicotomica

Set.c

```
int SETsearch(SET s, Key k) {
    int l = 0, m, r = s->N -1;
    while (l <= r) {
        m = l + (r-l)/2;
        if (KEYeq(key(s->v[m]), k))
            return 1;
        if (KEYless(key(s->v[m]), k))
            l = m+1;
        else
            r = m-1;
    }
    return 0;
}
```

Set.c

```
SET SETunion(SET s1, SET s2) {  
    int i=0, j=0, k=0, size1=SETsize(s1);  
    int size2=SETsize(s2);  
    SET s;  
    s = SETinit(size1+size2);  
    for(k = 0; (i < size1) || (j < size2); k++)  
        if (i >= size1) s->v[k] = s2->v[j++];  
        else if (j >= size2) s->v[k] = s1->v[i++];  
        else if (ITEMless(s1->v[i], s2->v[j]))  
            s->v[k] = s1->v[i++];  
        else if (ITEMless(s2->v[j], s1->v[i]))  
            s->v[k] = s2->v[j++];  
        else { s->v[k] = s1->v[i++]; j++; }  
    s->N = k;  
    return s;  
}
```

strategia simile alla
Merge del MergeSort

Set.c

```
SET SETintersection(SET s1, SET s2) {
    int i=0, j=0, k=0, size1=SETsize(s1);
    int size2=SETsize(s2), minsize;
    SET s;
    minsize = min(size1, size2);
    s = SETinit(minsize);
    while ((i < size1) && (j < size2)) {
        if (ITEMeq(s1->v[i], s2->v[j])) {
            s->v[k++] = s1->v[i++]; j++;
        }
        else if (ITEMless(s1->v[i], s2->v[j])) i++;
        else j++;
    }
    s->N = k;
    return s;
}
```

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

Implementazione con lista non ordinata

Set.c

```
typedef struct SETnode *link;  
  
struct set { link head; int N; };  
struct setNode { Item val; link next; };  
  
SET SETinit(int maxN) {  
    SET s = malloc(sizeof *s);  
    s->head = NULL;  
    s->N = 0;  
    return s;  
}  
void SETfree(SET s) {  
    link x, t;  
    for (x=s->head; x!=NULL; x=t) {  
        t = x->next; free(x);  
    }  
    free(s);  
}
```

wrapper

dimensione massima per uniformità ma non usata

ricerca lineare

Set.c

```
int SETsearch(SET s, Key k) {
    link x;
    x = s->head;
    while (x != NULL) {
        if (KEYeq(key(x->val), k))
            return 1;
        x = x->next;
    }
    return 0;
}
```

Set.c

```
SET SETunion(SET s1, SET s2) {
    link x1, x2; int founds2, counts2=0;
    SET s = SETinit(s1->N + s2->N);
    x1 = s1->head;
    while (x1 != NULL) {
        SETfill(s, x1->val); x1 = x1->next;}
    for (x2 = s2->head; x2 != NULL; x2 = x2->next) {
        x1 = s1->head;
        founds2 = 0;
        while (x1 != NULL) {
            if (ITEMeq(x1->val, x2->val)) founds2 = 1;
            x1 = x1->next;
        }
        if (founds2 == 0) {
            SETfill(s, x2->val); counts2++; }
    }
    s->N = s1->N + counts2;
    return s;
}
```

inserimento in testa
a lista non ordinata

inserimento in testa
a lista non ordinata

Set.c

```
SET SETintersection(SET s1, SET s2) {
    link x1, x2; int counts=0; SET s;
    s = SETinit(s1->N + s2->N);
    x1 = s1->head;
    while (x1 != NULL) {
        x2 = s2->head;
        while (x2 != NULL) {
            if (ITEMeq(x1->val, x2->val)) {
                SETfill(s, x1->val); counts++; break;}
            x2 = x2->next;
        }
        x1 = x1->next;
    }
    s->N = counts;
    return s;
}
```

inserimento in testa
a lista non ordinata

Code Generalizzate

CODA E STACK, CODA PRIORITARIA, TABELLE DI SIMBOLI

Le code generalizzate

Code generalizzate: collezioni di oggetti (dati) di tipo **Item** con operazioni principali:

- **Insert**: inserisci un nuovo oggetto nella collezione
- **Search**: ricerca se un oggetto è nella collezione
- **Delete**: cancella un oggetto della collezione

Altre operazioni:

- inizializzare la coda generalizzata
- conteggio oggetti (o verifica collezione vuota)
- distruzione della coda generalizzata
- copia della coda generalizzata

Criteri per operazione di Delete (extract)

- **cronologico:**
 - estrazione dell'elemento inserito più recentemente
 - politica LIFO: Last-In First-Out
 - **stack o pila**
 - inserzione (push) ed estrazione (pop) dalla testa
 - estrazione dell'elemento inserito meno recentemente
 - politica FIFO: First-In First-Out
 - **queue o coda**
 - inserzione (enqueue o put) in coda (tail) ed estrazione (dequeue o get) dalla testa (head)
- **priorità:**
 - l'inserzione garantisce che, estraendo dalla testa, si ottenga il dato a priorità massima (o minima)
 - **coda a priorità**

Criteri per operazione di Delete (extract)

- **caso:**
 - estraendo si ottiene un dato a caso
 - **coda casuale**
- **contenuto:**
 - l'estrazione ritorna un contenuto secondo determinati criteri
 - **tabella di simboli**

Pieno/vuoto

Controlli pieno/vuoto per evitare di inserire in coda piena o estrarre da coda vuota.

Scelta tra 2 strategie:

1. il client tiene conto del numero di dati nella coda oppure l'ADT fornisce funzioni di interfaccia per il controllo pieno/vuoto
2. l'ADT controlla la correttezza delle operazioni, indicando il successo/fallimento.

Nel seguito si adotta la prima strategia (la più semplice per l'implementazione).

L'ADT pila (stack)

Definizione: ADT che supporta operazioni di

- STACKpush: inserimento in cima
- STACKpop: preleva (e cancella) dalla cima l'oggetto inserito più di recente

Terminologia: la strategia di gestione dei dati è detta LIFO (Last In First Out)

Possibili versioni dell'ADT stack

- con vettore
 - quasi ADT
 - ADT di I classe
- con lista
 - quasi ADT
 - ADT di I classe

Vettore vs. lista: vantaggi/svantaggi

Spazio:

- vettore: spazio allocato sempre pari al massimo previsto, vantaggioso per stack quasi pieni
- lista: spazio utilizzato proporzionale al numero di elementi correnti, vantaggioso per stack che cambiano rapidamente dimensione

Tempo:

- push e pop **T(n) = O(1)**

Quasi ADT vs. ADT I classe

Quasi ADT

- implementazione mediante variabili **globali** (dichiarate fuori da funzioni) e **invisibili** da altri file sorgenti (**static**)

ADT di I classe

- una **struct** puntata (da handle), contenente, come campi, le variabili globali del quasi ADT.

Implementazione con vettore

- inizializzazione dello stack (**STACKinit**): array dinamico la cui dimensione viene ricevuta (come parametro `maxN`) dal programma client
- NON viene controllato il rispetto dei casi limite (pop da stack vuoto o push in stack pieno)
- suggerimento: implementare i controlli

Quasi ADT

stack.c

stack.h

```
void STACKinit(int maxN);
int STACKempty();
void STACKpush(Item val);
Item STACKpop();
```

```
static Item *s;
static int N;

void STACKinit(int maxN) {
    s = malloc(maxN*sizeof(Item));
    N=0;
}

int STACKempty() {
    return N == 0;
}
void STACKpush(Item val) {
    s[N++] = val;
}
Item STACKpop() {
    return s[--N];
}
```

quasi ADT

stack.c

variabili globali:
1 solo stack

```
static Item *s;  
static int N;
```

```
void STACKinit(int maxN) {  
    s = malloc(maxN*sizeof(Item));  
    N=0;  
}  
  
int STACKempty() {  
    return N == 0;  
}  
void STACKpush(Item val) {  
    s[N++] = val;  
}  
Item STACKpop() {  
    return s[--N];  
}
```

stack.h

```
void STACKinit(int maxN);  
int STACKempty();  
void STACKpush(Item val);  
Item STACKpop();
```

ADT I classe

stack.c

le variabili globali del quasi ADT sono diventate campi della struct

stack.h

```
typedef struct stack *STACK;  
  
STACK STACKinit(int maxN);  
int STACKempty(STACK s);  
void STACKpush(STACK s,  
              Item val);  
Item STACKpop (STACK s);
```

```
struct stack { Item *s; int N; };  
  
STACK STACKinit(int maxN) {  
    STACK sp = malloc(sizeof *sp) ;  
    sp->s = malloc(maxN*sizeof(Item));  
    sp->N=0;  
    return sp;  
}  
int STACKempty(STACK sp) {  
    return sp->N == 0;  
}  
void STACKpush(STACK s, Item val) {  
    sp->s[sp->N++] = val;  
}  
Item STACKpop(STACK s) {  
    return sp->s[--(sp->N)];  
}
```

Implementazione con lista

Stack di elementi in lista concatenata:

- coda della lista: primo elemento inserito
- testa della lista: ultimo elemento inserito
- push: inserzione in testa
- pop: estrazione dalla testa

La dimensione dello stack è (virtualmente) illimitata.

- inizializzazione dello stack come lista vuota (`maxN` non viene utilizzato)
- funzione `NEW` per creare (dinamicamente) un nuovo elemento
- NON viene controllato il rispetto del caso limite (pop da stack vuoto)

quasi ADT

stack.c

variabili globali:
1 solo stack

```
typedef struct STACKnode* link;
struct STACKnode {Item val; link next;};
static link head;
```

```
static link NEW (Item val, link next){
    link x = (link) malloc(sizeof *x);
    x->val = val; x->next = next;
    return x;
}
void STACKinit(int maxN) { head = NULL; }
int STACKempty() {return head == NULL; }
void STACKpush(Item val) {
    head = NEW(val, head);
}
```

stack.h

```
void STACKinit(int maxN);
int STACKempty();
void STACKpush(Item val);
Item STACKpop();
```

quasi ADT

stack.c

```
typedef struct STACKnode* link;
struct STACKnode {Item val; link next;};

static link head;

...

Item STACKpop() {
    Item tmp;
    tmp = head->val;
    link t = head->next;
    free(head);
    head = t;
    return tmp;
}
```

stack.h

```
void STACKinit(int maxN);
int STACKempty();
void STACKpush(Item val);
Item STACKpop();
```

ADT I classe

stack.c

le variabili globali del quasi ADT sono diventate campi della struct

stack.h

```
typedef struct stack *STACK;

STACK STACKinit(int maxN);
int STACKempty(STACK s);
void STACKpush(STACK s,
               Item val);
Item STACKpop (STACK s);
```

```
typedef struct STACKnode* link;
struct STACKnode {Item val; link next;};

struct stack { link head; };

static link NEW (Item val, link next){
    link x = (link) malloc(sizeof *x);
    x->val = val; x->next = next;
    return x;
}

STACK STACKinit(int maxN) {
    STACK s = malloc(sizeof *s) ;
    s->head = NULL;
    return s;
}

int STACKempty(STACK s) {
    return s->head == NULL; }
```

ADT I classe

stack.c

stack.h

```
typedef struct stack *STACK;

STACK STACKinit(int maxN);
int STACKempty(STACK s);
void STACKpush(STACK s,
               Item val);
Item STACKpop (STACK s);
```

```
typedef struct STACKnode* link;
struct STACKnode {Item val; link next;};

struct stack { link head; };

...

void STACKpush(STACK s, Item val) {
    s->head = NEW(val, s->head); }

Item STACKpop (STACK s) {
    Item tmp;
    tmp = s->head->val;
    link t = s->head->next;
    free(s->head);
    s->head = t;
    return tmp;
}
```

L'ADT coda (queue)

Definizione: ADT che supporta operazioni di:

- **enqueue/put**: inserisci un elemento (QUEUEput)
- **dequeue/get**: preleva (e cancella) l'elemento che è stato inserito meno recentemente (QUEUEget)

Terminologia: la strategia di gestione dei dati è detta FIFO (First In First Out).

Possibili versioni dell'ADT queue

- con vettore
 - quasi ADT
 - ADT di I classe
- con lista
 - quasi ADT
 - ADT di I classe

Vettore vs. lista: vantaggi/svantaggi

Spazio:

- vettore: spazio allocato sempre pari al massimo previsto, vantaggioso per code quasi piene
- lista: spazio utilizzato proporzionale al numero di elementi correnti, vantaggioso per code che cambiano rapidamente dimensione

Tempo:

- put e get **T(n) = O(1)**

Quasi ADT vs. ADT I classe

Quasi ADT

- implementazione mediante variabili **globali** (dichiarate fuori da funzioni) e **invisibili** da altri file sorgenti (**static**)

ADT di I classe

- una **struct** puntata (da handle), contenente, come campi, le variabili globali del quasi ADT.

Accesso a testa e coda

Servono 2 variabili `head` e `tail`:

- `head` permette di accedere all'elemento in testa, cioè il prossimo da estrarre
- `tail` permette di accedere:
 - implementazione a vettore: alla locazione che segue l'ultimo elemento in coda, cioè alla posizione della prossima inserzione
 - implementazione a lista: alla posizione dell'ultimo elemento in coda.

`head` e `tail` sono:

- indici nell'implementazione a vettore
- puntatori nell'implementazione a lista.

Implementazione con vettore ($O(n)$)

- `put` assegna alla prima cella libera, se esiste, in fondo al vettore con complessità $O(1)$. L'indice `tail` contiene il numero di elementi nella coda
- `get` da posizione fissa (`head = 0`), ma comporta scalare a sinistra tutti gli elementi restanti con costo $O(n)$

Implementazione con vettore ($O(1)$): buffer circolare

- put assegna alla prima cella libera, se esiste, in posizione indicata da indice tail ($O(1)$)
- get da posizione variabile (**head** assume valori tra 0 e $N-1$). Le celle del vettore occupate da elementi si spostano per via di put e get (**buffer circolare**).
 - head e tail sono incrementati MODULO N ($N-2, N-1, 0, 1, \dots$)
 - Non è più garantito $\text{head} \leq \text{tail}$:
 - coda vuota: head raggiunge tail
 - coda piena: tail raggiunge head

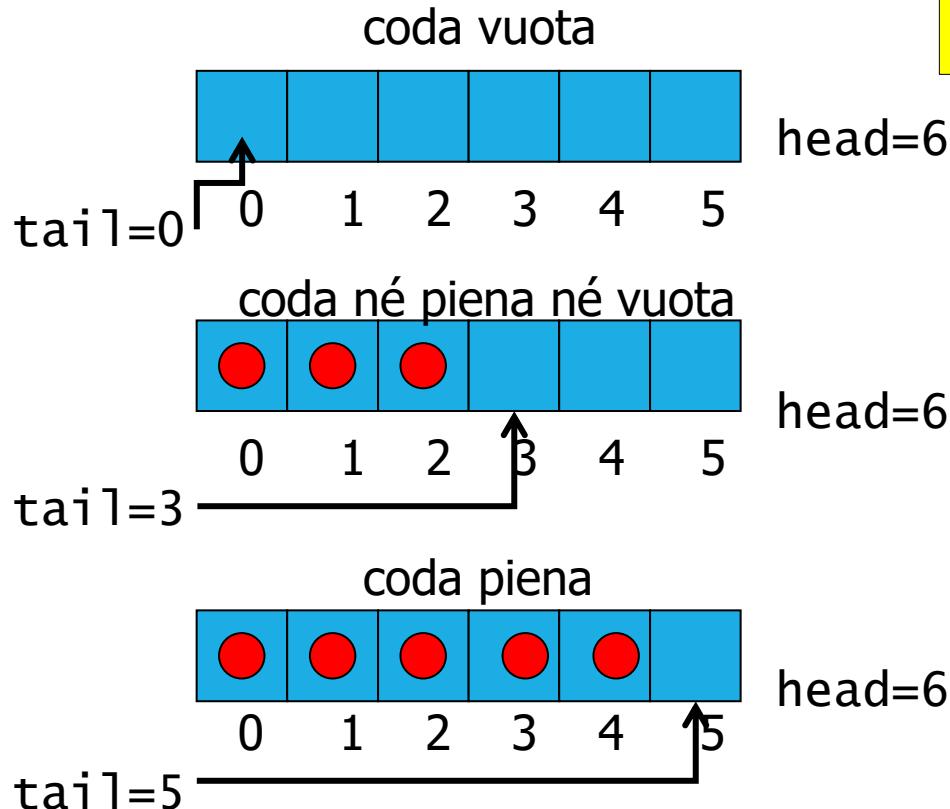
Strategie per «riconoscere» coda piena/vuota

- A. Usare un contatore di dati (terza variabile oltre e `head` e `tail`)
- B. Evitare lo riempimento: se si conosce il numero massimo «possibile» di dati in coda (`maxN`), allocare $N=maxN+1$ celle, di cui al massimo `maxN` usate.
 - o `tail` NON può raggiungere `head`
 - o La condizione `head==tail` può solo essere raggiunta per coda vuota.

Implementazione (versione B)

- Dimensione del vettore $N=maxN+1$. Si ammettono al massimo `maxN` elementi
- inizializzazione della coda (QUEUEinit): vettore dinamico di dimensione ricevuta (come parametro `maxN`) dal programma client
 - o inizialmente `head=N`, `tail=0`
 - o successivamente `head%N == tail` indica coda vuota (`tail` non può raggiungere `head` per coda piena, è proibito!)

maxN = 5
N = maxN+1 = 6



quasi ADT

variabili globali:
1 sola coda

queue.h

```
void QUEUEinit(int maxN);
int QUEUEempty();
void QUEUEput(Item val);
Item QUEUEget();
```

queue.c

```
static Item *q;
static int N, head, tail;

void QUEUEinit(int maxN) {
    q = malloc((maxN+1)*sizeof(Item));
    N = maxN+1;
    head = N; tail = 0;
}
int QUEUEempty() {
    return head%N == tail;
}
void QUEUEput(Item val) {
    q[tail++] = val;
    tail = tail%N;
}
Item QUEUEget() {
    head = head%N;
    return q[head++];
}
```

ADT I classe

le variabili globali del quasi ADT sono diventate campi della struct

queue.h

```
typedef struct queue *QUEUE;

QUEUE QUEUEinit(int maxN);
int QUEUEempty(QUEUE q);
void QUEUEput(QUEUE q,
              Item val);
Item QUEUEget (QUEUE q);
```

queue.c

```
struct queue {
    Item *q;
    int N, head, tail;
};

QUEUE QUEUEinit(int maxN) {
    QUEUE q = malloc(sizeof *q) ;
    q->q = malloc(maxN*sizeof(Item));
    q->N=maxN+1;
    q->head = N;
    q->tail = 0;
    return q;
}

int QUEUEempty(QUEUE q) {
    return (q->head)%(q->N) == q->tail;
}
...
```

ADT I classe

queue.h

```
typedef struct queue *QUEUE;

QUEUE QUEUEinit(int maxN);
int QUEUEempty(QUEUE q);
void QUEUEput(QUEUE q,
              Item val);
Item QUEUEget (QUEUE q);
```

queue.c

```
struct queue {
    Item *q;
    int N, head, tail;
};

...

void QUEUEput(QUEUE q, Item val) {
    q->q[tail++] = val;
    q->tail = q->tail%N;
}

Item QUEUEget(QUEUE q) {
    q->head = q->head%N;
    return q->q[q->head++];
}
```

Implementazione con lista

Coda di elementi in lista concatenata:

- testa della lista (head): primo elemento inserito
- coda della lista (tail): ultimo elemento inserito
- put: inserzione in coda
- get: estrazione dalla testa

La dimensione della coda è (virtualmente) illimitata:

- inizializzazione della coda come lista vuota (maxN non viene utilizzato, è mantenuto per uniformità con la versione basata su vettore)
- funzione NEW per creare (dinamicamente) un nuovo elemento
- put e get sono funzioni standard di inserzione in fondo ad una lista ed estrazione dalla testa della lista.

quasi ADT

queue.h

```
void QUEUEinit(int maxN);  
int QUEUEempty();  
void QUEUEput(Item val);  
Item QUEUEget();
```

variabili globali:
1 sola coda

queue.c

```
typedef struct QUEUEnode *link;  
struct QUEUEnode{Item val; link next;};  
  
static link head, tail;  
  
link NEW (Item val, link next) {  
    link x = malloc(sizeof *x);  
    x->val = val;  
    x->next = next;  
    return x;  
}  
void QUEUEinit(int maxN) {  
    head = tail = NULL;  
}  
int QUEUEempty() {  
    return head == NULL;  
}  
...
```

quasi ADT

queue.h

```
void QUEUEinit(int maxN);
int QUEUEempty();
void QUEUEput(Item val);
Item QUEUEget();
```

queue.c

```
typedef struct QUEUEnode *link;
struct QUEUEnode{Item val; link next;};

static link head, tail;
...
void QUEUEput(Item val) {
    if (head == NULL) {
        head = (tail = NEW(val, head));
        return;
    }
    tail->next = NEW(val, tail->next);
    tail = tail->next;
}
Item QUEUEget() {
    Item tmp = head->val;
    link t = head->next;
    free(head); head = t;
    return tmp;
}
```

ADT I classe

le variabili globali del quasi ADT sono diventate campi della struct

queue.h

```
typedef struct queue *QUEUE;  
  
QUEUE QUEUEinit(int maxN);  
int QUEUEempty(QUEUE q);  
void QUEUEput(QUEUE q,  
              Item val);  
Item QUEUEget (QUEUE q);
```

queue.c

```
typedef struct QUEUEnode *link;  
struct QUEUEnode{ Item val; link next; };  
  
struct queue { link head; link tail; };  
  
link NEW(Item val, link next) {  
    link x = malloc(sizeof *x) ;  
    x->val = val; x->next = next;  
    return x;  
}  
QUEUE QUEUEinit(int maxN) {  
    QUEUE q = malloc(sizeof *q) ;  
    q->head = NULL;  
    return q;  
}  
int QUEUEempty(QUEUE q) {  
    return q->head == NULL;  
}
```

ADT I classe

queue.h

```
typedef struct queue *QUEUE;  
  
QUEUE QUEUEinit(int maxN);  
int QUEUEempty(QUEUE q);  
void QUEUEput(QUEUE q,  
              Item val);  
Item QUEUEget (QUEUE q);
```

queue.c

```
...  
  
void QUEUEput (QUEUE q, Item val) {  
    if (q->head == NULL){  
        q->tail = NEW(val, q->head) ;  
        q->head = q->tail;  
        return;  
    }  
    q->tail->next = NEW(val,q->tail->next);  
    q->tail = q->tail->next;  
}  
Item QUEUEget(QUEUE q) {  
    Item tmp = q->head->tmp;  
    link t = q->head->next;  
    free(q->head); q->head = t;  
    return tmp;  
}
```

L'ADT coda a priorità

Definizione: ADT che supporta operazioni di:

- **insert**: inserisci un elemento (PQinsert)
- **extract**: preleva (e cancella) l'elemento a priorità massima (o minima) (PQextractmax o PQextractmin).

Terminologia: la strategia di gestione dei dati è detta priority-first.

Altre operazioni:

- inizializzare la coda a priorità
- verifica se vuota
- visualizzazione senza estrazione di elemento a massima/minima priorità
- cambio della priorità di un elemento

Possibili versioni dell'ADT coda a priorità

- con vettore/lista
 - ordinato/non ordinato
 - quasi ADT/ADT di I classe
- con heap.

Trattato più avanti

Complessità

- implementazione con vettore/lista NON ordinato:
 - inserzione in testa alla lista o in coda al vettore -> $O(1)$
 - estrazione/visualizzazione del massimo/minimo con scansione -> $O(N)$
 - cambio di priorità: richiede ricerca dell'elemento con scansione -> $O(N)$
- implementazione con vettore/lista ordinato:
 - inserzione ordinata nella lista o nel vettore mediante scansione -> $O(N)$
 - estrazione/visualizzazione del massimo/minimo se memorizzato in testa alla lista o in coda al vettore con accesso diretto -> $O(1)$
 - cambio di priorità:
 - lista: richiede ricerca dell'elemento mediante scansione ($O(N)$), eliminazione ($O(1)$), reinserimento ($O(N)$), globalmente complessità $O(N)$
 - vettore: richiede ricerca dell'elemento mediante ricerca dicotomica ($O(\log N)$), eliminazione ($O(N)$), reinserimento ($O(N)$), globalmente complessità $O(N)$

Complessità

- implementazione con heap:
 - inserzione/estrazione del massimo/minimo con complessità $O(\log N)$
 - visualizzazione del massimo/minimo con complessità $O(1)$
 - cambio di priorità: richiede ricerca dell'elemento (con tabella di hash complessità media $O(1)$), globalmente complessità $O(\log N)$.

ADT I classe coda a priorità

Implementazione con liste ordinate

PQ.h

```
typedef struct pqueue *PQ;

PQ PQinit(int maxN);
int PQempty(PQ pq);
void PQinsert(PQ pq, Item data);
Item PQextractMax(PQ pq);
Item PQshowMax(PQ pq);
void PQdisplay(PQ pq);
void PQchange(PQ pq, Item data);
```

PQ.c

```
typedef struct PQnode *link;
struct PQnode{ Item val; link next; };

struct pqueue { link head; };

link NEW(Item val, link next) {
    link x = malloc(sizeof *x);
    x->val = val; x->next = next;
    return x;
}
PQ PQinit(int maxN) {
    PQ pq = malloc(sizeof *pq);
    pq->head = NULL;
    return pq;
}
int PQempty(PQ pq) {
    return pq->head == NULL;
}
Item PQshowMax(PQ pq) {
    return pq->head->val;
}
```

```
void PQdisplay(PQ pq) {
    link x;
    for (x=pq->head; x!=NULL; x=x->next)
        ITEMdisplay(x->val);
    return;
}
void PQinsert (PQ pq, Item val) {
    link x, p;
    Key k = KEYget(val);
    if (pq->head==NULL || KEYless(KEYget(pq->head->val),k)) {
        pq->head = NEW(val, pq->head);
        return;
    }
    for(x=pq->head->next, p=pq->head;
        x!=NULL&&KEYless(k,KEYget(x->val));
        p=x, x=x->next);
    p->next = NEW(val, x);
    return;
}
```

PQ.c

```
typedef struct PQnode *link;
struct PQnode{ Item val; link next; };

struct pqueue { link head; };

...
Item PQextractMax(PQ pq) {
    Item tmp;
    link t;
    if (PQempty(pq)) {
        printf("PQ empty\n");
        return ITEMsetvoid();
    }
    tmp = pq->head->val;
    t = pq->head->next;
    free(pq->head);
    pq->head = t;
    return tmp;
}
```

```
void PQchange (PQ pq, Item val) {
    link x, p;
    if (PQempty(pq)) {
        printf("PQ empty\n");
        return;
    }
    for(x=pq->head, p=NULL; x!=NULL;
        p=x, x=x->next) {
        if (ITEMeq(x->val, val)){
            if (x==pq->head)
                pq->head = x->next;
            else
                p->next = x->next;
            free(x);
            break;
        }
    }
    PQinsert(pq, val);
    return;
}
```

Tabelle di Simboli

IMPLEMENTAZIONI BASATE SU VETTORI E LISTE

L'ADT Tabella di Simboli

Definizione: ADT che supporta operazioni di:

- **insert**: inserisci un dato (item) (STinsert)
- **search**: ricerca dato con certa chiave (STsearch)
- **delete**: cancella il dato con una certa chiave (STdelete).

Talora la tabella di simboli è detta **dizionario**.

Altre operazioni:

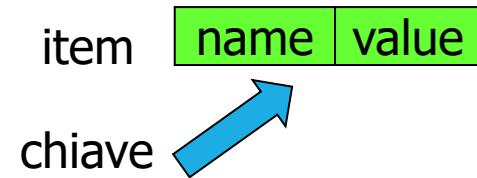
- inizializzare la tabella
- distruggere la tabella
- contare il numero di dati
- visualizzare della tabella
- se sulla chiave è definita una relazione d'ordine:
 - ordinare la tabella
 - selezionare la chiave di rango r (r -esima più piccola chiave)

Applicazioni delle tabelle di simboli

Applicazione	scopo: trovare	chiave	valore ritornato
dizionario	la definizione	parola	definizione
indice libro	pagine rilevanti	termine	lista pagine
DNS	indirizzo IP dato URL	URL	IP address
DNS inverso	URL dato indirizzo IP	IP address	URL
file system	file su disco	nome file	localizzazione disco
web search	pagine web	parola chiave	lista di pagine

Item

- Quasi ADT Item
- Dati:
 - Nome (stringa), valore (intero)
 - Chiave = nome
 - Tipologia 3 (composto per valore)



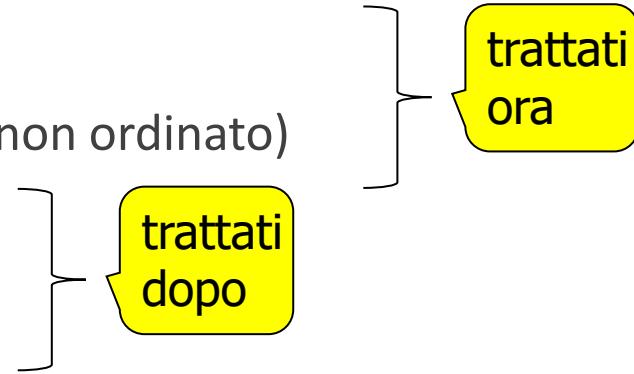
ADT di I classe Tabella di simboli

ST.h

```
typedef struct symboltable *ST;  
  
ST STinit(int maxN);  
void STfree(ST st);  
int STcount(ST st);  
void STinsert(ST st, Item val);  
Item STsearch(ST st, Key k);  
void STdelete(ST st, Key k);  
Item STselect(ST st, int r);  
void STdisplay(ST st);
```

Possibili versioni dell'ADT tabella di simboli

- tabelle ad accesso diretto
- strutture lineari (vettore/lista ordinato/non ordinato)
- strutture ad albero
 - alberi binari di ricerca (BST) e loro varianti
- tabelle di hash.



Complessità

	caso peggiore		
	inserim.	ricerca	selezione
Tabelle ad accesso diretto	1	1	maxN
Array non ordinato	1	n	
Array ordinato e ricerca lineare	n	n	1
Array ordinato e ricerca binaria	n	logn	1
Lista non ordinata	1	n	
Lista ordinata	n	n	n
BST	n	n	n
RB-tree	logn	logn	logn
Hashing	1	n	

Complessità

	caso medio		
	inserim.	ricerca	selezione
Tabelle ad accesso diretto	1	1	$\text{maxN}/2$
Array non ordinato	1	$n/2$	
Array ordinato e ricerca lineare	$n/2$	$n/2$	$n/2$
Array ordinato e ricerca binaria	$n/2$	$\log n$	$\log n$
Lista non ordinata	1	$n/2$	
Lista ordinata	$n/2$	$n/2$	$n/2$
BST	$\log n$	$\log n$	$\log n$
RB-tree	$\log n$	$\log n$	$\log n$
Hashing	1	1	

ADT di I classe Tabella ad accesso diretto

- insieme universo U con $M = \text{card}(U) = \text{maxN}$ elementi
- corrispondenza biunivoca tra ciascuna delle chiavi $k \in U$ e gli interi tra 0 e $M-1$ (funzione int GETindex(Key k)). L'intero funge da indice in un vettore
- vettore $\text{st} \rightarrow a[]$ di dimensione maxN :
- se la chiave k è nella tabella, essa è in posizione $\text{st} \rightarrow a[\text{GETindex}(k)]$, altrimenti $\text{st} \rightarrow a[\text{GETindex}(k)]$ contiene l'elemento vuoto
- si memorizza un insieme di N chiavi ($N \leq M$). La cardinalità è N ritornata dalla funzione $\text{st} \rightarrow \text{size}$.

Esempi di GETindex

se le chiavi sono le lettere maiuscole dell'alfabeto inglese A..Z ($M = 26$)

```
int GETindex(Key k) {  
    int i;  
    i = k - 'A';  
    return i;  
}
```

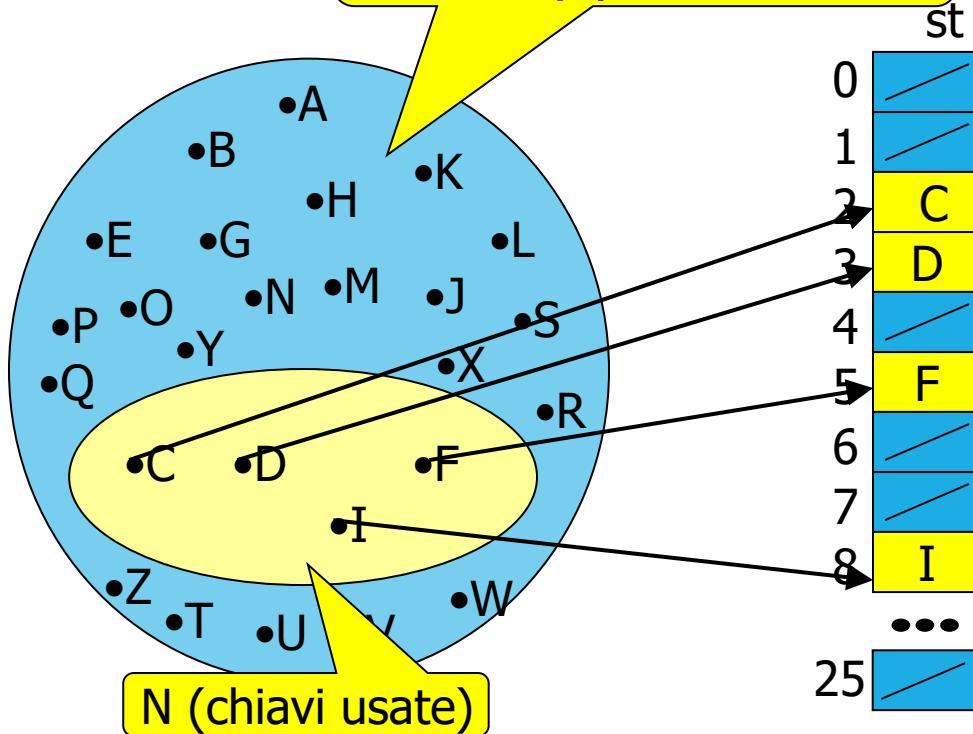
se le chiavi sono interi tra 0 e $M-1$

```
int GETindex(Key k) {  
    int i;  
    i = (int) k;  
    return i;  
}
```

- se le chiavi sono stringhe di lunghezza l fissa e corta, si trasformano in intero valutandole come polinomio di grado l in base 26 ($M=26^l$)

```
int GETindex(Key k) {  
    int i = 0, b = 26;  
  
    for ( ; *k != '\0'; k++)  
        i = (b * i + (*k - ((int) 'A')));  
    return i;  
}
```

U (universo delle chiavi)
 $M = \text{card}(U) = \text{maxN} = 26$



ST.c

```
struct symbtab {Item *a; int N; int M;};

ST STinit(int maxN) {
    ST st; int i;
    st = malloc(sizeof(*st));
    st->a = malloc(maxN * sizeof(Item) );
    for (i = 0; i < maxN; i++)
        st->a[i] = ITEMsetvoid();
    st->M = maxN;
    st->N= 0;
    return st;
}

int STcount(ST st) {
    return st->N;
}
```

```
void STfree(ST st) {
    free(st->a);
    free(st);
}

void STinsert(ST st, Item val) {
    int index = GETindex(KEYget(val));
    st->a[index] = val;
    st->N++;
}

Item STsearch(ST st, Key k) {
    int index = GETindex(k);
    return st->a[index];
}
```

```
void STdelete(ST st, Key k) {
    st->a[GETindex(k)] = ITEMsetvoid();
    st->N--;
}
Item STselect(ST st, int r) {
    int i;
    for (i = 0; i < st->M; i++)
        if ((ITEMcheckvoid(st->a[i]))==0) &&
            (r-- == 0))
            return st->a[i];
    return NULL;
}
void STdisplay(ST st){
    int i;
    for (i = 0; i < st->M; i++)
        if (ITEMcheckvoid(st->a[i])==0)
            ITEMstore(st->a[i]);
}
```

Vantaggi/svantaggi

- Complessità delle operazioni di inserimento, ricerca e cancellazione: $T(n) = \Theta(1)$
- Complessità delle operazioni di inizializzazione e selezione: $T(n) = \Theta(\text{card}(U)) = \Theta(M)$
- Occupazione di memoria $S(n) = \Theta(\text{card}(U)) = \Theta(M)$
 - applicabile per M piccolo
 - spreco di memoria per $N \ll M$
- Molto usate in pratica per trasformare chiavi in interi e viceversa a costo unitario.

ADT di I classe Tabella di simboli (vettore)

- Vettore non ordinato:
 - inserzione in fondo per avere complessità $O(1)$
 - realloc per ridimensionare la tabella se piena in inserzione
 - ricerca lineare preliminare alla cancellazione con complessità $O(N)$
 - la selezione non ha senso (non è ordinato)
- Vettore ordinato:
 - inserzione con scansione con complessità $O(N)$
 - ricerca dicotomica preliminare alla cancellazione con complessità $O(\log N)$.

```
struct symbtab {Item *a; int maxN; int size;};

ST STinit(int maxN) {
    ST st; int i;
    st = malloc(sizeof(*st));
    st->a = malloc(maxN * sizeof(Item) );
    for (i = 0; i < maxN; i++)
        st->a[i] = ITEMsetvoid();
    st->maxN = maxN;
    st->size = 0;

    return st;
}

int STcount(ST st) {
    return st->size;
}
```

```
void STfree(ST st) {  
    free(st->a);  
    free(st);  
}  
void STdisplay(ST st){  
    int i;  
    for (i = 0; i < st->size; i++)  
        ITEMstore(st->a[i]);  
}  
void STdelete(ST st, Key k) {  
    int i, j=0;  
    while (KEYcmp(KEYget(&st->a[j]), k)!=0)  
        j++;  
    for (i = j; i < st->size-1; i++)  
        st->a[i] = st->a[i+1];  
    st->size--;  
}
```

È responsabilità del client cancellare solo dopo aver accertato che la chiave è presente

Inserzione e ricerca in vettore non ordinato

```
void STinsert(ST st, Item val) {
    int i = st->size;
    if (st->size >= st->maxN) {
        st->a=realloc(st->a,(2*st->maxN)*sizeof(Item));
        if (st->a == NULL) return;
        st->maxN = 2*st->maxN;
    }
    st->a[i] = val; st->size++;
}
Item STsearch(ST st, Key k) {
    int i;
    if (st->size == 0) return ITEMsetvoid();
    for (i = 0; i < st->size; i++)
        if (KEYcmp(k, KEYget(&st->a[i]))==0) return st->a[i];
    return ITEMsetvoid();
}
```

Inserzione, selezione e ricerca in vettore ordinato

```
void STinsert(ST st, Item val) {
    int i = st->size++;
    if (st->size > st->maxN) {
        st->a=realloc(st->a,(2*st->maxN)*sizeof(Item));
        if (st->a == NULL)
            return;
        st->maxN = 2*st->maxN;
    }
    while((i>0)&&KEYcmp(KEYget(&val),KEYget(&st->a[i-1]))==-1){
        st->a[i] = st->a[i-1];
        i--;
    }
    st->a[i] = val;
}
Item STselect(ST st, int r) {
    return st->a[r];
}
```

```
Item STsearch(ST st, Key k) {
    return searchR(st, 0, st->size-1, k) ;
}

Item searchR(ST st, int l, int r, Key k) {
    int m;
    m = (l + r)/2;
    if (l > r)
        return ITEMsetvoid();
    if (KEYcmp(k, KEYget(&st->a[m]))==0)
        return st->a[m];
    if (l == r)
        return ITEMsetvoid();
    if (KEYcmp(k, KEYget(&st->a[m]))==-1)
        return searchR(st, l, m-1, k);
    else
        return searchR(st, m+1, r, k);
}
```

Vantaggi/svantaggi (vettore non ordinato)

- Complessità dell'operazione di inizializzazione e inserimento: $T(n) = \Theta(1)$
- Complessità delle operazioni di ricerca, cancellazione: $T(n) = O(N)$
- Occupazione di memoria $S(n) = \Theta(\max N)$
 \Rightarrow spreco di memoria per $|K| \ll \max N$

dimensione massima
presunta

Vantaggi/svantaggi (vettore ordinato)

- ogni inserzione ordinata ha costo lineare $T(n) = O(N)$, costo quadratrico complessivo per N inserzioni $T(n) = O(N^2)$
- ricerca dicotomica con costo logaritmico
- $T(n) = O(\log N)$
- ricerca lineare con interruzione non appena possibile $T(n) = O(N)$
- cancellazione con costo lineare $T(n) = O(N)$
- selezione banale: rango e indice coincidono.

Implementazione in funzione del contesto:

- **dinamico**: con molte inserzioni/cancellazioni: inserimento ordinato (con spostamento di una posizione degli elementi più grandi) con costo quadratico
- **statico**: con inserzioni solo in fase di lettura da file, nessuna cancellazione e molte ricerche: inserzione in fondo e ordinamento una sola volta con algoritmo $O(N \log N)$.

ADT di I classe Tabella di simboli (lista)

- Ricerca preliminare alla cancellazione sempre lineare $O(N)$
- Lista non ordinata:
 - inserzione in testa per avere complessità $O(1)$
 - la selezione non ha senso
- Lista ordinata:
 - inserzione con scansione con complessità $O(N)$

ST.c

```
typedef struct STnode* link;
struct STnode { Item val; link next; } ;
typedef struct { link head; int size; } list;
struct symbtab { list tab; };
static link NEW( Item val, link next) {
    link x = malloc(sizeof(*x));
    if (x == NULL) return NULL;
    x->val = val;    x->next = next;
    return x;
}
ST STinit(int maxN) {
    ST st;
    st = malloc(sizeof(*st));
    if(st == NULL) return NULL;
    st->tab.size = 0;  st->tab.head = NULL;
    return st;
}
```

```
void STfree(ST st) {
    link x, t;
    for (x = st->tab.head; x != NULL; x = t) {
        t = x->next;
        free(x);
    }
    free(st);
}

int STcount(ST st) {
    return st->tab.size;
}

void STdisplay(ST st) {
    link x;
    for (x = st->tab.head; x != NULL; x = x->next)
        ITEMstore(x->val);
}
```

```
Item STsearch(ST st, Key k) {
    link x;
    if (st == NULL)
        return ITEMsetvoid();
    if (st->tab.head == NULL)
        return ITEMsetvoid();

    for (x = st->tab.head; x != NULL; x = x->next)
        if (KEYcmp( KEYget(&x->val), k) ==0)
            return x->val;
    return ITEMsetvoid();
}
```

```
void STdelete(ST st, Key k) {  
    link x, p;  
    if (st == NULL) return;  
    if (st->tab.head == NULL) return;  
  
    for (x=st->tab.head, p=NULL; x!=NULL; p=x, x=x->next) {  
        if (KEYcmp(k, KEYget(&x->val)) == 0) {  
            if (x == st->tab.head)  
                st->tab.head = x->next;  
            else  
                p->next = x->next;  
            free(x);  
            break;  
        }  
    }  
    st->tab.size--;  
}
```

È responsabilità del client cancellare solo dopo aver accertato che la chiave è presente

Inserzione in lista non ordinata

```
void STinsert(ST st, Item val) {  
    if (st == NULL)  
        return;  
    st->tab.head = NEW(val, st->tab.head);  
    st->tab.size++;  
}
```

Selezione in lista ordinata

```
Item STselect(ST st, int r) {  
    int i;  
    link x = st->tab.head;  
    for (i = r; i>0; i--)  
        x = x->next;  
    return x->val;  
}
```

Inserzione in lista ordinata

```
void STinsert(ST st, Item val) {
    link x, p;
    if (st == NULL)
        return;

    if ((st->tab.head == NULL) ||
        (KEYcmp(KEYget(&st->tab.head->val), KEYget(&val))==1))
        st->tab.head = NEW(val,st->tab.head);
    else {
        for (x = st->tab.head->next, p = st->tab.head;
             x!=NULL&& (KEYcmp(KEYget(&val), KEYget(&x->val))==1);
             p = x, x = x->next);
        p->next = NEW(val, x);
    }
    st->tab.size++;
}
```

Vantaggi/svantaggi

- complessità dell'operazione di inizializzazione e inserimento: $T(n) = \Theta(1)$ (inserimento in lista ordinata $T(n) = O(n)$)
- complessità delle operazioni di ricerca e cancellazione:
 $T(n) = O(n)$
- complessità dell'operazione di selezione con lista ordinata:
 $T(n) = O(n)$
- occupazione di memoria $S(n) = \Theta(n)$

Gestione dei duplicati nelle tabelle di simboli

Casistica:

- le chiavi sono per sé distinte (IBAN, matricola, codice fiscale). In inserzione:
 - “**si ignora il nuovo elemento**” : si prosegue come se l’istanza (di inserimento) non sia stata avanzata = si ignora l’inserimento
 - “**si dimentica il vecchio elemento**” : si cancella (o sovrascrive) l’elemento già presente, poi si procede al nuovo inserimento
- le chiavi possono essere rese distinte (per es. si aggiunge il prefisso della nazione al numero telefonico. Si ricade nel caso precedente

- nel modello chiavi duplicate hanno senso (per es. numero di crediti superati condiviso da più studenti). Ci si riconduce al modello precedente creando una chiave univoca e un riferimento alla lista degli elementi che la condividono. L'inserzione è in 2 passi:
 - data la chiave, si identifica la lista ad essa associata
 - si inserisce nella lista
- nel modello hanno senso elementi che condividono la stessa chiave (per es. nome e cognome cui sono associati diversi indirizzi di e-mail). In ricerca è il client che decide cosa viene ritornato:
 - il primo elemento con quella chiave
 - un qualsiasi elemento con quella chiave
 - tutti gli elementi con quella chiave.

Gestione dei duplicati in pile e code

Bisogna tener presente il criterio temporale:

- un elemento con chiave duplicata potrebbe essere considerato diverso in quanto inserito a un tempo diverso
- prevale la duplicazione sul tempo: bisogna decidere
 - se scartare l'elemento
 - estrarre quello già presente e inserire quello nuovo al tempo corrente
 - modificare l'elemento già presente lasciandone invariata la posizione
 - etc.

Debug di errori nella gestione della memoria in C

(G.Cabodi)

Questo documento ha come obiettivo fornire una semplice e concisa panoramica (non esaustiva) di problemi ed errori legati all'uso di puntatori e allocazione dinamica.

Premessa

La "correttezza" di un programma è un concetto "*relativo*", nel senso che un programma va considerato corretto in riferimento a un insieme di requisiti, obiettivi, formulati in modo più o meno rigoroso/formale. In altri termini, un programma è corretto quando realizza ciò che ci si aspetta e lo fa senza errori: la descrizione di modi/metodi per formulare requisiti/specifiche di un programma, così come prassi e approcci per testarne la correttezza *NON sono obiettivo di questo documento*.

Obiettivi

Gli errori considerati in questo documento sono quelli che di solito generano "*crash*" del programma (il programma si "pianta", si interrompe), a causa di uso errato di puntatori e/o memoria (allocata in modo sia statico che dinamico). Ci si propone quindi semplicemente di rimuovere le ragioni di un crash, non necessariamente di far sì che un programma realizzi effettivamente ciò per cui è stato scritto.

Attenzione: nonostante quanto appena detto, è sempre bene aver chiaro che cosa si vuole realizzare nel programma. Difficilmente su può pensare di rimuovere gli errori senza aver la consapevolezza (almeno in parte) di ciò che si vuol realizzare nel programma. Si noti poi che gli strumenti che aiutano ad analizzare gli errori di memoria richiedono una certa consapevolezza nell'utilizzo: possono quindi rappresentare un'arma a doppio taglio, se utilizzati con imperizia.

Il crash di un programma

Con il termine crash si intende una chiusura (terminazione) anomala (il programma si pianta/blocca/chiude improvvisamente) di un programma (il crash, più in generale, può essere anche a livello hardware, di sistema operativo, rete o altro), solitamente a causa di un problema hardware o software non gestito correttamente. I problemi legati alla gestione della memoria sono una delle possibili cause di crash, altre cause potrebbero essere: errori aritmetici (ad esempio divisione per 0), istruzioni illegali, violazioni di privilegio/sicurezza, problemi sul file system (ad esempio il tentativo di scrivere su un disco pieno), ecc.

Si veda ad esempio

[https://en.wikipedia.org/wiki/Crash_\(computing\)](https://en.wikipedia.org/wiki/Crash_(computing))

contenente una rapida e concisa panoramica sul concetto di crash.

Rimuovere una causa di crash implica di solito un ragionevole compromesso tra analizzare il programma e adottare opportune tecniche di debug: scoprire la vera causa di un crash non è infatti cosa banale, in quanto spesso un errore si manifesta non immediatamente, ma in modo indiretto e a valle dell'esecuzione di successive istruzioni.

La pagina web

<https://www.eventhelix.com/embedded/debugging-software-crashes/>

contiene una descrizione di diverse tipologie di software crash e di tecniche di debug utilizzabili per analizzare ed eventualmente correggere i problemi. Si noti che una parte significativa di tale pagina tratta di problemi legati alla memoria, descritti *successivamente* in questo documento.

Nei semplici programmi C affrontati nell'ambito del corso di Algoritmi e Strutture Dati, sono frequenti i crash causati da uso errato di puntatori e accessi a memoria.

Tipi di errori legati alla memoria.

I problemi di crash (software) legati all'uso della memoria, possono essere classificati in vari modi, ad esempio in base al tipo di memoria:

- globale
- heap
- stack

oppure in base al problema che scatena il crash

- memory corruption (scrittura in memoria e successivo utilizzo di dati non validi)
- storage violation (accesso illegale a memoria protetta e/o non accessibile)

Descrizioni delle diverse tipologie di errori sono reperibili, per memory corruption, su

https://en.wikipedia.org/wiki/Memory_corruption

<https://www.proggen.org/doku.php?id=security:memory-corruption:start>

mentre per la storage violation (e il cosiddetto “segmentation fault”, uno dei tipi di storage violation), su

<https://en.wikipedia.org/wiki/StorageViolation>

https://it.wikipedia.org/wiki/Errore_di_segmentazione

I “memory leak”

Si tratta di un caso molto particolare di errore, che tuttavia **non genera solitamente un crash**, ma che, potenzialmente, innescava un sottoutilizzo della risorsa memoria, e quindi un potenziale degrado di prestazioni. In generale, i “leak” (perdite) sono legati all'uso dell'heap e

dell'allocazione dinamica, e quindi sono connessi a una o più mancate chiamate alla funzione "free". Attenzione, i **crash** legati alle chiamate di "free" non sono a rigore dei leak/leakage, ma problemi di "memory corruption" o "illegal storage", in quanto causati da un tentativo di liberare una parte di memoria non allocata o già liberata in precedenza.

Per una descrizione del problema dei memory leak, si vedano ad esempio le pagine web

https://it.wikipedia.org/wiki/Memory_leak

https://en.wikipedia.org/wiki/Memory_leak

Fare debug di errori di memoria

Obiettivo di ognuna delle descrizioni sopra citate dovrebbe essere quello di capire le tipologie dei potenziali problemi, per cercare di evitarli, tuttavia non sempre questo è possibile. Ciò fa sì che, in molti casi, possa essere estremamente utile, oltre alle buone norme di programmazione e di revisione del codice scritto, l'utilizzo di opportuni strumenti di debug, orientati a individuare e rimuovere errori di memoria. Un debugger orientato a problemi di accesso a memoria è purtroppo uno strumento "intrusivo", nel senso che richiede di attivare, durante l'esecuzione di un programma, moduli software in grado di monitorare e verificare il corretto utilizzo delle varie aree di memoria su cui lavora un programma. In pratica il programma "gira" in un contesto diverso da quello privo di debugger e ciò talvolta può produrre tracce di esecuzione diverse, tra le modalità con e senza debug.

Una panoramica di debugger per problemi di memoria è reperibile su

https://en.wikipedia.org/wiki/Memory_debugger#List_of_memory_debugging_tools

Descrizioni di singoli strumenti sono consultabili su

<https://valgrind.org/docs/manual/mc-manual.html>

<https://en.wikipedia.org/wiki/AddressSanitizer>

https://www.cprogramming.com/tutorial/memory_debugging_parallel_inspector.html

Quasi tutti gli strumenti di debug della memoria includono una parte orientata ai leak. Una discussione sul problema specifico dei leak può ad esempio essere letta su

<https://www.bogotobogo.com/cplusplus/CppCrashDebuggingMemoryLeak.php>

Consigli pratici

Tra i vari strumenti di debug ci si limita a suggerire, ad un programmatore non ancora esperto, due degli strumenti (non commerciali e ampiamente disponibili) citati nel paragrafo precedente:

- valgrind
- address sanitizer

Valgrind

Si tratta di uno strumento open source, sviluppato in ambito Linux, ma disponibile su piattaforme Windows e Mac. Non è parte di un debugger, ma uno strumento esterno, un "profiler", che ha come scopo raccogliere statistiche di esecuzione di un programma, al fine di caratterizzarne le prestazioni.

Ad esempio, un programma eseguibile es2, avente due argomenti al main (ad esempio "a b") potrebbe essere eseguito, mediante valgrind, con il comando:

```
valgrind es2 a b
```

Valgrind include molteplici strumenti, quello adibito a verificare problemi di memoria si chiama "memcheck". Lo si può attivare, ad esempio (attenzione al doppio trattino --), con

```
valgrind --tool=memcheck es2 a b
```

Problema non trascurabile, per il programmatore inesperto, può essere l'interpretazione dei dati visualizzati da valgrind, nonché il fatto che l'individuazione di errori non è garantita al 100%.

Valgrind può essere integrato all'interno di altri strumenti di debug (es. Clion), a patto di effettuarne correttamente l'installazione.

AddressSanitizer

Si tratta di uno strumento sviluppato da Google, integrato nei compilatori Clang, GCC e Xcode, in modo tale che un programma, a patto di attivare le opportune opzioni di compilazione/link, venga eseguito in una forma in cui vengono controllati gli accessi alla memoria. Si noti che sono presenti più di un sanitizer (esiste ad esempio anche il leak sanitizer, ma non tutti sono presenti su tutte le piattaforme e i sistemi operativi)

AddressSanitizer può essere attivato ed utilizzato ad esempio, in CodeBlocks, Clion e Xcode, come descritto nel seguente link:

Compilazione in linea con GCC

Si può abilitare il sanitizer direttamente in compilazione sulla linea di comando gcc (quindi in ambito Linux, Mac, oppure Windows con ambiente Cygwin o WSL), aggiungendo le opzioni di compilazione:

```
-g -fsanitize=address
```

Se ad esempio si volesse generare l'eseguibile es1 a partire dai file es1.c e ts.c, si potrebbero usare i comandi:

```
gcc -c -g -fsanitize=address ts.c (compila ts.c)
gcc -c -g -fsanitize=address es1.c (compila es1.c)
gcc -o es1 -g -fsanitize=address ts.o es1.o (link -> eseguibile es1)
```

oppure l'unico comando (per compilazione e link insieme)

```
gcc -o es1 -g -fsanitize=address ts.c es1.c
```

Come conseguenza, al termine dell'esecuzione vengono visualizzate statistiche sull'uso della memoria ed eventuali anomalie/errori

CodeBlocks

Occorre configurare il compilatore in modo da attivare il tool e fornire sufficienti informazioni al debugger. Supponendo di utilizzare come compilatore GCC e come debugger GDB, occorre

- modificare le opzioni di compilazione in “project->build options...->compiler settings” (oppure “settings->compiler”, qualora si voglia rendere l’opzione globale per tutti i progetti), selezionato il compilatore “GNU GCC compiler”, nella sezione “other compiler options”, inserire le opzioni:
-ggdb -fsanitize=address
- modificare le opzioni di link (“project->build options...->compiler settings”, oppure “settings->compiler”, quindi selezionare “linker settings”): nella sezione “other linker options”, inserire:
-lasan
- In caso di crash, le informazioni sull’eventuale problema legato alla memoria vengono visualizzate sulla finestra di output (la console). Per evitare che, al termine dell’esecuzione, la finestra scompaia immediatamente, si consiglia di abilitare l’opzione “Pause when execution ends” nella finestra “project->properties->Build targets”, sotto la selezione di “Type: Console application”

Clion

Le istruzioni su come abilitare i Google Sanitizers sono visibili su

<https://www.jetbrains.com/help/clion/google-sanitizers.html>

In breve:

- In windows occorre utilizzare WSL-Ubuntu (vedere <https://www.jetbrains.com/help/clion/how-to-use-wsl-development-environment-in-product.html>)
- in CMakeLists.txt occorre aggiungere una riga:
`set(CMAKE_C_FLAGS=" ${CMAKE_C_FLAGS} -fsanitize=address -g")`
Attenzione a usare eventualmente CMAKE_CXX_FLAGS nel caso si stia utilizzando il compilatore C++
 - in alternativa, in Settings -> Build, Execution, Deployment -> CMake -> CMake options, aggiungere
`-DCMAKE_C_FLAGS="${CMAKE_C_FLAGS} -fsanitize=address"`
- Nelle preferenze (Settings -> Build, Execution, Deployment -> Dynamic Analysis Tools -> Sanitizers) abilitare "Use visual representation for Sanitizer's output"

ATTENZIONE: evitare il copia/incolla diretto dal testo sopra, in quanto i doppi apici vengono spesso codificati in modo non compatibile con quanto richiesto da CMAKE!!! Meglio riscrivere oppure fare un copia incolla passando da un editor di puro testo.

Xcode

In Xcode è sufficiente abilitare address sanitizer in "Product->Scheme->Edit scheme", dove, per la configurazione Run oppure Test, tra le opzioni "Diagnostics", è possibile configurare "Runtime Sanitization"

Richiami di Matematica Discreta: grafi e alberi

Paolo Camurati



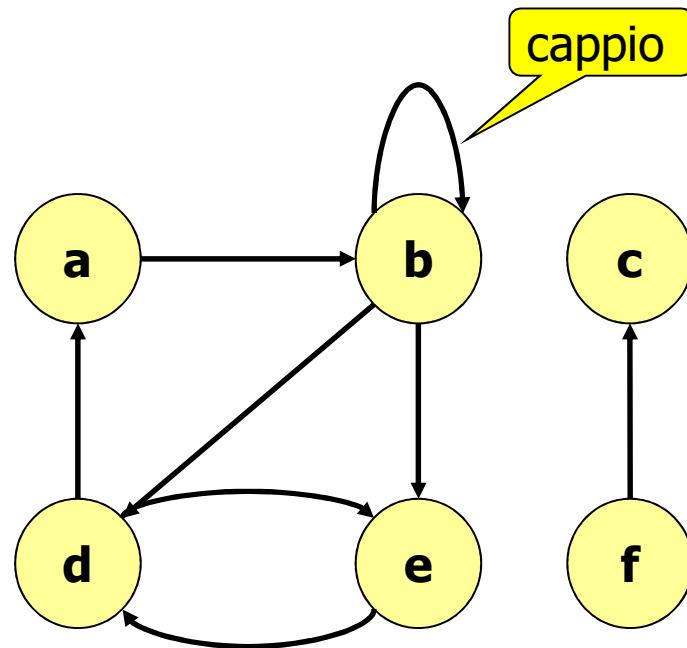
Grafi

- Definizione: $G = (V, E)$
 - V : insieme finito e non vuoto di vertici (contenenti dati semplici o composti)
 - E : insieme finito di archi, che definiscono una relazione binaria su V
- Grafi orientati/non orientati:
 - orientati: arco = coppia ordinata di vertici $(u, v) \in E$ e $u, v \in V$
 - non orientati: arco = coppia non ordinata di vertici $(u, v) \in E$ e $u, v \in V$

Grafi come modelli

Dominio	Vertice	Arco
comunicazioni	telefono, computer	fibra ottica, cavo
circuiti	porta, registro, processore	filo
meccanica	giunto	molla
finanza	azioni, monete	transazioni
trasporti	aeroporto, stazione	corridoio aereo, linea ferroviaria
giochi	posizione sulla scacchiera	mossa lecita
social networks	persona	amicizia
reti neurali	neurone	sinapsi
composti chimici	molecola	legame

Esempio: grafo orientato

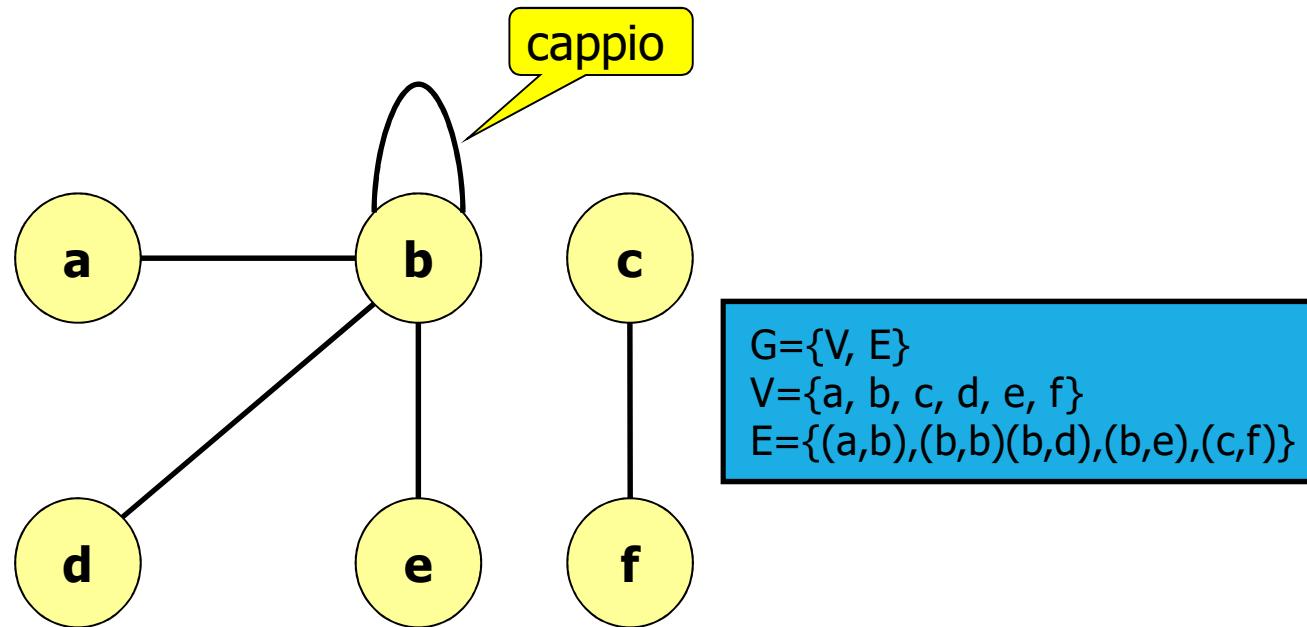


```
G={V, E}  
V={a, b, c, d, e, f}  
E={(a,b),(b,b),(b,d),(b,e),  
(d,a),(d,e),(e,d),(f,c)}
```

NB: in alcuni contesti i cappi possono essere vietati.

Se il contesto ammette i cappi, ma il grafo ne è privo, esso si dice **SEMPLICE**.

Esempio: grafo non orientato



NB: in alcuni contesti i cappi possono essere vietati.

Se il contesto ammette i cappi, ma il grafo ne è privo, esso si dice **SEMPLICE**.

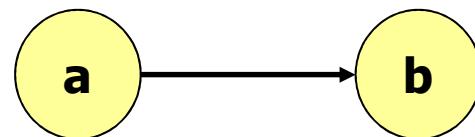
Incidenza e adiacenza

Arco (a, b):

- entrante (incidente) nel vertice b
 - uscente da vertice a
 - insistente sui vertici a e b
- } solo per archi orientati

Vertici a e b **adiacenti**:

$$a \rightarrow b \Leftrightarrow (a, b) \in E$$

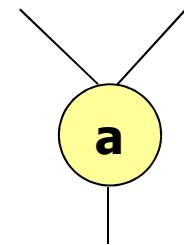


Grado di un vertice

grafo non orientato:

- $\text{degree}(a)$ = numero di archi incidenti

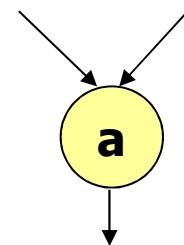
$$\text{degree}(a) = 3$$



grafo orientato:

- $\text{in_degree}(a)$ =numero di archi entranti
- $\text{out_degree}(a)$ =numero di archi uscenti
- $\text{degree}(a)=\text{in_degree}(a) + \text{out_degree}(a)$.

$$\begin{aligned}\text{in_degree}(a) &= 2 \\ \text{out_degree}(a) &= 1 \\ \text{degree}(a) &= 3\end{aligned}$$



Cammini e raggiungibilità

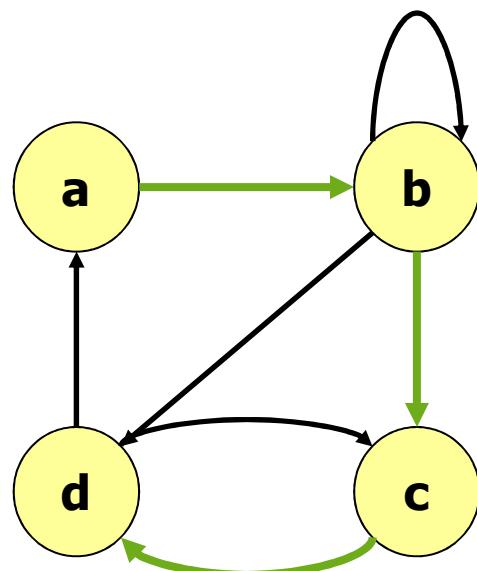
Cammino p : $u \rightarrow_p u'$ in $G = (V, E)$:

$\exists (v_0, v_1, v_2, \dots, v_k) \mid$

$u = v_0, u' = v_k, \forall i = 1, 2, \dots, k \ (v_{i-1}, v_i) \in E.$

- k = **lunghezza** del cammino.
- u' è **raggiungibile** da $u \Leftrightarrow \exists p: u \rightarrow_p u'$
- cammino p **semplice**: $(v_0, v_1, v_2, \dots, v_k) \in p$ distinti.

Esempio



$$G = (V, E)$$

$$p: a \xrightarrow{p} d : (a, b), (b, c), (c, d)$$

$$k = 3$$

d è raggiungibile da a
p semplice.

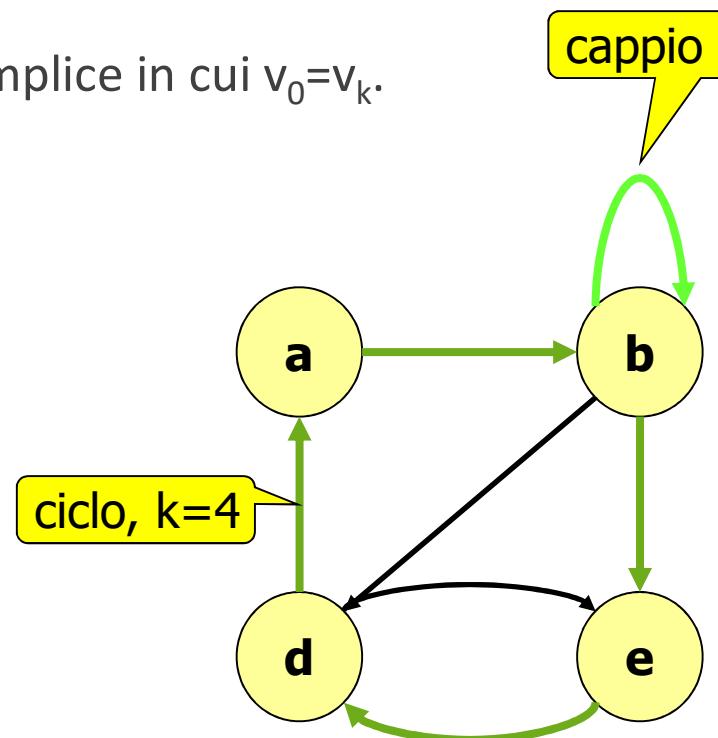
Cicli

Ciclo = cammino in cui $v_0=v_k$.

Ciclo semplice = cammino semplice in cui $v_0=v_k$.

Cappio = ciclo di lunghezza 1.

Un grafo senza cicli = **aciclico**.



Connessione nei grafi non orientati

Grafo non orientato **connesso**:

$$\forall v_i, v_j \in V \quad \exists p \quad v_i \rightarrow_p v_j$$

Componente connessa: sottografo connesso massimale (= \nexists sottoinsiemi per cui vale la proprietà che lo includono).

Grafo non orientato connesso: una sola componente connessa.

Connessione nei grafi orientati

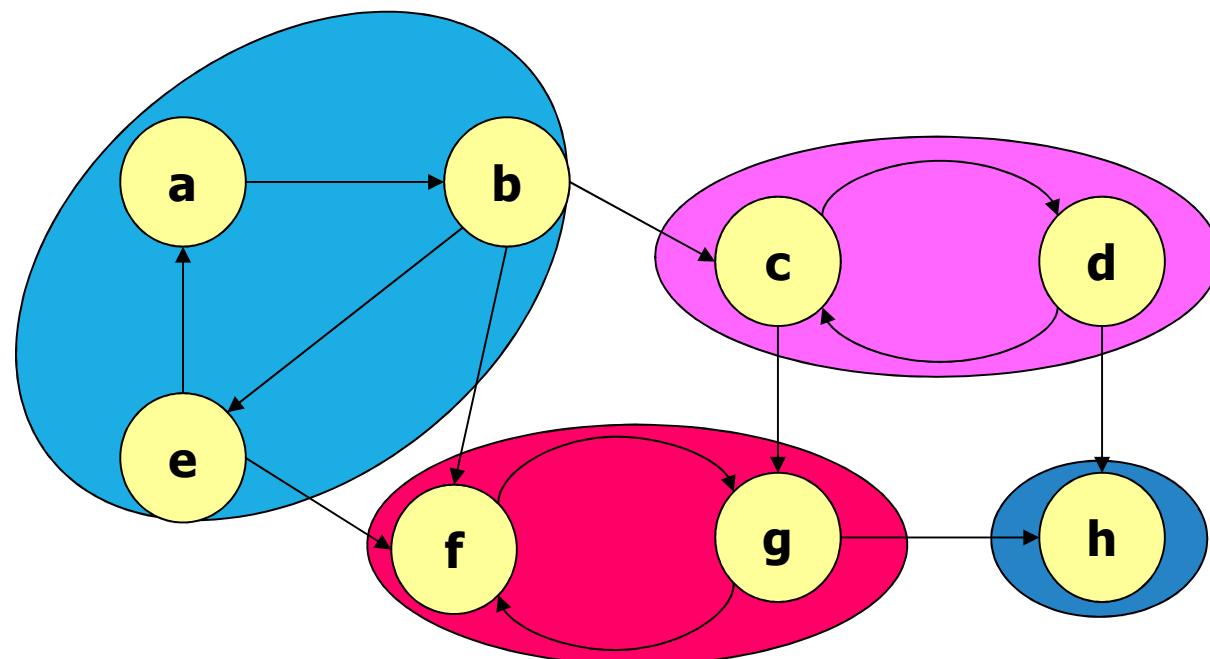
Grafo orientato **fortemente connesso**:

$$\forall v_i, v_j \in V \quad \exists p, p' \quad v_i \rightarrow_p v_j \text{ e } v_j \rightarrow_{p'} v_i$$

Componente **fortemente connessa**: sottografo fortemente connesso massimale.

Grafo orientato fortemente connesso: una sola componente fortemente connessa.

Esempio



Grafo completo $K_{|V|}$

Definizione:

$$\forall v_i, v_j \in V \quad \exists (v_i, v_j) \in E$$

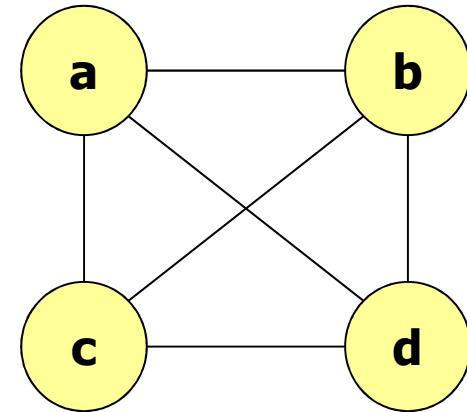
Quanti archi?

grafo completo non orientato:

$|E|$ = numero di **combinazioni** di $|V|$ elementi a 2 a 2

$$|E| = \frac{|V|!}{(|V|-2)!*2!} = \frac{|V|*(|V|-1)*(|V|-2)!}{(|V|-2)!*2!} = \frac{|V|*(|V|-1)}{2}$$

l'ordine non conta



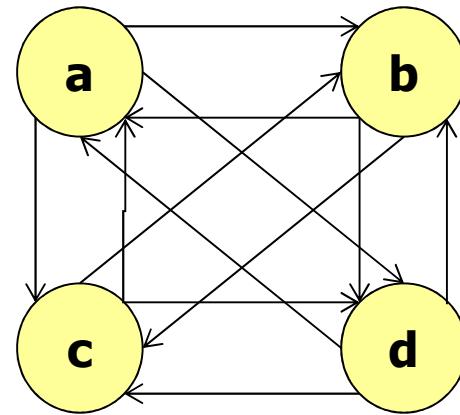
Quanti archi?

grafo completo orientato:

$|E|$ = numero di **disposizioni** di $|V|$ elementi a 2 a 2

$$|E| = \frac{|V|!}{(|V|-2)!} = \frac{|V| * (|V|-1) * (|V|-2)!}{(|V|-2)!} = |V| * (|V|-1)$$

l'ordine conta



Grafi densi/sparsi

Dato grafo $G = (V, E)$

$|V|$ = cardinalità dell'insieme V

$|E|$ = cardinalità dell'insieme E

e il corrispondente grafo completo con $|V|$ vertici $K_{|V|}$

la densità: $d(G) = |E_G| / |E_K|$

- grafo denso: $|E| \cong |V|^2$
- grafo sparso: $|E| \ll |V|^2$

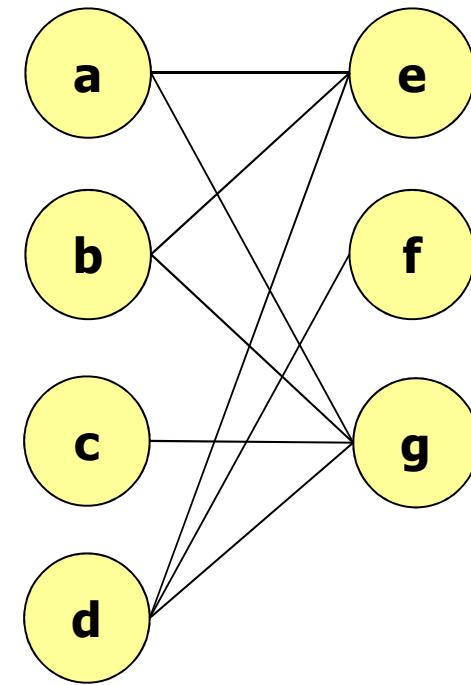
Grafo bipartito

Definizione:

Grafo non orientato in cui l'insieme V può essere partizionato in 2 sottoinsiemi V_1 e V_2 , tali per cui

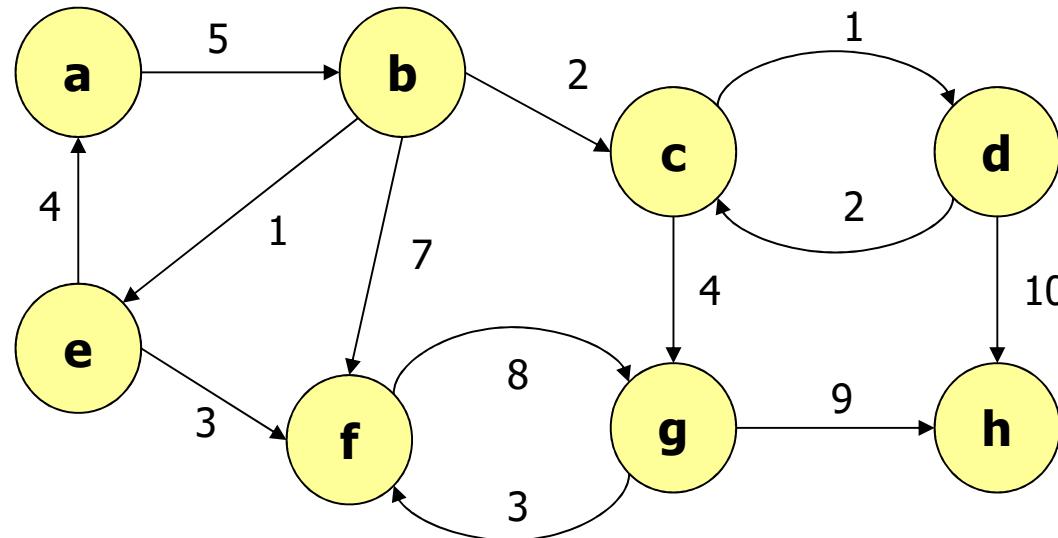
$$\forall (v_i, v_j) \in E$$

$$\begin{aligned} v_i &\in V_1 \text{ && } v_j \in V_2 \\ || \\ v_j &\in V_1 \text{ && } v_i \in V_2 \end{aligned}$$

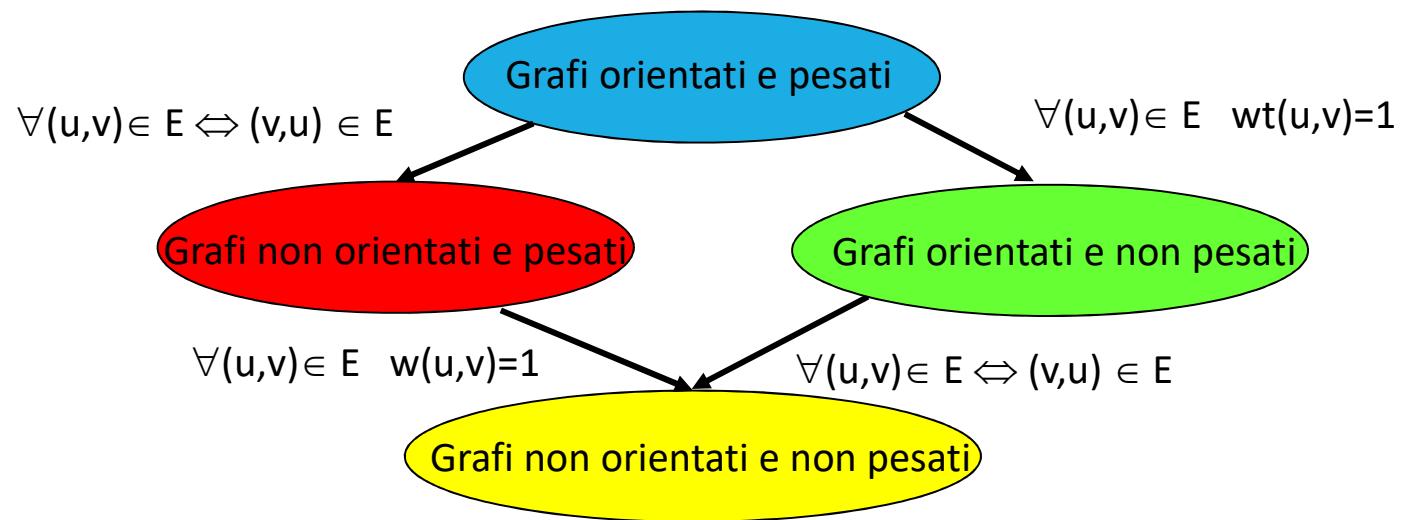


Grafo pesato

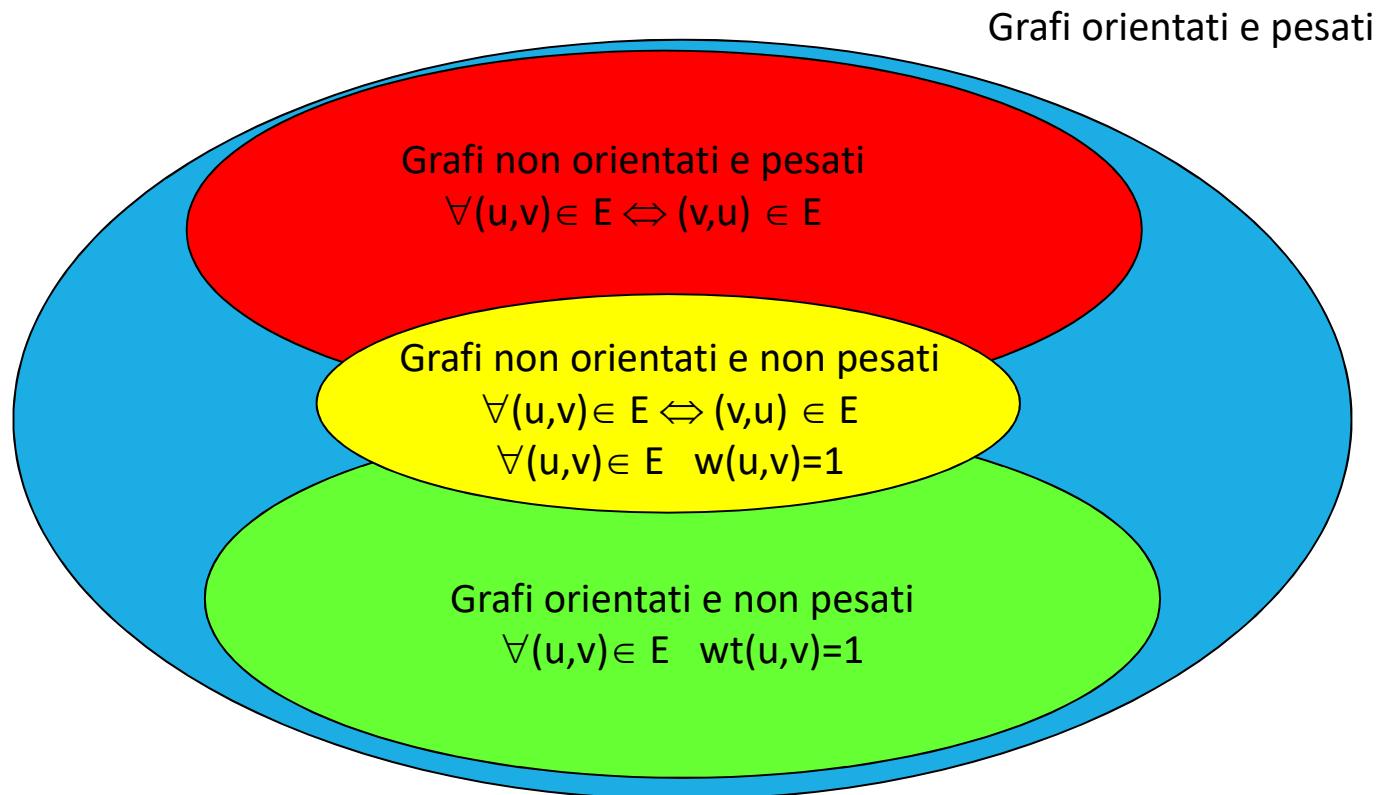
$\exists \text{wt} : E \rightarrow \mathbb{R} \cup \{-\infty, +\infty\} \mid \text{wt}(u,v) = \text{peso dell'arco } (u, v)$



Tipologie (visione a grafo)

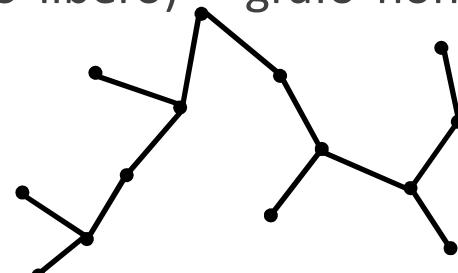


Tipologie (visione a insieme)

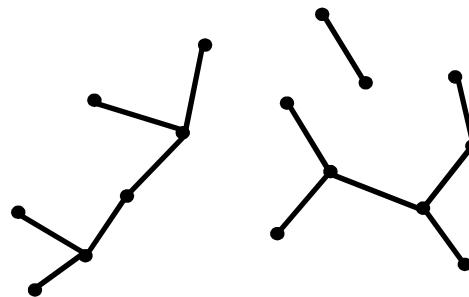


Alberi non radicati (liberi)

Albero non radicato (o libero) = grafo non orientato, connesso, aciclico



Forest = grafo non orientato, aciclico



Proprietà

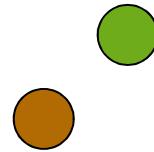
$G = (V, E)$ grafo non orientato $|E|$ archi, $|V|$ nodi:

- G = albero non radicato
- ogni coppia di nodi connessa da un unico cammino semplice
- G connesso, la rimozione di un arco lo sconnette
- G connesso e $|E| = |V| - 1$
- G aciclico e $|E| = |V| - 1$
- G aciclico, l'aggiunta di un arco introduce un ciclo.

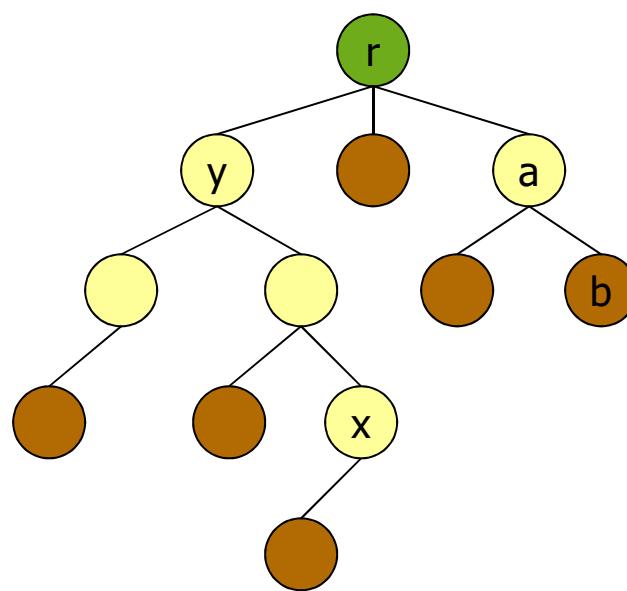
Alberi radicati

\exists nodo r detto radice

- che induce una relazione di parentela tra nodi:
 - y antenato di x se y appartiene al cammino da r a x. x discendente di y
 - antenato proprio se $x \neq y$
 - padre/figlio: nodi adiacenti
- radice: no padre
- foglie no figli



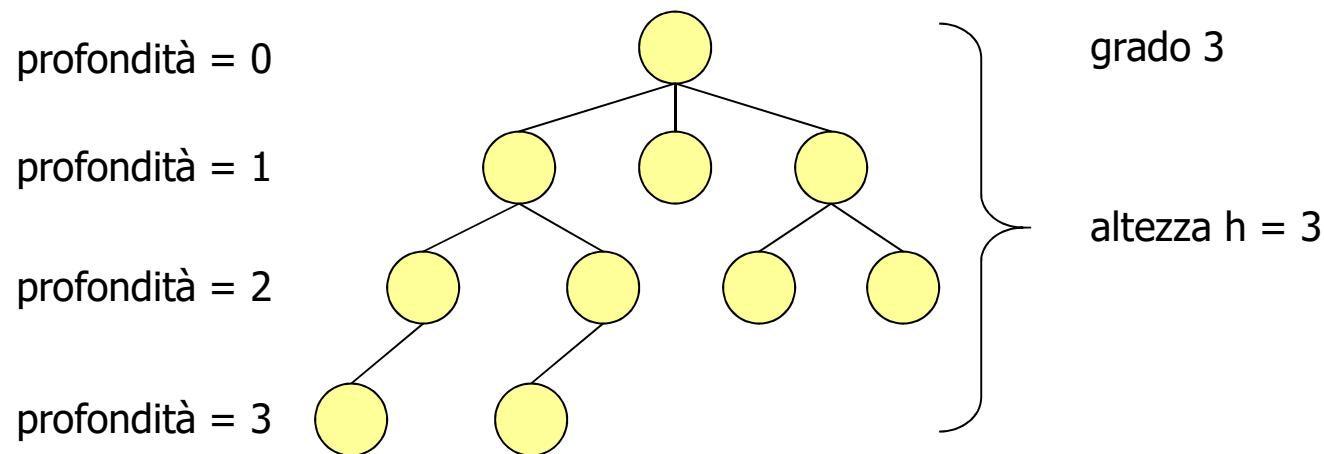
Esempio



r radice
y antenato proprio di x
x discendente proprio di y
a padre di b
b figlio di a

Proprietà di un albero T

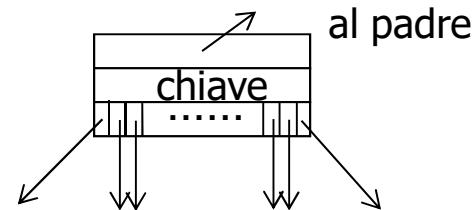
- **grado(T)** = numero max di figli
- **profondità(x)** = lunghezza del cammino da r a x
- **altezza(T)** = profondità massima.



Rappresentazione di alberi

Rappresentazione di un nodo di un albero di grado(T) = k

- puntatore al padre, chiave, k puntatori ai k figli



k puntatori ai k figli, eventualmente a NULL

Inefficiente in termini di spazio se solo pochi nodi hanno davvero grado k (spazio per tutti i k puntatori allocato, ma molti a NULL).

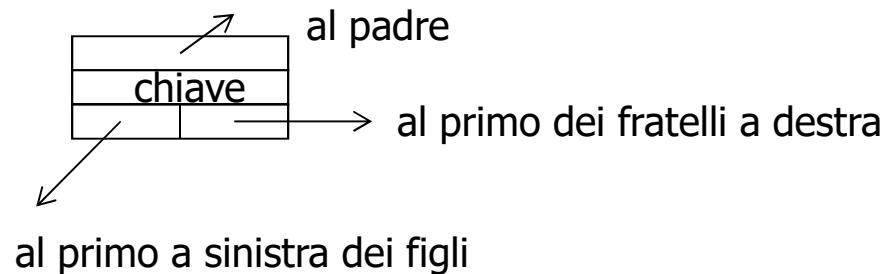
Valutazione:

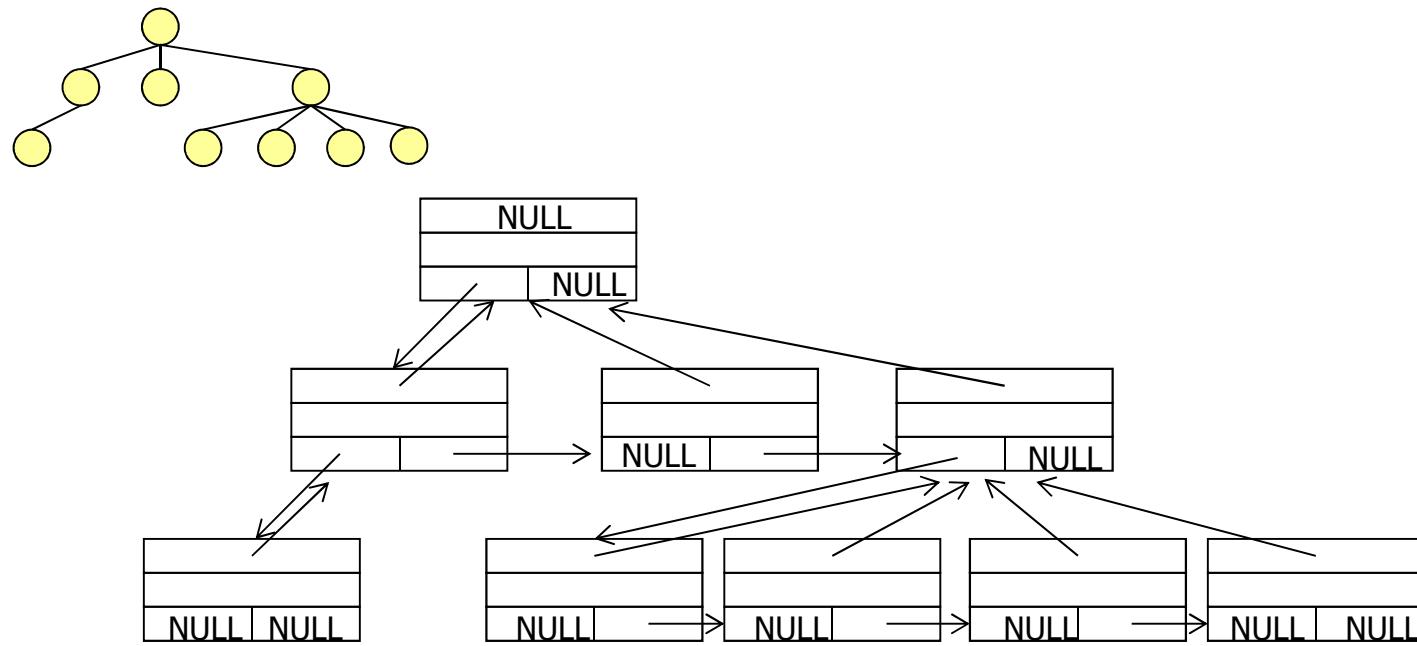
- inefficiente in termini di spazio se solo pochi nodi hanno davvero grado k (spazio per tutti i k puntatori allocato, ma molti a NULL)
- efficiente in termini di tempo (da padre a figlio e viceversa con costo $O(1)$)

Rappresentazione left-child right sibling

Rappresentazione di un nodo di un albero di grado(T) = k

- puntatore al padre, chiave, 1 puntatore al figlio sinistro, 1 puntatore al fratello a destra





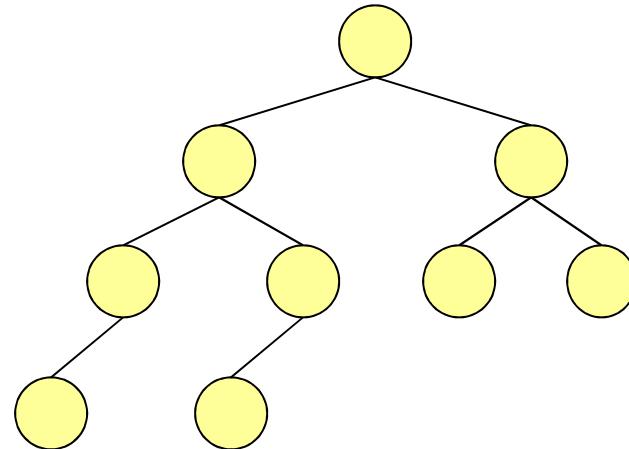
Valutazione:

- efficiente in termini di spazio: sempre solo 2 puntatori, indipendentemente dal grado dell'albero
- inefficiente in termini di tempo (da padre a figlio e viceversa con costo $O(k)$).

Albero binario

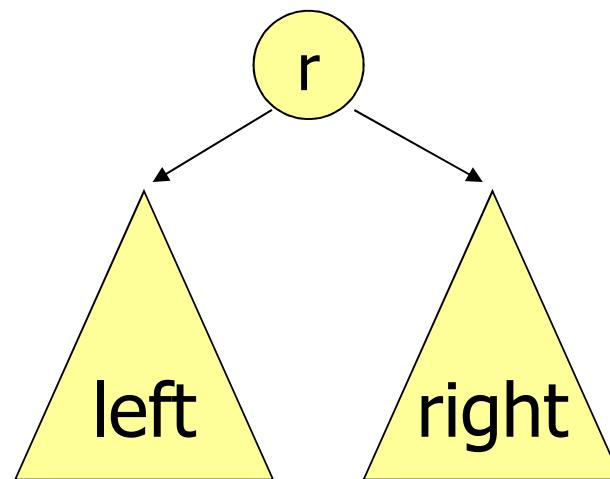
Definizione:

- Albero di grado 2: ogni nodo ha 0, 1 o 2 figli



Definizione ricorsiva:

- Un albero binario T è:
 - insieme di nodi vuoto
 - una terna formata da radice, sottoalbero sinistro, sottoalbero destro.



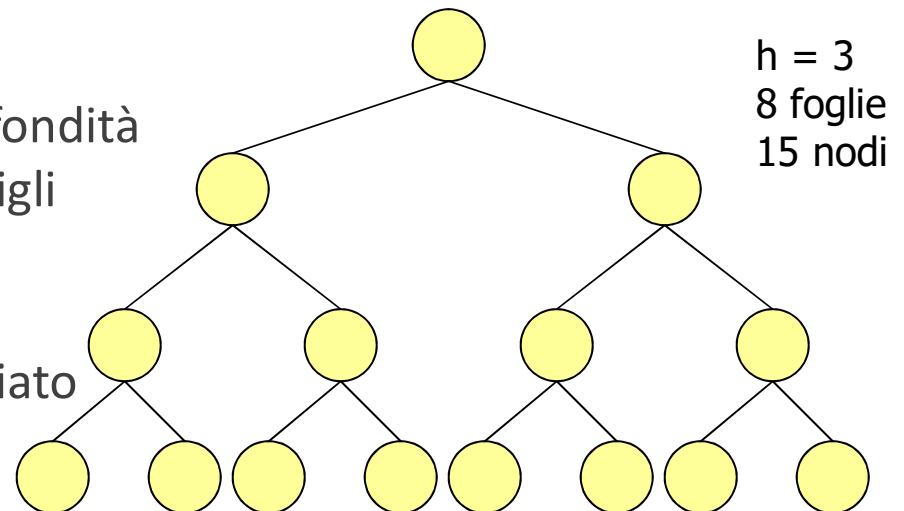
Albero binario completamente bilanciato (pieno)

Due condizioni:

- tutte le foglie hanno la stessa profondità
- ogni nodo o è una foglia o ha 2 figli

Albero binario completamente bilanciato
(pieno) di altezza h :

- numero di foglie: 2^h
- numero di nodi: $\sum_{0 \leq i \leq h} 2^i = 2^{h+1} - 1$



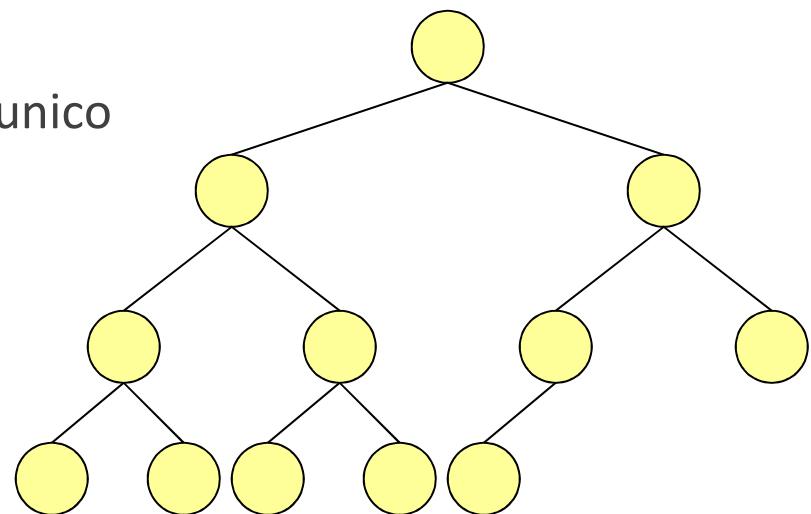
progressione geometrica
finita di ragione 2



Albero binario completo (a sinistra)

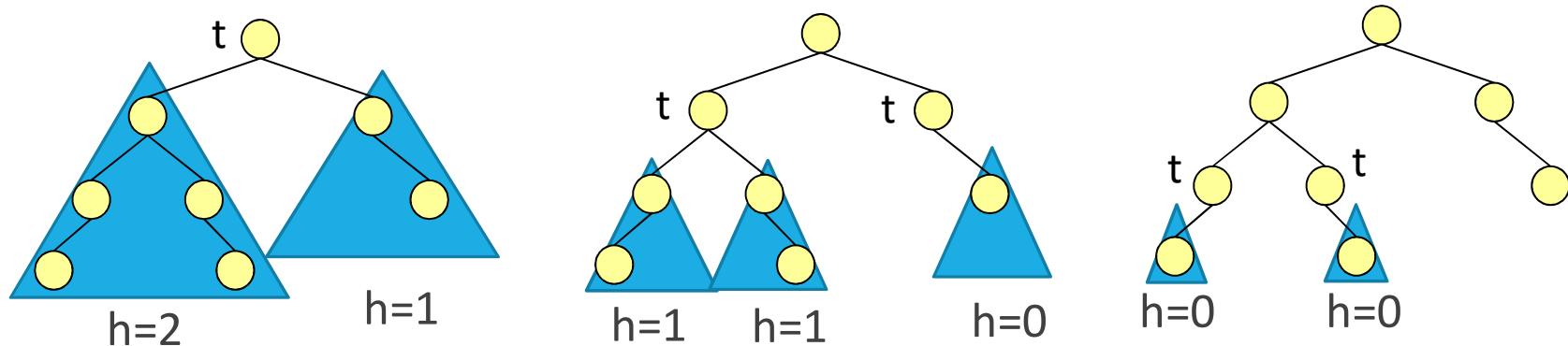
Tutti i livelli sono completi (hanno tutti i nodi) eccetto l'ultimo che è riempito da sinistra a destra.

Dato un numero di nodi n esiste ed è unico l'albero completo (a sinistra).

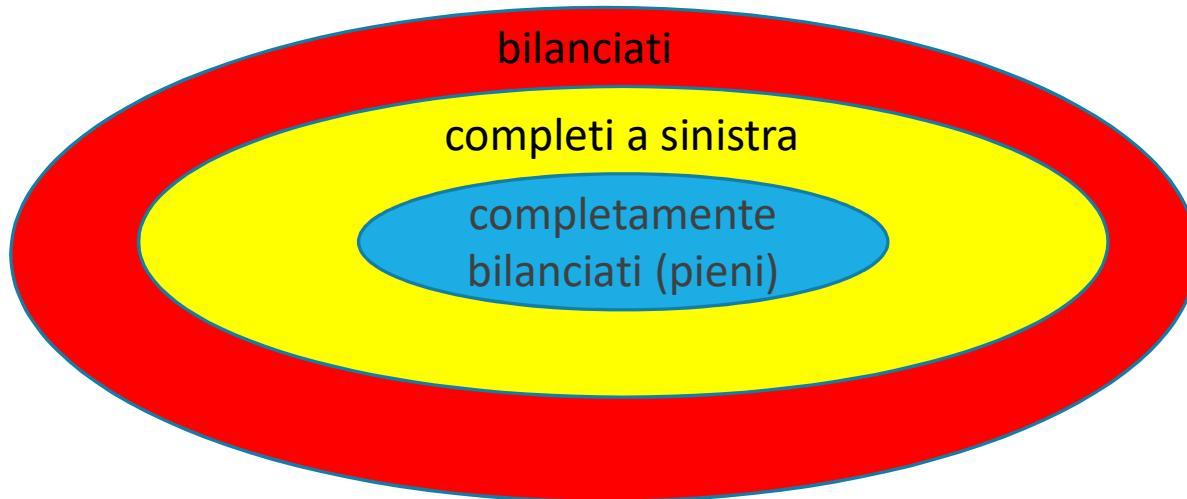


Albero binario bilanciato (in altezza)

Un albero è **bilanciato in altezza** (in breve **bilanciato**) se e solo se, per ogni sottoalbero t radicato in un suo nodo, l'altezza del sottoalbero sinistro di t differisce di al più di 1 dall'altezza del sottoalbero destro di t .

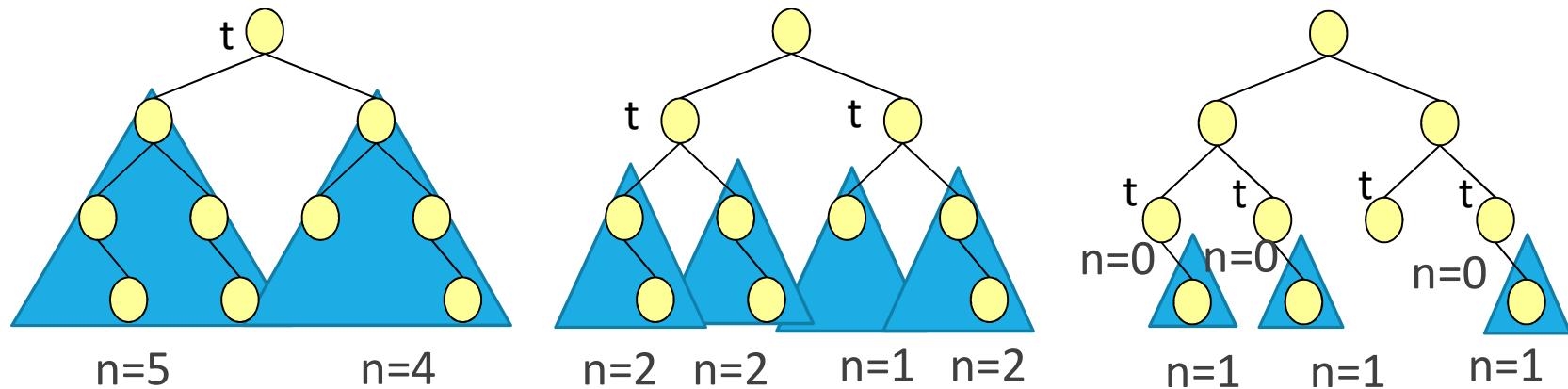


Gli alberi completamente bilanciati (pieni) sono un sottoinsieme proprio degli alberi completi (a sinistra) che a loro volta sono un sottoinsieme proprio degli alberi bilanciati.



Albero binario bilanciato in nodi

Un albero binario è **bilanciato in nodi** se e solo se, per ogni sottoalbero t radicato in un suo nodo, il numero di nodi del sottoalbero sinistro di t differisce di al più di 1 dal numero di nodi del sottoalbero destro di t .



Sequenze lineari

- Insieme finito di elementi disposti consecutivamente in cui a ogni elemento è associato univocamente un indice
 - $a_0, a_1, \dots, a_i, \dots, a_{n-1}$
- Sulle coppie di elementi è definita una relazione predecessore/successore:
 - $a_{i+1} = \text{succ}(a_i)$ $a_i = \text{pred}(a_{i+1})$

Memorizzazione e accesso

Vettori o array:

- Modalità di memorizzazione: dati **contigui** in memoria
- **Accesso diretto**:
 - dato l'indice i , si accede all'elemento a_i senza dover scorrere la sequenza lineare
 - il costo dell'accesso non dipende dalla posizione dell'elemento nella sequenza lineare, quindi è **$O(1)$**

Liste:

- Modalità di memorizzazione: dati **non contigui** in memoria
- **Accesso sequenziale:**
 - dato l'indice i , si accede all'elemento a_i scorrendo la sequenza lineare a partire da uno dei suoi 2 estremi, solitamente quello di SX
 - il costo dell'accesso dipende dalla posizione dell'elemento nella sequenza lineare, quindi è **$O(n)$** nel caso peggiore

Operazioni sulle liste

- **ricerca** di un elemento il cui campo **chiave di ricerca** è uguale a una chiave data
- **inserzione** di un elemento:
 - **in testa** alla lista non ordinata
 - **in coda** alla lista non ordinata
 - **nella posizione** tale da garantire l'invarianza della proprietà di ordinamento per una lista ordinata
- **estrazione** di un elemento:
 - che si trova **in testa** alla lista non ordinata
 - che ha **un campo** con contenuto uguale a quello di una chiave di cancellazione (tale operazione richiede solitamente una ricerca preventiva dell'elemento da cancellare).

Collezioni di dati

Code generalizzate: collezioni di oggetti (dati) con operazioni principali:

- **Insert:** inserisci un nuovo oggetto nella collezione
- **Search:** ricerca se un oggetto è nella collezione
- **Delete:** cancella un oggetto della collezione

Altre operazioni:

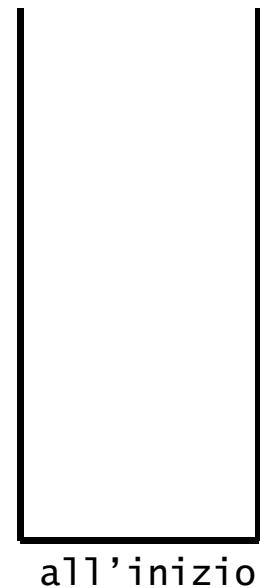
- Inizializzare la coda generalizzata
- Conteggio oggetti (o verifica collezione vuota)
- Distruzione della coda generalizzata
- Copia della coda generalizzata

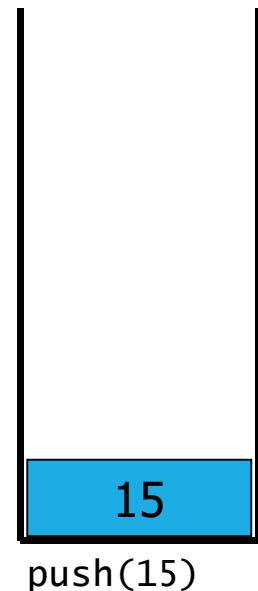
Criteri per operazione di **Delete**:

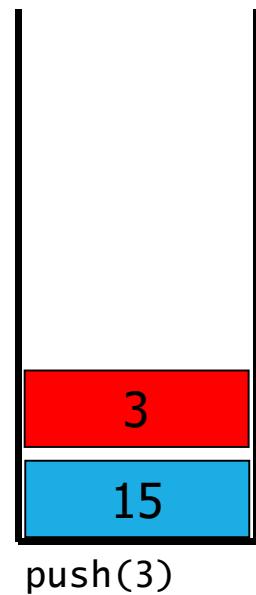
- **cronologico**:
 - estrazione dell'elemento inserito più recentemente
 - politica LIFO: Last-In First-Out
 - **stack o pila**
 - inserzione (push) ed estrazione (pop) dalla testa
 - estrazione dell'elemento inserito meno recentemente
 - politica FIFO: First-In First-Out
 - **queue o coda**
 - inserzione (enqueue) in coda (tail) ed estrazione (dequeue) dalla testa (head)
- **priorità**:
 - l'inserzione garantisce che, estraendo dalla testa, si ottenga il dato a priorità massima (o minima)
 - **coda a priorità**

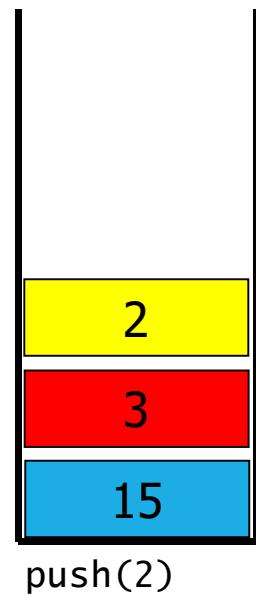
- **contenuto:**
 - l'estrazione ritorna un contenuto secondo determinati criteri
 - **tabella di simboli** (più avanti nel Corso).

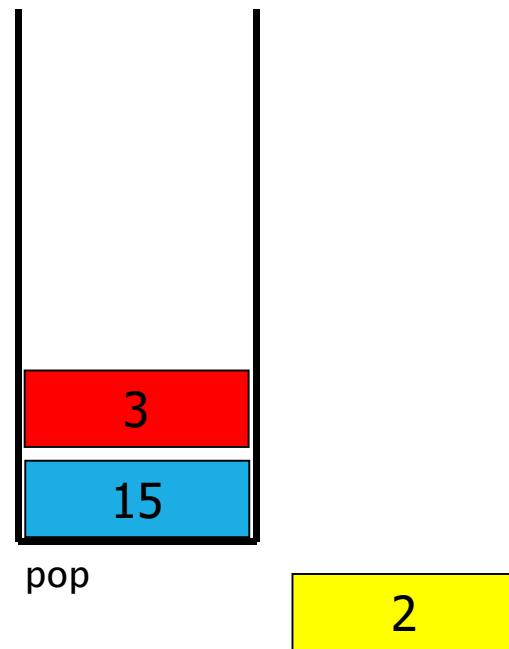
Esempio: politica LIFO (pila-stack)



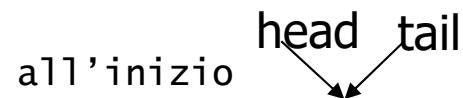


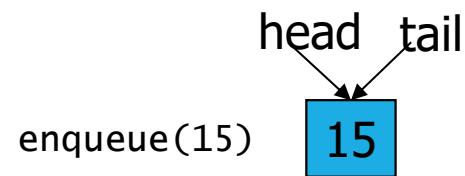


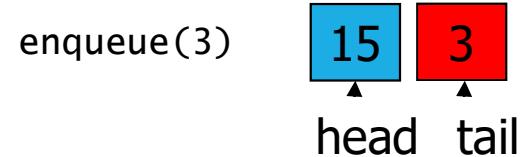




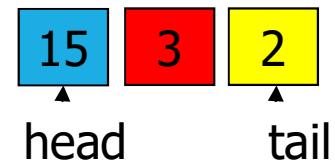
Esempio: FIFO (coda-queue)





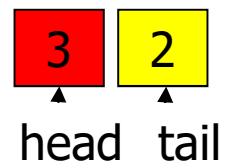


enqueue(2)



dequeue

15

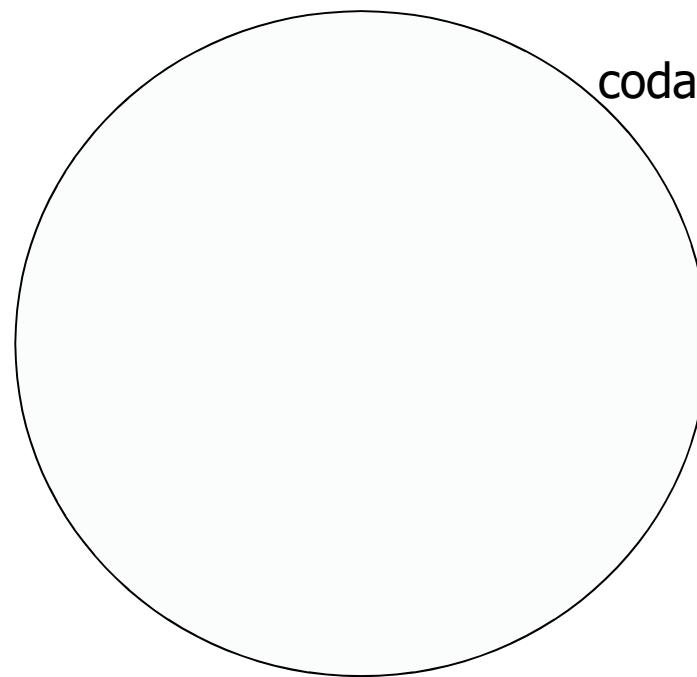


Esempio: coda a priorità

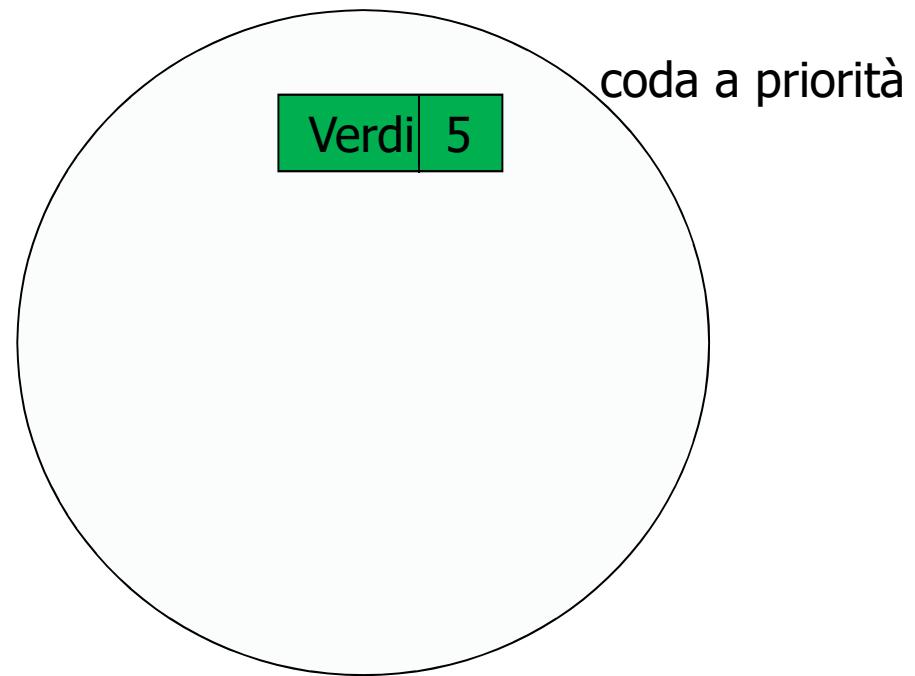
dato	Rossi	15
campo (cognome)		campo (priorità)

all'inizio

coda a priorità

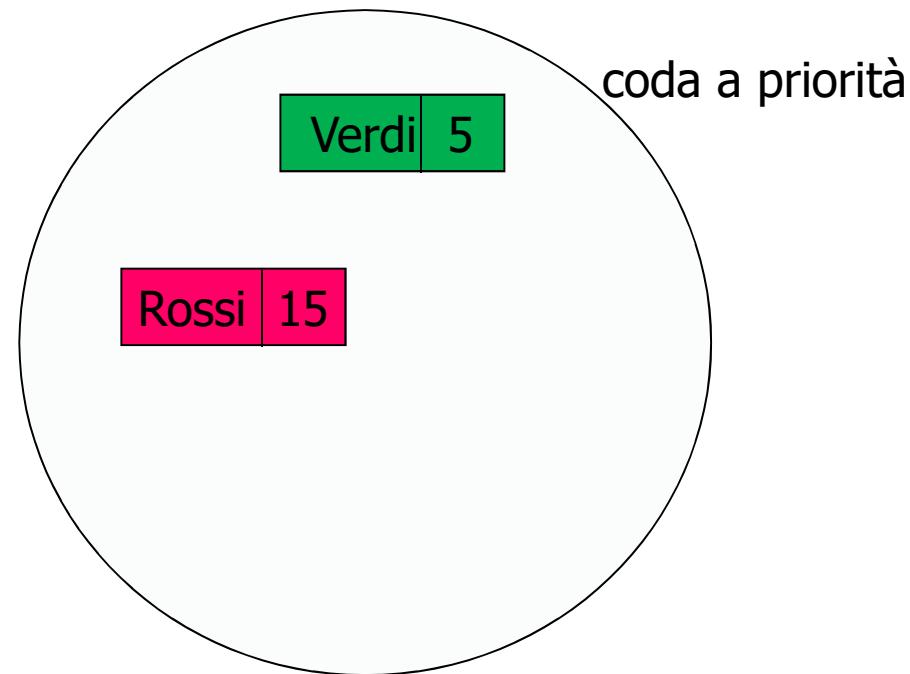


`insert(verdi, 5)`



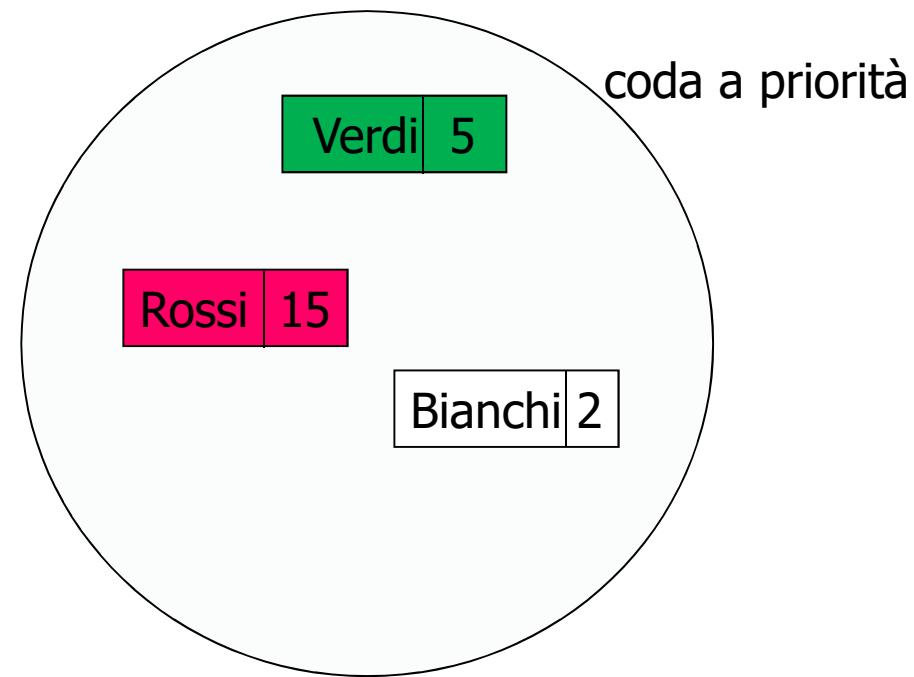
coda a priorità

`insert(Rossi, 15)`



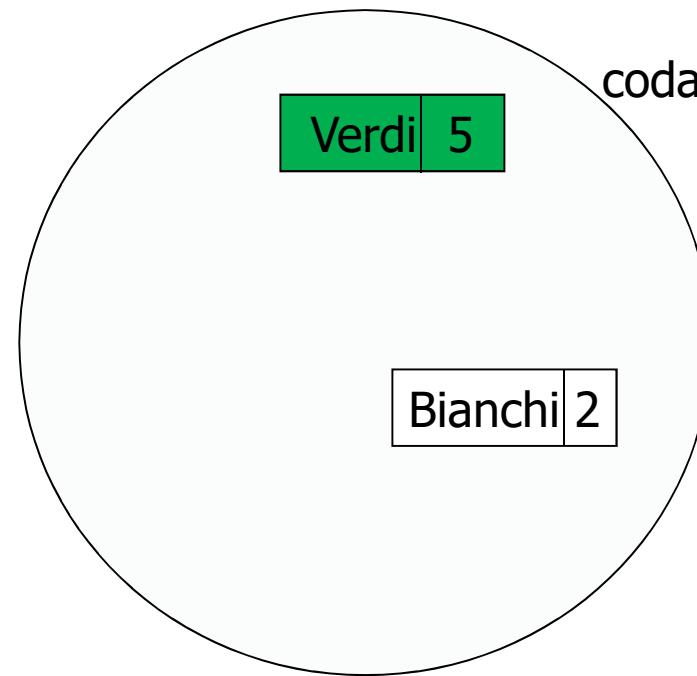
coda a priorità

```
insert(Bianchi, 2)
```



extract

Rossi | 15



coda a priorità

Riferimenti

- Grafi:
 - Cormen 5.4
 - Sedgewick Part 5 17.1
- Alberi:
 - Cormen 5.5
 - Sedgewick 5.4
- Liste, pile, code, code a priorità:
 - Cormen 10.1, 10.2, 6.5
 - Sedgewick 3.3, 4.2, 4.6, 9
- Code generalizzate:
 - Sedgewick 4.1

Prerequisiti per Algoritmi e Strutture Dati da Tecniche di Programmazione

Paolo Camurati



- Algoritmo = strategia
- Approccio al problem-solving
- Tipologie di problemi
- Classi di complessità
- Analisi asintotica di complessità di caso peggiore
- Online connectivity
- Matematica discreta
- Ordinamenti

Algoritmo = strategia

- Scegliere un algoritmo spesso significa individuare la miglior strategia per risolvere un problema
- La scelta si basa su:
 - classificazione del problema in base a problemi simili noti
 - conoscenza di algoritmi noti in letteratura
 - conoscenza delle operazioni elementari e delle funzioni di libreria fornite dal linguaggio C
 - conoscenza dei costrutti linguistici (tipi di dato, costrutti condizionali e iterativi)
 - esperienza su tipi diversi di problemi.

Approccio al problem-solving

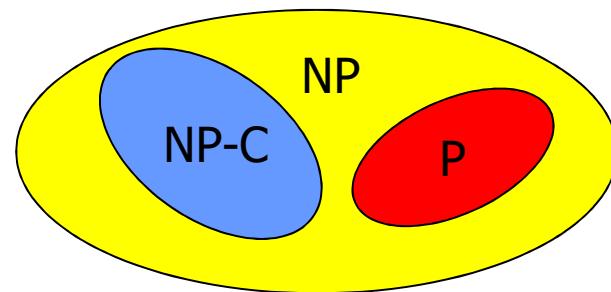
1. analisi del problema: lettura delle specifiche, comprensione del problema, identificazione della classe di problemi noti cui il problema in esame appartiene
2. identificazione delle metodologie: scelta tra paradigmi algoritmici noti (incrementale, divide et impera, programmazione dinamica, greedy, etc.)
3. selezione dell'approccio migliore in base ad un'analisi di complessità
4. decomposizione in sottoproblemi: identificazione dei sottoproblemi e delle loro interazioni in un'ottica di modularità
5. definizione dell'algoritmo risolutivo: identificazione della sequenza di passi elementari e dei dati su cui opera e dimostrazione della correttezza
6. riflessione critica: ripensamento per identificare criticità, migliorie, etc.

Tipologie di problemi

- Problemi di decisione:
 - risposta sì/no, decidibili/non decidibili, trattabili/non trattabili
- Problemi di ricerca:
 - se esiste, identificazione di una soluzione valida
 - ricerca in uno spazio delle soluzioni (con pruning)
 - verifica di validità della soluzione
- Problemi di verifica: verifica di validità della soluzione
- Problemi di ottimizzazione:
 - se esiste, identificazione di una soluzione valida e ottima
 - ricerca in uno spazio delle soluzioni (con pruning)
 - verifica di validità della soluzione
 - enumerazione esaustiva dello spazio delle soluzioni per identificare una soluzione ottima

Classi di complessità

- Classe P: problemi di decisione decidibili e trattabili per cui esiste un algoritmo polinomiale
- Classe NP (Non-deterministico Polinomiale): problemi di decisione decidibili per cui:
 - esistono algoritmi di soluzione esponenziali
 - non conosciamo algoritmi polinomiali, non possiamo però escludere che esistano
 - esistono algoritmi polinomiali di **verifica**
- Classe NP-C: problemi di decisione che
 - appartengono a NP
 - tali per cui ogni altro problema in NP è riducibile ad un problema di NP-C attraverso una trasformazione polinomiale



Analisi di complessità

- Previsione delle risorse (memoria, tempo) richieste dall'algoritmo per la sua esecuzione
- indipendente dalla macchina, asintotica, di caso peggiore
- notazione O , Ω , Θ
- limiti laschi/stretti.

Online Connectivity

Concetti ripresi:

- relazione di equivalenza: componenti fortemente connesse nei grafi orientati
- rappresentazione: algoritmo di Kruskal per gli alberi ricoprenti minimi (Minimum Spanning Trees) in Teoria dei Grafi

Matematica discreta

Sul Portale sono disponibili lucidi che raccolgono ed estendono i concetti relativi a grafi ed alberi introdotti in Tecniche di Programmazione.

Ordinamenti

- Relazione d'ordine
- stabilità/in loco
- limite inferiore alla complessità e ottimalità degli algoritmi
- algoritmo linearitmico: bottom-up mergesort

Ricorsione e paradigma divide et impera

Paolo Camurati



Definizione

Funzione *ricorsiva*:

- all'interno della propria definizione chiamata alla funzione stessa (*ricorsione diretta*)
- chiamata ad almeno una funzione la quale, direttamente o indirettamente, chiama la funzione stessa (*ricorsione indiretta*)

Algoritmo *ricorsivo*: si basa su funzioni ricorsive.

La soluzione di un problema S applicato ai dati D è ricorsiva se si può esprimere come:



$$S(D) = f(S(D'))$$

$$S(D_0) = S_0$$

$$D \neq D_0$$

D' più semplice di D

condizione di terminazione

Motivazioni

- Natura di molti problemi:
 - risoluzione di sotto-problemi analoghi a quello di partenza (ma più piccoli)
 - combinazione di soluzioni parziali nella soluzione del problema originario
 - ricorsione come base del paradigma di problem-solving noto come **divide et impera**.
- Eleganza matematica della soluzione.

Condizione di terminazione

Ogni algoritmo deve terminare \Rightarrow ricorsione finita.

Sottoproblemi semplici e risolvibili:

- banali (es.: insiemi di 1 solo elemento)
- esaurimento delle scelte lecite (es.: nel grafo è terminata la lista delle adiacenze).

Il Paradigma Divide et Impera

Divide

- da problema di dimensione n in $a \geq 1$ *problem*
indipendenti e di ugual natura di dimensione $n' < n$

Impera

- risoluzione di problema elementare (condizione di terminazione)

Combina

- ricostruzione di soluzione complessiva combinando le soluzioni parziali.

Risolfi(Problema):

- Se il problema è elementare:
 - Return Soluzione = **Risolvì_banale**(Problema)
- Altrimenti:
 - Sottoproblema_{1,2,3,...,a} = **Dividi**(Problema) ;
 - Per ciascun Sottoproblema_i:
 - Sottosoluzione_i = **Risolvì**(Sottoproblema_i) ;
 - Return Soluzione = **Combina**(Sottosoluzione_{1,2,3,...,a}) ;

condizione di terminazione

“a” sottoproblemi,
ciascuno più piccolo
del problema originale

chiamata ricorsiva

Valori di a

- $a=1$: ricorsione lineare
- $a>1$: ricorsione multi-via

Valori di n' : ad ogni passo la dimensione si riduce di:

- un **valore** costante, non sempre uguale per tutti i sottoproblemi
- un **fattore** costante, in generale lo stesso per tutti i sottoproblemi
- una quantità variabile, sovente difficile da stimare.

Terminologia incontrata in letteratura:

- **Divide and conquer**: fattore di riduzione in generale costante
- **Decrease and conquer**: valore di riduzione in generale costante.

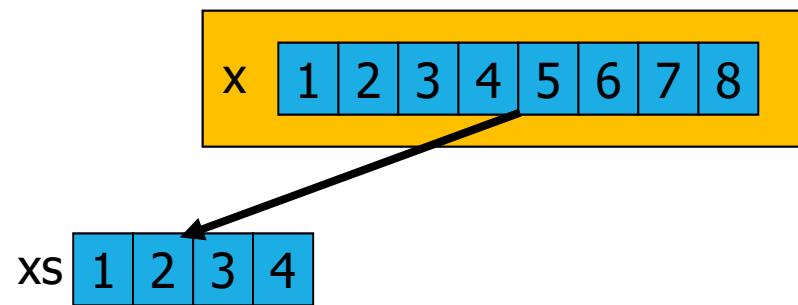
L'albero della ricorsione

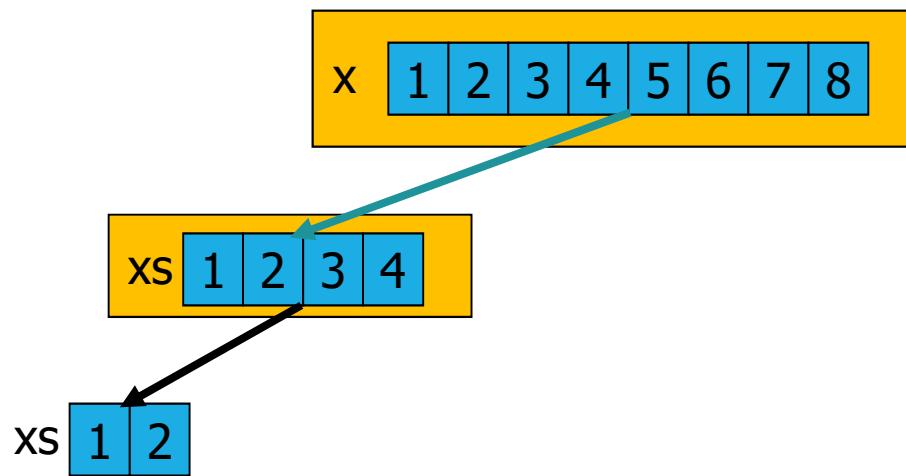
divide and conquer
 $a = 2$ $b = 2$

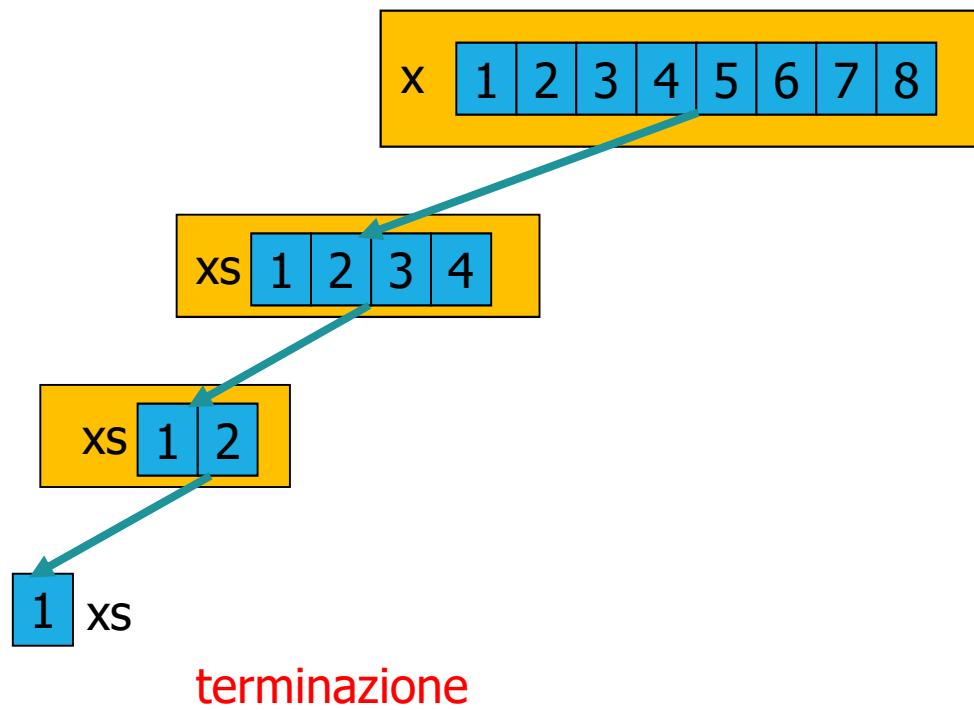
Esempio 1:

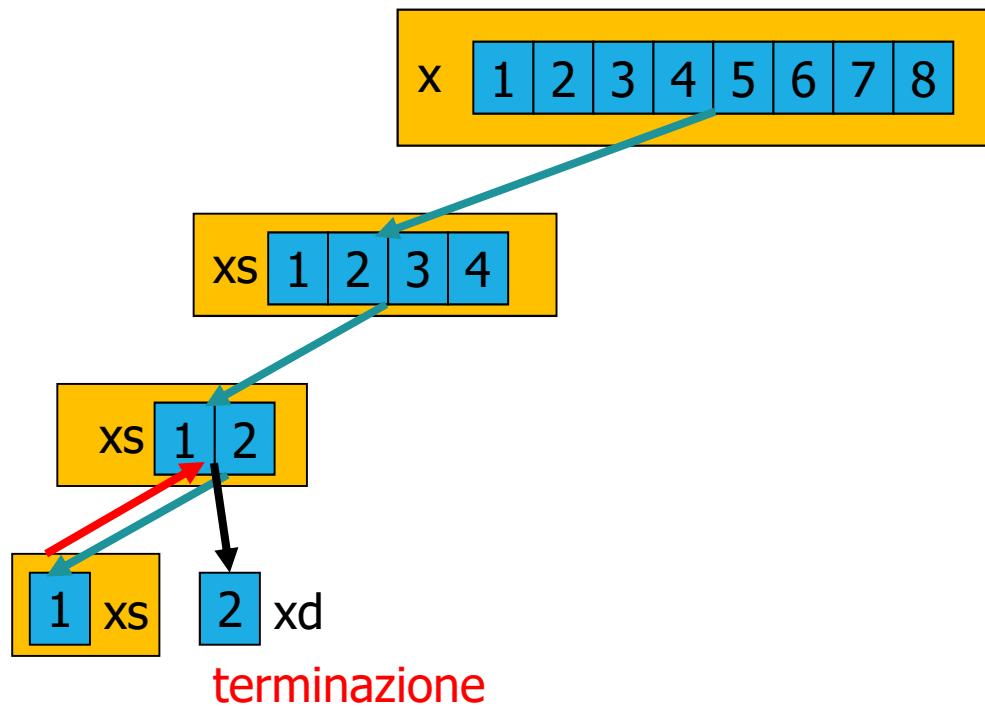
dato un vettore di $n=2^k$ interi, suddividerlo ricorsivamente in sottovettori di dimensione metà, fino alla condizione di terminazione (sottovettore di 1 sola cella).

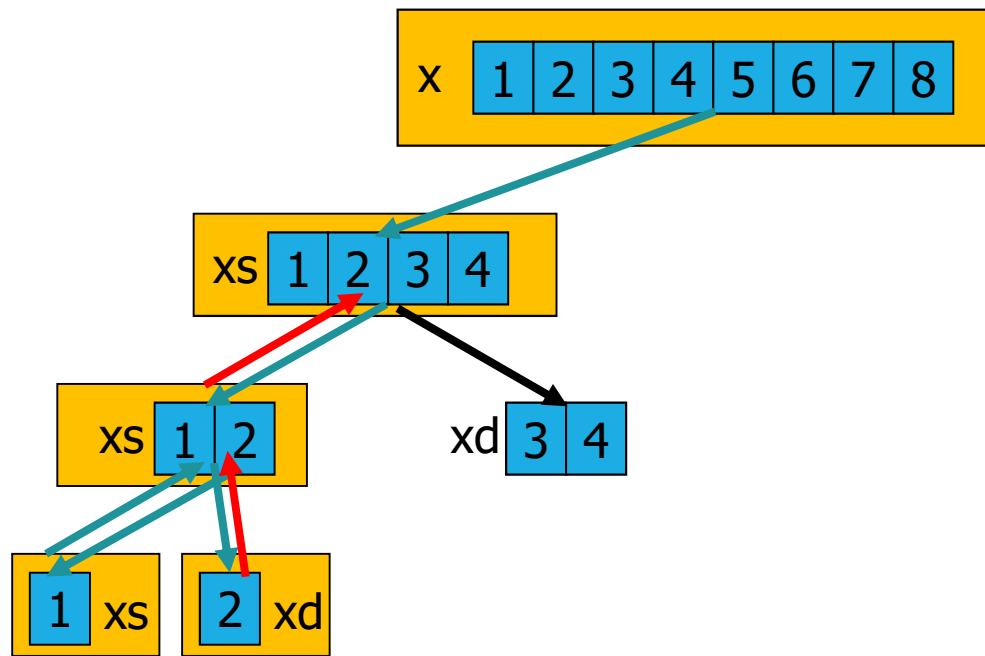
x [1|2|3|4|5|6|7|8]

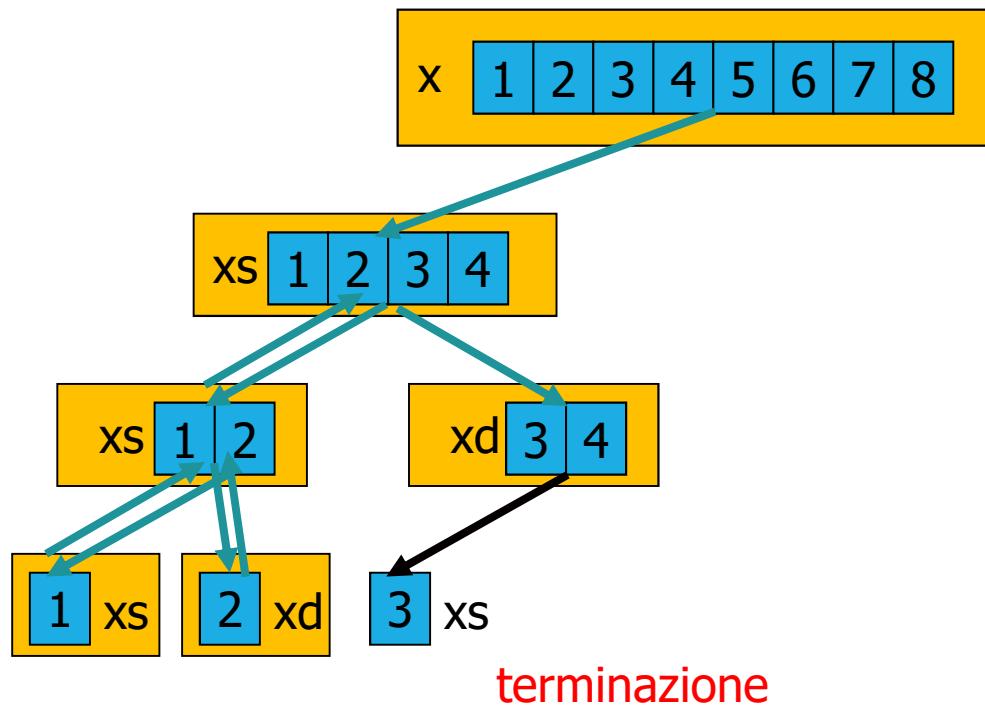


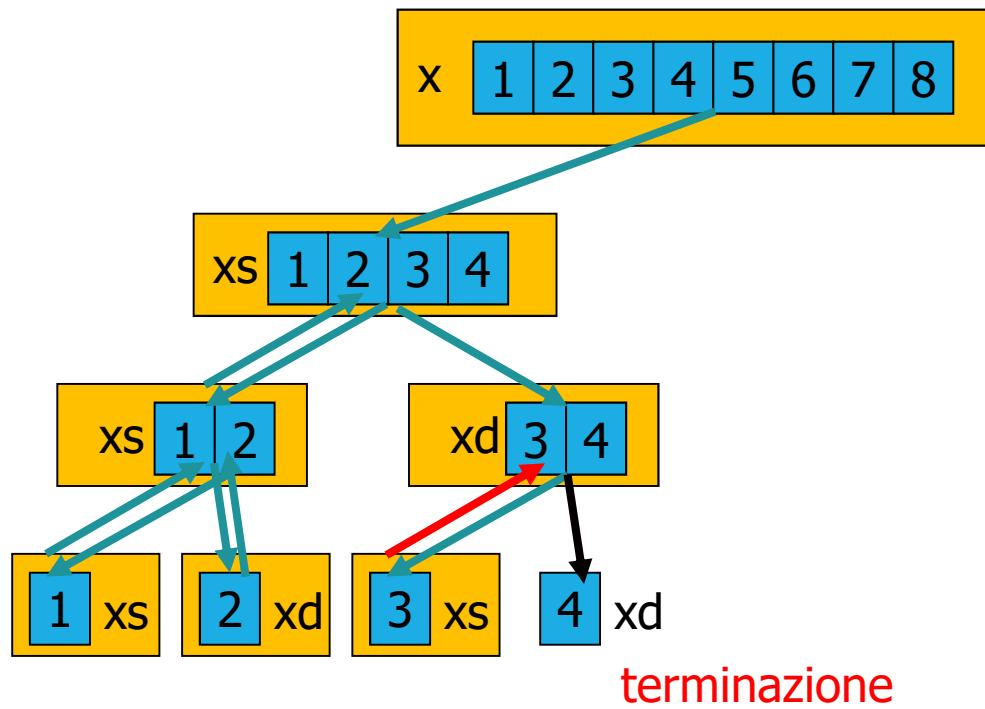


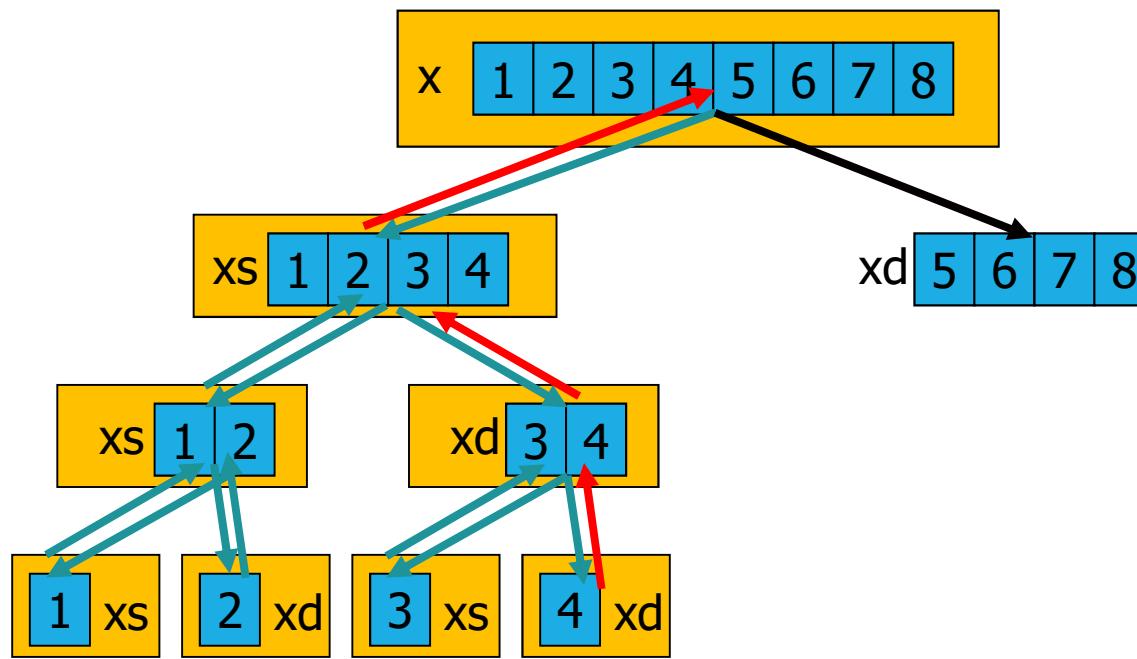


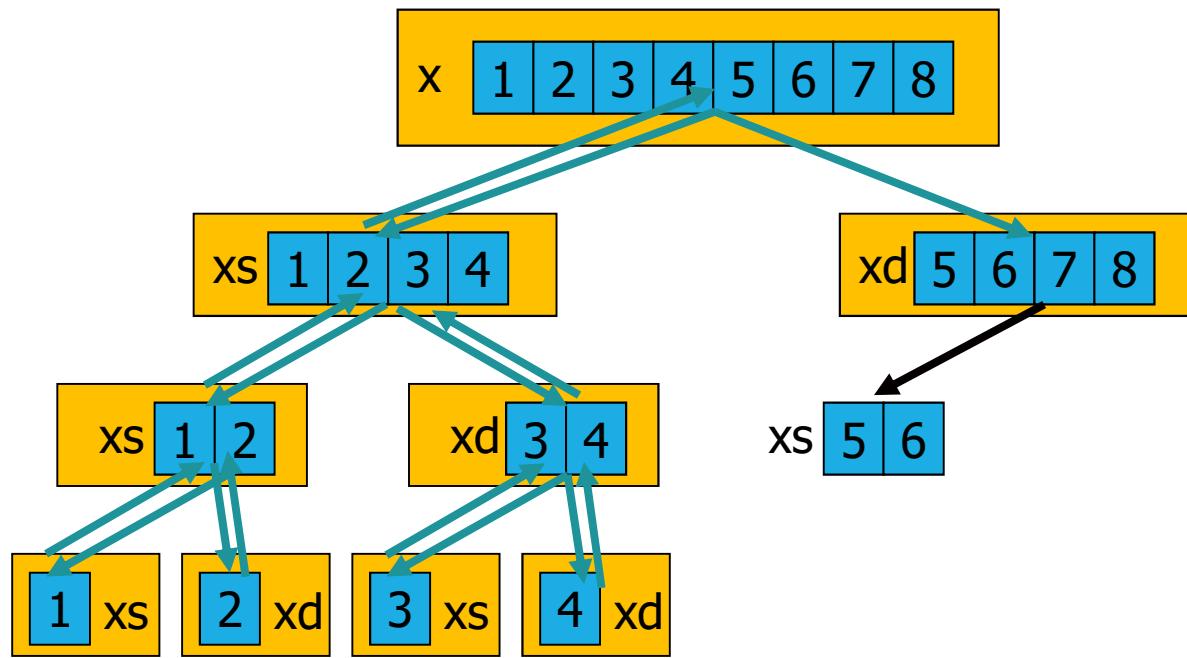


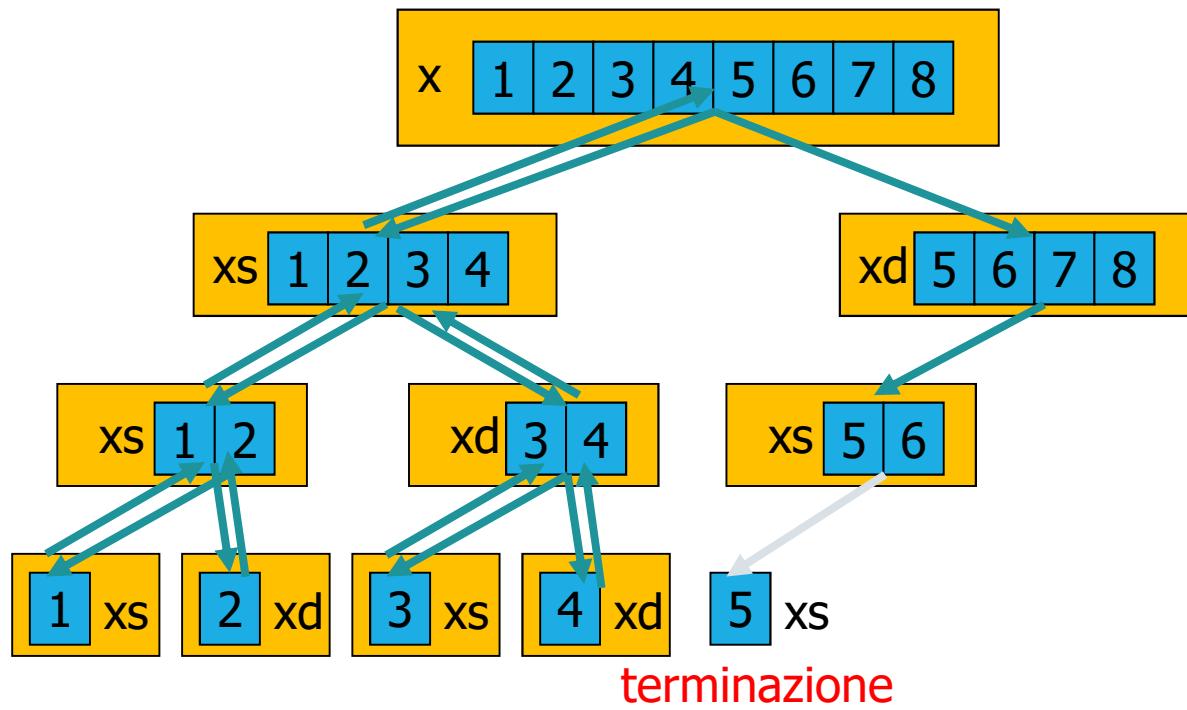


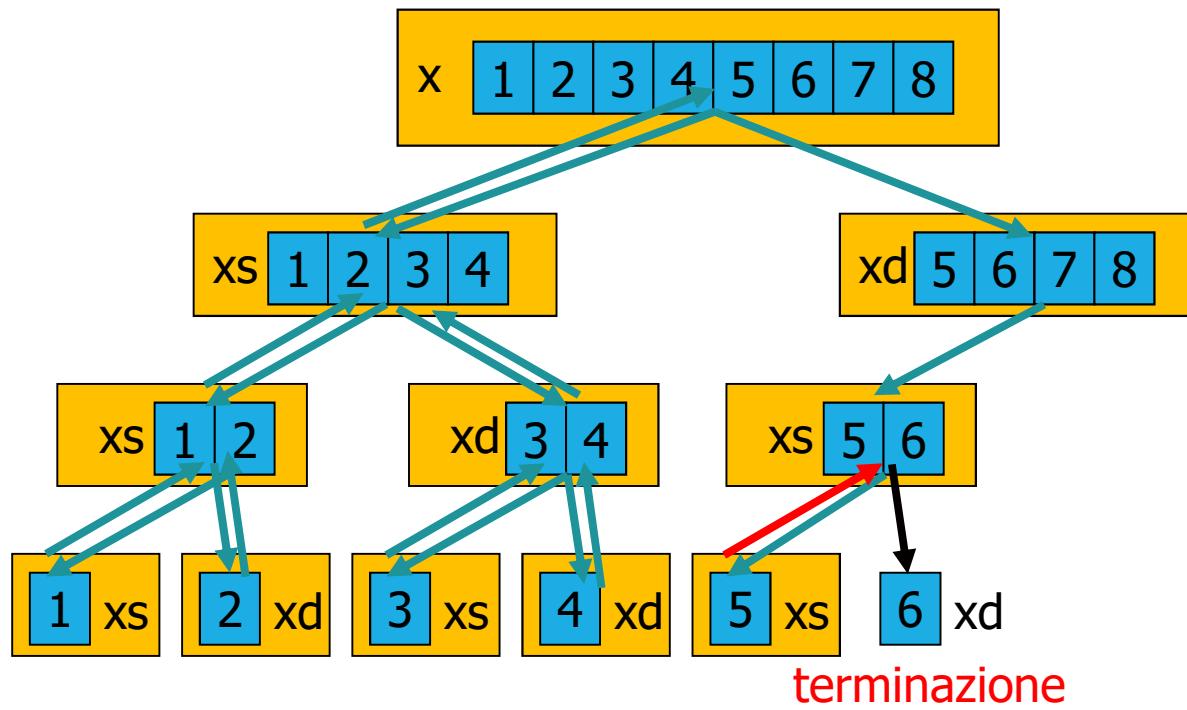


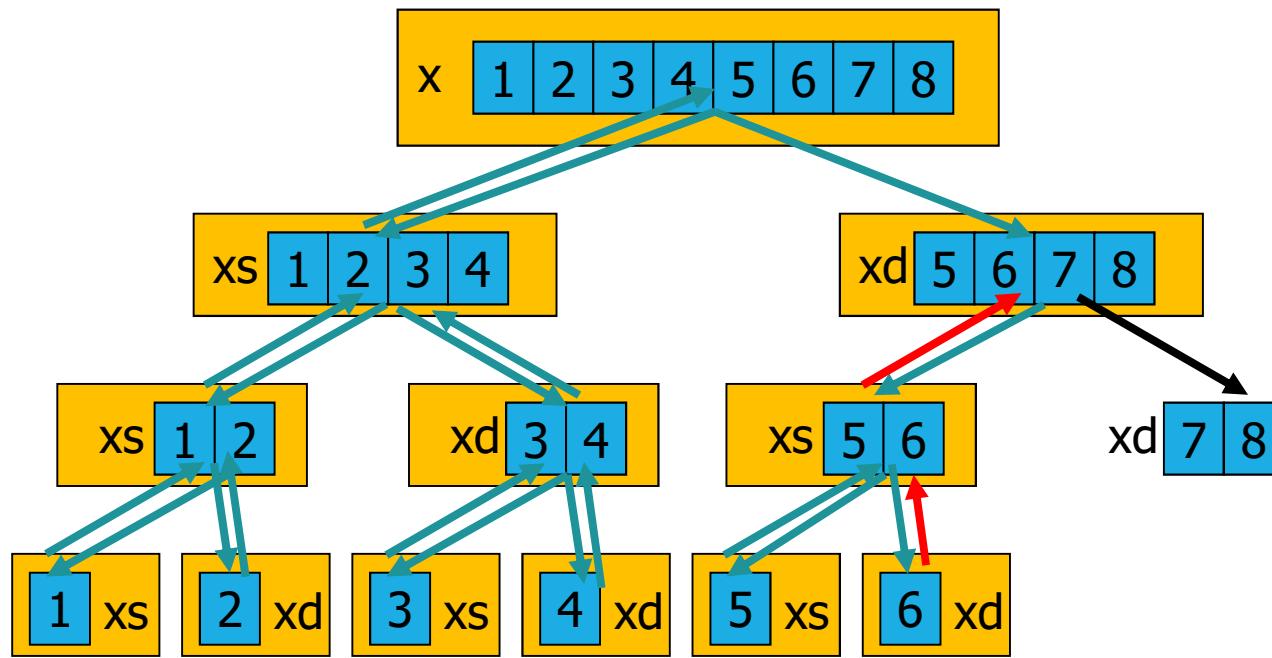


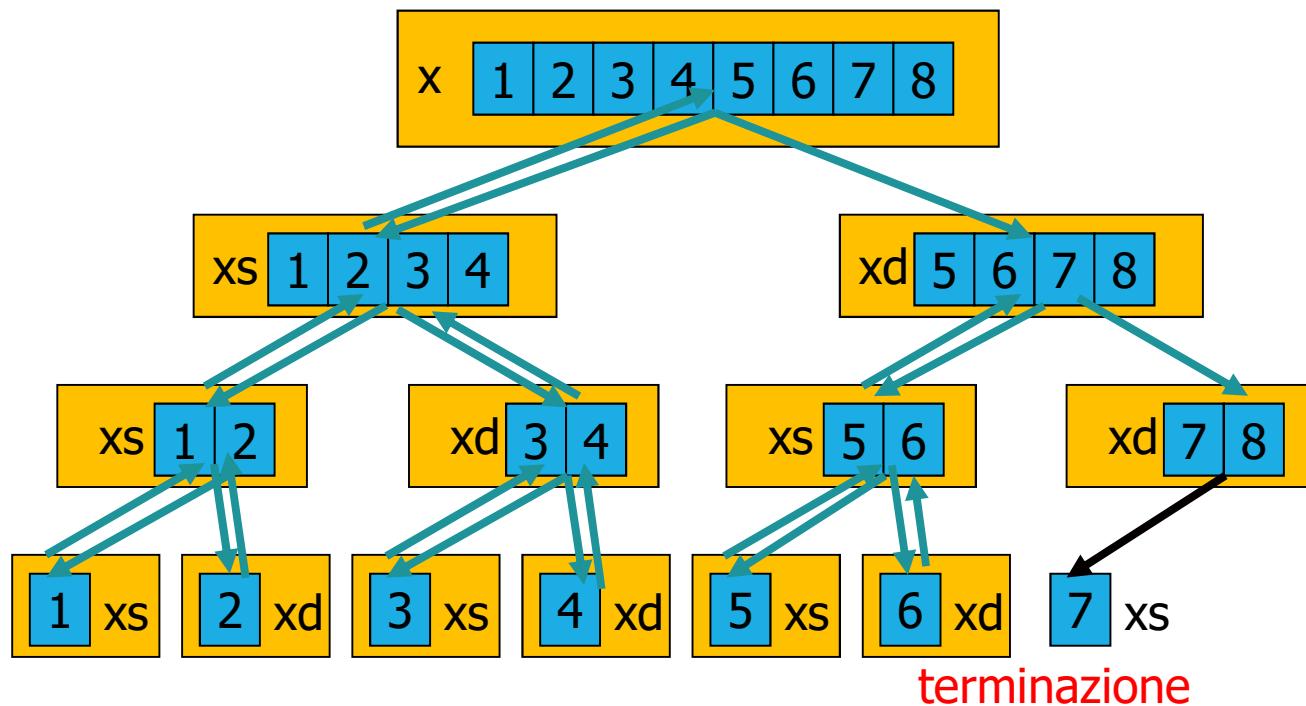


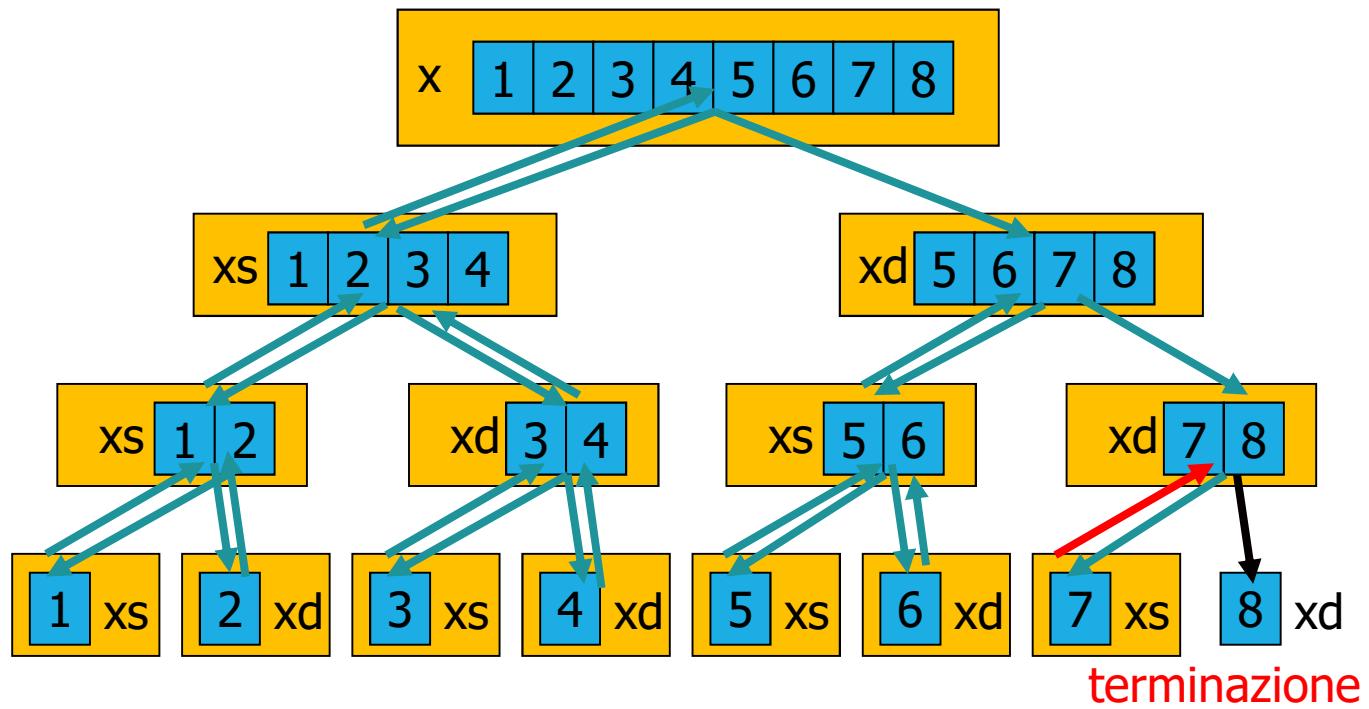


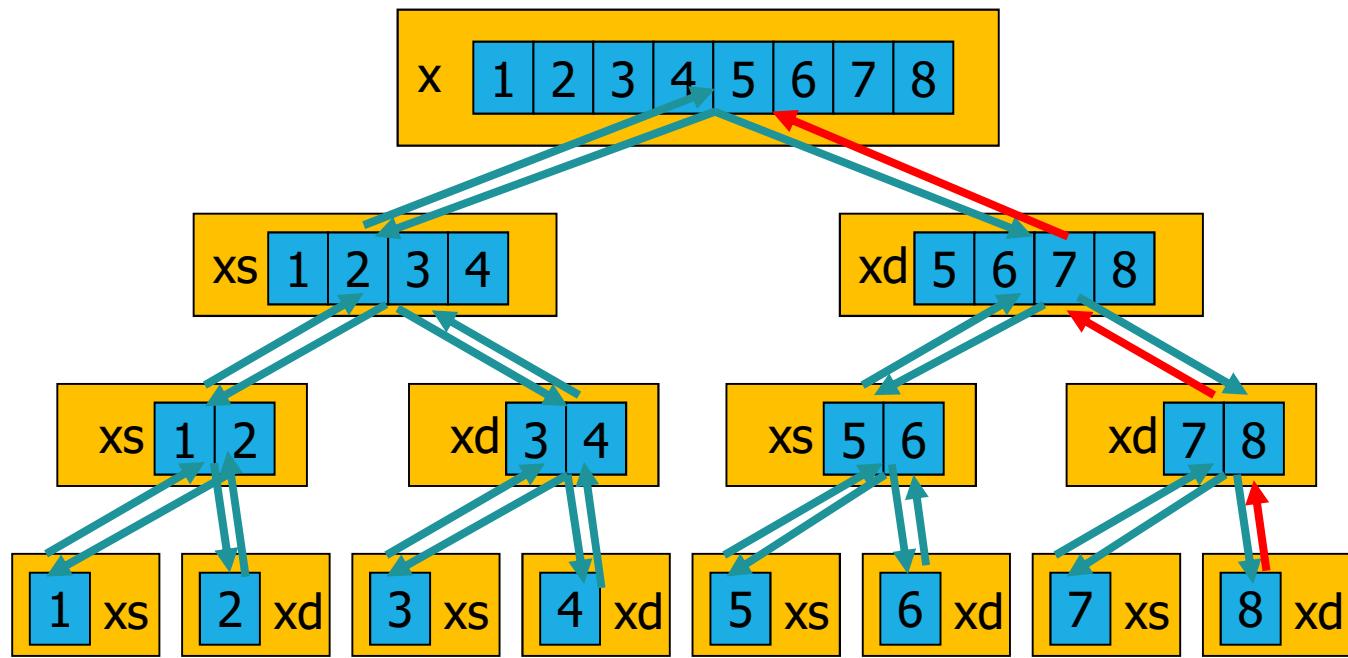












```

void show(int x[], int l, int r) {
    int i, c;
    if (l >= r) → condizione di terminazione
        return;
    c = (r+1)/2; → divisione
    printf("xs = ");
    for (i=l; i <= c; i++)
        printf("%d", x[i]);
    printf("\n");
    show(x, l, c); → chiamata ricorsiva
    printf("xd = ");
    for (i=c+1; i <= r; i++)
        printf("%d", x[i]);
    printf("\n");
    show(x, c+1, r); → chiamata ricorsiva
    return;
}

```

00show_recursion_tree.c



divide and conquer
 $a = 2$ $b = 2$

Esempio 2:

dato un vettore di $n=2^k$ interi, determinarne il massimo.

Massimo di un vettore di interi:

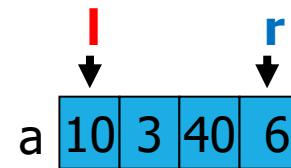
- se la dimensione è $n=1$, l'unico elemento è anche il massimo
- per $n>1$
 - dividere il vettore in due sottovettori metà
 - applicare ricorsivamente la ricerca del massimo a ciascun sottovettore
 - confrontare i risultati e restituire il più grande.

Nel main:

```
result = max(a, 0, 3);
```

$n = 2^2$

$l = 0 \quad r = 3$



```
int max(int a[], int l, int r){  
    int u, v;  
    int m = (l + r)/2;  
    if (l == r)  
        return a[l];  
    u = max (a, l, m);  
    v = max (a, m+1, r);  
    if (u > v)  
        return u;  
    else  
        return v;  
}
```

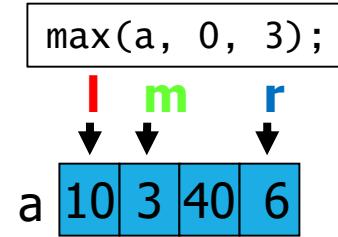
Tracciamento della ricorsione

ridondante, solo per chiarezza.
Alternativa:
`if (u > v)
 return u;
return v;`



01max_array.c

I = 0 **r** = 3 **m** = 1



```
int max(int a[],int l,int r){  
    int u, v;  
    int m = (l + r)/2;  
    if (l == r)  
        return a[l];  
    u = max (a, l, m);  
    v = max (a, m+1, r);  
    if (u > v)  
        return u;  
    else  
        return v;  
}
```

```
max(a, 0, 1);
```

l = 0 r = 3 m = 1

a **10 | 3 | 40 | 6**

chiamata ricorsiva

```
int max(int a[],int l,int r){  
    int u, v;  
    int m = (l + r)/2;  
    if (l == r)  
        return a[l];  
    u = max (a, l, m);  
    v = max (a, m+1, r);  
    if (u > v)  
        return u;  
    else  
        return v;  
}
```

```
int max(int a[],int l,int r){  
    int u, v;  
    int m = (l + r)/2;  
    if (l == r)  
        return a[l];  
    u = max (a, l, m);  
    v = max (a, m+1, r);  
    if (u > v)  
        return u;  
    else  
        return v;  
}
```

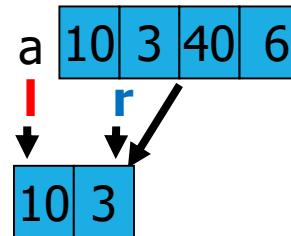
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 0 r = 1

max(a, 0, 1);



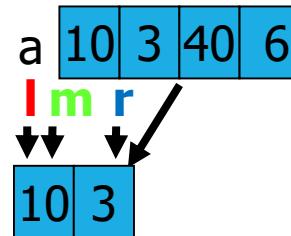
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 0 r = 1 m = 0

max(a, 0, 1);



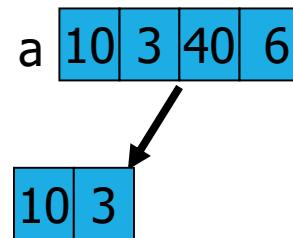
```

int max(int a[], int l, int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max(a, l, m);
    v = max(a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 0 r = 1 m = 0

max(a, 0, 1);



chiamata ricorsiva

max(a, 0, 0);

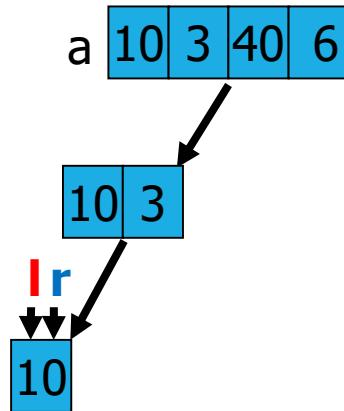
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 0 r = 0

max(a, 0, 0);



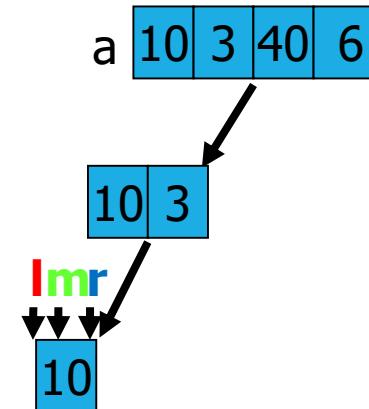
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 0 r = 0 m = 0

max(a, 0, 0);



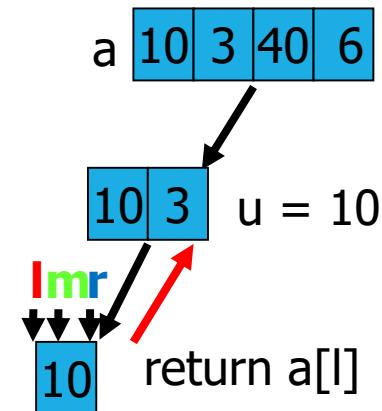
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 0 r = 0 m = 0

max(a, 0, 0);



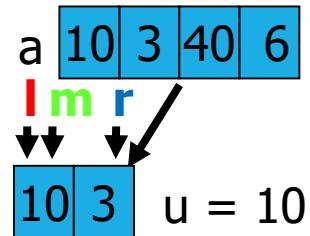
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 0 r = 1 m = 0

max(a, 0, 1);



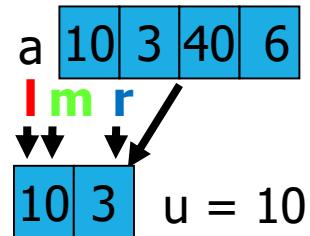
```

int max(int a[], int l, int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max(a, l, m);
    v = max(a, m+1, r); // Line 10
    if (u > v)
        return u;
    else
        return v;
}

```

l = 0 r = 1 m = 0

max(a, 0, 1);



chiamata ricorsiva

max(a, 1, 1);

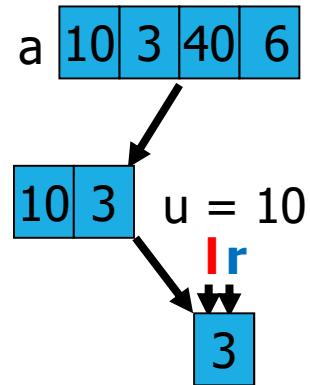
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 1 r = 1

max(a, 1, 1);



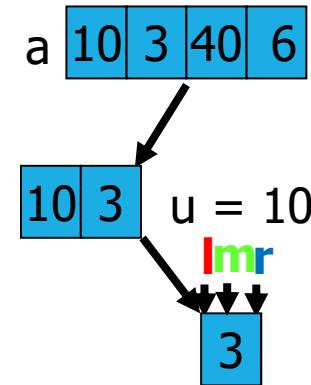
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 1 r = 1 m = 1

max(a, 1, 1);



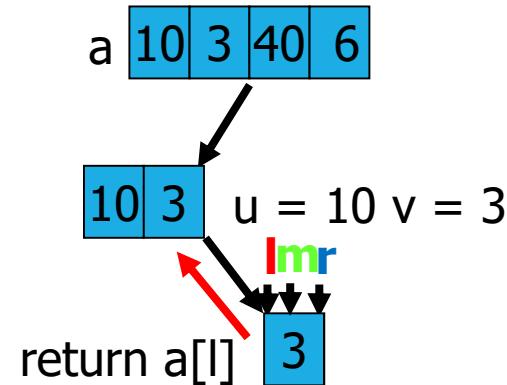
```

int max(int a[], int l, int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max(a, l, m);
    v = max(a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 1 r = 1 m = 1

max(a, 1, 1);



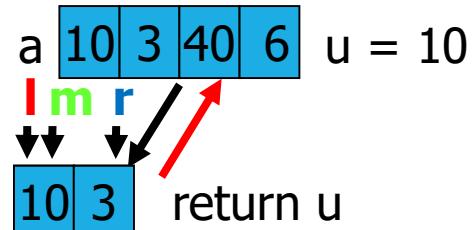
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

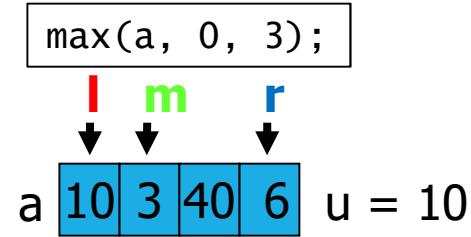
```

l = 0 r = 1 m = 0

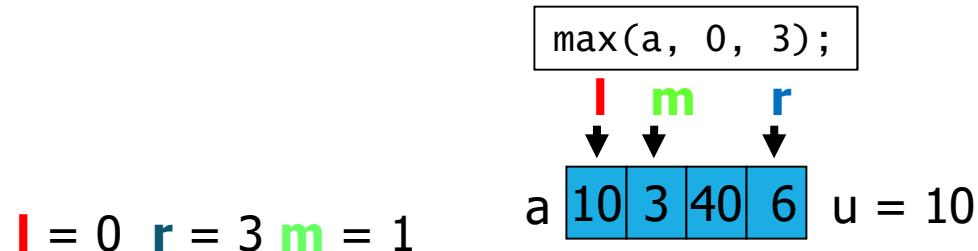
max(a, 0, 1);



I = 0 **r** = 3 **m** = 1



```
int max(int a[],int l,int r){  
    int u, v;  
    int m = (l + r)/2;  
    if (l == r)  
        return a[l];  
    u = max (a, l, m);  
    v = max (a, m+1, r);  
    if (u > v)  
        return u;  
    else  
        return v;  
}
```



```

int max(int a[], int l, int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max(a, l, m);
    v = max(a, m+1, r); // Chiamata ricorsiva
    if (u > v)
        return u;
    else
        return v;
}
  
```

chiamata ricorsiva

$\boxed{\max(a, 2, 3)}$

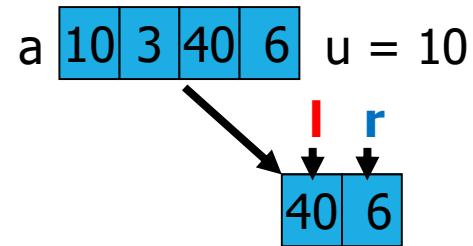
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 2 r = 3

max(a, 2, 3);



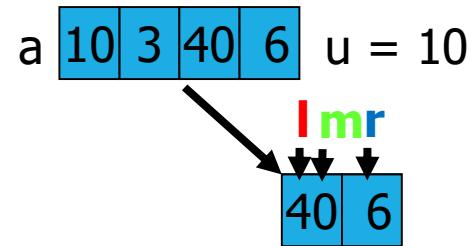
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 2 r = 3 m = 2

max(a, 2, 3);



```

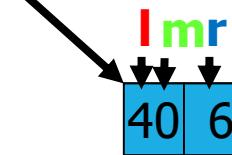
int max(int a[], int l, int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max(a, l, m);
    v = max(a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 2 r = 3 m = 2

max(a, 2, 3);

a 10 3 40 6 u = 10



chiamata ricorsiva

max(a, 2, 2);

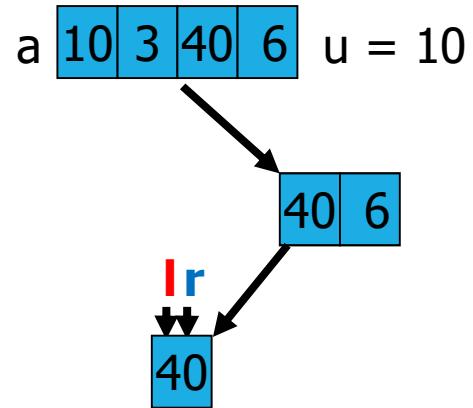
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 2 r = 2

max(a, 2, 2);



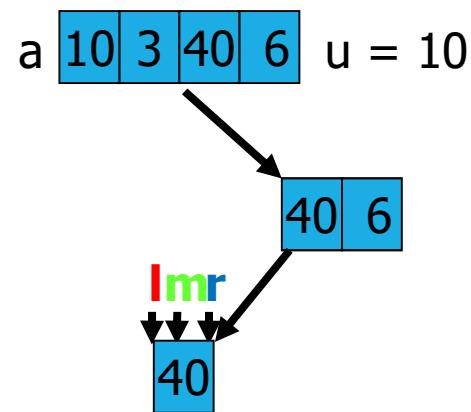
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 2 r = 2 m = 2

max(a, 2, 2);



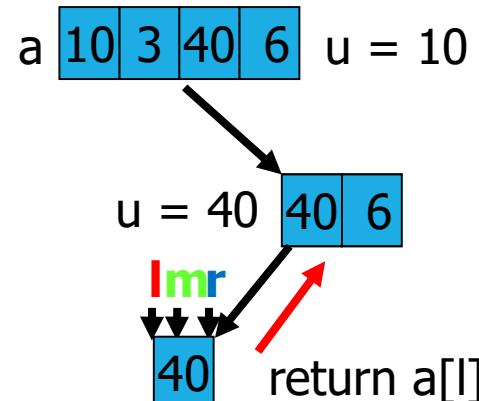
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 2 r = 2 m = 2

max(a, 2, 2);



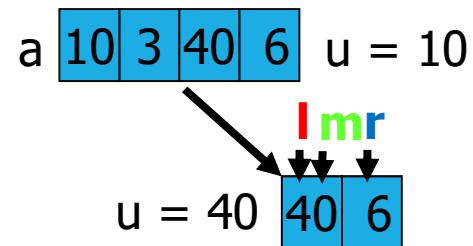
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 2 r = 3 m = 2

max(a, 2, 3);



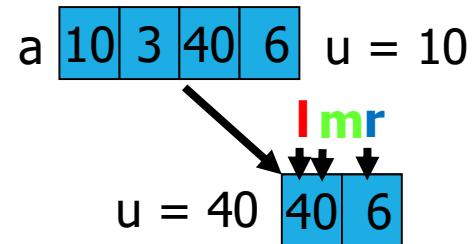
```

int max(int a[], int l, int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max(a, l, m);
    v = max(a, m+1, r); // Line 10
    if (u > v)
        return u;
    else
        return v;
}

```

l = 2 r = 3 m = 2

max(a, 2, 3);



chiamata ricorsiva

max(a, 3, 3);

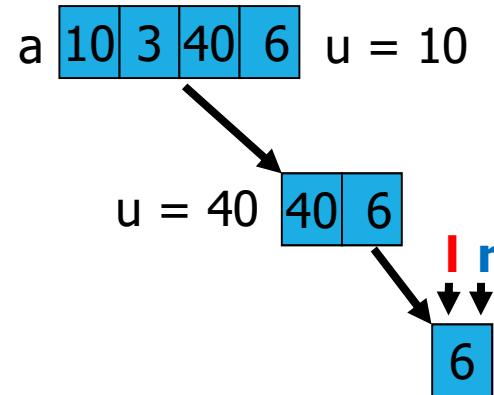
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 3 r = 3

max(a, 3, 3);



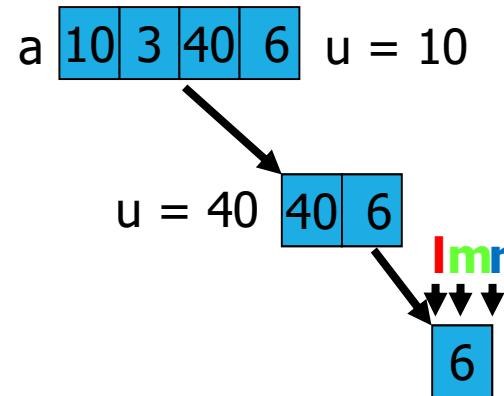
```

int max(int a[],int l,int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max (a, l, m);
    v = max (a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 3 r = 3 m = 3

max(a, 3, 3);



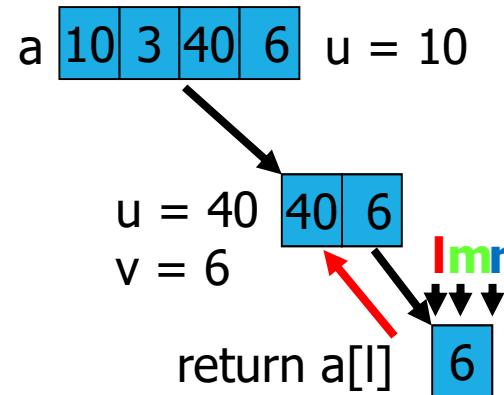
```

int max(int a[], int l, int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max(a, l, m);
    v = max(a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

```

l = 3 r = 3 m = 3

max(a, 3, 3);



```

int max(int a[], int l, int r){
    int u, v;
    int m = (l + r)/2;
    if (l == r)
        return a[l];
    u = max(a, l, m);
    v = max(a, m+1, r);
    if (u > v)
        return u;
    else
        return v;
}

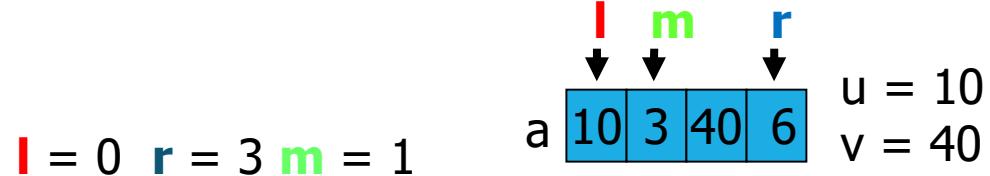
```

l = 2 r = 3 m = 2

max(a, 2, 3);

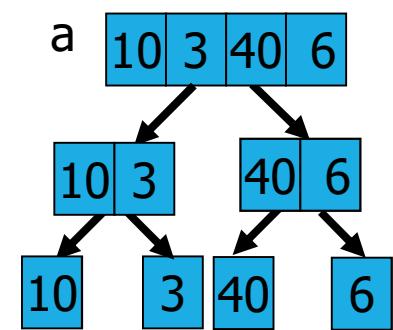
a [10|3|40|6] u = 10
 v = 40
 l|m|r
 ↓↓↓
 40|6
 return u
 u = 40
 v = 6

```
result = max (A, 0, 3); ➔ result = 40
```



```
int max(int a[],int l,int r){  
    int u, v;  
    int m = (l + r)/2;  
    if (l == r)  
        return a[l];  
    u = max (a, l, m);  
    v = max (a, m+1, r);  
    if (u > v)  
        return u;  
    else  
        return v;  
}
```

return v



Analisi di Complessità

Equazione alle Ricorrenze:

$T(n)$ viene espressa in termini di:

- $D(n)$: costo della divisione
- tempo di esecuzione per input più piccoli (ricorsione)
- $C(n)$: costo della ricombinazione

Si suppone che il costo della soluzione elementare sia unitario $\Theta(1)$.

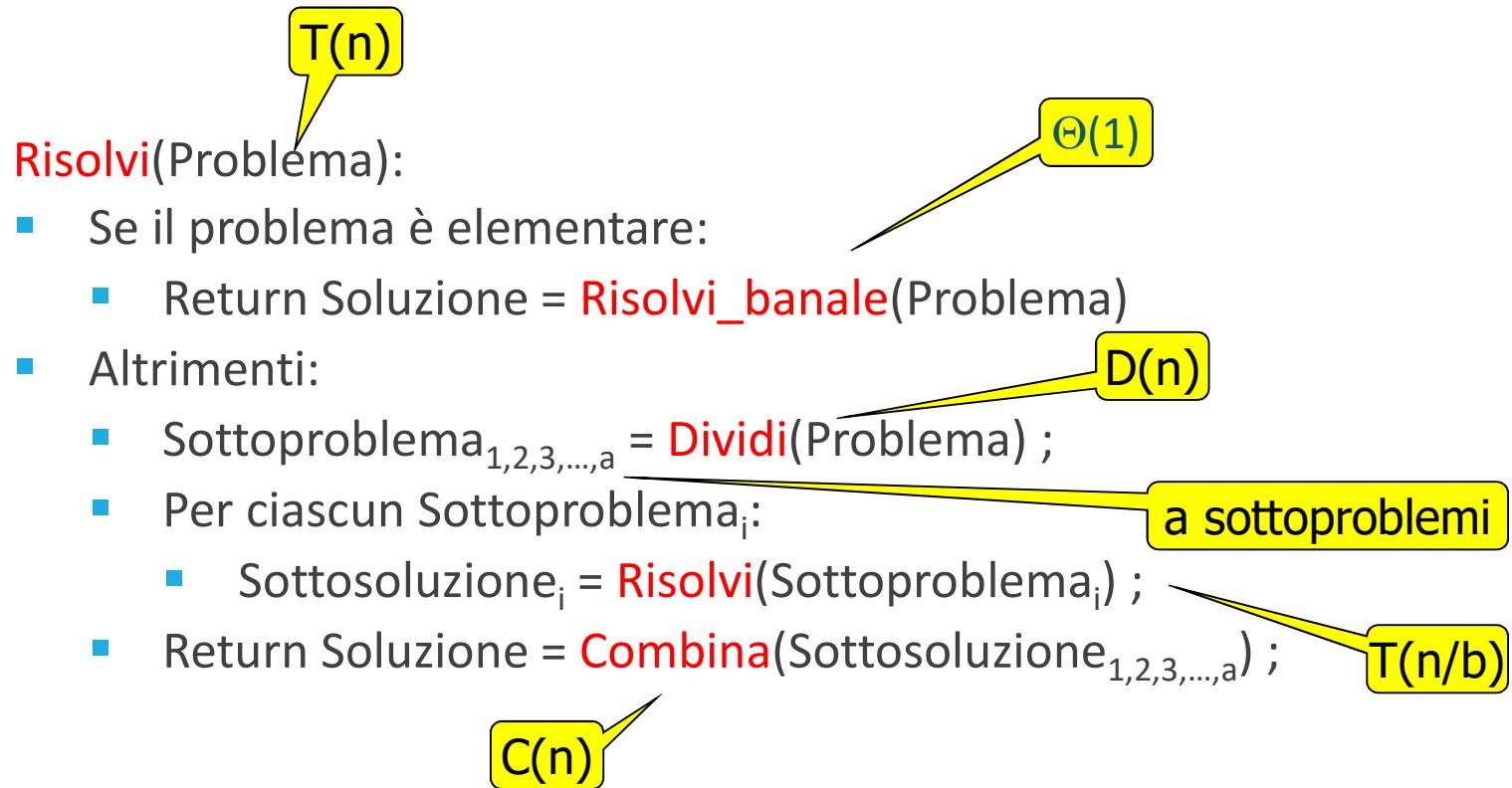
Se:

- a è il numero di sottoproblemi che risulta dalla fase di Divide
- b è il fattore di riduzione, quindi n/b è la dimensione di ciascun sottoproblema

l'equazione alle ricorrenze ha forma:

$$T(n) = D(n) + a T(n/b) + C(n) \quad n > c$$

$$T(n) = \Theta(1) \quad n \leq c$$

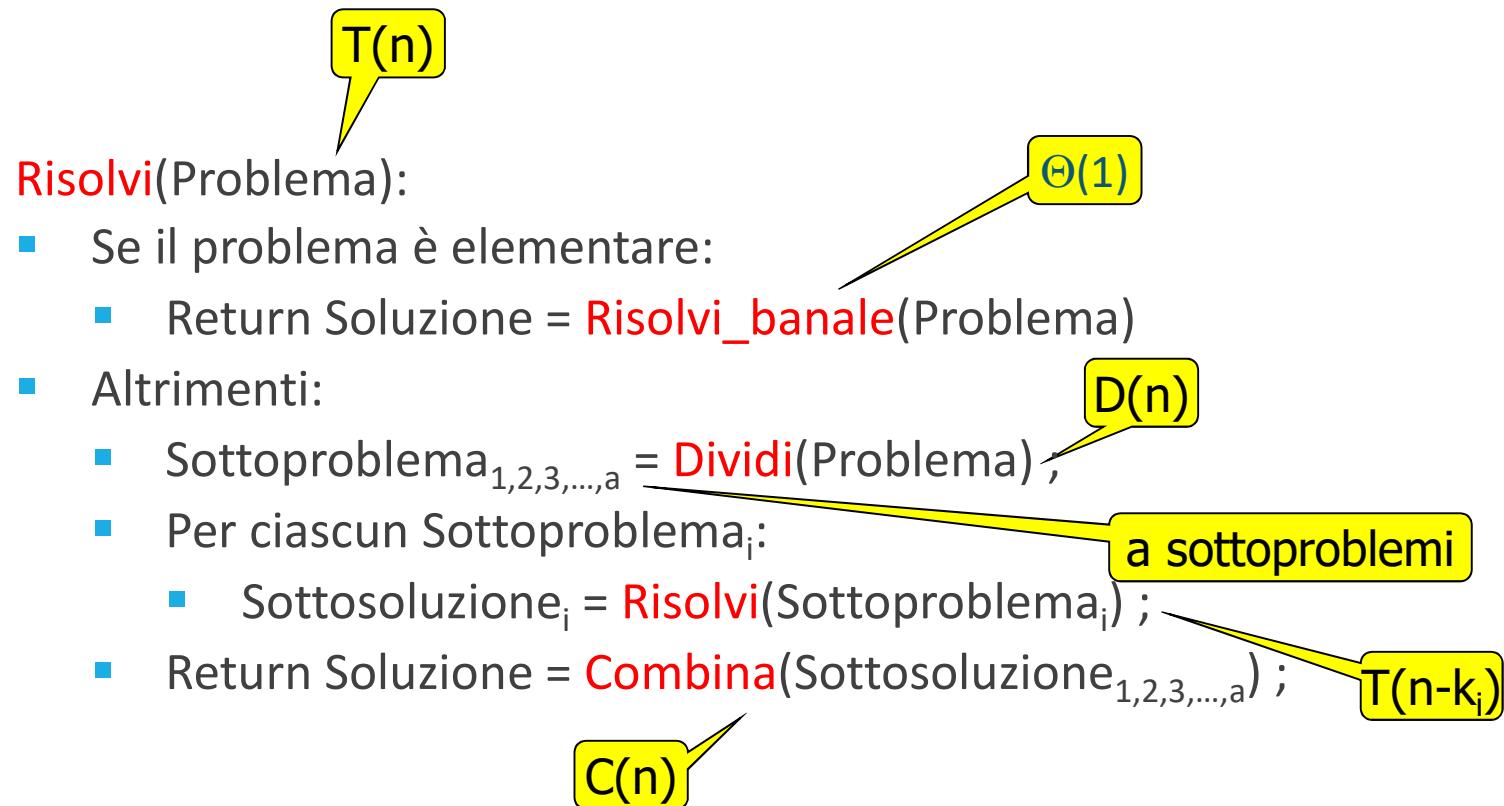


Se:

- a è il numero di sottoproblemi che risulta dalla fase di Divide
 - la riduzione è di un valore k_i , che può variare di passo in passo
- l'equazione alle ricorrenze ha forma:

$$T(n) = D(n) + \sum_{i=0}^{a-1} T(n-k_i) + C(n) \quad n > c$$

$$T(n) = \Theta(1) \quad n \leq c$$



Problemi ricorsivi semplici

Matematici:

- fattoriale
- numeri di Fibonacci
- massimo comun divisore
- prodotto di 2 interi positivi.

Il fattoriale

decrease and conquer
 $a = 1$ $k_i = 1$

Fattoriale (definizione ricorsiva)

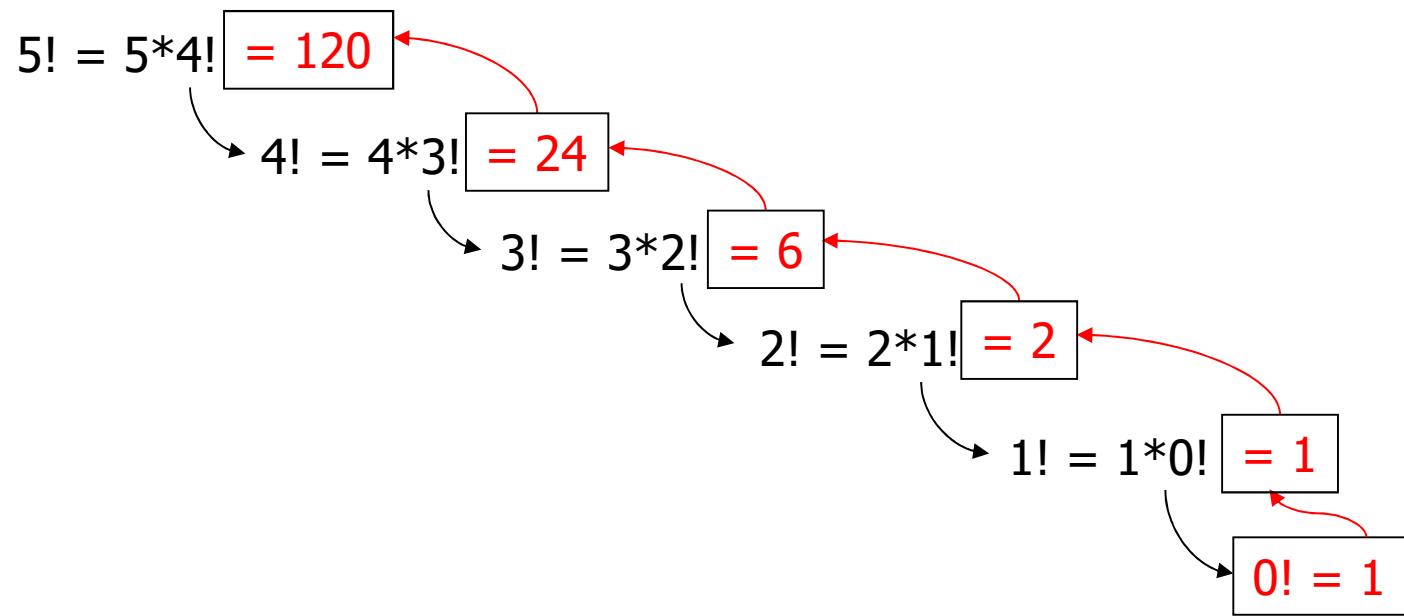
$$n! \equiv n * (n-1)! \quad n \geq 1$$

$$0! \equiv 1! \equiv 1$$

Fattoriale (definizione iterativa)

$$n! \equiv \prod_{i=0}^{n-1} (n - i) = n * (n-1) * \dots * 2 * 1$$

Esempio



```
unsigned long fact(int n) {  
    if ((n == 0) || (n == 1))  
        return 1;  
    return n*fact(n-1);  
}
```



02recursive_factorial.c

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 1$, $k_i = 1$
- equazione alla ricorrenze:

$$T(n) = \Theta(1) + T(n-1) \quad n > 1$$
$$T(1) = \Theta(1)$$

Risoluzione per sviluppo (unfolding):

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

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

$$T(n-2) = 1 + T(n-3)$$

Sostituendo in $T(n)$

$$T(n) = 1+1+1+T(n-3) = \sum_{i=0}^{n-1} 1 + (n-1) = n$$

terminazione:
 $n-i = 1$
 $i = n - 1$

Quindi:

$$T(n) = O(n)$$

Esistono anche altre tecniche di risoluzione di equazioni alle ricorrenze che permettono di esprimere limiti stretti di tipo Θ anziché O .

I numeri di Fibonacci

Numeri di Fibonacci:

$$FIB_n = FIB_{n-2} + FIB_{n-1} \quad n > 1$$

$$FIB_0 = 0$$

$$FIB_1 = 1$$

decrease and conquer
 $a = 2$ $k_i = 1$ $k_{i-1} = 2$

Leonardo Pisano (Fibonacci)

- Leonardo detto “Bigollo” Pisano (figlio di Bonaccio \Rightarrow Fibonacci) (circa 1170-1240)
- Matematico pisano che introdusse in Europa la numerazione indo-arabica nel Liber abaci (1202)
- Quesito: «*quante coppie di conigli nasceranno in un anno, a partire da un'unica coppia, se ogni mese ciascuna coppia dà alla luce una nuova coppia che diventa produttiva a partire dal secondo mese?*». Si ipotizza che nessun coniglio muoia.



Mese 0

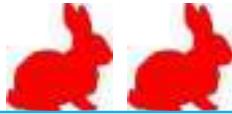
$FIB_0 = 0$

rosso: coppia non fertile **verde**: coppia fertile
↓ diventa fertile ↓ sopravvive ↗ genera

Mese 0

$$FIB_0 = 0$$

Mese 1



$$FIB_1 = 1$$

rosso: coppia non fertile **verde:** coppia fertile
↓ diventa fertile ↓ sopravvive ↗ genera

Mese 0

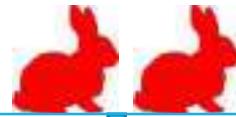
$$FIB_0 = 0$$

Mese 1

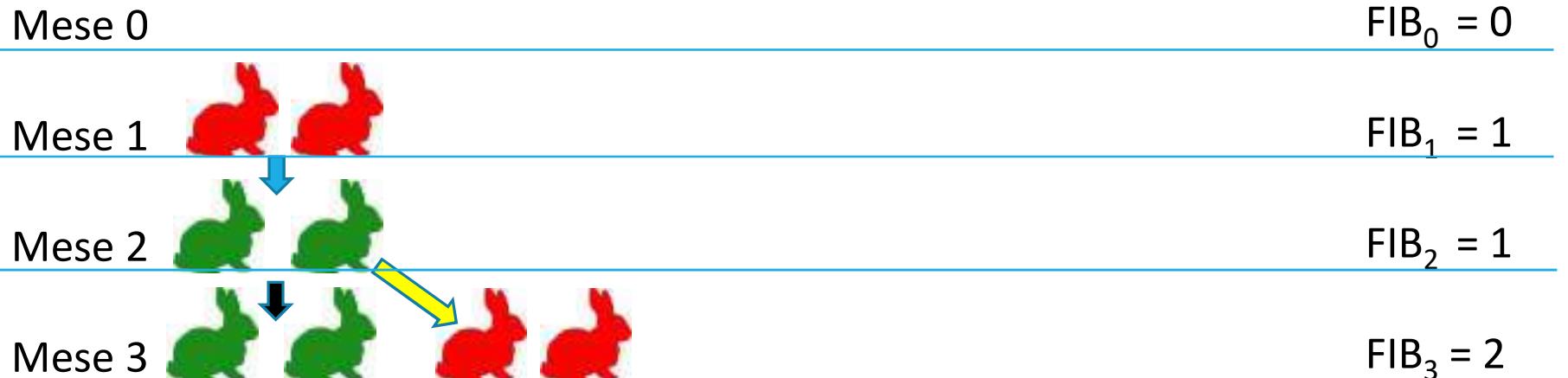
$$FIB_1 = 1$$

Mese 2

$$FIB_2 = 1$$



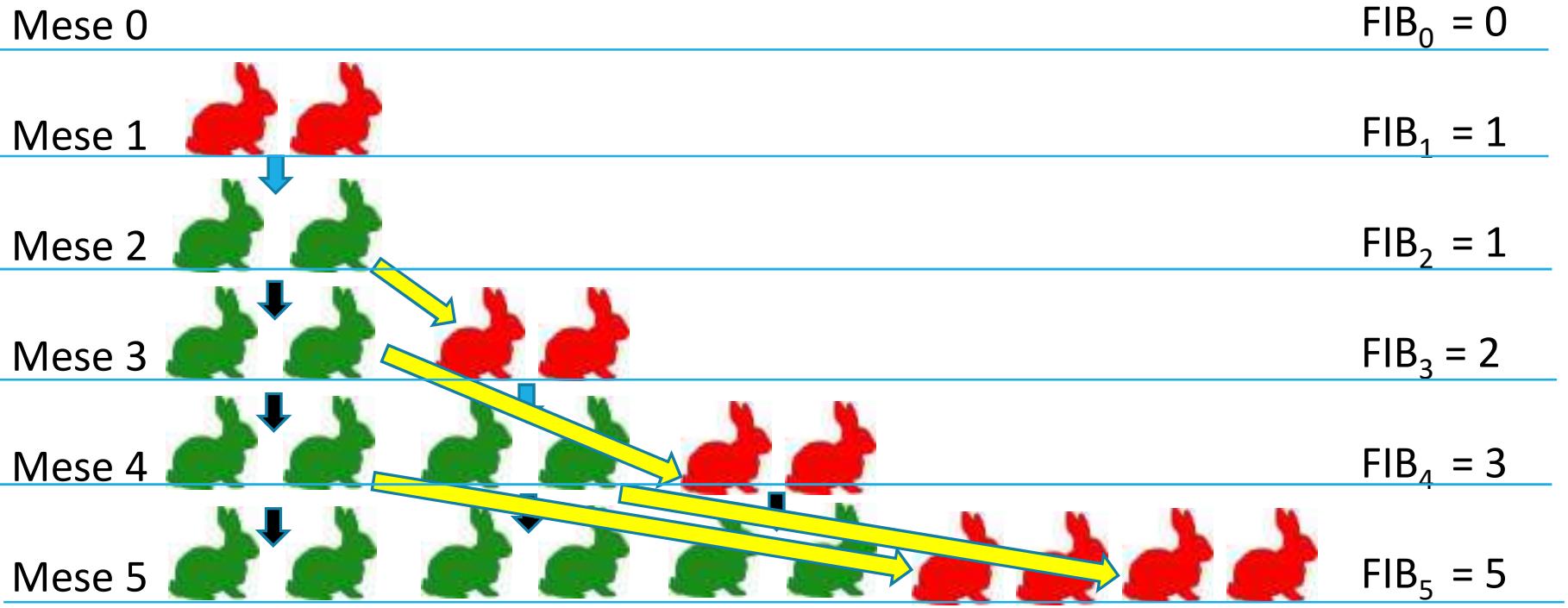
rosso: coppia non fertile **verde:** coppia fertile
 diventa fertile sopravvive genera



rosso: coppia non fertile **verde:** coppia fertile
 ↓ diventa fertile ↓ sopravvive → genera



rosso: coppia non fertile **verde:** coppia fertile
 ↓ diventa fertile ↓ sopravvive → genera

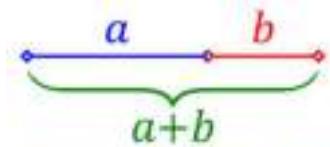


.....
rosso: coppia non fertile **verde**: coppia fertile
 ↓ diventa fertile ↓ sopravvive → genera

La sezione aurea $\varphi = a/b$

La sezione aurea o rapporto aureo o numero aureo o costante di Fidia o proporzione divina è il rapporto fra due segmenti diversi a e b tale per cui il maggiore (a) è **medio proporzionale** tra il minore (b) e la somma dei due. Lo stesso rapporto esiste anche tra il minore e la differenza.

$$(a+b) : a = a : b = b : (a-b)$$



Ponendo $\varphi = a/b$, $(a+b)/a = a/b$ diventa:

$$(\varphi + 1)/\varphi = \varphi \text{ quindi } \varphi^2 - \varphi - 1 = 0$$

Risolvendo e considerando solo la soluzione positiva

$$\varphi = (1 + \sqrt{5})/2 = 1.61803$$

La sezione aurea e i numeri di Fibonacci

Considerando anche la soluzione negativa:

$$\varphi' = (1 - \sqrt{5})/2 = -0.61803$$

l' n -esimo numero di Fibonacci FIB_n si può esprimere come:

$$FIB_n = (\varphi^n - \varphi'^n) / \sqrt{5}$$

La sezione aurea nella storia

- fra Luca Pacioli (1509) "De divina proportione"
- l'uomo vitruviano (1490) di Leonardo da Vinci
- Modulor (1948) di Le Corbusier
- Lateralus (2001) dei Tool



Lateralus (Tool, 2001)

Traccia 9: Keenan usa le sillabe per formare i primi sei numeri di Fibonacci (partendo da 1):

[1] black

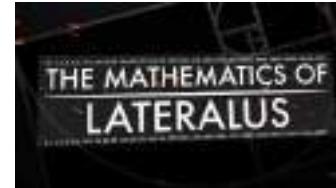
[1] then

[2] white are

[3] all I see

[5] in my infancy

[8] red and yellow then came to be



<https://youtu.be/uOHkeH2VaE0>

PS: la numerologia secondo Umberto Eco

<https://www.cicap.org/n/articolo.php?id=273203>

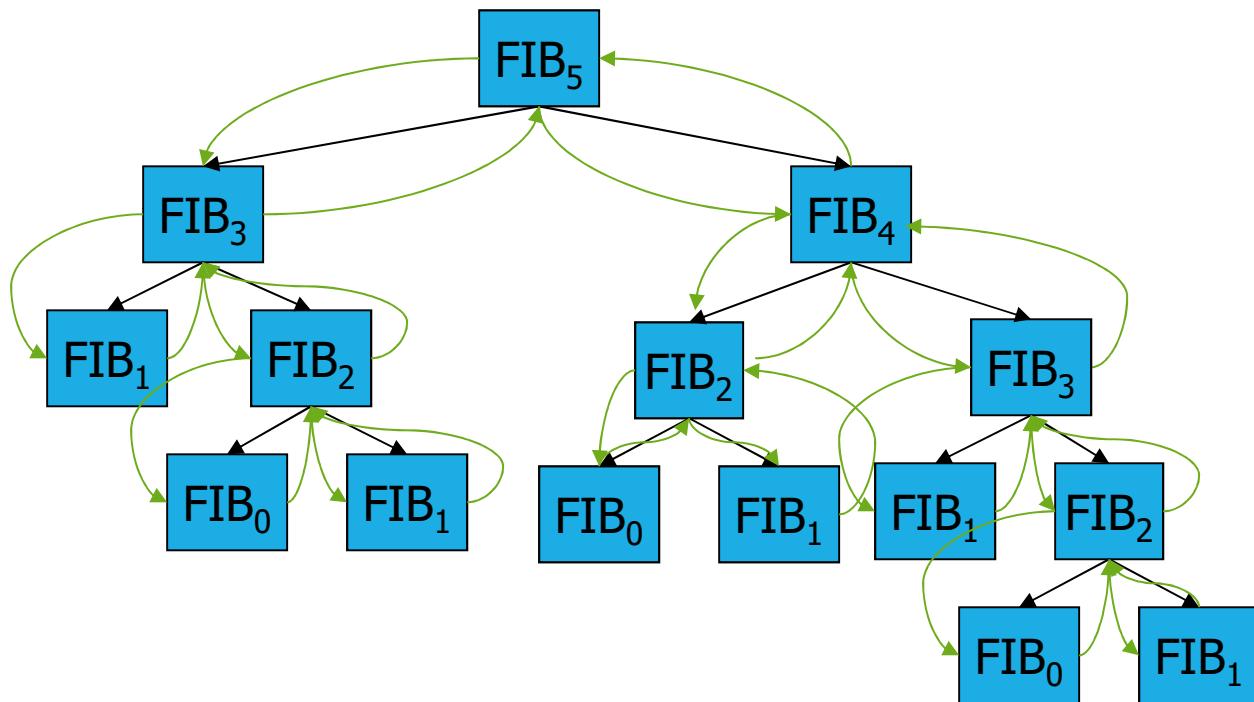
Ricorsione e numeri di Fibonacci

```
unsigned long fib(int n){  
    if(n == 0 || n == 1)  
        return(n);  
    return(fib(n-2) + fib(n-1));  
}
```



03recursive_fibonacci.c

Esempio



Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 2$, $k_i = 1$, $k_{i-1} = 2$
- equazione alla ricorrenze:

$$T(n) = 1 + T(n-1) + T(n-2) \quad n > 1$$

$$T(0) = 1 \quad T(1) = 1$$

- approssimazione conservativa: essendo

$$T(n-2) \leq T(n-1)$$

lo sostituisco con $T(n-1)$ e l'equazione diventa

$$T(n) = 1 + 2T(n-1) \quad n > 1$$

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

Risoluzione per sviluppo (unfolding):

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

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

$$T(n-2) = 1 + 2T(n-3)$$

Sostituendo in $T(n)$

$$T(n) = 1 + 2 + 4 + 2^3T(n-3) = \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

Quindi:

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

Stima migliore: $T(n) = O(\varphi^n)$.

terminazione:

$$n-i = 1$$

$$i = n - 1$$

Il massimo comun divisore

decrease and conquer
 $a = 1$ k_i variabile

Il massimo comun divisore gcd di due interi x e y non entrambi nulli è il più grande dei divisori comuni di x e y . Si assume che inizialmente $x > y$.

Esempio: $gcd(600,54) = 6$

Algoritmo inefficiente basato sulla scomposizione in fattori primi:

$$x = p_1^{e_1} \cdot p_2^{e_2} \cdots p_r^{e_r} \quad y = p_1^{f_1} \cdot p_2^{f_2} \cdots p_r^{f_r}$$

$$gcd(x,y) = p_1^{\min(e_1, f_1)} \cdot p_2^{\min(e_2, f_2)} \cdots p_r^{\min(e_r, f_r)}$$

Algoritmo di Euclide (IV sec. aC):

versione 1: sottrazione

- se $x > y$
 $\text{gcd}(x, y) = \text{gcd}(x-y, y)$
- altrimenti
 $\text{gcd}(x, y) = \text{gcd}(x, y-x)$
- terminazione:
se $x=y$ ritorna x

```
int gcd(int x, int y) {  
    if(x == y)  
        return x;  
    if (x > y)  
        return gcd(x-y, y);  
    return gcd(x, y-x);  
}
```



04recursive_gcd.c

Algoritmo di Euclide-Lamé (1844)-Dijkstra:

versione 2: resto della divisione intera

- $\text{gcd}(x, y) = \text{gcd}(y, x \% y)$
- terminazione:
se $y=0$ ritorna x

```
int gcd(int x, int y) {  
    if(y == 0)  
        return x;  
    return gcd(y, x % y);  
}
```

else ridondante eliminato

Esempi:

$\text{gcd}(600, 54)$

$\text{gcd}(54, 6)$

$\text{gcd}(6, 0)$

return 6

$\text{gcd}(314159, 271828)$

$\text{gcd}(271828, 42331)$

$\text{gcd}(42331, 17842)$

$\text{gcd}(17842, 6647)$

$\text{gcd}(6647, 4548)$

$\text{gcd}(4548, 2099)$

$\text{gcd}(2099, 350)$

$\text{gcd}(350, 349)$

$\text{gcd}(349, 1)$

$\text{gcd}(1, 0)$ return 1

314159 e 271828 sono coprimi tra di loro

Analisi di complessità

- $D(x,y) = \Theta(1)$, $C(x,y) = \Theta(1)$
- $a = 1$, riduzione variabile
- Caso peggiore: x e y sono 2 numeri di Fibonacci consecutivi (il loro gcd è 1):

$$x = \text{FIB}(n+1) \quad y = \text{FIB}(n)$$

Equazione alle ricorrenze

$$\begin{aligned} T(x,y) &= T(\text{FIB}(n+1), \text{FIB}(n)) \\ &= 1 + T(\text{FIB}(n), \text{FIB}(n+1) \% \text{FIB}(n)) \quad T(x,0) = 1 \end{aligned}$$

$$\text{ma } \text{FIB}(n+1) \% \text{FIB}(n) = \text{FIB}(n-1)$$

terminazione:
n passi

$$\begin{aligned}T(x,y) &= T(FIB(n+1), FIB(n)) \\&= 1 + T(FIB(n), FIB(n-1)) \\&= \sum_{i=0}^{n-1} 1 = n\end{aligned}$$

$T(x, y) = O(n)$, ma, visto che
 $y = FIB(n) = (\varphi^n - \varphi'^n)/\sqrt{5} = \Theta(\varphi^n)$, allora
 n è una funzione di $\log_\varphi(y)$

Quindi:

$$T(n) = O(\log(y)) \text{ e } T(x, y) = O(\log(y))$$

Il massimo di un vettore

Analisi di complessità:

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 2, b = 2$

divide and conquer
 $a = 2$ $b = 2$

Equazione alle ricorrenze

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

$$T(1) = 1$$

Risoluzione per sviluppo (unfolding):

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

$$T(n/2) = 1 + 2T(n/4)$$

$$T(n/4) = 1 + 2T(n/8)$$

Sostituendo in $T(n)$

$$T(n) = 1 + 2 + 4 + 2^3 T(n/8)$$

$$= \sum_{i=0}^{\log_2 n} 2^i = (2^{\log_2 n + 1} - 1)/(2-1)$$

$$= 2 * 2^{\log_2 n} - 1 = 2n - 1$$

Quindi:

$$T(n) = O(n)$$

terminazione:

$$n/2^i = 1$$

$$i = \log_2 n$$

$$\sum_{i=0}^k x^i = (x^{k+1} - 1)/(x-1)$$

Il prodotto di 2 interi

Moltiplicazione di 2 interi positivi x e y :

- Algoritmo elementare: moltiplicazione di ogni cifra del moltiplicatore per le cifre del moltiplicando, scalamento a sinistra (moltiplicazione per 10) e somme. Richiede la conoscenza delle tabelline pitagoriche
- Algoritmo ricorsivo elementare
 - se $y = 1$ allora $x * 1 = x$
 - altrimenti $x * y = x + x * (y - 1)$

divide and conquer
 $a = 4$ $b = 2$

Algoritmo ricorsivo per la moltiplicazione di 2 interi positivi x e y di n cifre (con $n = 2^k$):

- se la dimensione è $n=1$, calcola $x * y$ (tabelline pitagoriche)
- per $n > 1$
 - dividi x in 2: $x = 10^{n/2} * x_s + x_d$
 - dividi y in 2: $y = 10^{n/2} * y_s + y_d$
 - calcola ricorsivamente $x_s * y_s$, $x_s * y_d$, $x_d * y_s$, $x_d * y_d$,
 - calcola $x * y = 10^n * x_s * y_s + 10^{n/2} * (x_s * y_d + x_d * y_s) + x_d * y_d$

Esempio

$$1356 * 2410 = 3.267.960$$

x

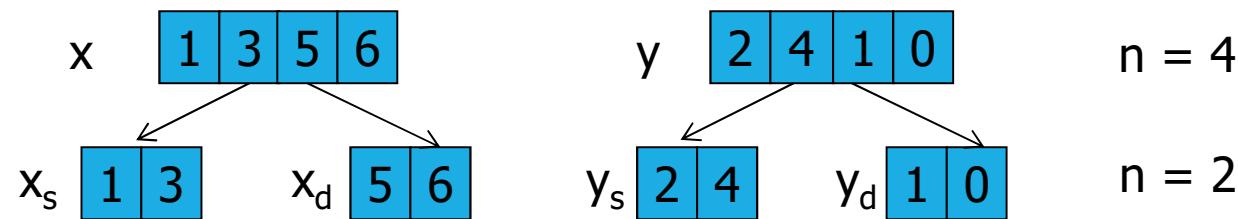
1	3	5	6
---	---	---	---

 * y

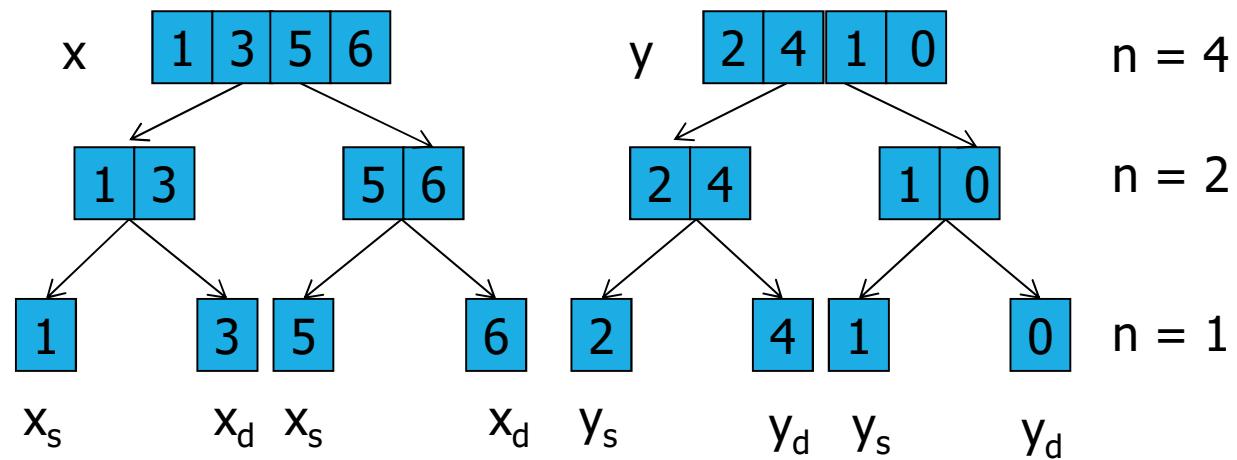
2	4	1	0
---	---	---	---

 n = 4

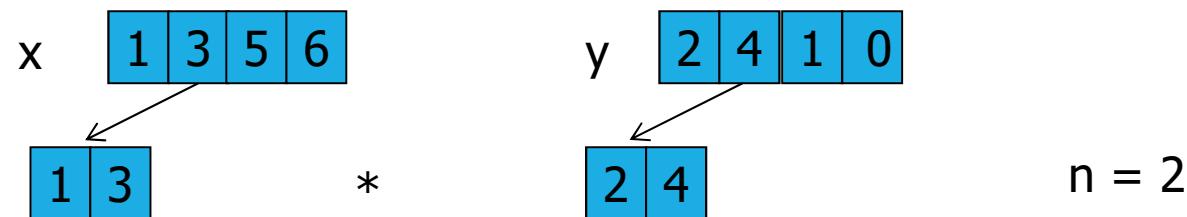
$$1356 * 2410 = 3.267.960$$



$$1356 * 2410 = 3.267.960$$



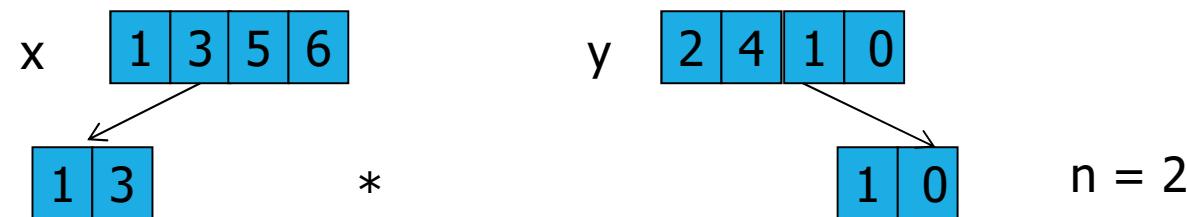
$$1356 * 2410 = 3.267.960$$



$$\begin{aligned}
 & 10^n \quad x_s \quad y_s \quad 10^{n/2} \quad x_s \quad y_d \quad x_d \quad y_s \quad x_d \quad y_d \\
 & 10^2 * \boxed{1} * \boxed{2} + 10^1 * (\boxed{1} * \boxed{4} + \boxed{3} * \boxed{2}) + \boxed{3} * \boxed{4}
 \end{aligned}$$

$$13 * 24 = 10^2 * 2 + 10^1 * 10 + 12 = 312$$

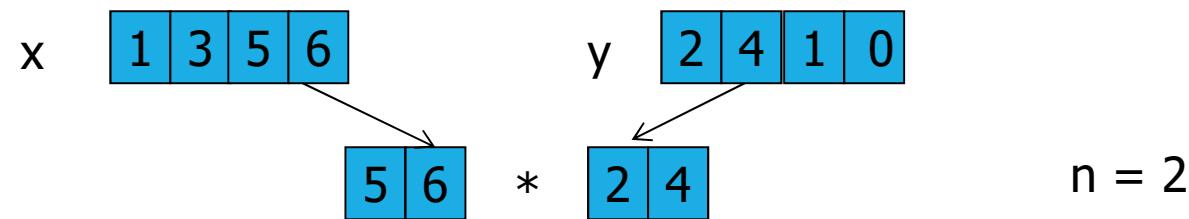
$$1356 * 2410 = 3.267.960$$



$$\begin{array}{ccccccccc}
 10^n & x_s & y_s & 10^{n/2} & x_s & y_d & x_d & y_s & x_d & y_d \\
 10^2 * 1 & * 1 & + 10^1 * (& 1 & * 0 & + 3 & * 1 &) + 3 & * 0
 \end{array}$$

$$13 * 10 = 10^2 * 1 + 10^1 * 3 + 0 = 130$$

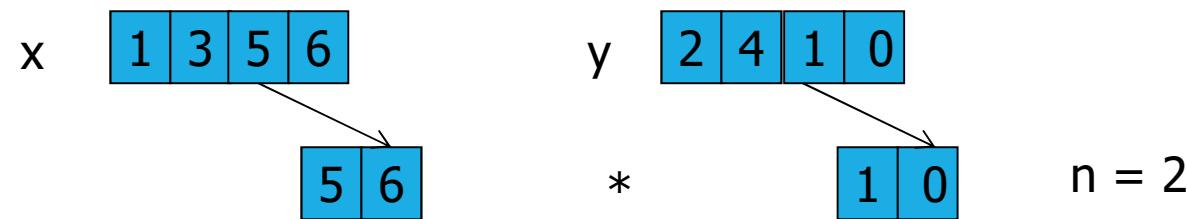
$$1356 * 2410 = 3.267.960$$



$$\begin{aligned}
 & 10^n \quad x_s \quad y_s \quad 10^{n/2} \quad x_s \quad y_d \quad x_d \quad y_s \quad x_d \quad y_d \\
 & 10^2 * \boxed{5} * \boxed{2} + 10^1 * (\boxed{5} * \boxed{4} + \boxed{6} * \boxed{2}) + \boxed{6} * \boxed{4}
 \end{aligned}$$

$$56 * 24 = 10^2 * 10 + 10^1 * 32 + 24 = 1344$$

$$1356 * 2410 = 3.267.960$$



$$\begin{array}{ccccccccc}
 10^n & x_s & y_s & 10^{n/2} & x_s & y_d & x_d & y_s & x_d & y_d \\
 10^2 * 5 & * 1 & + 10^1 * (5 * 0 + 6 * 1) + 6 * 0
 \end{array}$$

$$56 * 10 = 10^2 * 5 + 10^1 * 6 + 0 = 560$$

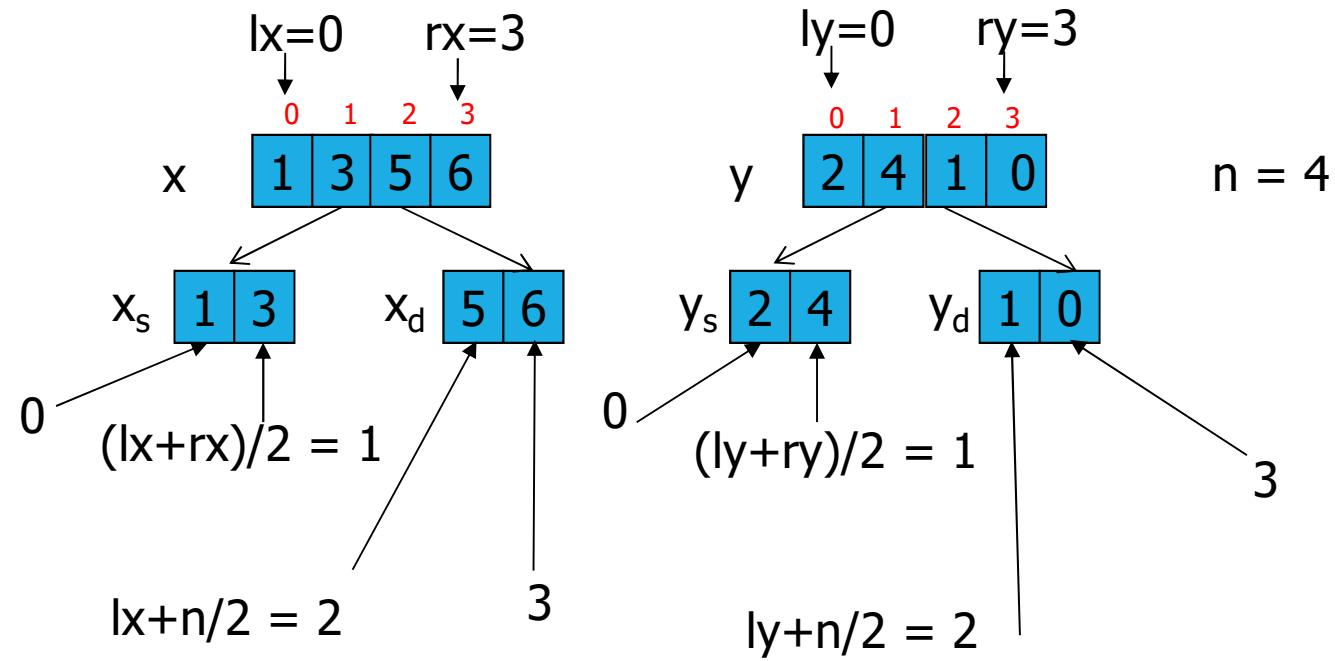
$$1356 * 2410 = 3.267.960$$

$$x \quad \boxed{1 \mid 3 \mid 5 \mid 6} \quad * \quad y \quad \boxed{2 \mid 4 \mid 1 \mid 0} \quad n = 4$$

$$\begin{array}{cccccccccc} 10^n & x_s & y_s & 10^{n/2} & x_s & y_d & x_d & y_s & x_d & y_d \\ 10^4 * 13 & * 24 & + 10^2 * (& 13 & * 10 & + 56 & * 24 &) + 56 & * 10 \end{array}$$

$$1356 * 2410 = 10^4 * 312 + 10^2 * (130 + 1344) + 560 = 3.267.960$$

Identificazione dei sottovettori sinistro e destro:



```

long prod(int *x,int lx,int rx,int *y,int ly,int ry,int n) {
    long t1, t2, t3;
    if (n == 1)
        return (*x+lx)*(*y+ly);
    t1 = prod(x, lx, (lx+rx)/2, y, ly, (ly+ry)/2, n/2);
    t2 = prod(x, lx, (lx+rx)/2, y, ly+n/2, ry, n/2)
        + prod(x, lx+n/2, rx, y, ly, (ly+ry)/2, n/2);
    t3 = prod(x, lx+n/2, rx, y, ly + n/2, ry, n/2);
    return t1 * pow(10,n) + t2 * pow (10, n/2) + t3;
}

```

operazione considerata
elementare



05recursive_integer_product.c

Analisi di complessità

- moltiplicazione per 10^k : shift a sinistra (ogni shift è $\Theta(1)$, in totale è $\Theta(k)$. Al più $k=n$ e sono necessarie 2 moltiplicazioni, il costo è quindi $\Theta(n)$)
- la somma di numeri su k cifre è $\Theta(k)$. Due numeri di n cifre danno un prodotto su $2n$ cifre. Al più $k=2n$ e sono necessarie 3 somme, il costo è quindi $\Theta(n)$)
- moltiplicazione: costo della ricorsione (4 moltiplicazioni)
- $D(n) = \Theta(1)$, $C(n) = \Theta(n)$, $D(n) + C(n) = \Theta(n)$
- $a = 4$, $b = 2$

Equazione alle ricorrenze:

$$T(n) = 4T(n/2) + n \quad n > 1$$

$$T(1) = 1$$

Risoluzione per sviluppo (unfolding)

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

$$T(n/4) = 4T(n/8) + n/4 \text{ etc.}$$

terminazione:
 $n/2^i = 1$
 $i = \log_2 n$

$$\begin{aligned} T(n) &= n + 4*(n/2) + 4^2 * (n/4) + 4^3 * T(n/8) \\ &= \sum_{0 \leq i \leq \log_2 n} 4^i / 2^i * n = n * \sum_{0 \leq i \leq \log_2 n} 2^i \\ &= n * (2^{\log_2 n + 1} - 1) / (2 - 1) = n * (2 * 2^{\log_2 n} - 1) = 2n^2 - n \end{aligned}$$

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

$$\sum_{i=0}^k x^i = (x^{k+1} - 1)/(x - 1)$$

divide and conquer
 $a = 3$ $b = 2$

Algoritmo di Karatsuba e Ofman (1962):

Ottimizzazione dell'algoritmo di base mediante riduzione del numero di moltiplicazioni:

$$x_s * y_d + x_d * y_s = x_s * y_s + x_d * y_d - (x_s - x_d) * (y_s - y_d)$$

3 moltiplicazioni ricorsive, anziché 4.

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(n)$
- $D(n) + C(n) = \Theta(n)$
- $a = 3$, $b = 2$

Equazione alle ricorrenze:

$$T(n) = 3T(n/2) + n \quad n > 1$$

$$T(1) = 1$$

Risoluzione per sviluppo (unfolding)

$$T(n/2) = 3T(n/4) + n/2$$

$$T(n/4) = 3T(n/8) + n/4 \text{ etc.}$$

$$T(n) = n + 3*(n/2) + 3^2 * (n/4) + 3^3 * T(n/8)$$

$$= \sum_{0 \leq i \leq \log_2 n} 3^i / 2^i * n = n * \sum_{0 \leq i \leq \log_2 n} (3/2)^i$$

$$= n * ((3/2)^{\log_2 n + 1} - 1) / (3/2 - 1)$$

$$= 2n * (3/2 * (3^{\log_2 n} / 2^{\log_2 n}) - 1)$$

$$= 2n * (3 * n^{\log_2 3} / 2^{n-1})$$

$$= 3n^{\log_2 3} - 2n$$

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

terminazione:
 $n/2^i = 1$
 $i = \log_2 n$

$$\sum_{i=0}^k x^i = (x^{k+1} - 1)/(x - 1)$$

$$a^{\log_b n} = n^{\log_b a}$$

Problemi ricorsivi semplici

Informatici:

- ricerca binaria o dicotomica
- stampa in ordine inverso
- elaborazione di liste concatenate
 - conteggi
 - attraversamenti
 - cancellazione
- alberi binari
 - calcolo di parametri
 - visite
 - espressioni

La ricerca binaria

divide and conquer
 $a = 1$ $b = 2$

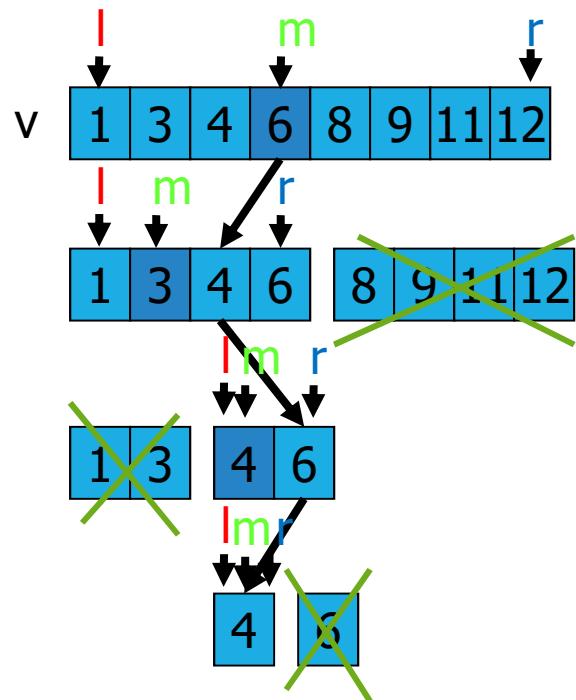
Ricerca binaria o dicotomica: la chiave k è presente all'interno di un vettore ordinato $v[n]$? Sì/No

Approccio

ipotesi: $n = 2^p$

- ad ogni passo: confronto k con elemento centrale del vettore:
 - =: terminazione con successo
 - <: la ricerca prosegue nel sottovettore di SX
 - >: la ricerca prosegue nel sottovettore di DX
- Terminazione: il vettore non è più significativo (l'indice di sinistra ha scavalcato quello di destra)

Esempio

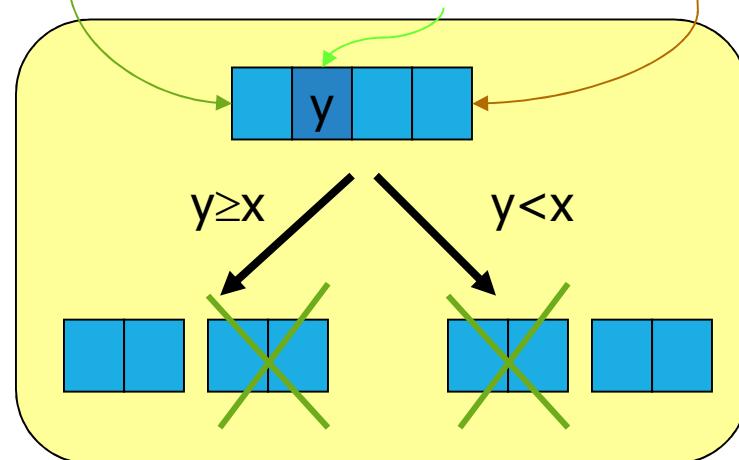


y = elemento di mezzo

| = indice estremo di SX

r = indice estremo di DX

m = indice elemento di mezzo



```
int BinSearch(int v[], int l, int r, int k) {  
    int m;  
    if (l > r)  
        return -1;  
  
    m = (l + r)/2;  
    if (k == v[m])  
        return (m);  
    if (k < v[m])  
        return BinSearch(v, l, m-1, k);  
  
    return BinSearch(v, m+1, r, k);  
}
```



07recursive_binsearch.c

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 1$, $b = 2$

Equazione alla ricorrenze:

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

$$T(1) = 1$$

Risoluzione per sviluppo (unfolding)

$$T(n/2) = T(n/4) + 1$$

$$T(n/4) = T(n/8) + 1$$

$$T(n) = 1 + 1 + 1 + T(n/8)$$

$$= \sum_{i=0}^{\log_2 n} 1$$

$$= 1 + \log_2 n$$

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

terminazione:

$$n/2^i = 1$$

$$i = \log_2 n$$

Stampa in ordine inverso

decrease and conquer
 $a = 1 \ k_i = 1$

- Leggere da input una stringa
- Stamparla in ordine inverso

```
void reverse_print(char *s) {  
    if(*s != '\0') {  
        reverse_print(s+1);  
        putchar(*s);  
    }  
    return;  
}
```



07reverse_print.c

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 1$, $k_i = 1$

Equazione alla ricorrenze:

$$T(n) = 1 + T(n-1) \quad n > 1$$

$$T(1) = 1$$

$$T(n) = O(n)$$

Problemi ricorsivi semplici

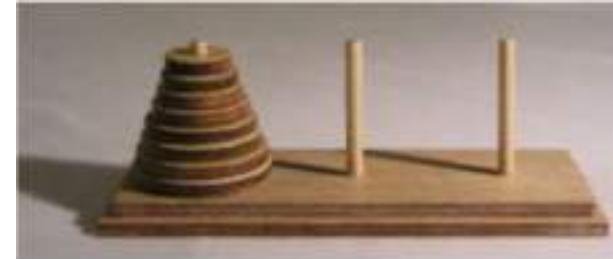
Matematica ricreativa:

- Le Torri di Hanoi
- Il righello.

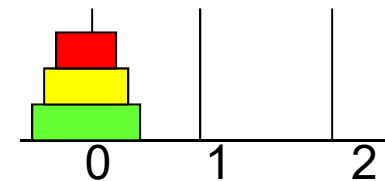
Le Torri di Hanoi (E. Lucas 1883)

decrease and conquer
 $a = 2$ $k_i = 1$

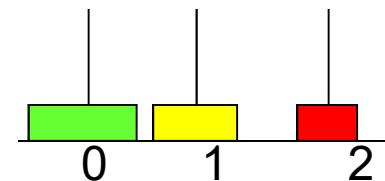
- Configurazione iniziale:
 - vi sono 3 pioli, 3 dischi di diametro decrescente sul primo piolo
- Configurazione finale:
 - 3 dischi sul terzo piolo
- Regole:
 - accesso solo al disco in cima
 - sopra ogni disco solo dischi più piccoli
- Generalizzabile a n dischi e k pioli.



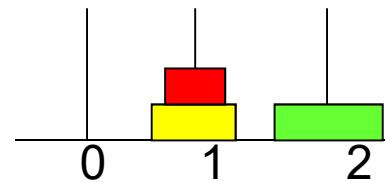
Esempio di soluzione



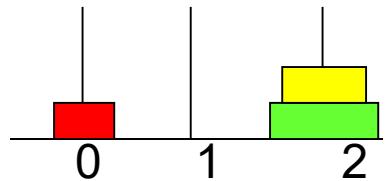
$0 \rightarrow 2$



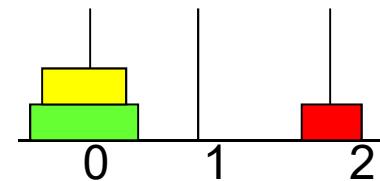
$2 \rightarrow 1$



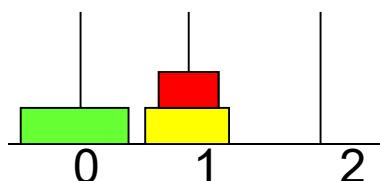
$1 \rightarrow 0$



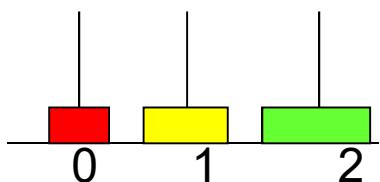
$0 \rightarrow 2$



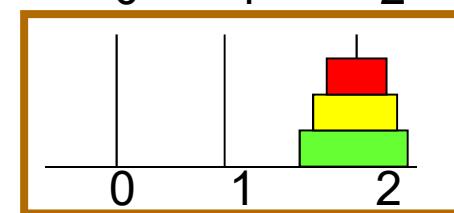
$0 \rightarrow 1$



$0 \rightarrow 2$

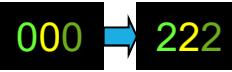


$1 \rightarrow 2$

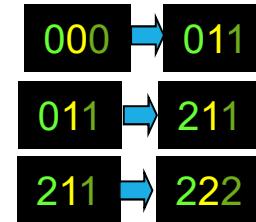


Strategia divide et impera

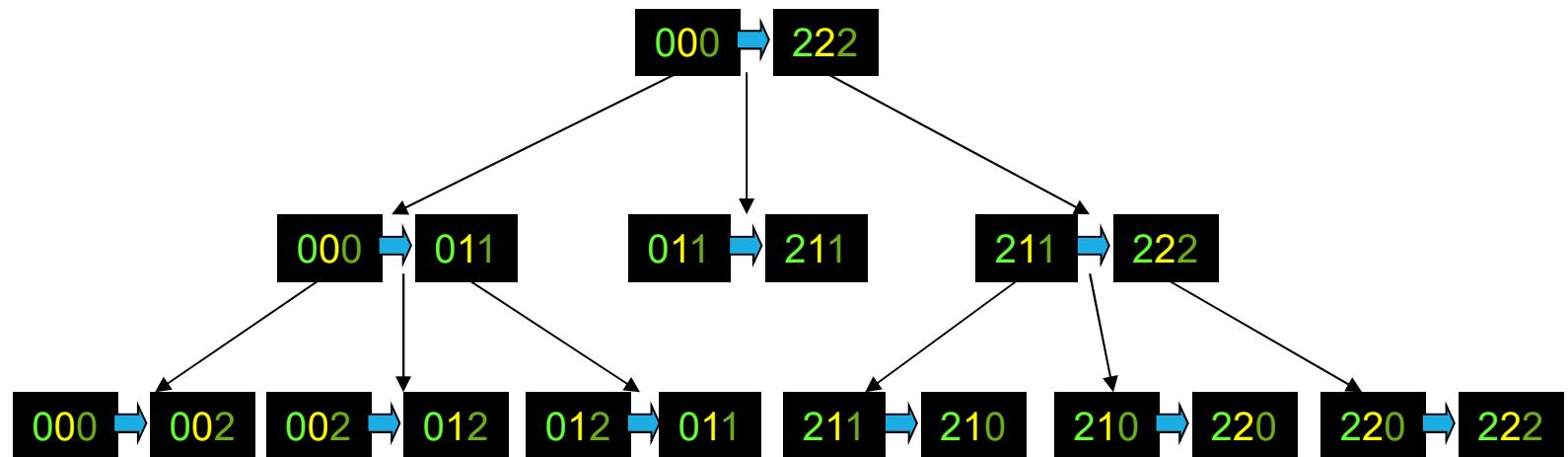
- Problema iniziale: spostare n dischi da 0 a 2
- Riduzione a sottoproblemi:
 - n-1 dischi da 0 a 1, 2 deposito
 - l'ultimo disco da 0 a 2
 - n-1 dischi da 1 a 2, 0 deposito
- Condizione di terminazione: si muove 1 solo disco.
 - 0, 1, 2: piolo 0, 1, 2
 - █ disco grande, █ disco medio, █ disco piccolo
 - 0 significa disco piccolo su piolo 0, 2 significa disco grande su piolo 2, etc.
 - stato 011
 - transizione di stato 

Problema  decomposto in 3 sottoproblemi di cui 1 elementare:

1. dischi medio e piccolo da 0 a 1
2. disco grande da 0 a 2
3. dischi medio e piccolo da 1 a 2.



Albero della ricorsione



```

void Hanoi(int n, int src, int dest) {
    int aux;
    aux = 3 - (src + dest);
    if (n == 1) {
        printf("src %d -> dest %d \n", src, dest);
        return;
    }
    Hanoi(n-1, src, aux); chiamata ricorsiva
    printf("src %d -> dest %d \n", src, dest);
    Hanoi(n-1, aux, dest);
}

```

divisione

terminazione

soluzione elementare

divisione

chiamata ricorsiva



8towers_of_hanoi.c

Analisi di complessità

- Dividi: considera $n-1$ dischi: $D(n) = \Theta(1)$
- Risolfi: risolve 2 sottoproblemi di dimensione $n-1$ ciascuno: $2T(n-1)$
- Terminazione: spostamento di 1 disco: $\Theta(1)$
- Combina: nessuna azione: $C(n) = \Theta(1)$

Equazione alle ricorrenze:

$$T(n) = 2T(n-1) + 1 \quad n > 1$$

$$T(1) = 1$$

$$\begin{aligned} T(n) &= 1 + 2 + 4 + 8T(n-3) \\ &= \sum_{0 \leq i \leq n-1} 2^i \\ &= 2^{n-1+1} - 1 / (2-1) \\ &= 2^n - 1 \end{aligned}$$

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

Il righello

divide and conquer
 $a = 2$ $b = 2$

Tracciare una tacca in ogni punto tra 0 e 2^n estremi esclusi, dove:

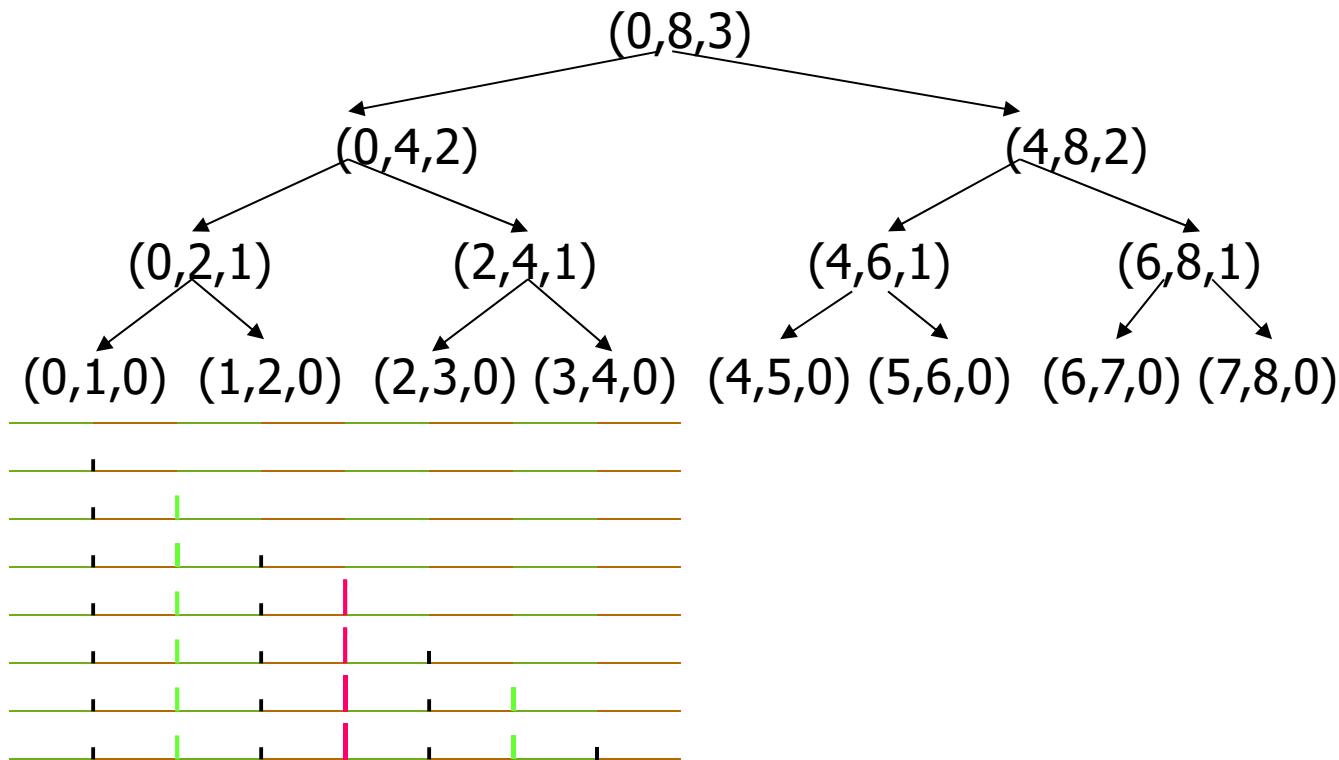
- la tacca centrale è alta n unità,
- le due tacche al centro delle due metà di destra e sinistra sono alte $n-1$
- etc.
- `mark(x, h)` traccia una tacca alta h unità in posizione x

```
void mark(int m, int h) {  
    int i;  
    printf("%d \t", m);  
    for (i = 0; i < h; i++)  
        printf("*");  
    printf("\n");  
}
```

Strategia divide et impera

- Dividiamo l'intervallo in due metà
- Disegniamo ricorsivamente le tacche (più corte) nella metà di SX
- Disegniamo la tacca (più lunga) al centro
- Disegniamo ricorsivamente le tacche (più corte) nella metà di DX
- Condizione di terminazione: tacche di altezza 0

Esempio



divisione

```
void ruler(int l, int r, int h) {  
    int m;  
    m = (l + r)/2;  
    if (h > 0) {  
        ruler(l, m, h-1);  
        mark(m, h);  
        ruler(m, r, h-1);  
    }  
}
```

discesa ricorsiva/terminazione

chiamata ricorsiva

soluzione elementare

chiamata ricorsiva



08ruler.c

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 2$, $b = 2$

Equazione alla ricorrenze:

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

$$T(1) = 1$$

$$T(n) = O(n)$$

Meccanismi computazionali per eseguire funzioni ricorsive

Chiamata a funzione: quando si chiama una funzione:

- si crea una nuova istanza della funzione chiamata
- si alloca memoria per i parametri e per le variabili locali
- si passano i parametri
- il controllo passa dal chiamante alla funzione chiamata
- si esegue la funzione chiamata
- al suo termine, il controllo ritorna al programma chiamante, che esegue l'istruzione immediatamente successiva alla chiamata a funzione.

È possibile che una funzione ne chiami un'altra. Serve un meccanismo per gestire le chiamate annidate e i relativi ritorni: lo **stack**.

Stack

- Definizione: tipo di dato astratto (ADT) che supporta operazioni di:
 - **Push**: inserimento dell'oggetto in cima allo stack
 - **Pop**: prelievo (e cancellazione) dalla cima dell'oggetto inserito più di recente
- Terminologia: la strategia di gestione dei dati è detta LIFO (Last In First Out)

Si chiama **stack frame** (o **record di attivazione**) la struttura dati che contiene almeno:

- i parametri formali
- le variabili locali
- l'indirizzo a cui si ritornerà una volta terminata l'esecuzione della funzione
- il puntatore al codice della funzione.

Lo stack frame viene creato alla chiamata della funzione e distrutto al suo termine.

Gli stack frame sono memorizzati nello stack di sistema.

Lo stack di sistema ha a disposizione una quantità prefissata di memoria. Quando oltrepassa lo spazio allocatogli, c'è **stack overflow**.

Lo stack cresce da indirizzi maggiori a indirizzi minori (quindi verso l'alto). Lo **stack pointer SP** è un registro che contiene l'indirizzo del primo stack frame disponibile.

Esempio

```
int f1(int x);
int f2(int x);

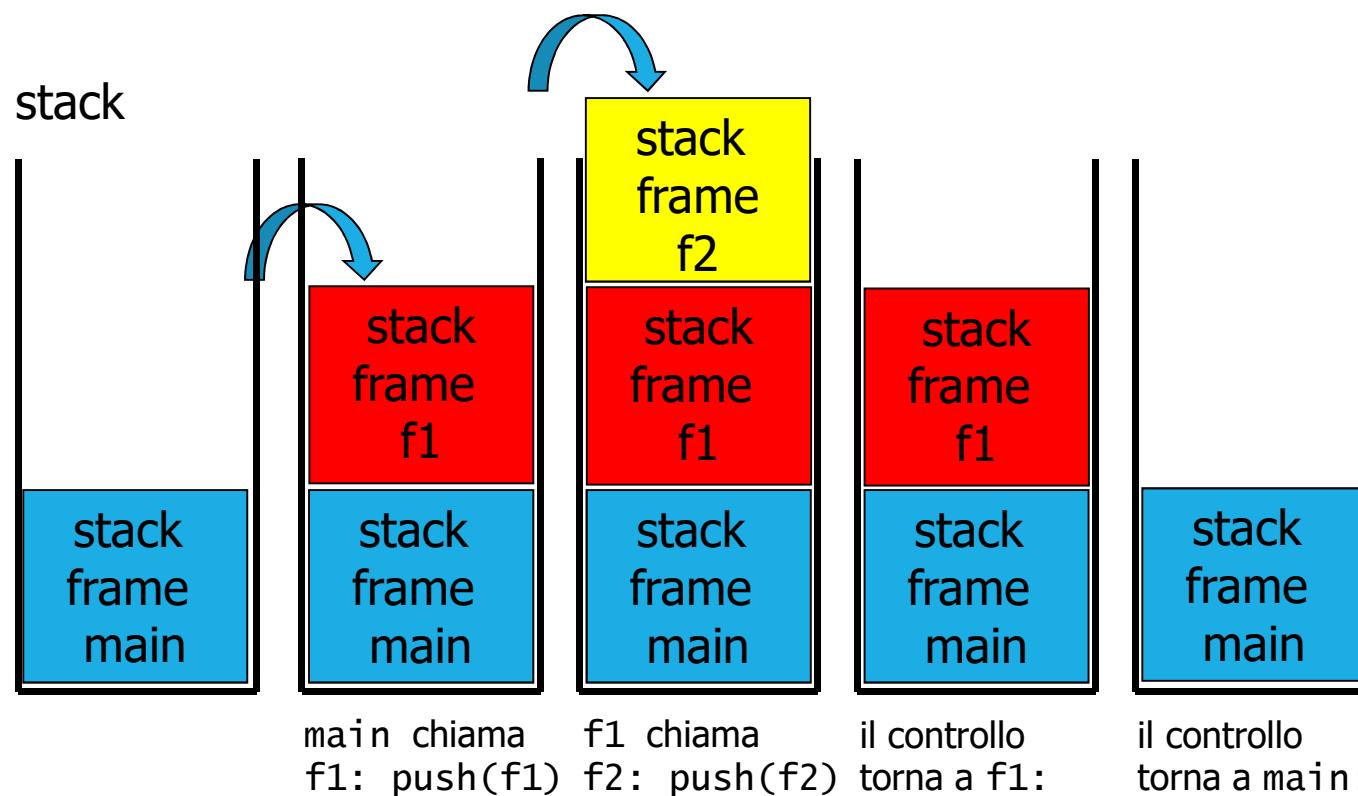
main() {
    int x, a = 10;
    x = f1(a);
    printf("x is %d \n", x);
}

int f1(int x) { return f2(x); }

int f2(int x) { return x+1; }
```



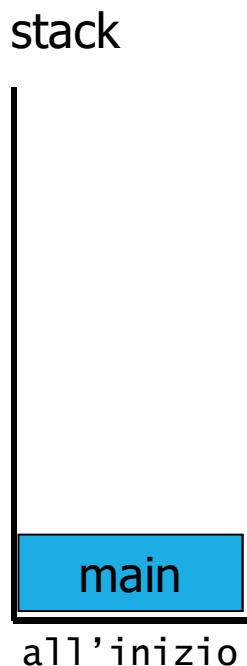
10function_call.c



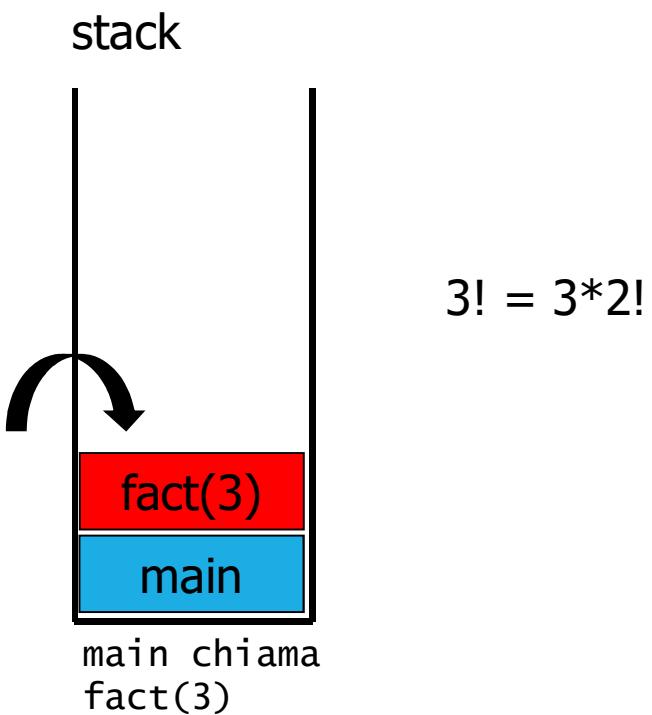
Funzioni ricorsive

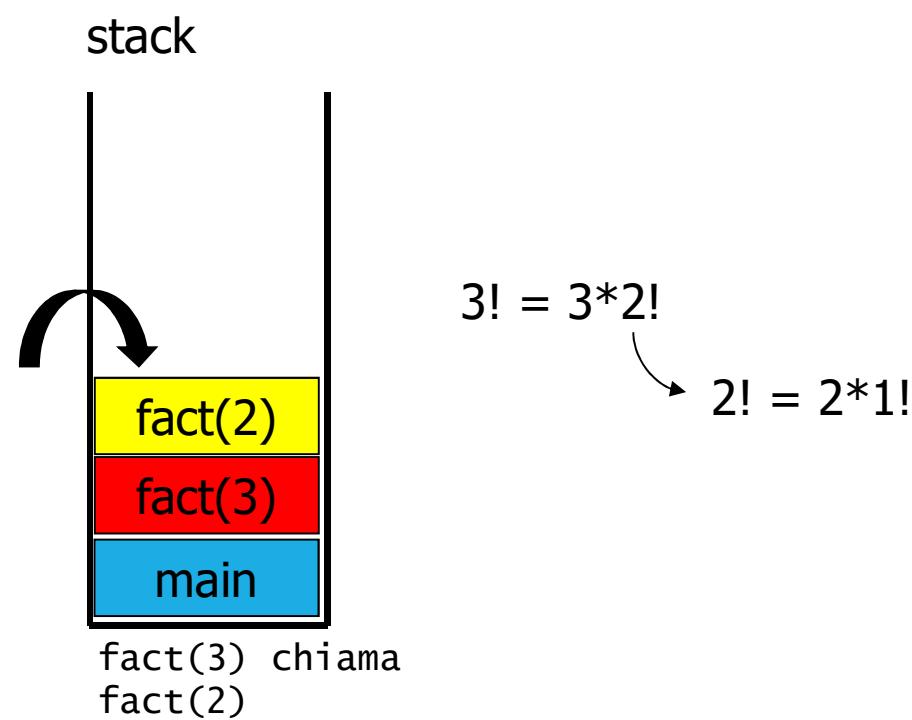
- Funzione chiamante e chiamata coincidono, operano però su valori diversi
- Si usa lo stack di sistema come in una qualsiasi chiamata a funzione
- Un numero eccessivo di chiamate ricorsive può portare allo stack overflow.

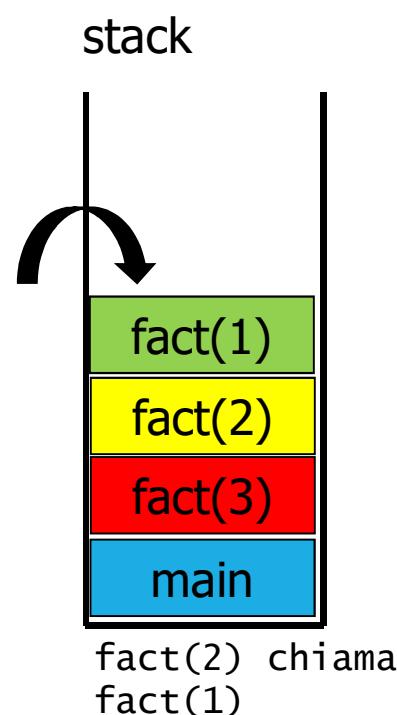
Esempio: calcolo di 3!



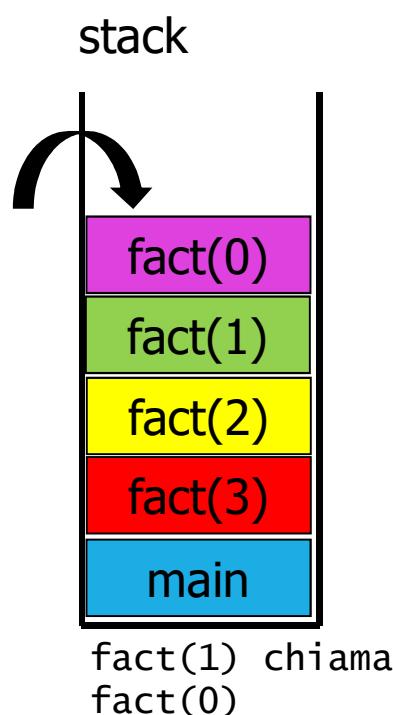
```
int main() {  
    int n;  
    printf("Input n (<=12): ");  
    scanf("%d", &n);  
    printf("%d %lu \n", n, fact(n));  
    return 0;  
}  
unsigned long fact(int n) {  
    if(n == 0)  
        return 1;  
    return n*fact(n-1);  
}
```







$$\begin{aligned}
 3! &= 3 * 2! \\
 &\quad \swarrow \\
 2! &= 2 * 1! \\
 &\quad \swarrow \\
 1! &= 1 * 0!
 \end{aligned}$$



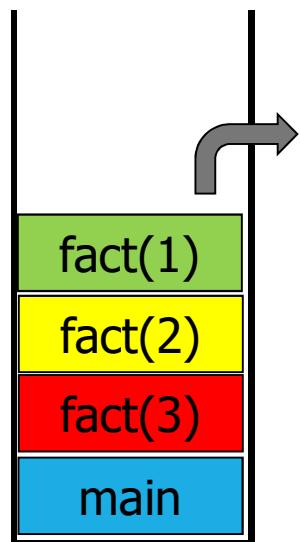
$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

stack



$$3! = 3 * 2!$$

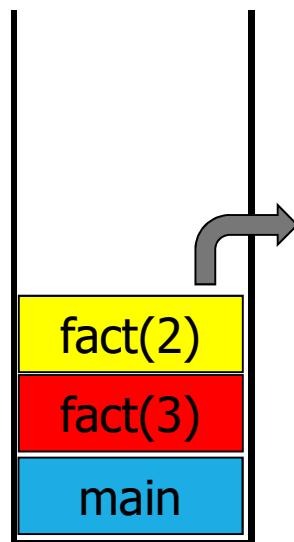
$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

fact(0) termina, restituisce
il valore 1 e torna il controllo
a fact(1)

stack



$$3! = 3 * 2!$$

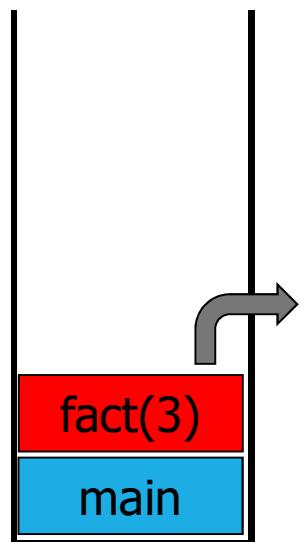
$$2! = 2 * 1!$$

$$1! = 1 * 0! = 1$$

$$0! = 1$$

fact(1) termina, restituisce
il valore 1 e torna il controllo
a fact(2)

stack



$$3! = 3 * 2!$$

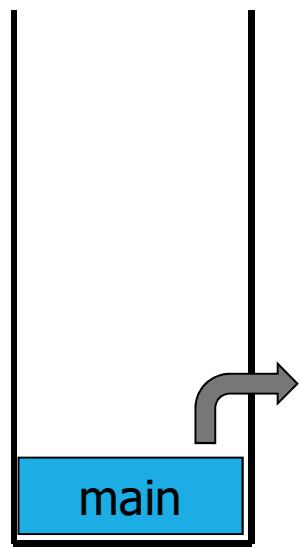
$$2! = 2 * 1! = 2$$

$$1! = 1 * 0! = 1$$

$$0! = 1$$

fact(2) termina, restituisce
il valore 2 e torna il controllo
a fact(3)

stack

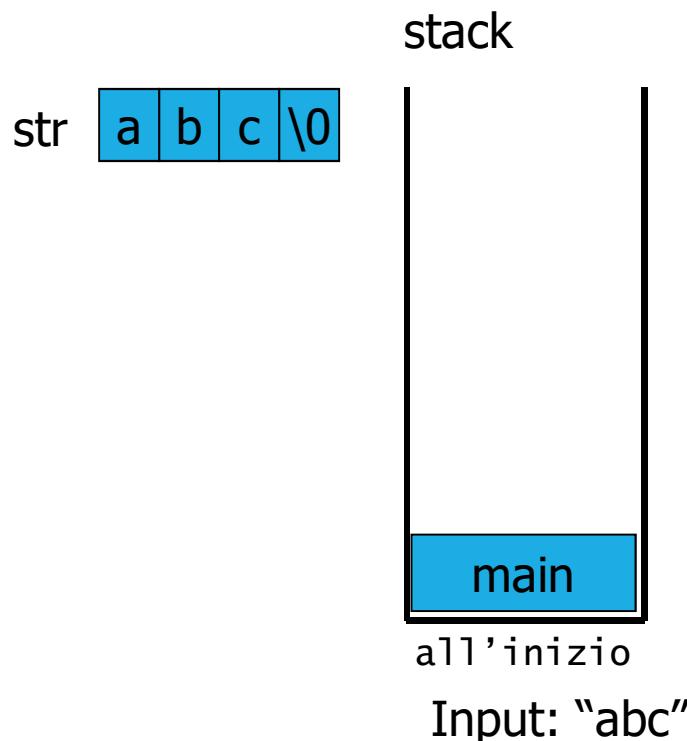


$$\begin{aligned}3! &= 3 * 2! = 6 \\2! &= 2 * 1! = 2 \\1! &= 1 * 0! = 1 \\0! &= 1\end{aligned}$$

The equations show the recursive calculation of factorials. The first equation is explicitly calculated, while the subsequent ones are shown as intermediate steps with arrows pointing from one step to the next.

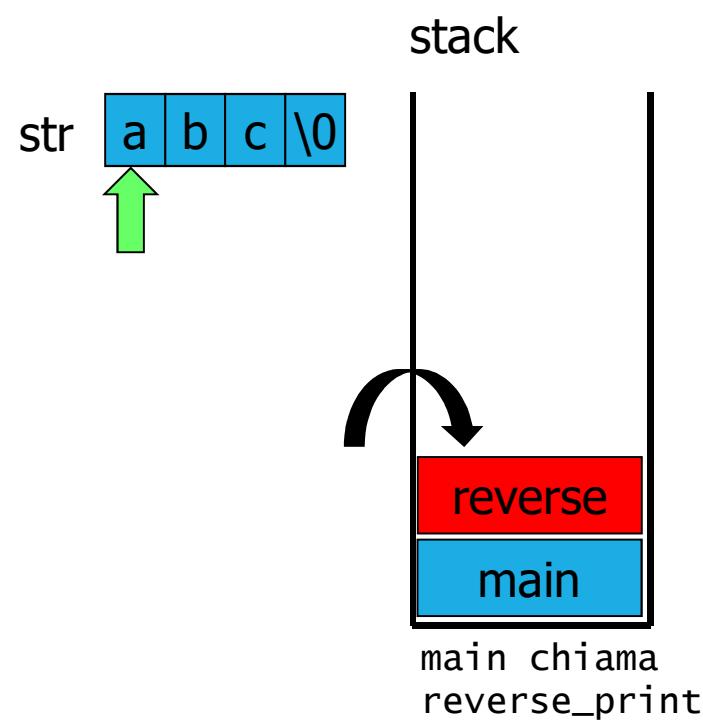
fact(3) termina, restituisce
il valore 6 e torna il controllo
al main

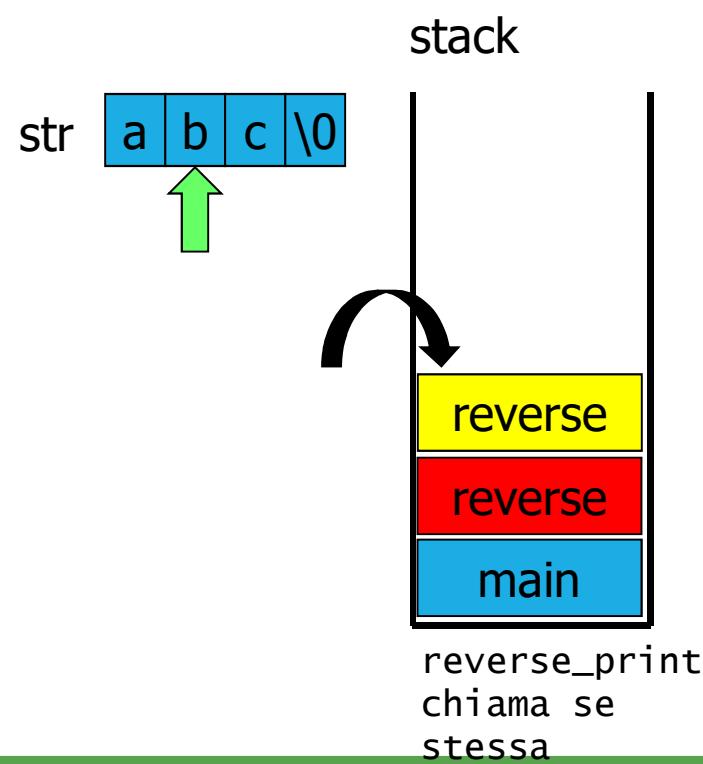
Esempio: reverse_print di "abc"

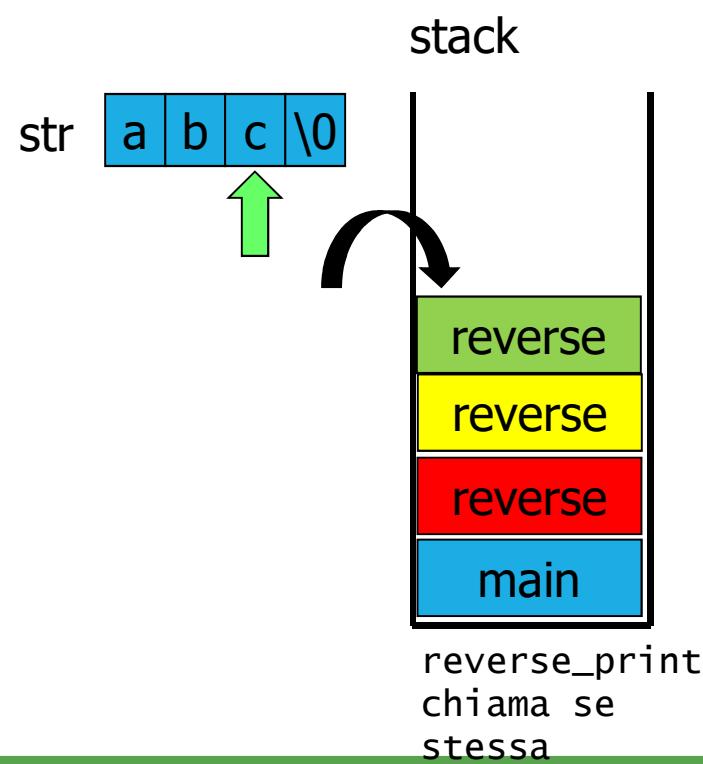


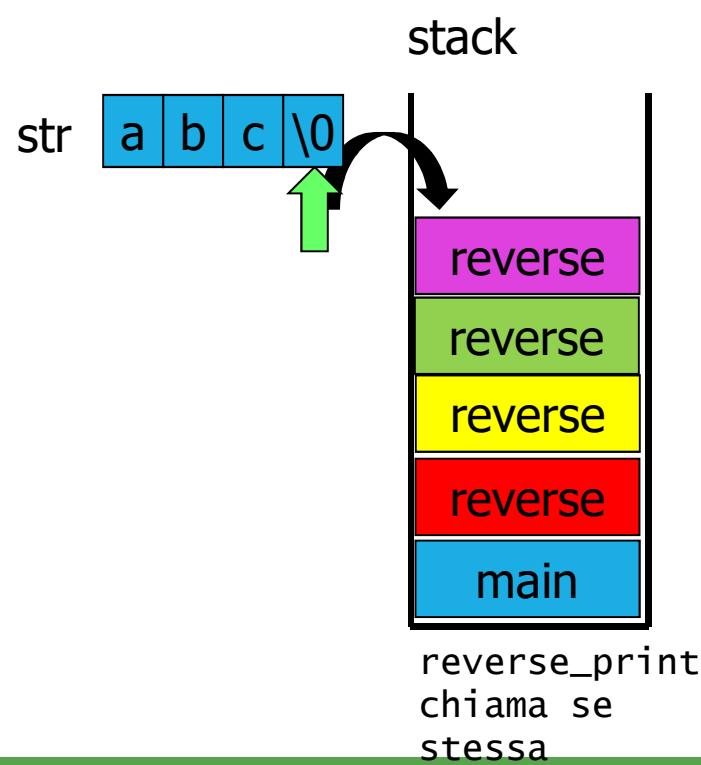
```
int main() {
    char str[max+1];
    printf("Input string: ");
    scanf("%s", str);
    printf("Reverse string is: ");
    reverse_print(str);
    return 0;
}

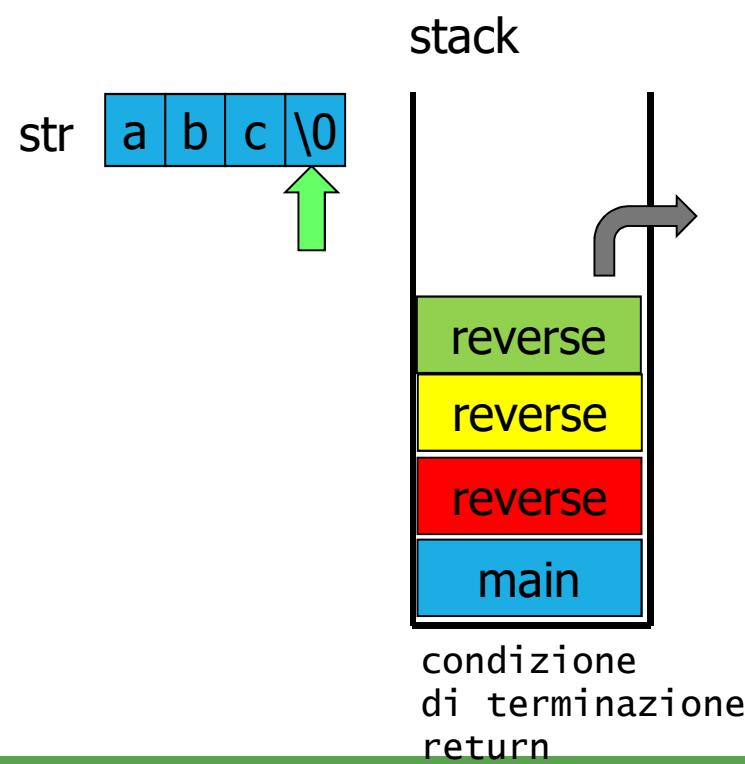
void reverse_print(char *s) {
    if(*s != '\0') {
        reverse_print(s+1);
        putchar(*s);
    }
    return;
}
```

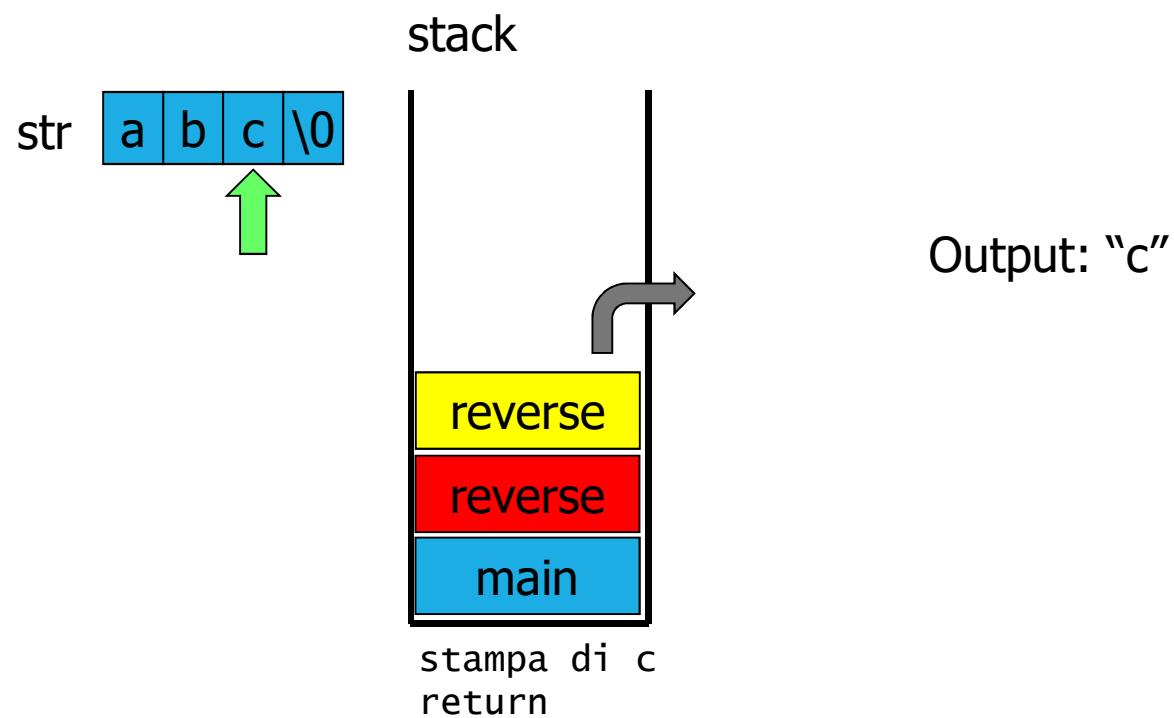


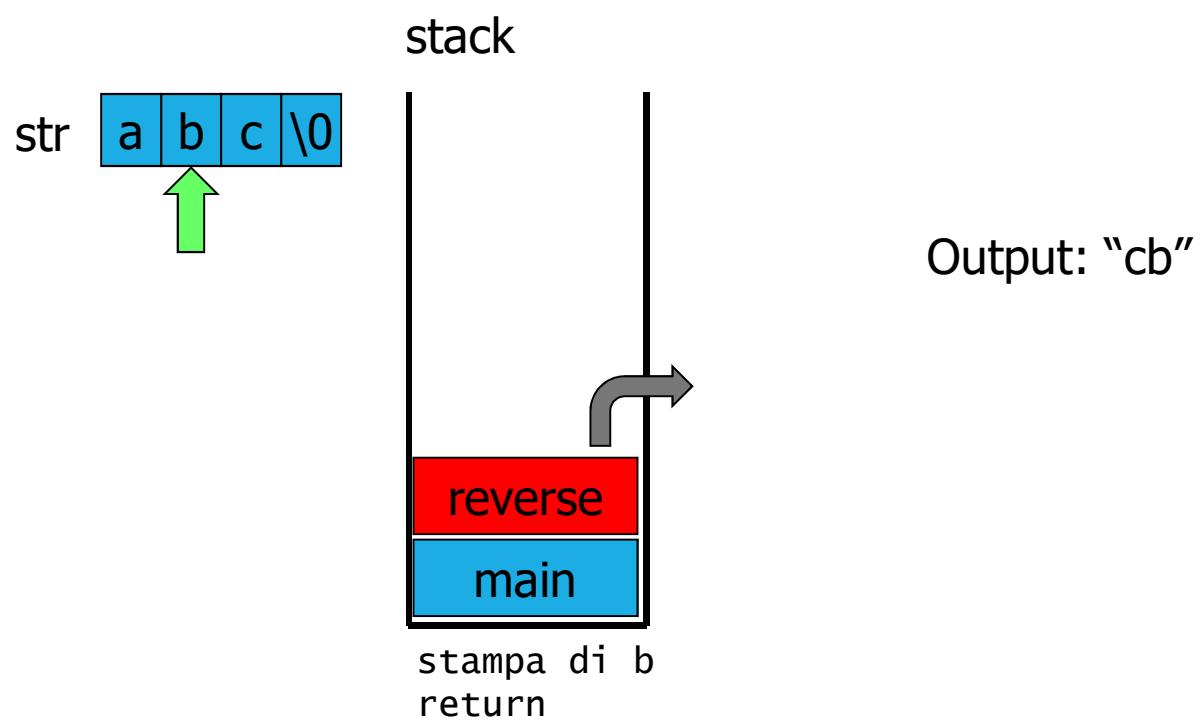


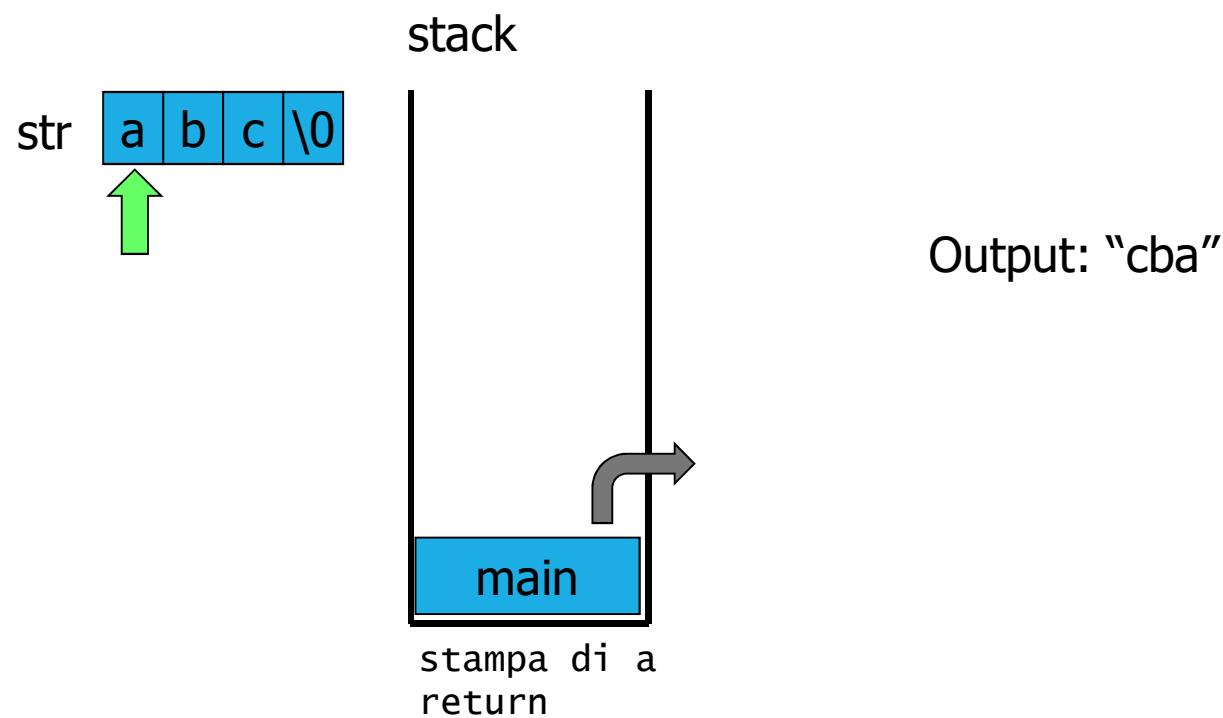












Funzioni non tail-recursive

Una funzione ricorsiva non è **tail-recursive** se la chiamata ricorsiva non è l'ultima operazione da eseguire

```
unsigned long fact(int n) {  
    if(n == 0)  
        return 1;  
    return n*fact(n-1);  
}
```

la moltiplicazione può essere eseguita solo dopo il ritorno della chiamata ricorsiva

i calcoli si fanno in fase di risalita

discesa

risalita

```
fact(3)
3 * fact(2)
3 * (2 * fact(1))
3 * (2 * (1 * fact(0)))
3 * (2 * (1 * 1))

3 * (2 * 1)
3 * 2
6
```

È necessario mantenere uno stack!

Funzioni tail-recursive

Nel main:

```
result = tr_fact(n, 1);
```

Una funzione ricorsiva è **tail-recursive** se la chiamata ricorsiva è l'ultima operazione da eseguire, eccezion fatta per **return**

```
unsigned long tr_fact(int n, unsigned long f)
{
    if(n == 0)
        return f;
    return tr_fact(n-1, n*f);
}
```

è tail-recursive perché
la moltiplicazione viene
eseguita prima della
chiamata ricorsiva



i calcoli si fanno in fase
di discesa



11tail_recursive_factorial.c

discesa

```
tr_fact(3,1)
tr_fact(2,3)
tr_fact(1,6)
tr_fact(0,6)
```

Se una funzione è tail-recursive la funzione chiamante (caller) deve solo ritornare il valore calcolato dalla funzione chiamata (callee)



l'operazione di pop dello stack frame della funzione chiamante avviene prima dell'operazione di push nello stack frame della funzione chiamata



lo stack frame della funzione chiamata rimpiazza semplicemente quello della funzione chiamante



l'occupazione di memoria non è più $O(n)$ dove n sono i livelli di ricorsione, bensì $O(1)$



non ci può più più essere stack overflow

Svantaggi:

- più difficili da scrivere
- non tutti i compilatori (tra cui quello del C) riescono a sfruttare le funzioni tail-recursive per ottimizzare il codice.

Dualità ricorsione - iterazione

Possibili soluzioni:

- iterativa “nativa”
- ricorsiva:
 - se tail-recursive trasformabile direttamente in iterativa senza uso di stack
 - se non tail-recursive trasformabile in iterativa con uso di stack. In generale soluzione meno efficienti di quelle iterative “native”.

direttamente in forma iterativa

Fibonacci:

$$FIB_0 = 0$$

$$FIB_1 = 1$$

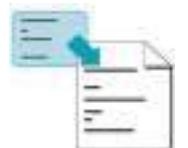
$$FIB_2 = FIB_0 + FIB_1 = 1$$

$$FIB_3 = FIB_1 + FIB_2 = 2$$

$$FIB_4 = FIB_2 + FIB_3 = 3$$

$$FIB_5 = FIB_3 + FIB_4 = 5$$

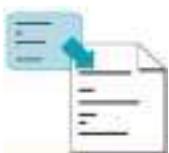
```
unsigned long fib(int n) {  
    unsigned long f1p=1, f2p=0, f;  
    int i;  
    if(n == 0 || n == 1)  
        return n;  
    f = f1p + f2p; /* n==2 */  
    for(i=3; i<= n; i++) {  
        f2p = f1p;  
        f1p = f;  
        f = f1p+f2p;  
    }  
    return f;  
}
```



12iterative_fibonacci.c

Ricerca binaria

```
int BinSearch(int v[], int N, int k) {  
    int m, found= 0, l=0, r=N-1;  
    while(l <= r && found == 0){  
        m = (l+r)/2;  
        if(v[m] == k)  
            found = 1;  
        if(v[m] < k)  
            l = m+1;  
        else  
            r = m-1;  
    }  
    if (found == 0)  
        return -1;  
    return m;  
}
```



13iterative_binsearch.c

direttamente in forma iterativa

Fattoriale

```
unsigned long fact(int n) {  
    unsigned long tot = 1;  
    int i;  
  
    for (i=2; i<=n; i++)  
        tot = tot * i;  
  
    return tot;  
}
```

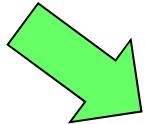


14iterative_factorial.c

da tail-recursive a forma iterativa

Fattoriale

```
unsigned long tr_fact(int n, unsigned long f) {  
    if(n == 0)  
        return f;  
    return tr_fact(n-1, n*f);  
}
```



```
unsigned long tr2iterfact(int n, unsigned long f){  
    while (n > 0) {  
        f = n * f;  
        n--;  
    }  
    return f;  
}
```



15tr2iterfact.c

```
unsigned long fact(int n)
{
    if(n == 0)
        return 1;
    return n*fact(n-1);
}
```

da non tail-recursive
a iterativo con stack
gestito dall'utente

ADT S (stack)

```
unsigned long fact (int n){
    unsigned long f = 1;
    S stack;
    stack = STACKinit(N);

    while (n>0) {
        STACKpush(stack, n);
        n--;
    }
    while (STACKsize(stack) > 0) {
        n = STACKpop(stack);
        f = n * f;
    }
    return f;
}
```

Riferimenti

- Ricorsione
 - Sedgewick 5.1
 - Deitel 5.14, 5.15
- Divide et Impera
 - Sedgewick 5.2
 - Cormen 1.3.1
- Risoluzione di Equazioni alle Ricorrenze:
 - Cormen 4.2
- Algoritmo di Karatsuba:
 - Crescenzi 2.5.2
- Chiamata di funzioni
 - Deitel 5.7

Gli Ordinamenti Ricorsivi

Paolo Camurati



I dati da ordinare

- I dati da ordinare non sono necessariamente sempre e solo interi.
- Generalizzazione: i dati da ordinare appartengono ad un tipo Item definito come struct
 - uno dei campi = *chiave* di ordinamento
 - restanti campi = dati aggiuntivi
- funzioni di interfaccia a oggetti di tipo Item
 - lettura/scrittura
 - generazione di valori casuali
 - accesso alla chiave
- operatori relazionali su oggetti di tipo Item

Item definito come:

- ADT di I classe
 - Quasi ADT
- } trattati più avanti

Tipologie:

1. semplice scalare e chiave coincidente
2. vettore dinamico di caratteri e chiave coincidente
3. scalare e vettore di caratteri sovradimensionato staticamente in una **struct**
4. scalare e vettore di caratteri allocato dinamicamente in una **struct**.

Cfr. *Puntatori e strutture dati dinamiche* cap. 6.2

Esempio: Quasi ADT – tipologia 1

```
...  
#define maxKey 100  
typedef int Item;
```

definizione di un
nuovo tipo **Item**

```
Item ITEMscan();  
int ITEMeq(Item A, Item B);  
int ITEMneq(Item A, Item B);  
int ITEMlt(Item A, Item B);  
int ITEMgt(Item A, Item B);  
void ITEMshow(Item A);  
Item ITEMrand();
```

prototipi di funzioni su
dati di tipo **Item**

```
...
int ITEMeq(Item A, Item B) {
    return (A == B);
}

int ITEMneq(Item A, Item B) {
    return (A != B);
}

int ITEMlt(Item A, Item B) {
    return (A < B);
}

int ITEMgt(Item A, Item B) {
    return (A > B);
}
...
```

implementazione
di funzioni su dati
di tipo Item

```
Item ITEMscan(){
    Item A;
    printf("item = "); scanf("%d", &A);
    return A;
}

void ITEMshow(Item A) {
    printf("%6d \n", A);
}

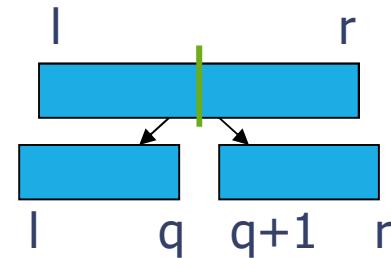
Item ITEMrand() {
    Item A= maxKey*((float)rand()/RAND_MAX);
    return A;
}
```

Merge Sort (von Neumann, 1945)



Divisione:

- due sottovettori SX e DX rispetto al centro del vettore.



Ricorsione

- condizione di terminazione: con 1 ($l=r$) o 0 ($l>r$) elementi il vettore è ordinato
- merge sort su sottovettore SX
- merge sort su sottovettore DX

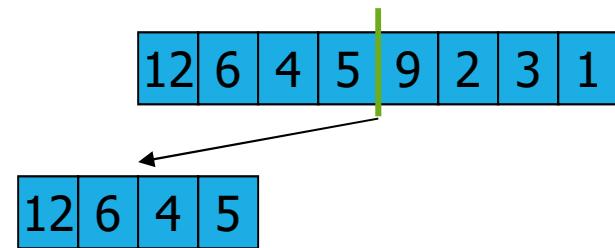
Ricombinazione:

- fondi i due sottovettori ordinati in un vettore ordinato.

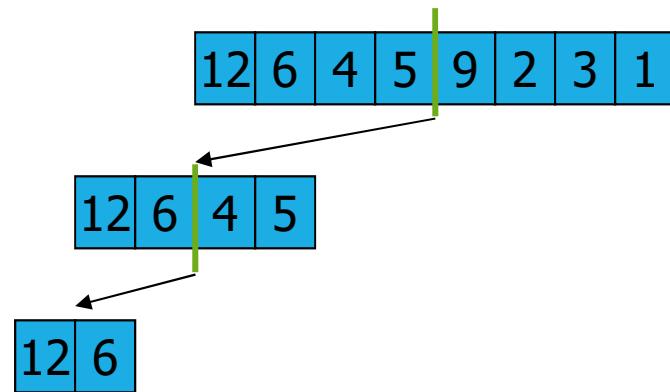
Esempio

12	6	4	5	9	2	3	1
----	---	---	---	---	---	---	---

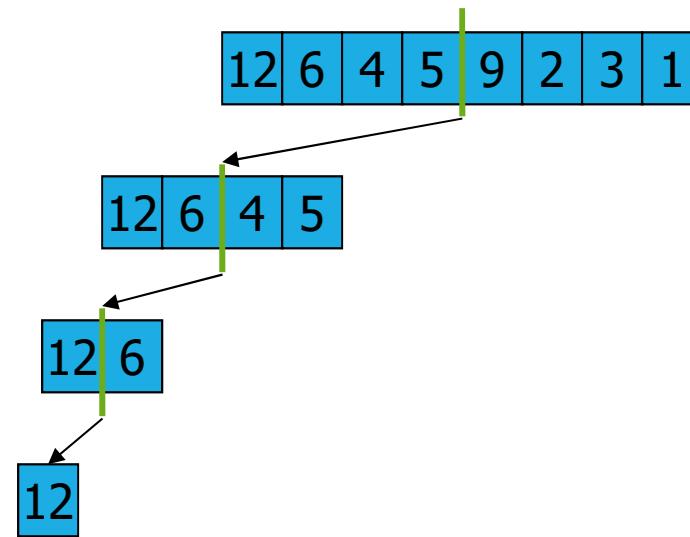
Divisione ricorsiva:



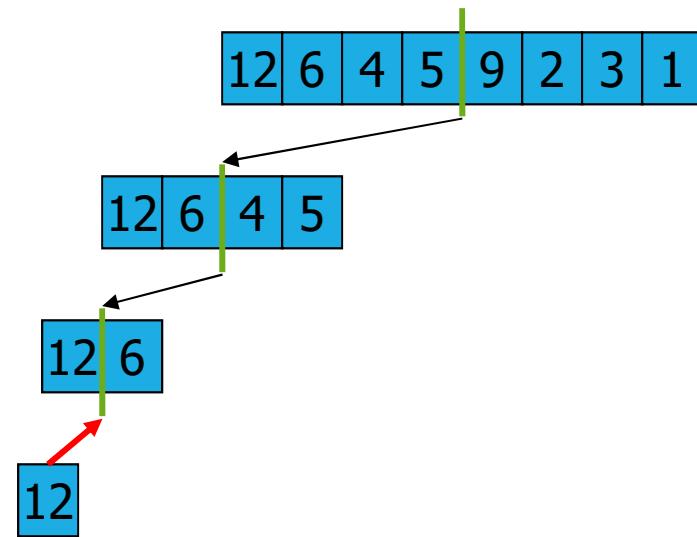
Divisione ricorsiva:



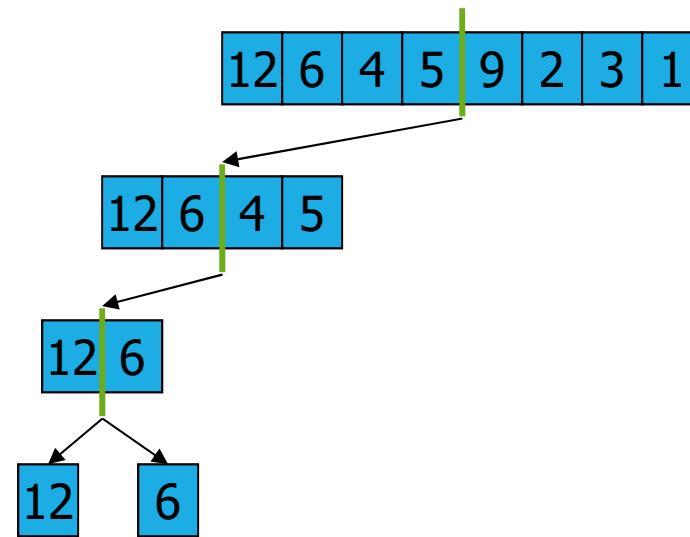
Divisione ricorsiva:



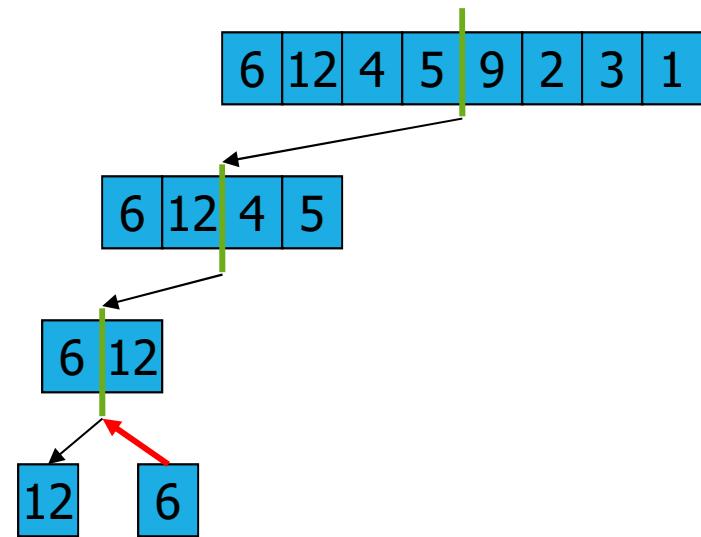
Ricombinazione:



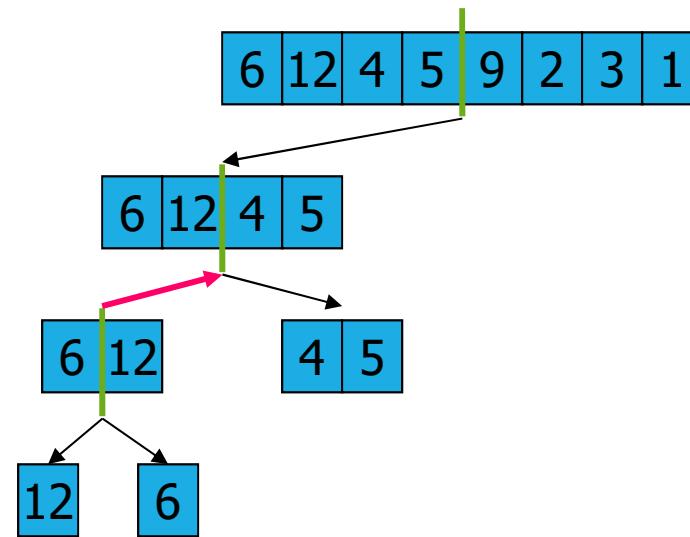
Divisione ricorsiva:



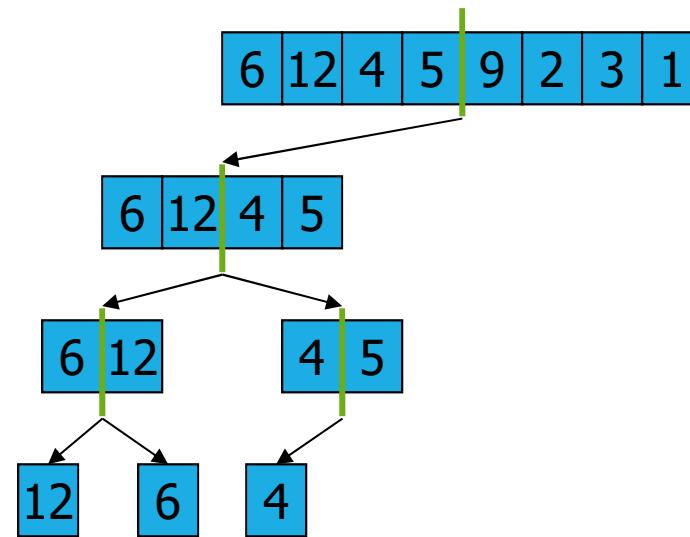
Ricombinazione:



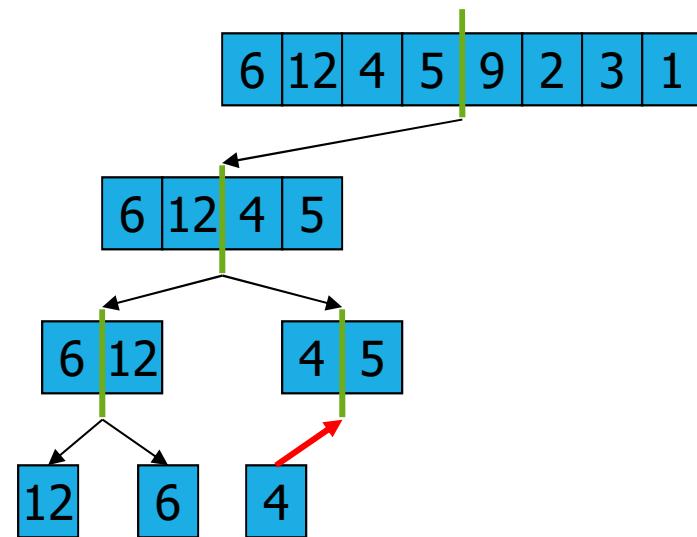
Divisione ricorsiva:



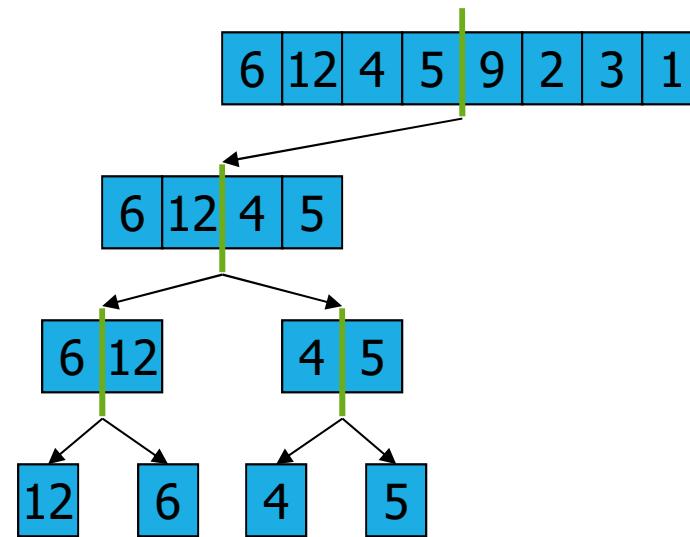
Divisione ricorsiva:



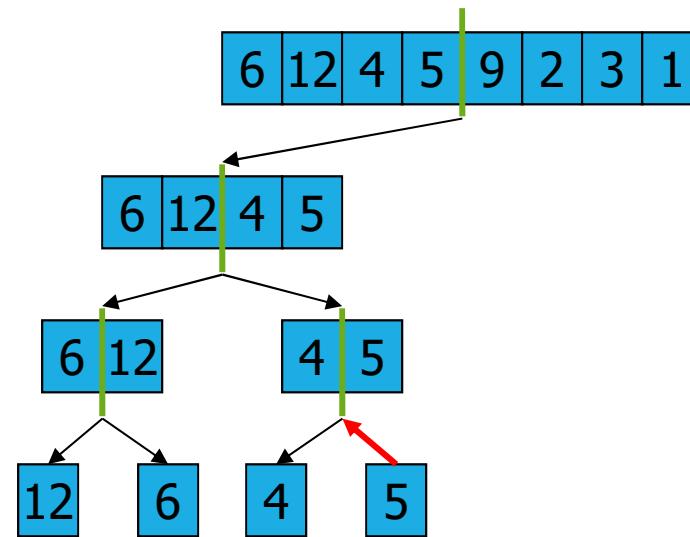
Ricombinazione:



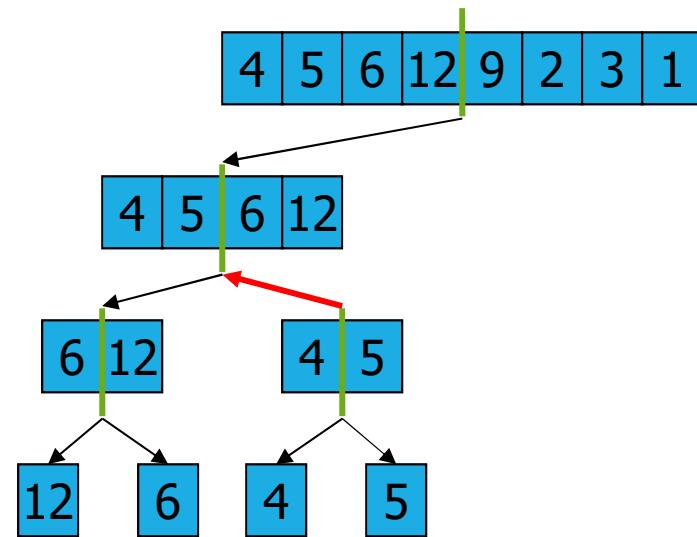
Divisione ricorsiva:



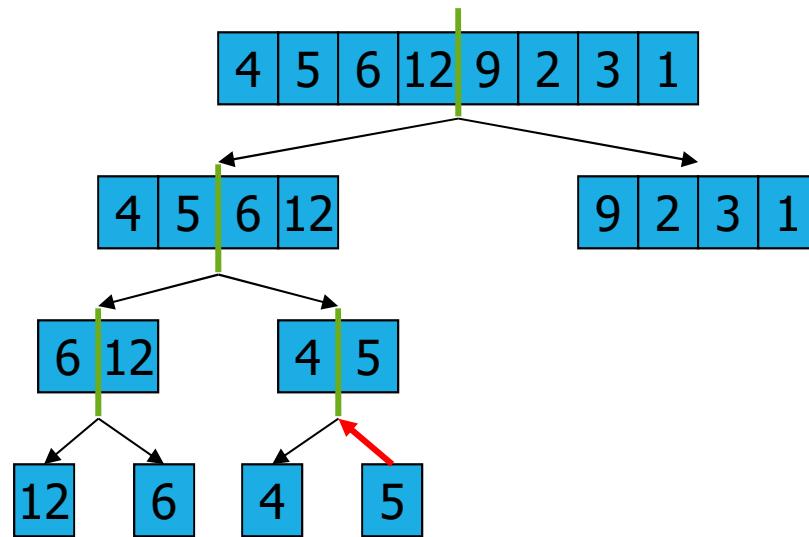
Ricombinazione:



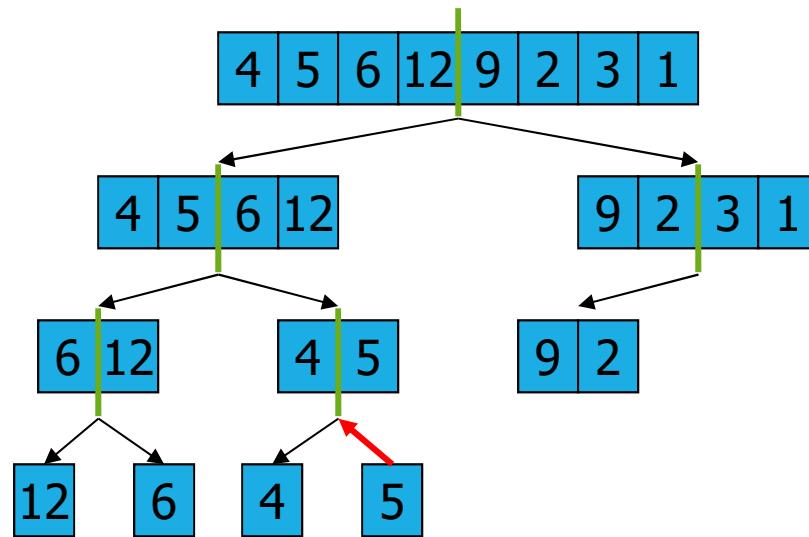
Ricombinazione:



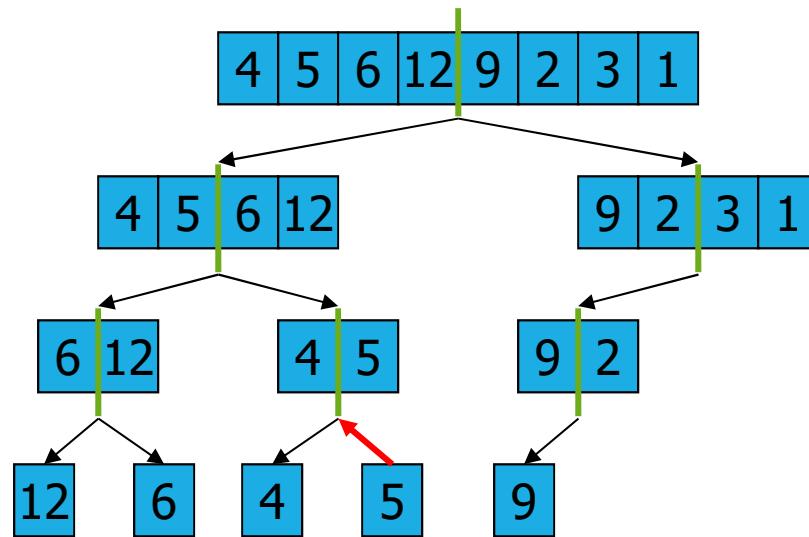
Discesa ricorsiva:



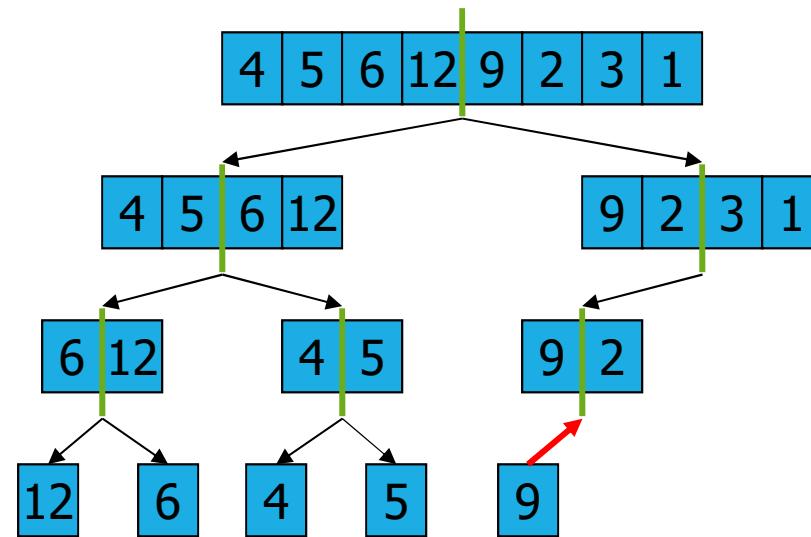
Discesa ricorsiva:



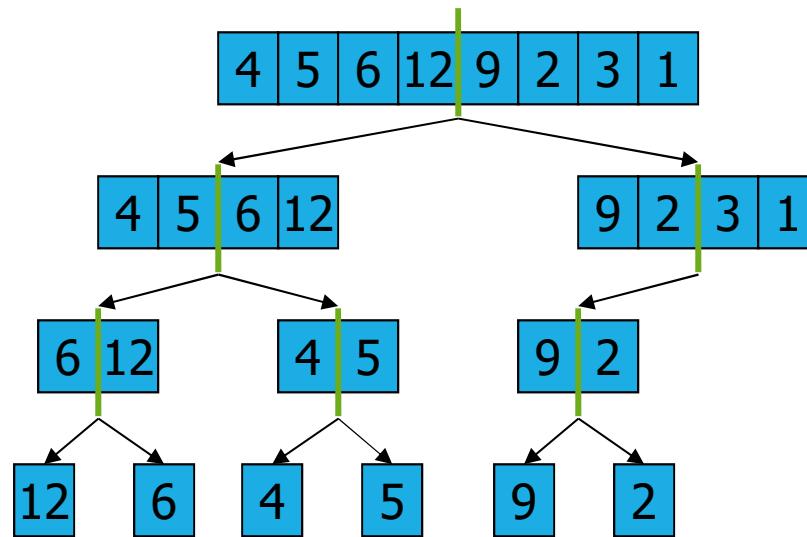
Discesa ricorsiva:



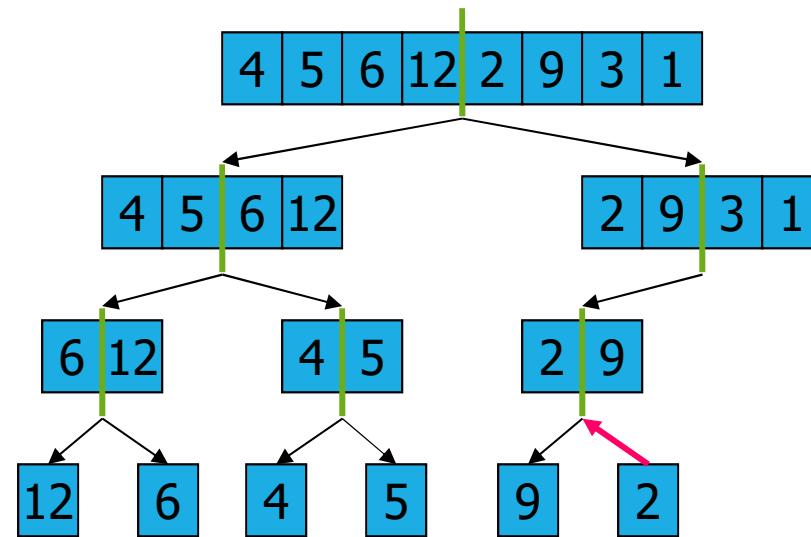
Ricombinazione:



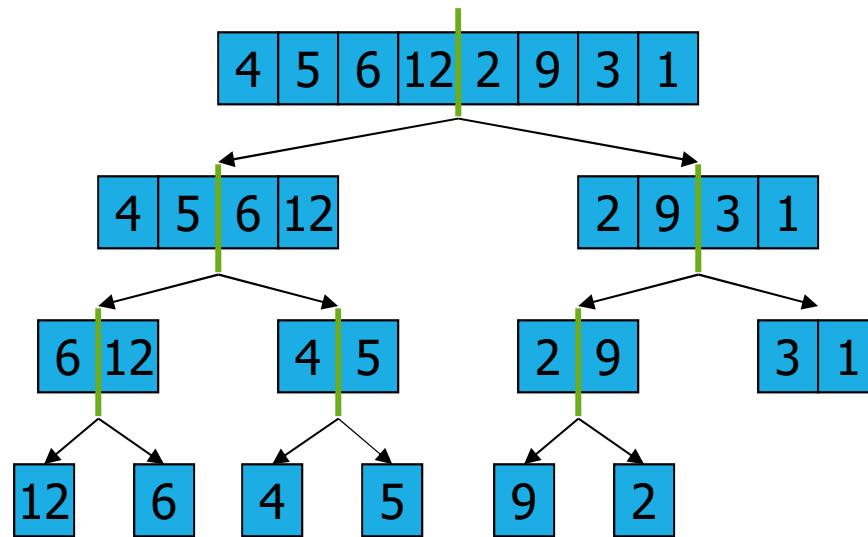
Discesa ricorsiva:



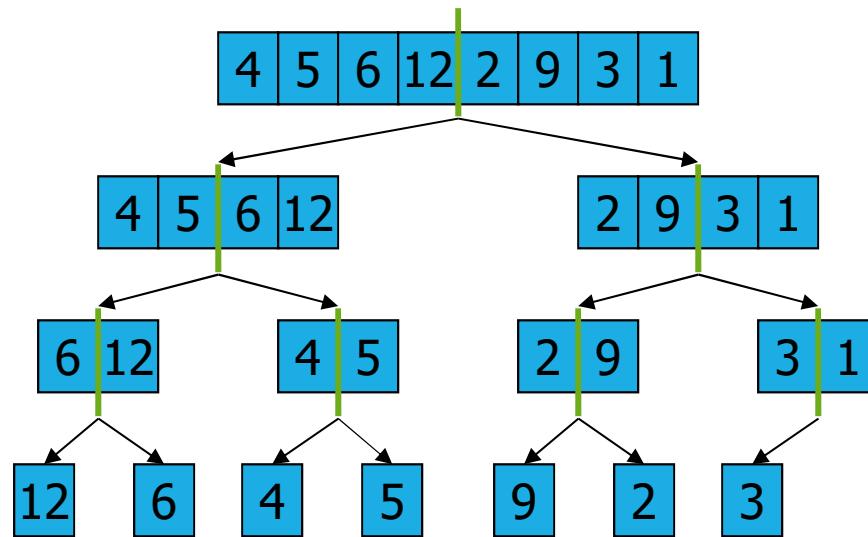
Ricombinazione:



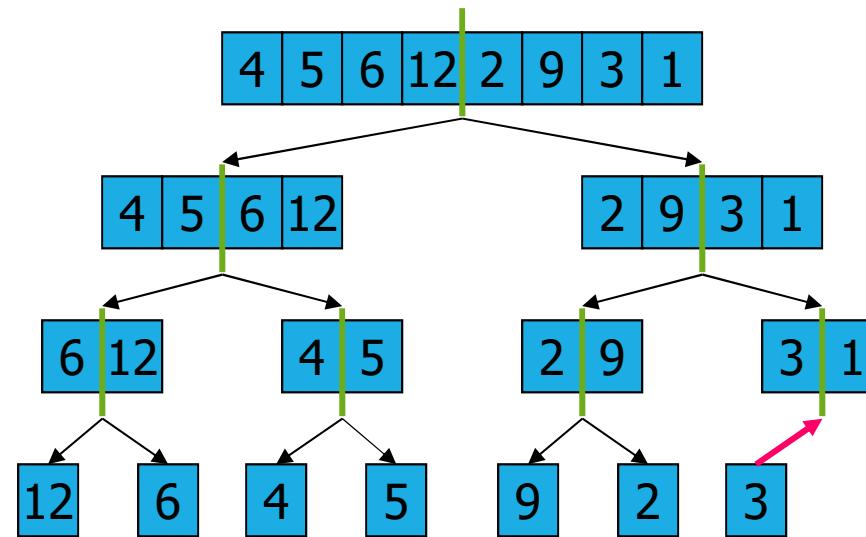
Discesa ricorsiva:



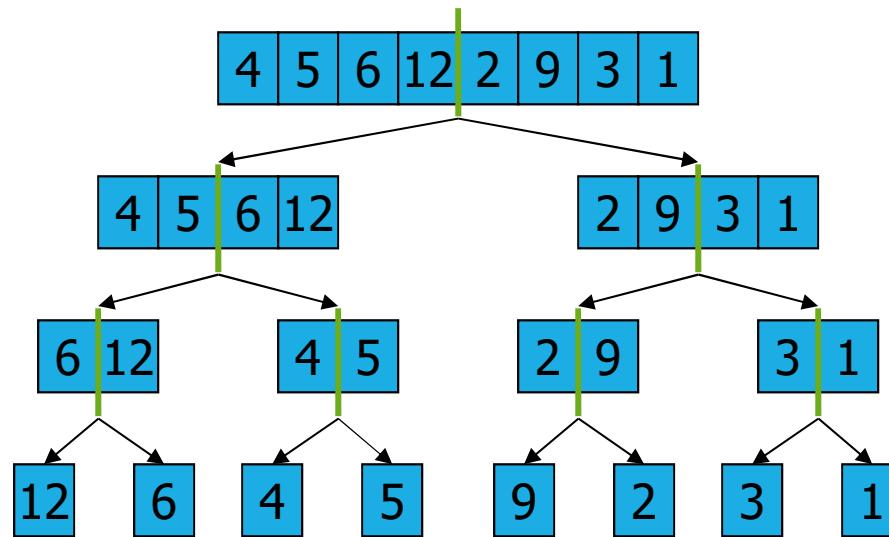
Discesa ricorsiva:



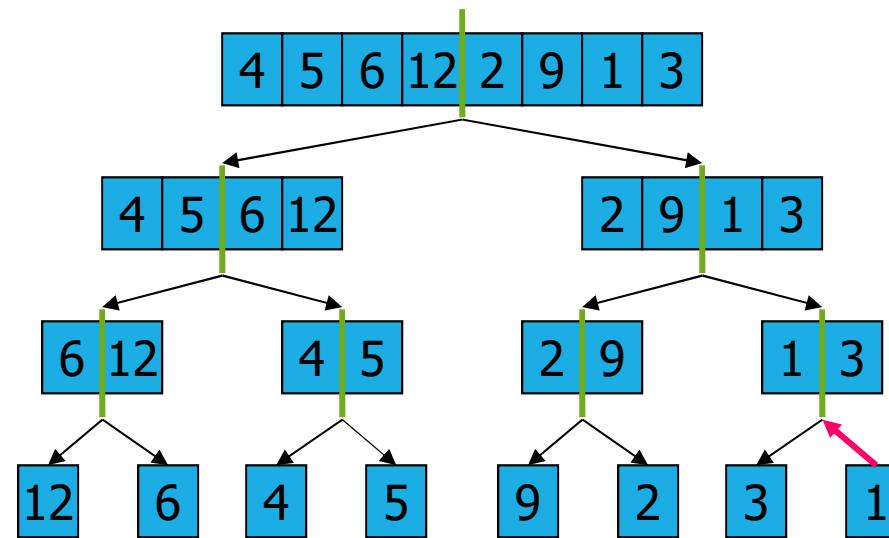
Ricombinazione:



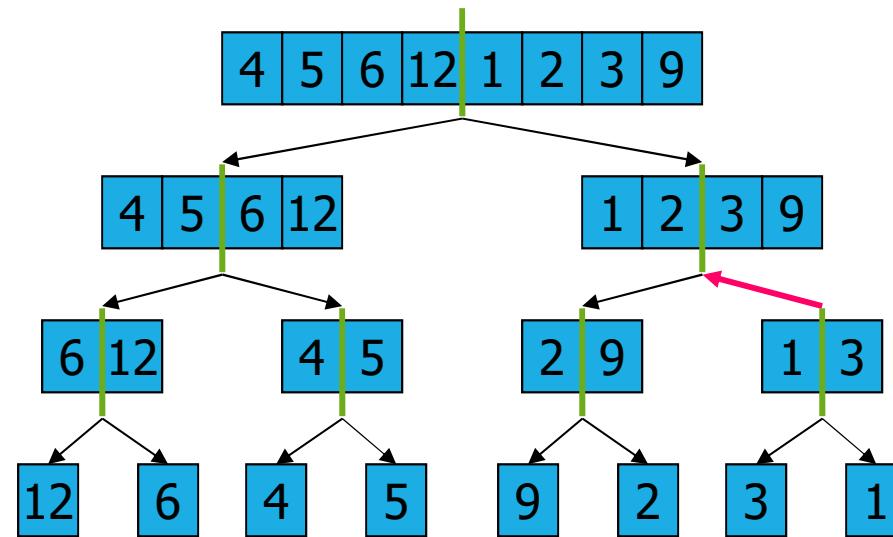
Discesa ricorsiva:



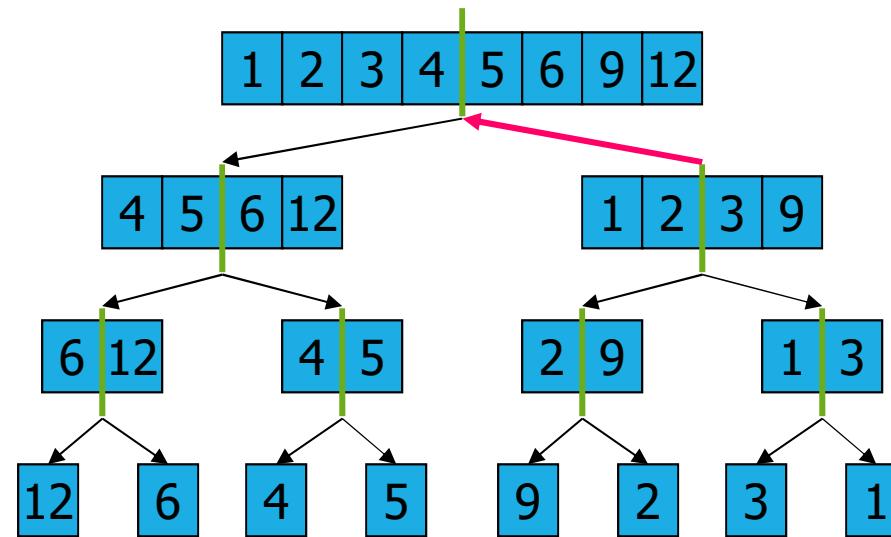
Ricombinazione:



Ricombinazione:



Ricombinazione:



```
void MergeSort(Item A[], Item B[], int N) {  
    int l=0, r=N-1;  
    MergeSortR(A, B, l, r);  
}
```

wrapper

vettore ausiliario

estremi

chiamata alla
funzione ricorsiva



RecursiveSort.c

```
void MergeSortR(Item A[], Item B[], int l, int r){  
    int q;  
    if (l >= r) —————— terminazione  
        return;  
    q = (l + r)/2;  
    MergeSortR(A, B, l, q);  
    MergeSortR(A, B, q+1, r);  
    Merge(A, B, l, q, r);  
}
```

terminazione

divisione

chiamata ricorsiva

chiamata ricorsiva

ricombinazione

2-way Merge

- ipotesi: la dimensione del vettore A è una potenza di 2 $N = 2^p$
- fusione di 2 (2-way) sottovettori di A ordinati di ugual dimensione per ottenere un vettore ordinato di dimensione doppia
- generalizzabile a k vettori (k-way Merge)
- indice q per dividere sottovettori di A a metà in 2 sottovettori sinistro e destro
- sottovettore sinistro con indice $l \leq i \leq q$
- sottovettore destro con indice $q+1 \leq j \leq r$
- vettore ausiliario B di dimensione N con indice $l \leq k \leq r$ per memorizzare i risultati delle fusioni passato come parametro

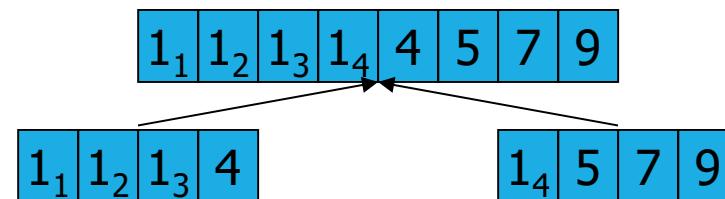
Approccio:

- scorrere i sottovettori sinistro e destro mediante gli indici i e j e il vettore B mediante l'indice k
 - se è esaurito il sottovettore sinistro ($i > q$), ricopiare gli elementi rimanenti del sottovettore destro in B
 - altrimenti se è finito il sottovettore destro ($j > r$), ricopiare gli elementi rimanenti del sottovettore sinistro in B
 - altrimenti confrontare l'elemento corrente $A[i]$ del sottovettore sinistro con quello del sottovettore destro $A[j]$
 - se $A[i] \leq A[j]$, ricopiare $A[i]$ in B e avanzare i , j resta invariato
 - altrimenti ricopiare $A[j]$ in B e avanzare j , i resta invariato
- I confronti avvengono mediante operatori relazionali su item.

```
void Merge(Item A[], Item B[], int l, int q, int r) {  
    int i, j, k;  
    i = l;           esaurito sottovettore SX  
    j = q+1;  
    for (k = l; k <= r; k++)  
        if (i > q)  
            B[k] = A[j++];  
        else if (j > r)  
            B[k] = A[i++];  
        else if (ITEMlt(A[i], A[j]) || ITEMeq(A[i], A[j]))  
            B[k] = A[i++];  
        else  
            B[k] = A[j++];  
    for (k = l; k <= r; k++)  
        A[k] = B[k];  
    return;  
}
```

Caratteristiche

- Non in loco (usa il vettore ausiliario B la cui dimensione è funzione di N)
- Stabile: in quanto la funzione merge prende dal sottovettore SX in caso di chiavi uguali:



Analisi di complessità

Ipotesi: $n = 2^k$ solo ai fini dell'analisi.

- Dividi: calcola la metà di un vettore $D(n) = \Theta(1)$
- Risolvi: risolve 2 sottoproblemi di dimensione $n/2$ ciascuno $2T(n/2)$
- Terminazione: semplice test $\Theta(1)$
- Combina: basata su Merge $C(n) = \Theta(n)$
- $C(n) + D(n) = \Theta(n)$

Equazione alle ricorrenze:

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

$$T(1) = 1 \quad n = 1$$

- soluzione per sviluppo (unfolding)

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

$$T(n/4) = 2T(n/8) + n/4$$

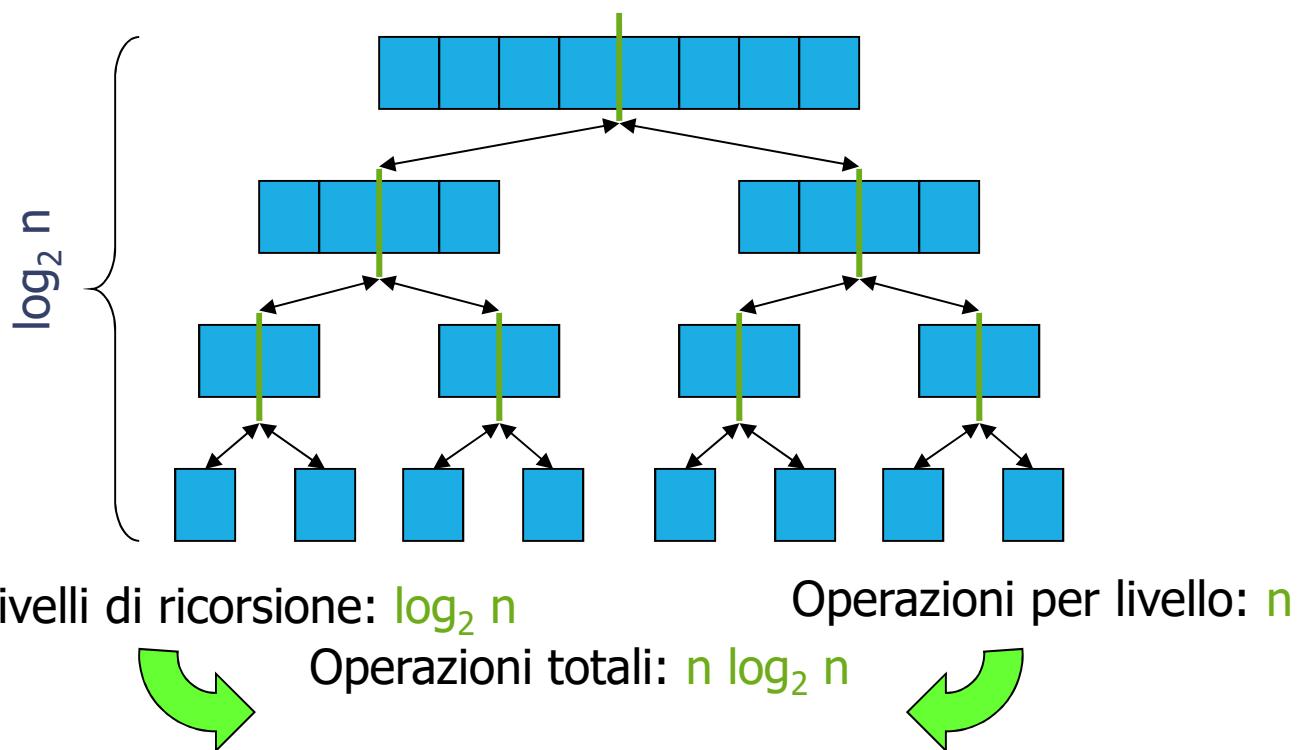
etc.

Terminazione: a ogni passo i dati si dimezzano, dopo i passi sono $n/2^i$. Si termina per $n/2^i = 1$, $i = \log_2 n$

$$\begin{aligned} T(n) &= n + 2*(n/2) + 2^2 * (n/4) + 2^3 * T(n/8) \\ &= \sum_{0 \leq i \leq \log_2 n} 2^i / 2^i * n = n * \sum_{0 \leq i \leq \log_2 n} 1 \\ &= n * (1 + \log_2 n) \end{aligned}$$

$T(n) = O(n \log n)$. Altri metodi di risoluzione dell'equazione alle ricorrenze portano a $T(n) = \Theta(n \log n)$.

Intuitivamente:



Quicksort (Hoare, 1961)



Divisione:

- partiziona il vettore $A[l..r]$ in due sottovettori SX e DX:
 - dato un elemento pivot $x = A[q]$
 - SX $A[l..q-1]$ contiene tutti elementi $\leq x$
 - DX $A[q+1..r]$ contiene tutti elementi $\geq x$
 - $A[q]$ si trova al posto giusto

La divisione non è necessariamente a metà, a differenza del mergesort.

Ricorsione

- quicksort su sottovettore SX $A[l..q-1]$
- quicksort su sottovettore DX $A[q+1..r]$
- condizione di terminazione: se il vettore ha 1 elemento è ordinato

Ricombinazione:

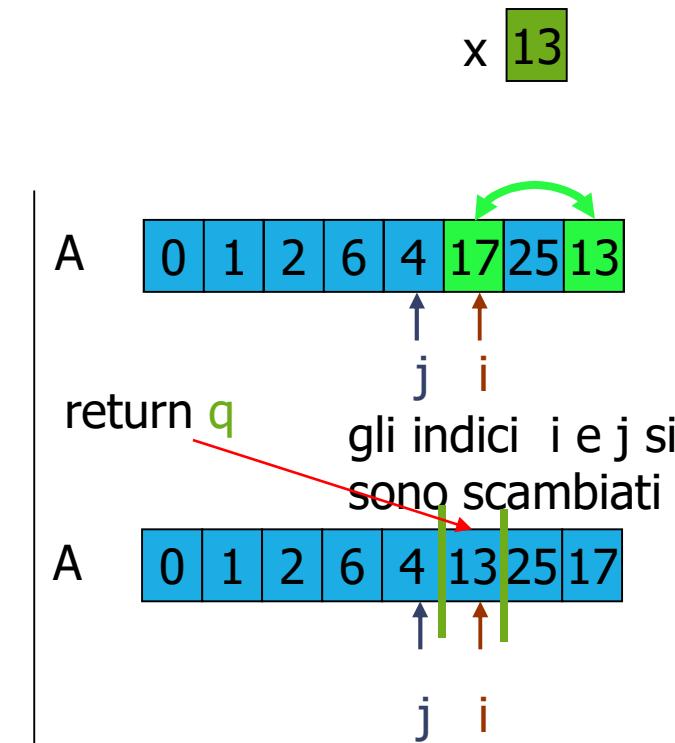
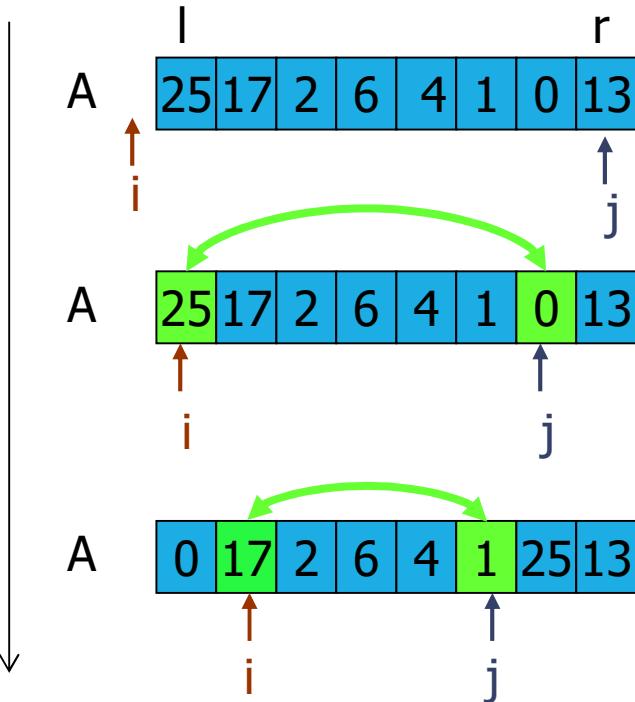
- nulla.

Partition (à la Hoare)

- Pivot $x = A[r]$
- i per scandire da SX a DX, j da DX a SX
- ripeti fintanto che $i < j$
 - individua $A[i]$ e $A[j]$ elementi “fuori posto”
 - ciclo ascendente su i fino a trovare un elemento maggiore del pivot x
 - ciclo discendente j fino a trovare un elemento minore del pivot x
 - scambia $A[i]$ e $A[j]$
- alla fine scambia $A[i]$ e il pivot x
- ritorna $q = i$
- $T(n) = \Theta(n)$.

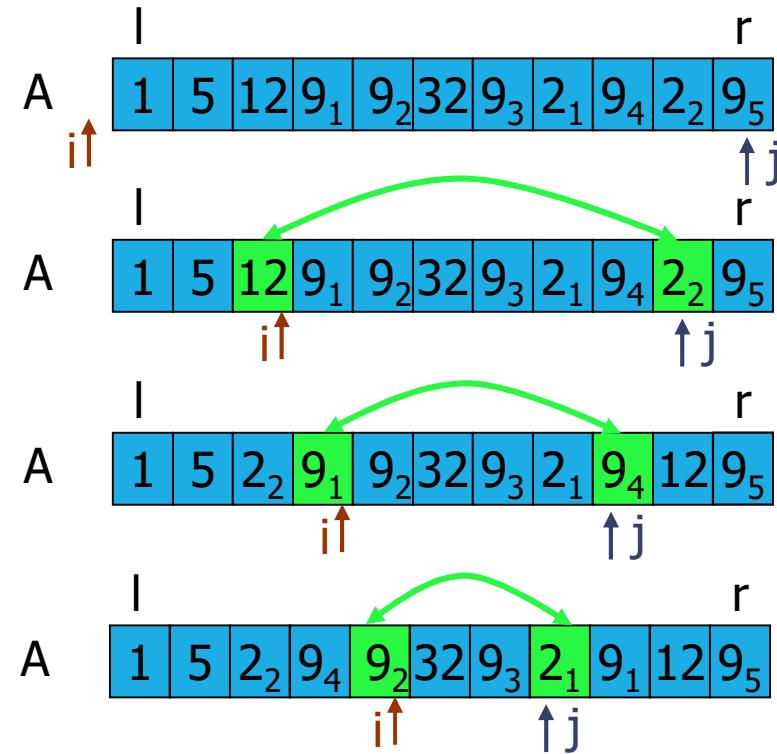
PS: esiste anche la partition à la Lomuto.

Esempio

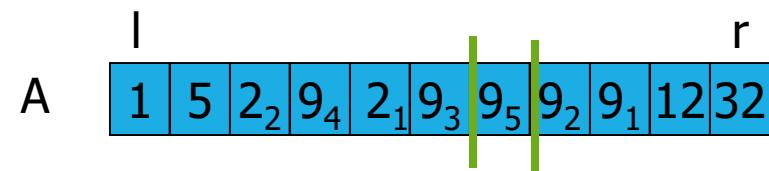
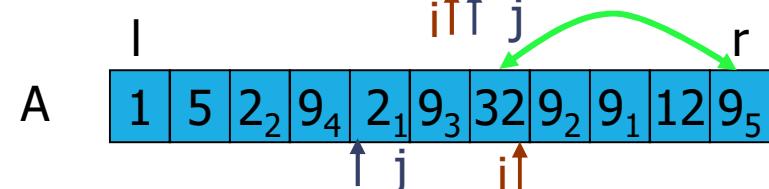
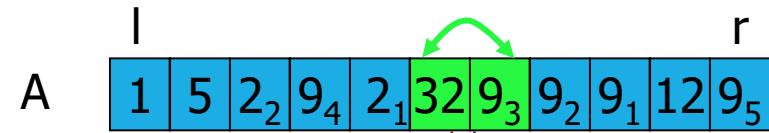


Si osservi che, se il pivot compare in più istanze, è irrilevante che esse si trovino alla fine della partition nel sottovettore sinistro o in quello destro, l'importante è che l'istanza scelta come pivot finisca nella posizione finale.

x **9₅**

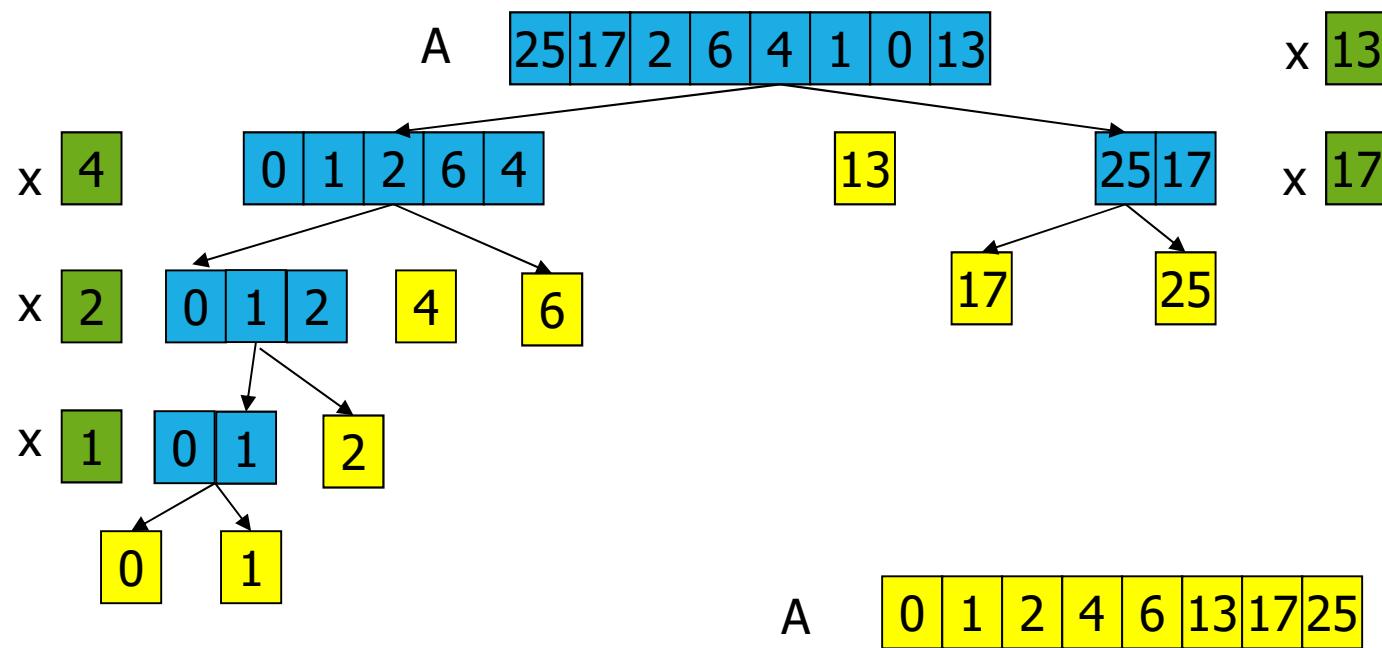


x **9₅**



gli indici i e j si
sono scambiati

Esempio



```

    wrapper
    estremi
    chiamata alla funzione ricorsiva
void Quicksort(Item A[], int N) {
    int l=0, r=N-1;
    quicksortR(A, l, r);
}
void quicksortR(Item A[], int l, int r ){
    int q;
    if (l >= r) terminazione
        return;
    q = partition(A, l, r);
    quicksortR(A, l, q-1);
    quicksortR(A, q+1, r);
    return;
}

```

divisione

chiamata ricorsiva

chiamata ricorsiva



RecursiveSort.c

```
int partition (Item A[], int l, int r ) {
    int i = l-1, j = r;
    Item x = A[r];
    for ( ; ; ) {
        while(ITEMlt(A[++i], x));
        while(ITEMgt(A[--j], x));
        if (i >= j)
            break;
        Swap(A, i, j);
    }
    Swap(A, i, r);
    return i;
}
```

```
void Swap(Item *v, int n1,
          int n2) {
    Item temp;
    temp = v[n1];
    v[n1] = v[n2];
    v[n2] = temp;
    return;
}
```

Caratteristiche

- In loco
- Non stabile: la funzione partition può provocare uno scambio tra elementi «lontani», facendo sì che un'occorrenza di una chiave duplicata si sposti a SX di un'occorrenza precedente della stessa chiave «scavalcandola».

Analisi di complessità

Efficienza legata al bilanciamento delle partizioni

A ogni passo partition ritorna:

- caso peggiore: un vettore da $n-1$ elementi e l'altro da 1 elemento
- caso migliore: due vettori da $n/2$ elementi
- caso medio: due vettori di dimensioni diverse.

Bilanciamento legato alla scelta del pivot.

Caso peggiore

Caso peggiore: pivot = minimo o massimo in vettore già ordinato in ordine opposto a quello desiderato

Equazione alle ricorrenze:

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

Risoluzione per sviluppo:

$$T(n) = n + (n-1) + (n-2) + \dots + 3 + 2 + 1 = \sum_{1 \leq i \leq n} i = n * (n+1)/2$$

Quindi $T(n) = O(n^2)$. Altri metodi di risoluzione dell'equazione alle ricorrenze portano a $T(n) = \Theta(n^2)$.

Caso migliore

Equazione alle ricorrenze:

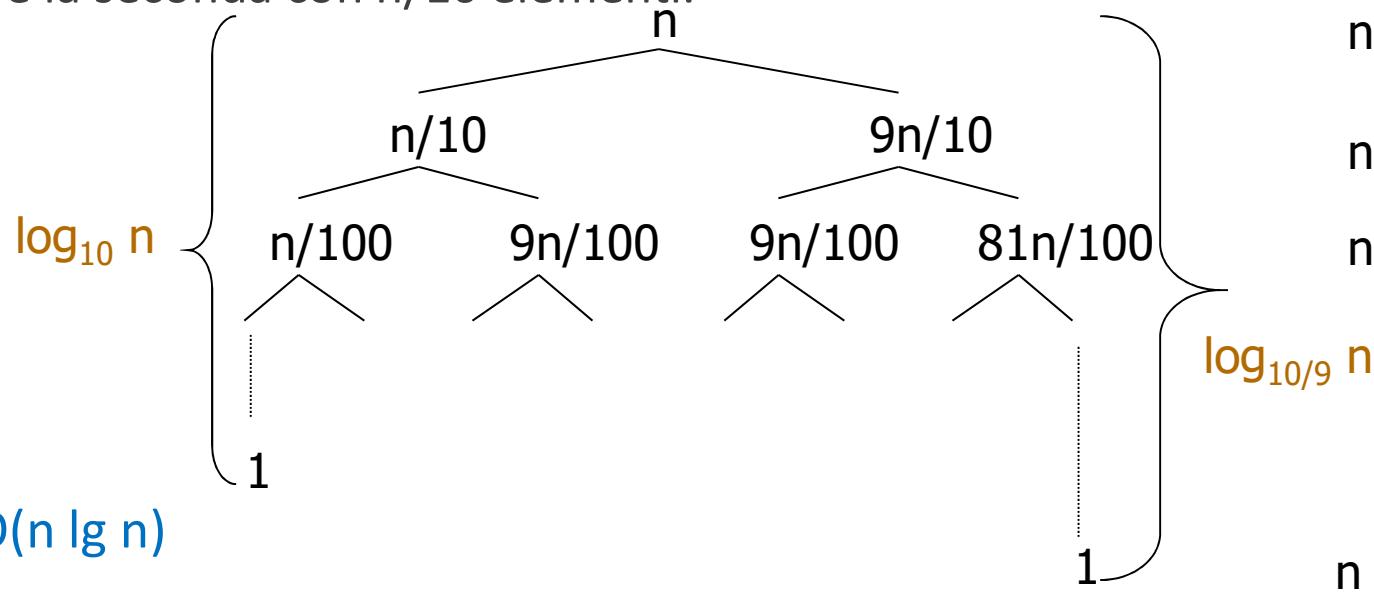
$$T(n) = 2T(n/2) + n \quad n \geq 2$$

$$T(1) = 1$$

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

Caso medio

- Purché non si ricada nel caso peggiore, anche se il partizionamento è molto sbilanciato, caso medio = caso migliore
- Esempio: ad ogni passo si generano 2 partizioni, la prima con $9/10 n$ e la seconda con $n/10$ elementi.



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

Scelta del pivot

- Elemento di mezzo: $x \leftarrow A[(l+r)/2]$
- Scegliere il valore medio tra min e max
- Scegliere la mediana tra 3 elementi presi a caso nel vettore
- ...

Se lo scopo è semplicemente di rendere molto improbabile il caso peggiore, basta generare un numero casuale i con $l \leq i \leq r$, poi scambiare $A[r]$ e $A[i]$ e infine usare come pivot $A[r]$.

Quadro riassuntivo

ALGORITMO	IN LOCO	STABILE	COMPLESSITÀ
Bubble Sort	Sì	Sì	$O(n^2)$
Selection Sort	Sì	No	$O(n^2)$
Insertion Sort	Sì	Sì	$O(n^2)$
Shell Sort	Sì	No	dipende
Merge Sort	No	Sì	$O(n \log n)$
Quick Sort	Sì	No	$O(n^2)$
Counting Sort	No	Sì	$O(n)$
Radix Sort	No	Sì	$O(n)$

Riferimenti

- Mergesort
 - Sedgewick 8.3 e 8.5
 - Cormen 2.3
- Quicksort
 - Sedgewick 7.1 e 7.2
 - Cormen 7.1, 7.2

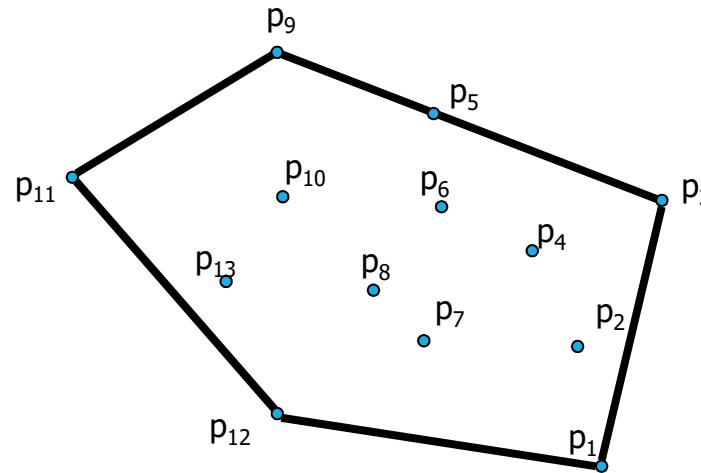
Esercizi di teoria

- 2. Ordinamenti ricorsivi
 - 2.1 Merge Sort
 - 2.2 Quick Sort



Un'applicazione: l'inviluppo convesso

Dato un insieme di punti Q , l'**inviluppo convesso** è il poligono convesso P di area minima per cui ogni punto di Q è interno a P o al più sul perimetro di P :



Soluzione brute-force:

- dato l'insieme dei punti, costruire l'insieme dei suoi sottoinsiemi (insieme delle parti)
- per ciascun sottoinsieme verificare che si tratti di un poligono convesso e se sì calcolarne l'area
- tenere traccia dell'area minima
- complessità esponenziale

Soluzione efficiente: Graham Scan (1972)

- ordinare i punti con Mergesort
- scansione lineare dei punti
- complessità $T(N) = O(N \lg N)$

Il Graham Scan è trattato nei lucidi di approfondimento disponibili sul Portale della Didattica.

Applicazioni degli Ordinamenti: l'Inviluppo Convesso



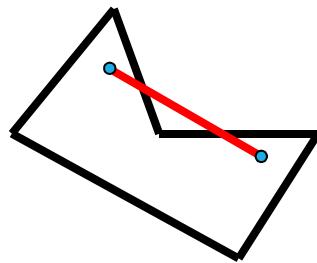
Geometria computazionale

Branca dell'Informatica che studia gli algoritmi atti a risolvere problemi geometrici:

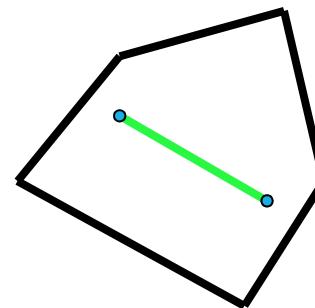
- intersezione di una coppia qualsiasi di segmenti
- inviluppo convesso
- ricerca della coppia di punti più vicini.

Poligono convesso

Un poligono si dice convesso se ogni segmento che congiunge due punti del poligono è interno al poligono stesso, incluso il bordo.



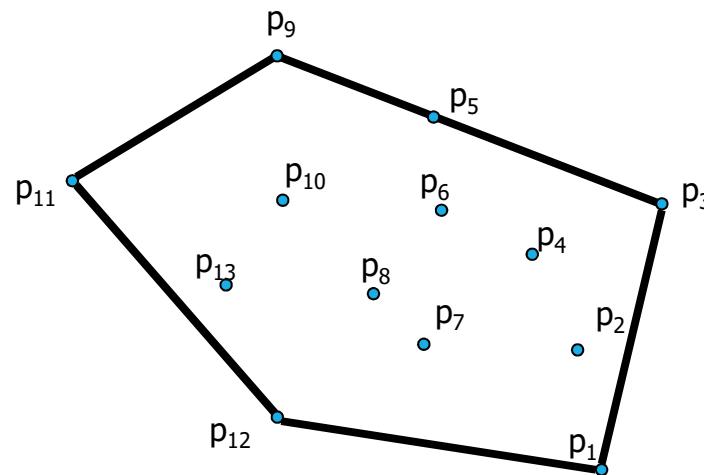
poligono concavo



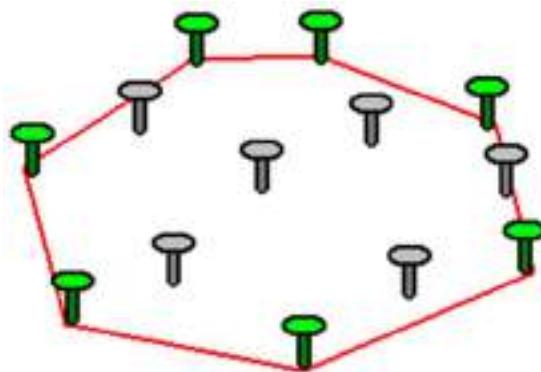
poligono convesso

Inviluppo (involucro) convesso – Convex hull

Dato un insieme di punti Q , l'inviluppo convesso è il poligono convesso P di area minima per cui ogni punto di Q è interno a P o al più sul perimetro di P :



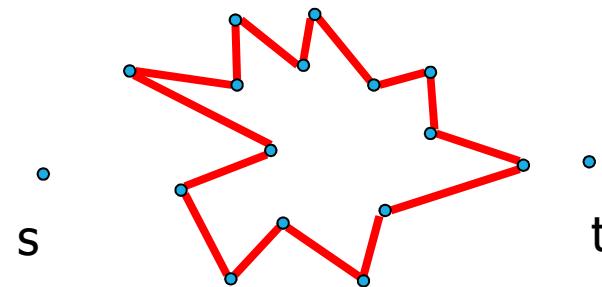
Soluzione «meccanica»: piantare chiodi perpendicolari al piano in ogni punto, racchiuderli con un elastico:



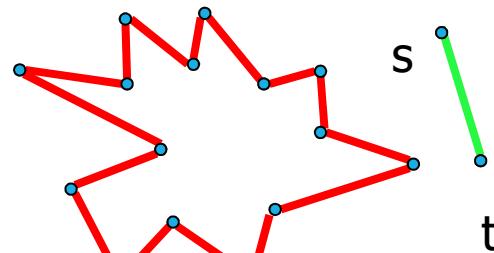
http://www.idlcoyote.com/math_tips/convexhull.html

Applicazione

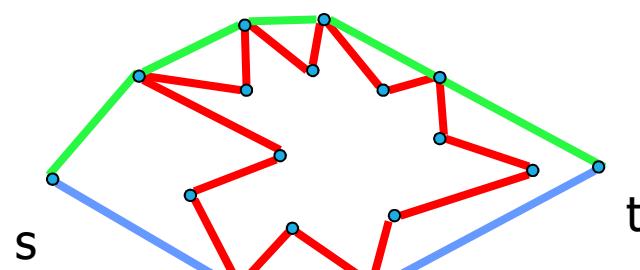
Pianificazione del moto di un robot con un cammino a lunghezza minima da un punto s a un punto t per evitare un ostacolo di forma poligonale:



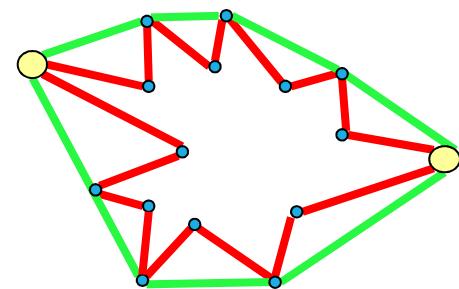
- o c'è un segmento s-t che evita l'ostacolo



- o è una delle 2 spezzate tra s e t lungo il contorno dell'inviluppo convesso



Dati N punti sul piano, trovare la coppia a distanza massima. Si dimostra che i 2 punti sono vertici dell'inviluppo convesso e che li si può trovare con complessità $T(N) = O(N)$.



Soluzione brute-force

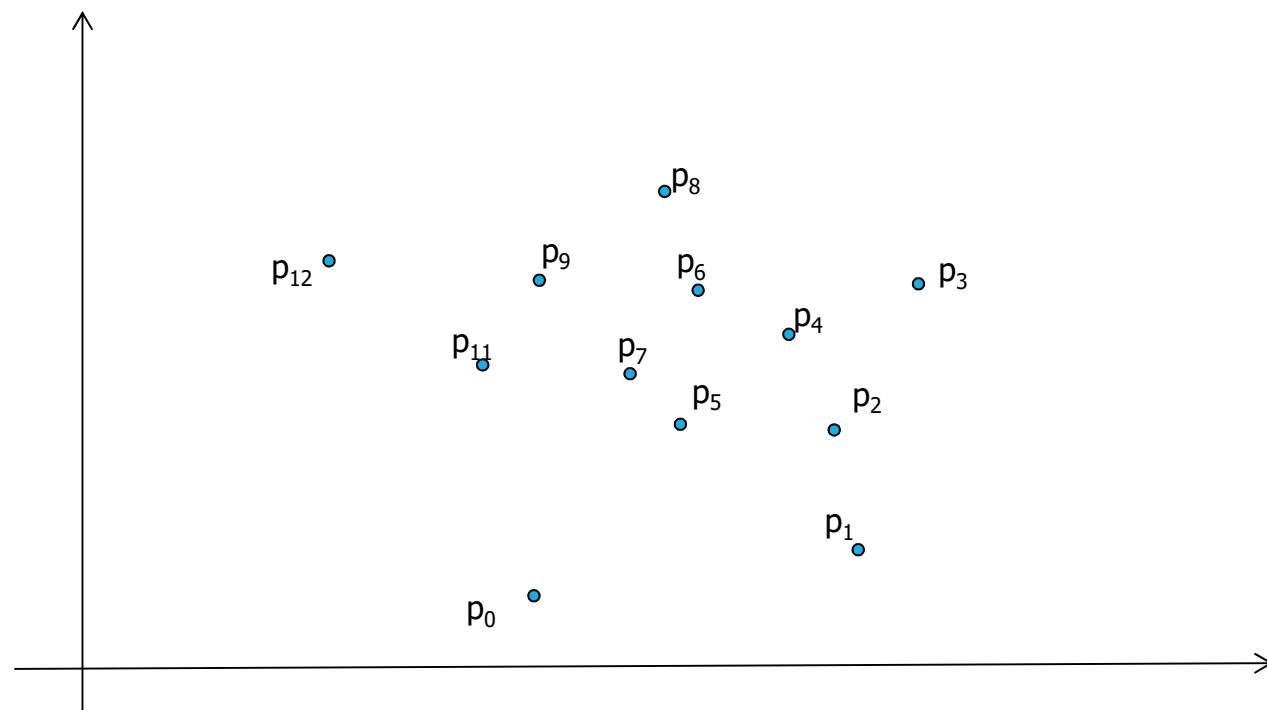
- dato l'insieme dei punti, costruire l'insieme dei suoi sottoinsiemi (insieme delle parti)
- per ciascun sottoinsieme verificare che si tratti di un poligono convesso e se sì calcolarne l'area
- tenere traccia dell'area minima
- complessità esponenziale.

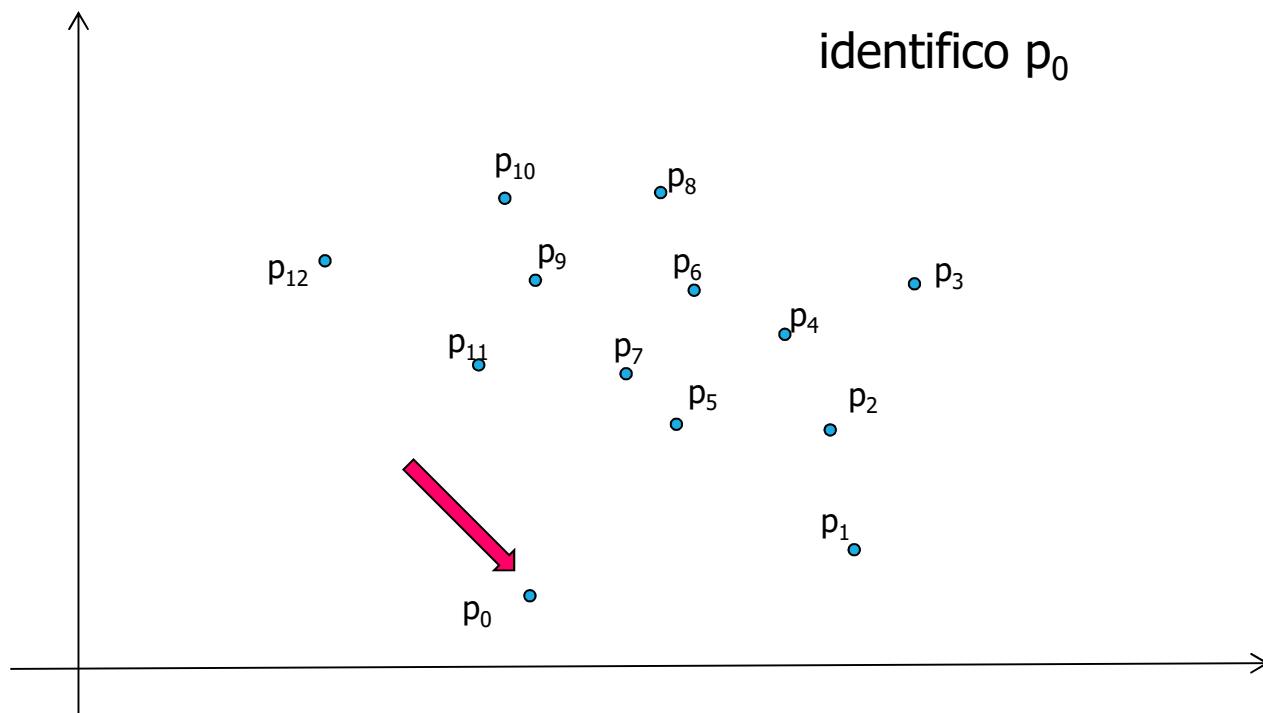
Graham Scan (1972)

Dato un insieme di punti sul piano $p_0 \dots p_N$

- identificare il punto a ordinata minima (il più a SX in caso di parità) e memorizzalo come p_0
- ordinare i punti rimanenti $p_1 \dots p_N$ per angolo polare crescente rispetto a p_0 . Per punti sulla stessa retta, tenere solo il più distante da p_0
- push sullo stack p_0, p_1, p_2
- per i rimanenti punti con $i = 3$ a N
 - fintanto che l'angolo formato dal punto sotto la cima dello stack, da quello in cima allo stack e da p_i non provoca una svolta antioraria (a sinistra), pop dallo stack
 - push sullo stack di p_i
- lo stack alla fine contiene i punti dell'inviluppo complesso in ordine antiorario

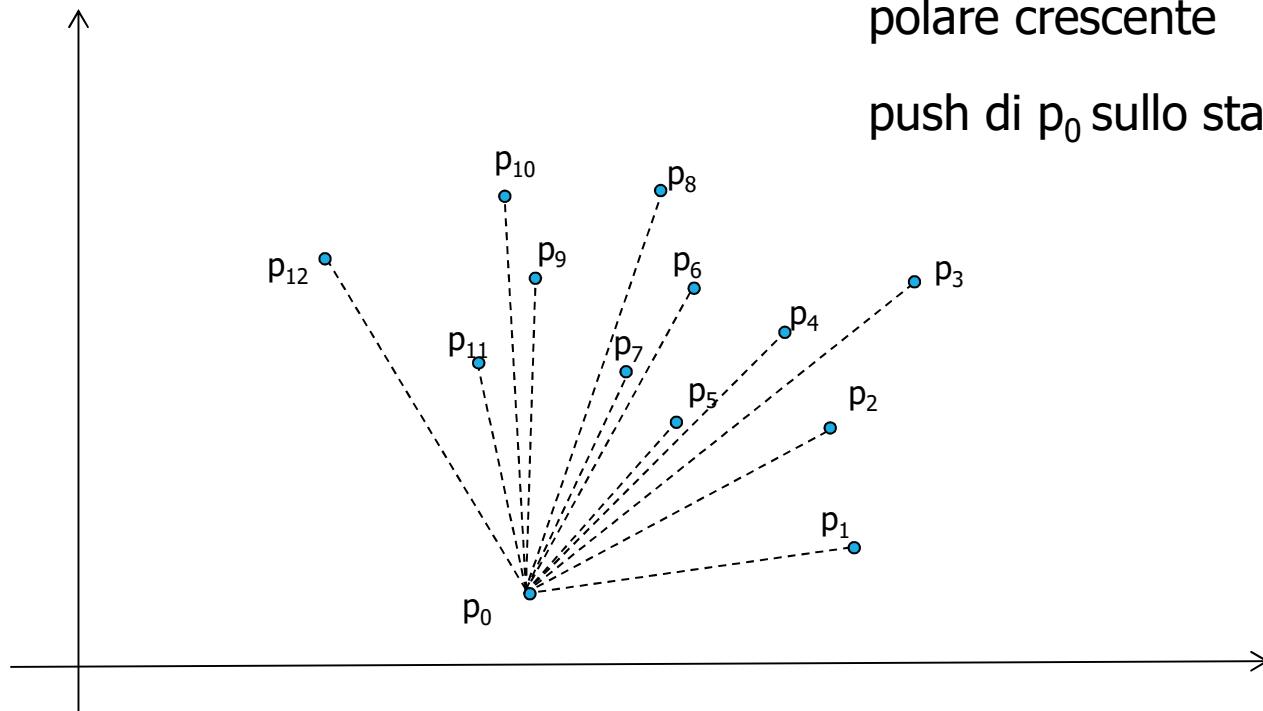
Esempio



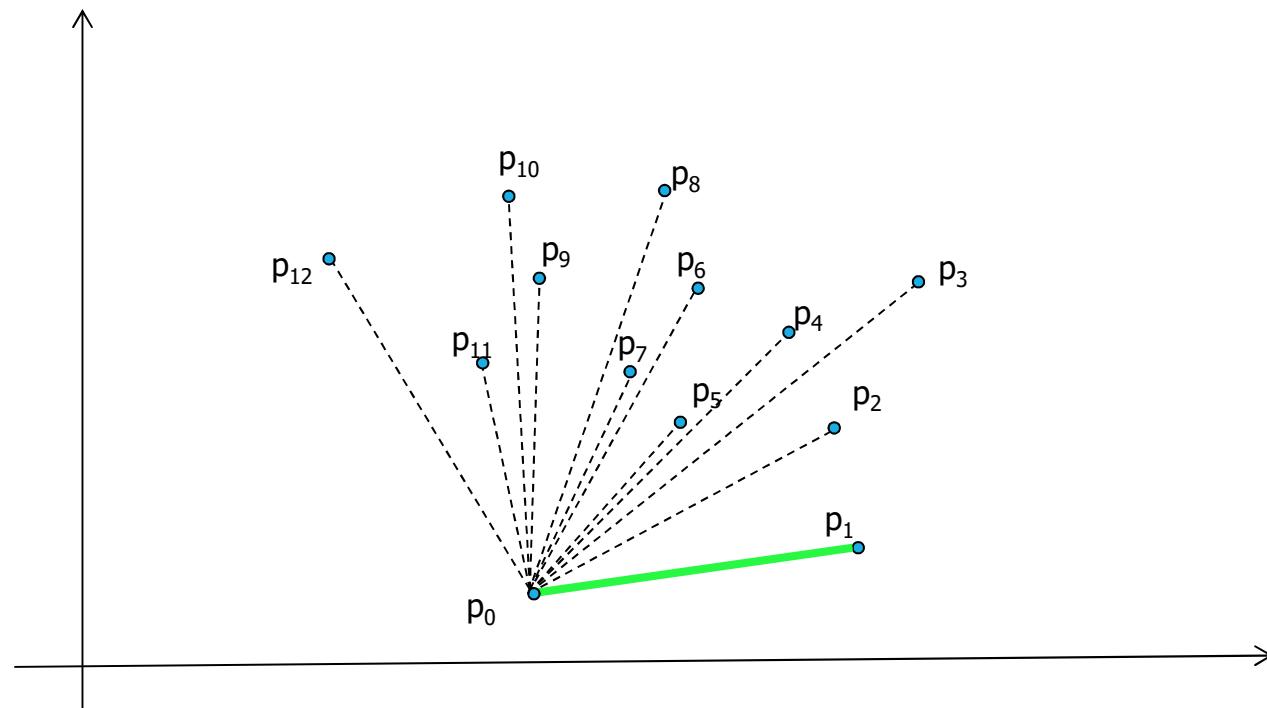


ordino i rimanenti
punti per angolo
polare crescente

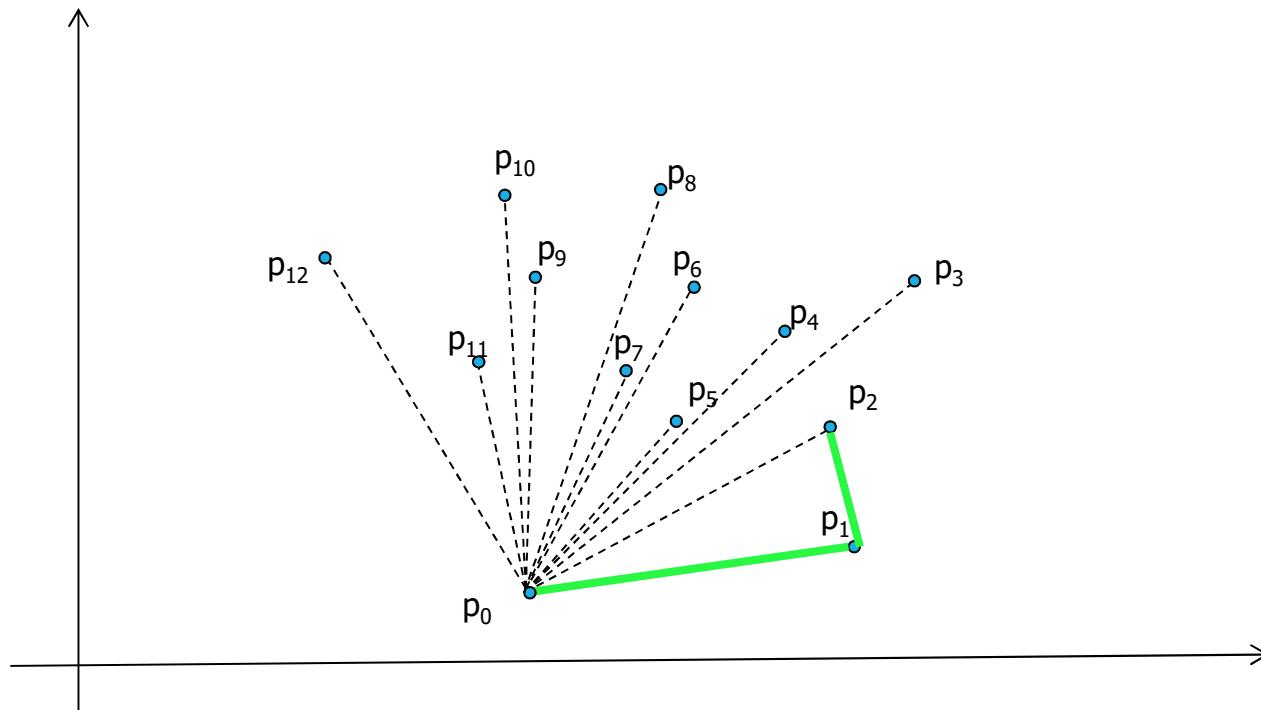
push di p_0 sullo stack



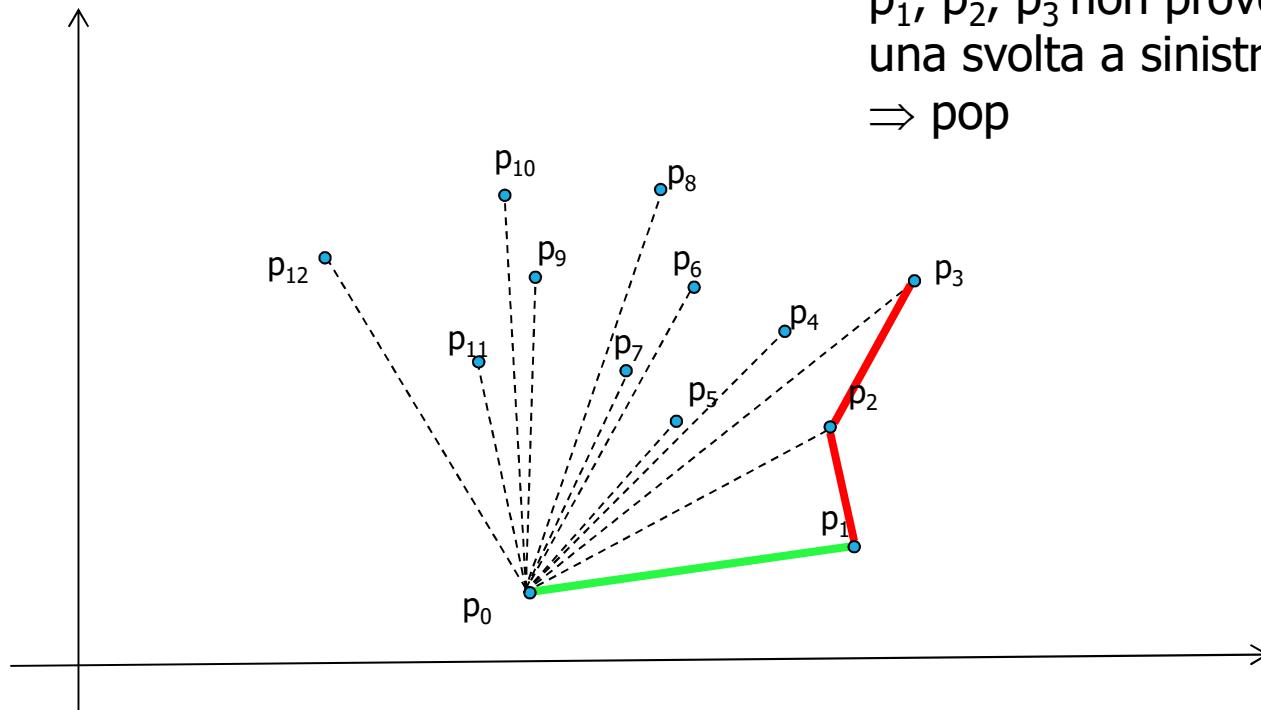
push di p_1 sullo stack



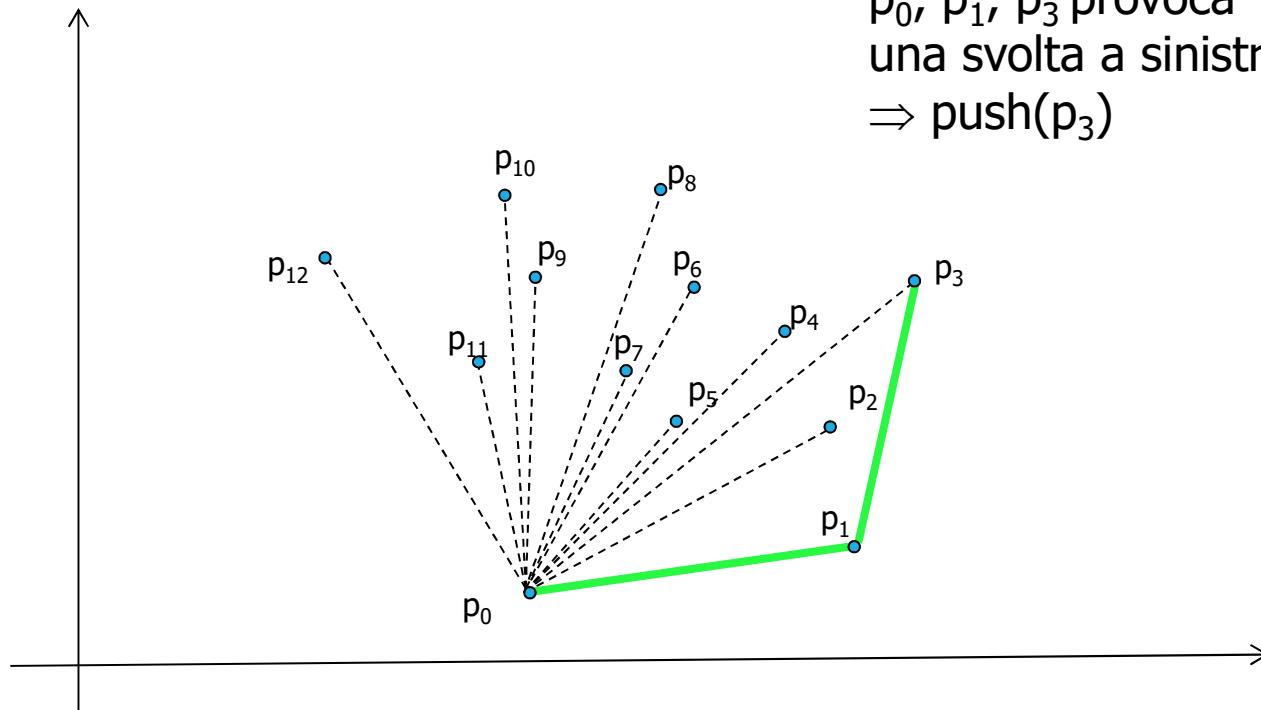
push di p_1 sullo stack



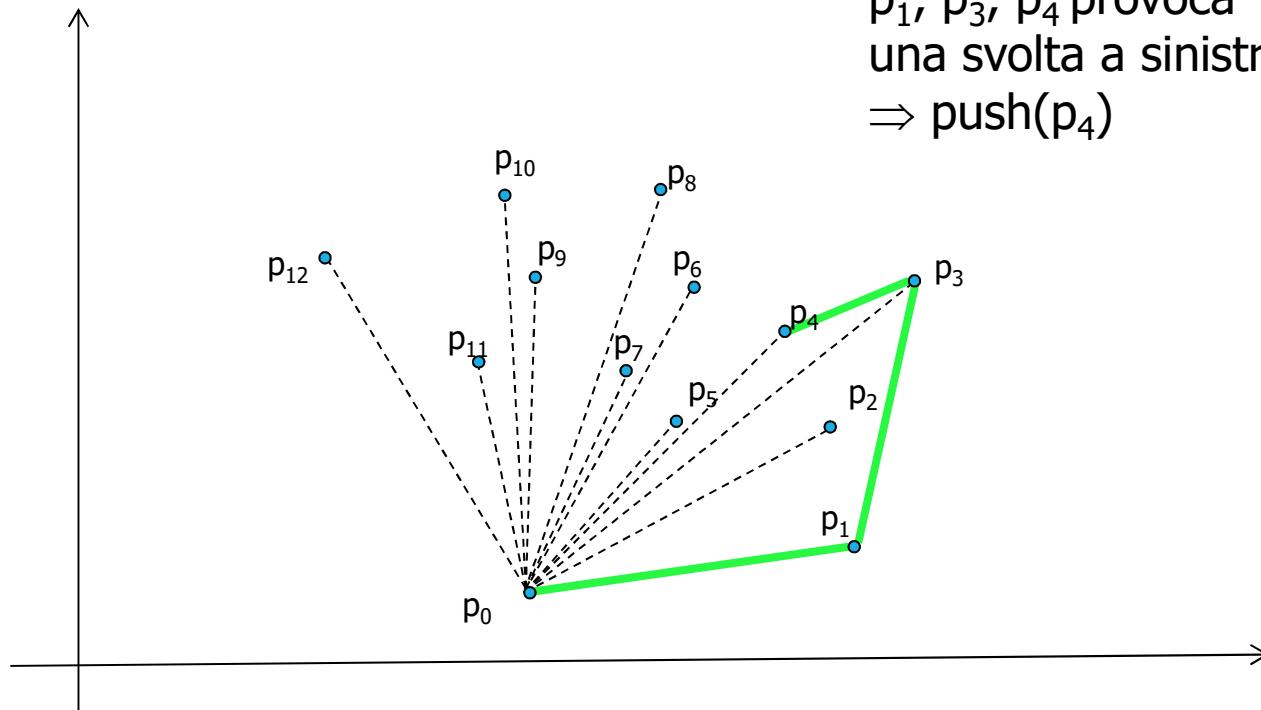
l'angolo formato da
 p_1, p_2, p_3 non provoca
una svolta a sinistra
 \Rightarrow pop



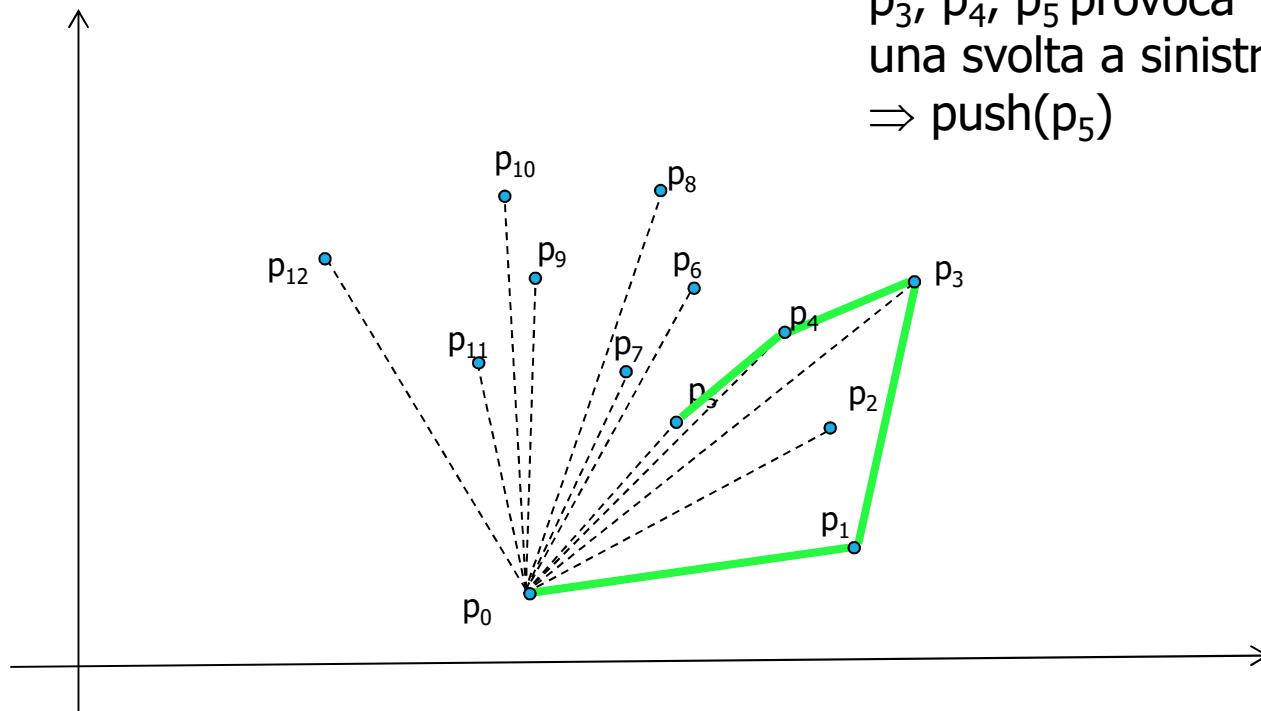
l'angolo formato da
 p_0, p_1, p_3 provoca
una svolta a sinistra
 $\Rightarrow \text{push}(p_3)$



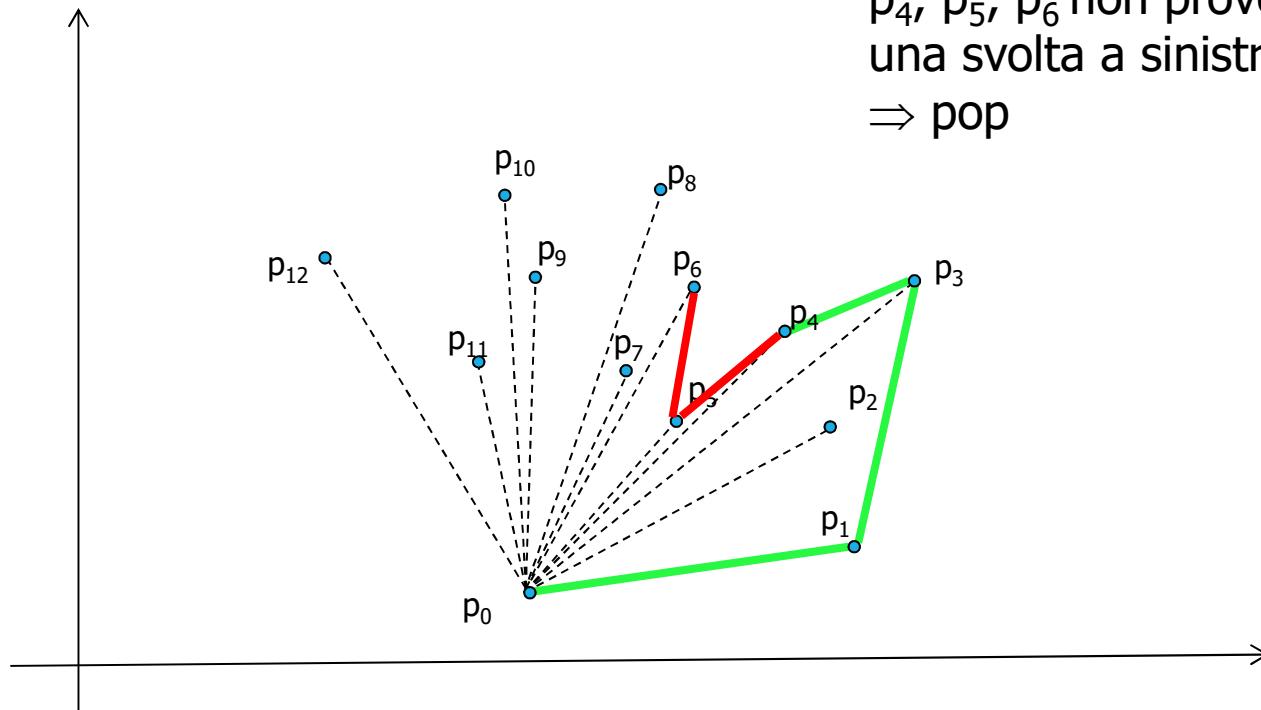
l'angolo formato da
 p_1, p_3, p_4 provoca
una svolta a sinistra
 $\Rightarrow \text{push}(p_4)$



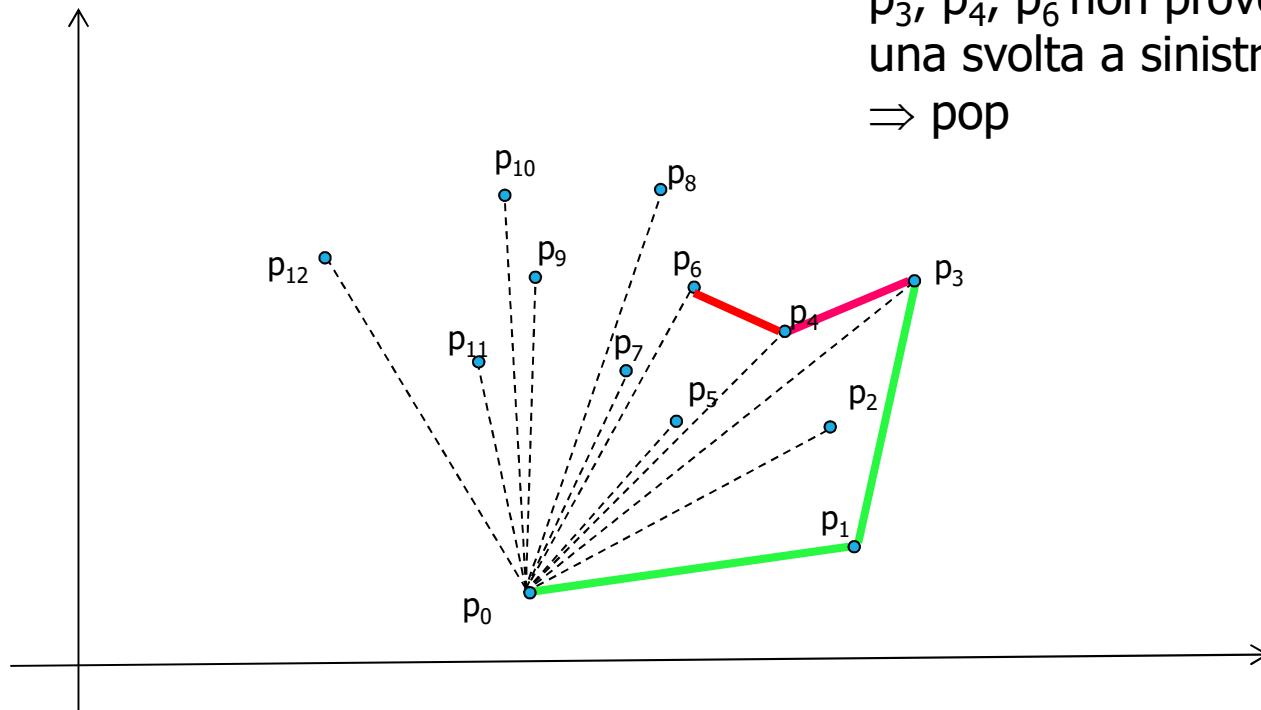
l'angolo formato da
 p_3, p_4, p_5 provoca
una svolta a sinistra
 $\Rightarrow \text{push}(p_5)$



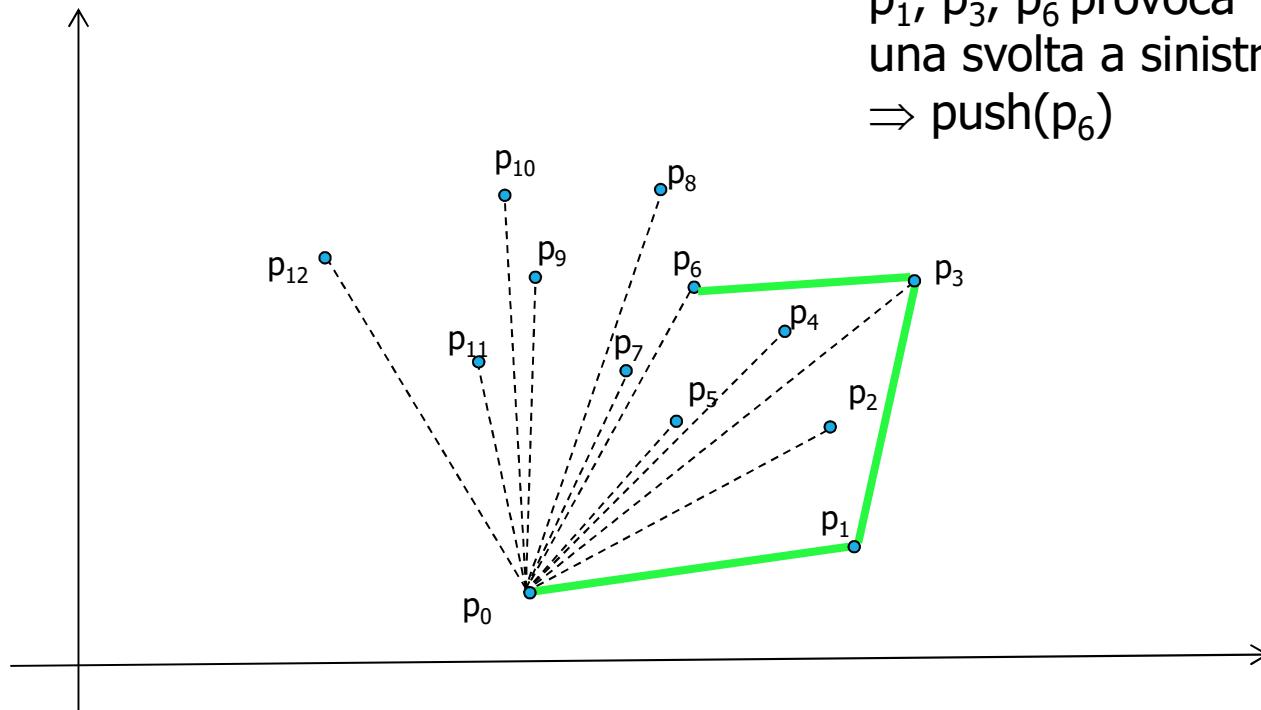
l'angolo formato da
 p_4, p_5, p_6 non provoca
una svolta a sinistra
 \Rightarrow pop



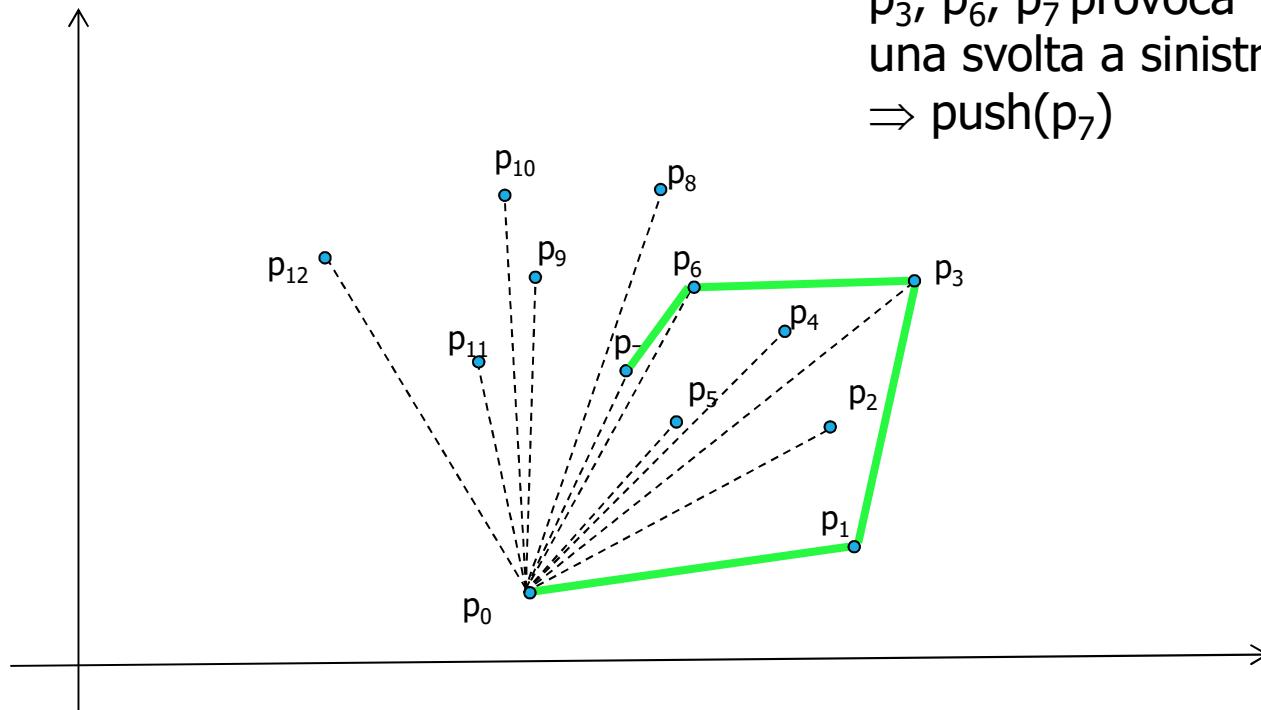
l'angolo formato da
 p_3, p_4, p_6 non provoca
una svolta a sinistra
 \Rightarrow pop



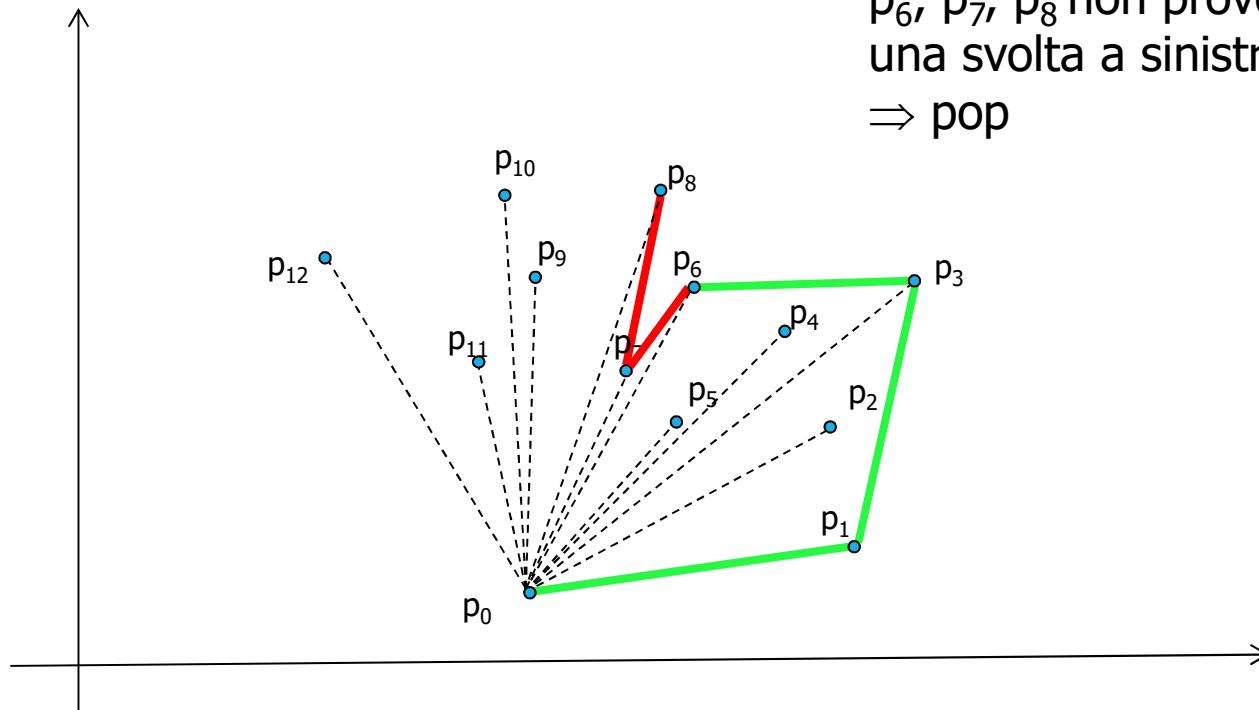
l'angolo formato da
 p_1, p_3, p_6 provoca
una svolta a sinistra
 $\Rightarrow \text{push}(p_6)$



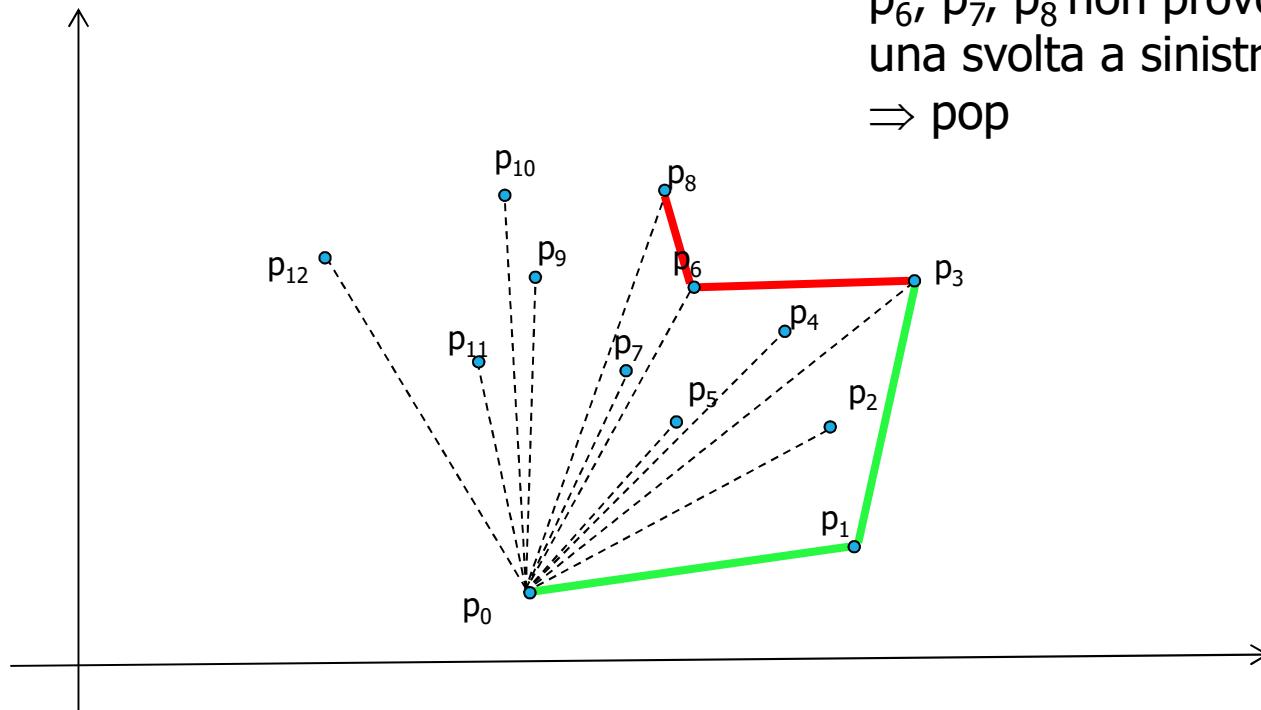
l'angolo formato da
 p_3, p_6, p_7 provoca
una svolta a sinistra
 $\Rightarrow \text{push}(p_7)$



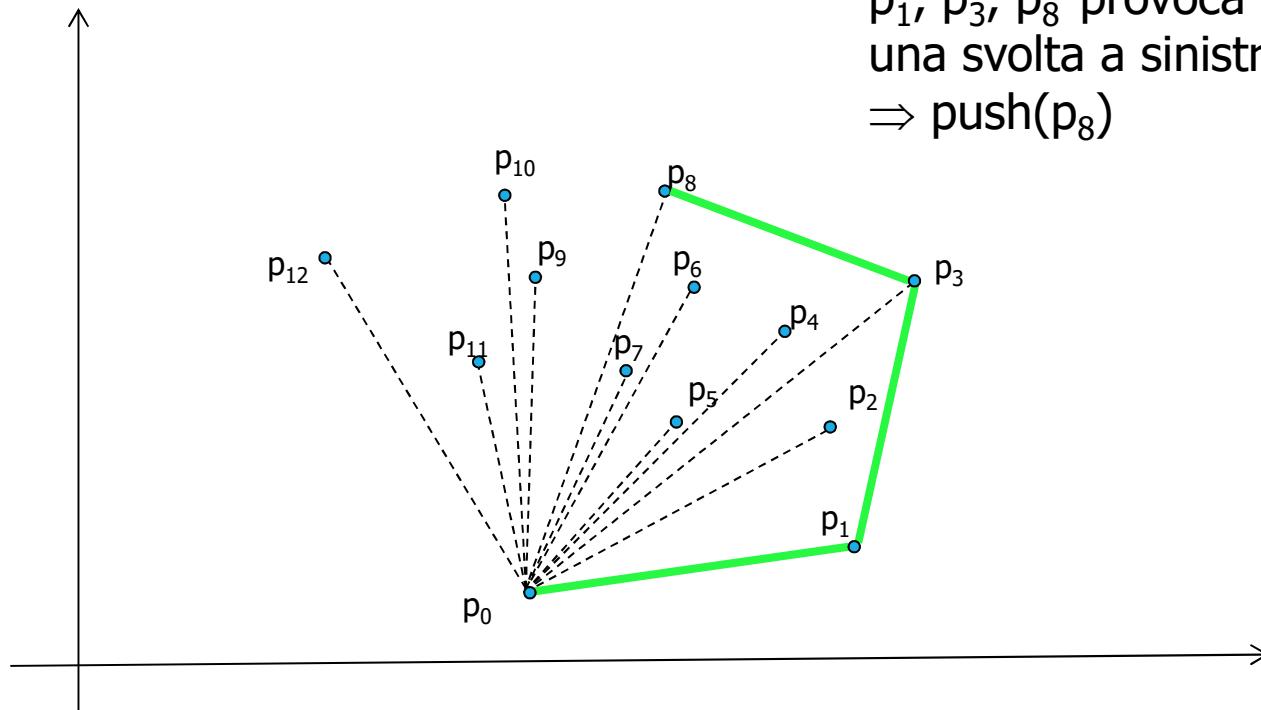
l'angolo formato da
 p_6, p_7, p_8 non provoca
una svolta a sinistra
 \Rightarrow pop



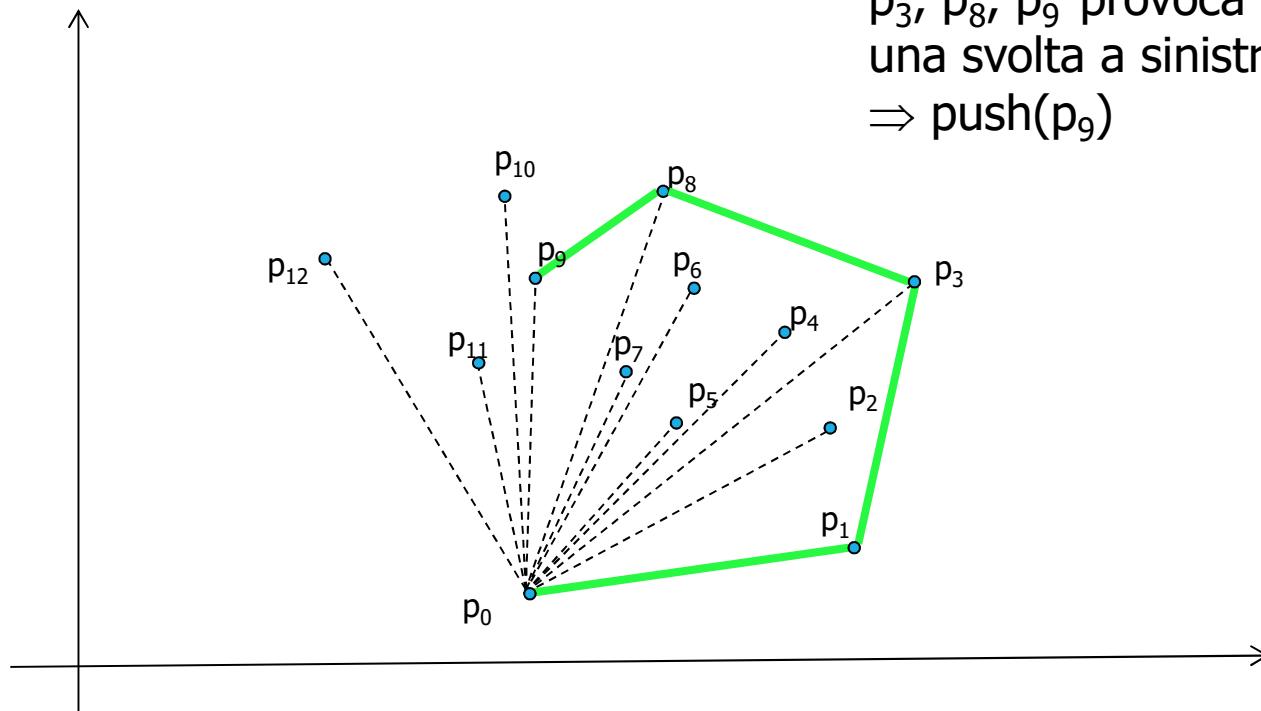
l'angolo formato da
 p_6, p_7, p_8 non provoca
una svolta a sinistra
 \Rightarrow pop



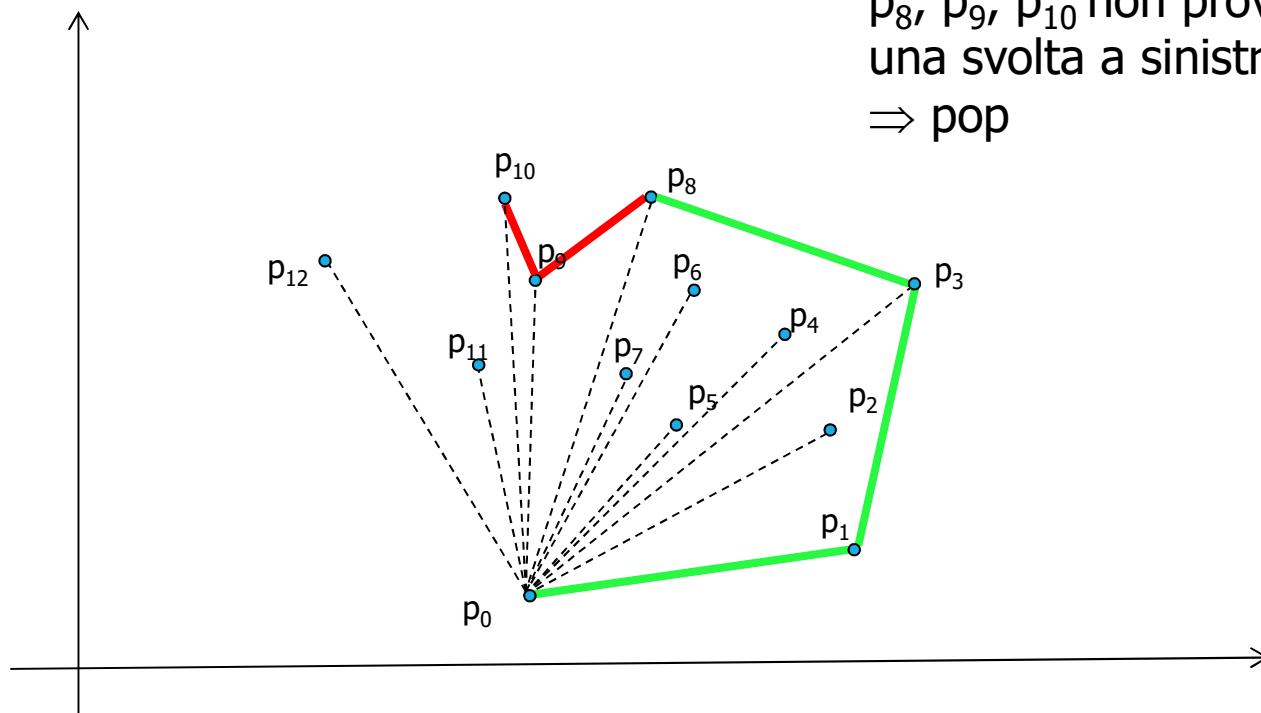
l'angolo formato da
 p_1, p_3, p_8 provoca
una svolta a sinistra
 $\Rightarrow \text{push}(p_8)$



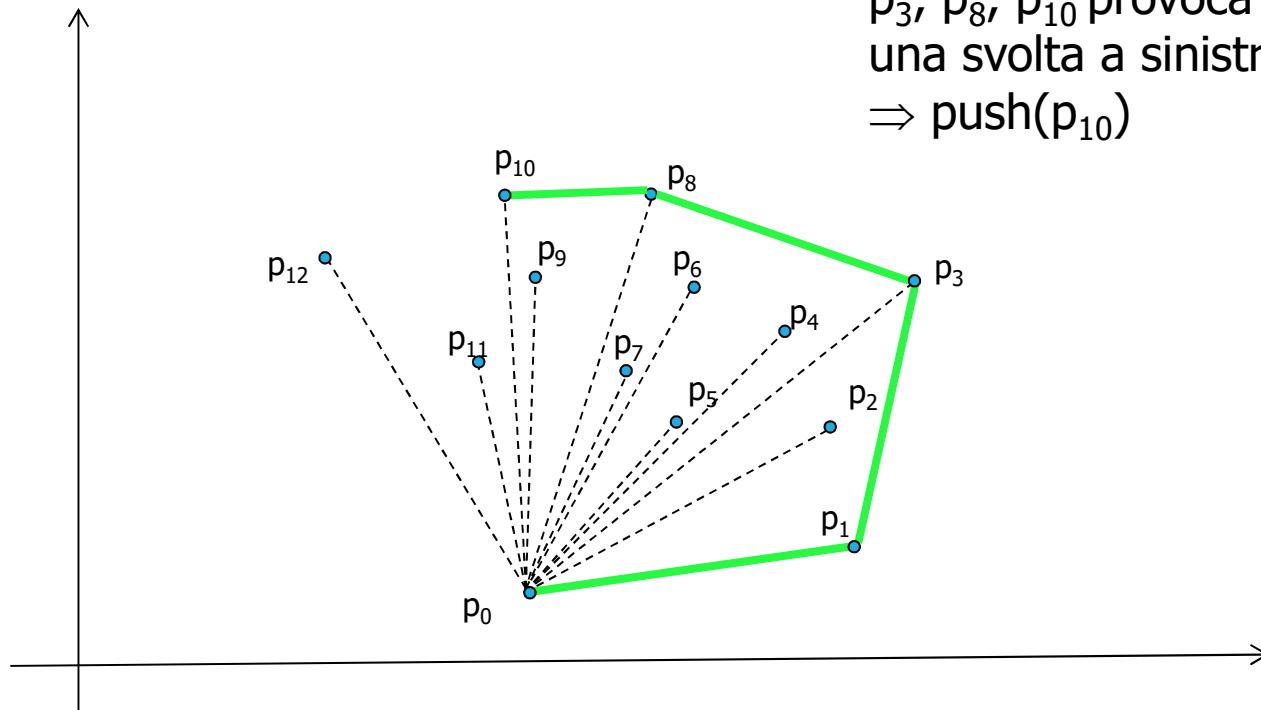
l'angolo formato da
 p_3, p_8, p_9 provoca
una svolta a sinistra
 $\Rightarrow \text{push}(p_9)$



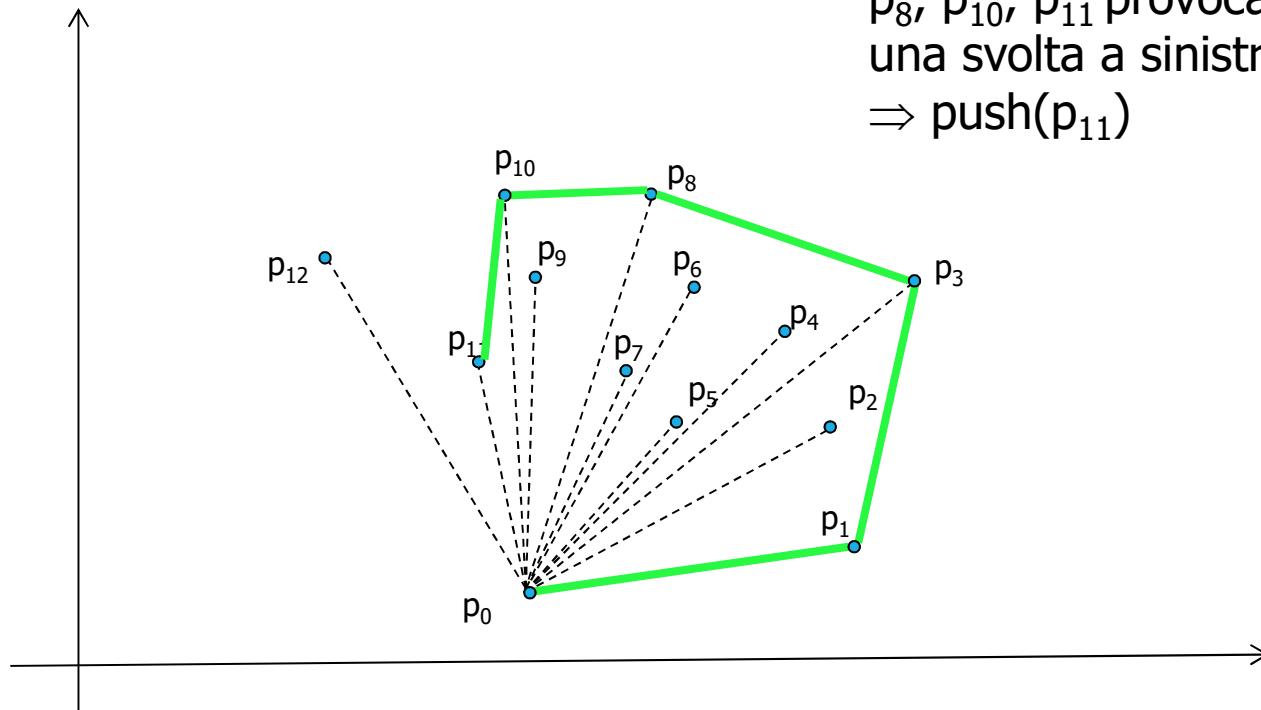
l'angolo formato da
 p_8, p_9, p_{10} non provoca
una svolta a sinistra
 \Rightarrow pop



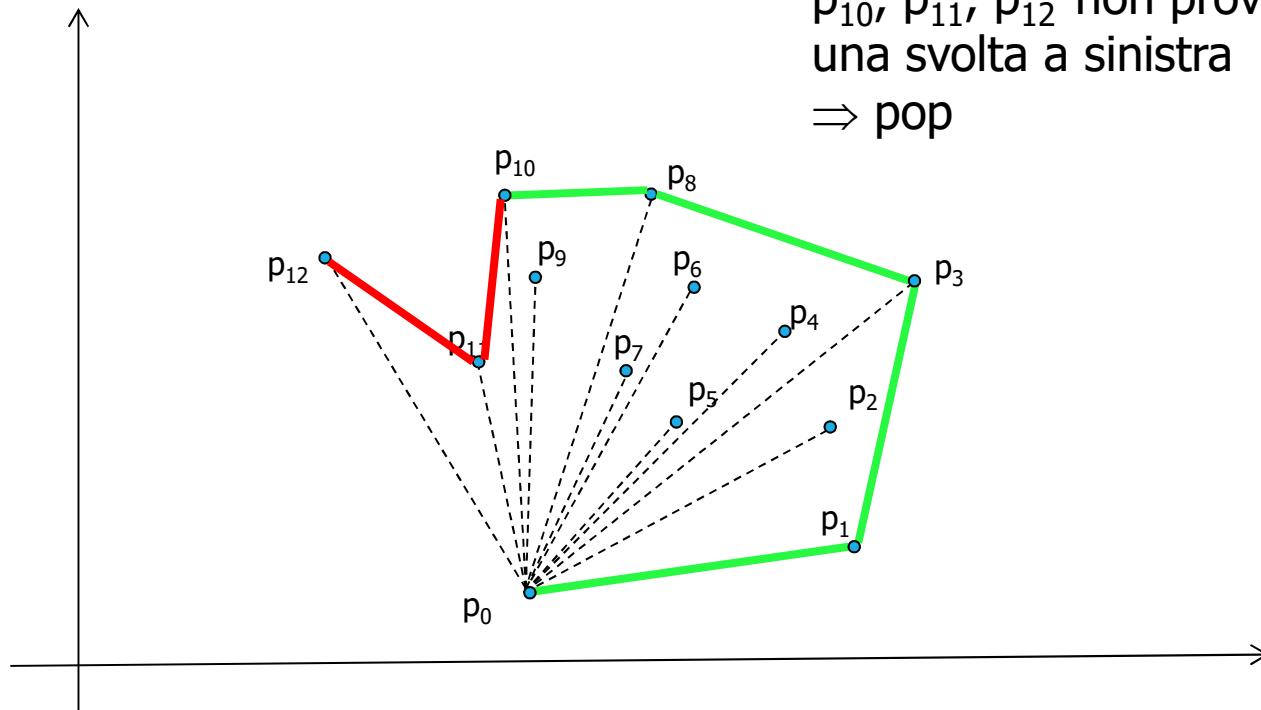
l'angolo formato da
 p_3, p_8, p_{10} provoca
una svolta a sinistra
 $\Rightarrow \text{push}(p_{10})$



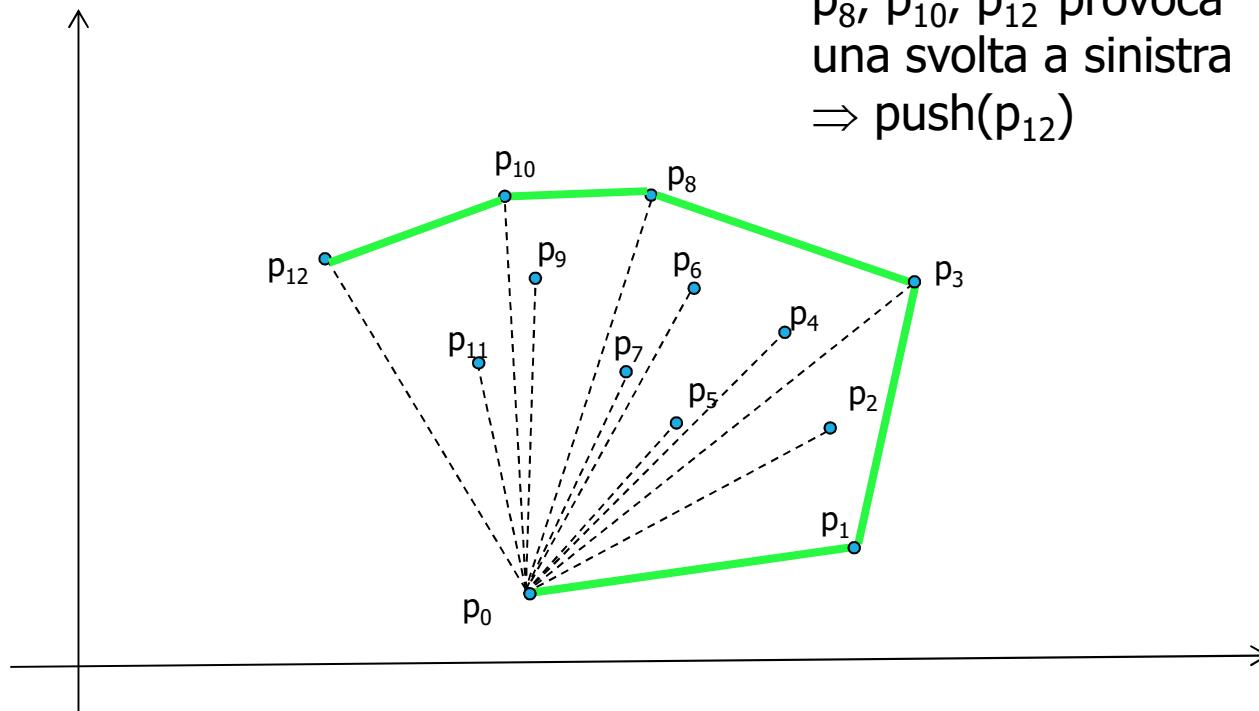
l'angolo formato da
 p_8, p_{10}, p_{11} provoca
una svolta a sinistra
 $\Rightarrow \text{push}(p_{11})$

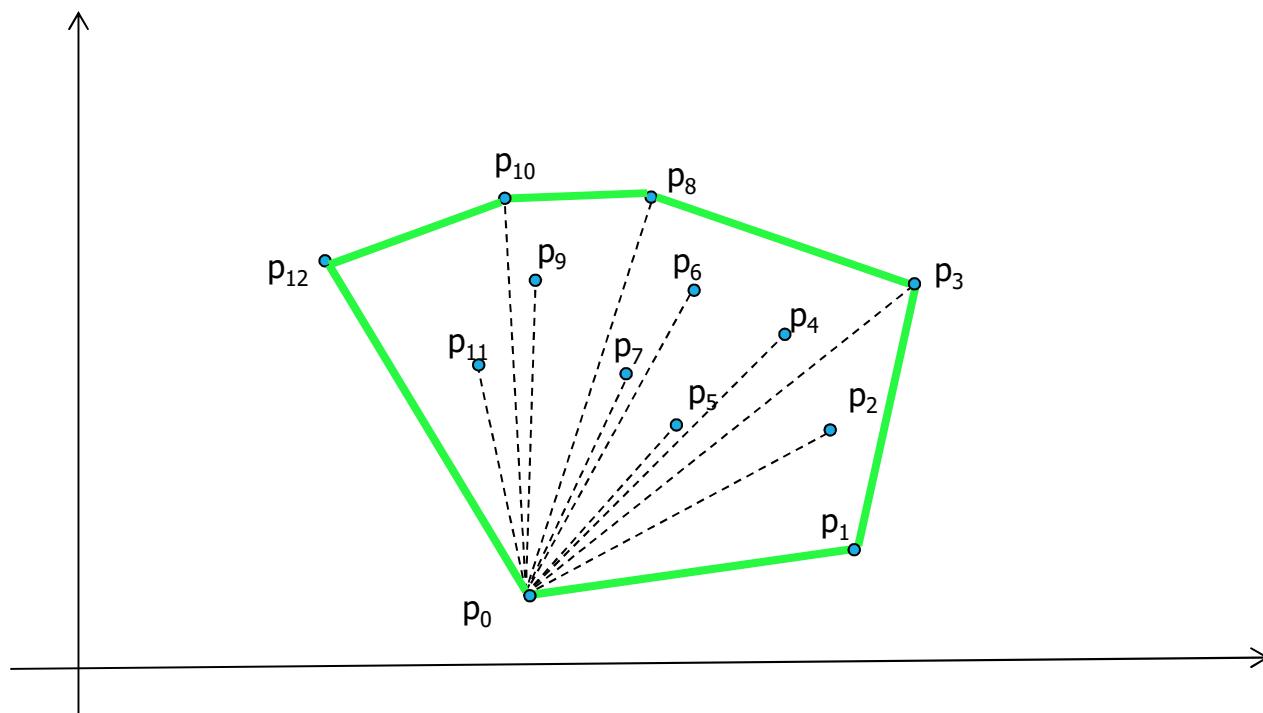


l'angolo formato da
 p_{10}, p_{11}, p_{12} non provoca
una svolta a sinistra
 \Rightarrow pop



l'angolo formato da
 p_8, p_{10}, p_{12} provoca
una svolta a sinistra
 $\Rightarrow \text{push}(p_{12})$





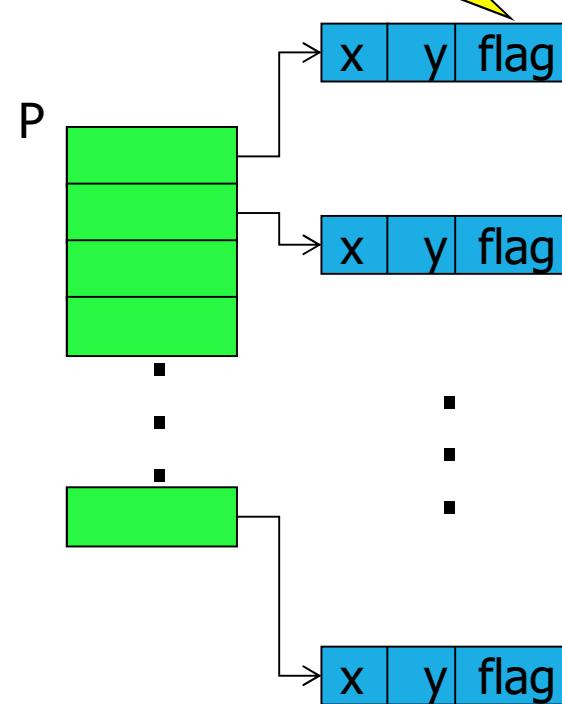
Strutture dati

Vettore di puntatori a punti:

```
typedef struct point *Point;
struct point {int x, y, flag;};
Point *P;

Point NEWpoint(int x, int y){
    Point p;
    p = malloc(sizeof(*p));
    p->x = x;
    p->y = y;
    p->flag = 1;
    return p;
}
```

indica se tenere il punto o no



Identificazione del punto a ordinata minima (il più a SX in caso di parità) e memorizzazione come p_0

```
void LookForP0(void) {
    int i, min = 0;
    Point tmp;
    for(i = 1; i < N; i++) {
        if(P[i]->y < P[min]->y)
            min = i;
        else if(P[i]->y == P[min]->y)
            if(P[i]->x < P[min]->x)
                min = i;
    }
    tmp = P[0]; P[0] = P[min]; P[min] = tmp;
    return;
}
```

Ordinamento dei punti rimanenti $p_1 \dots p_N$ per angolo polare crescente rispetto a p_0 . Per punti sulla stessa retta, tenere solo il più distante da p_0



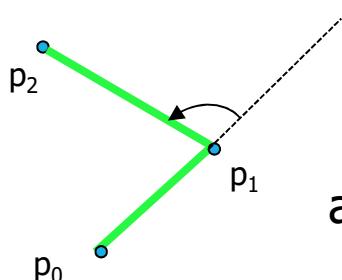
ordinamento con MergeSort, confronto in base all'angolo polare, senza funzioni trigonometriche costose (atan)

determinazione della svolta
a sinistra o a destra

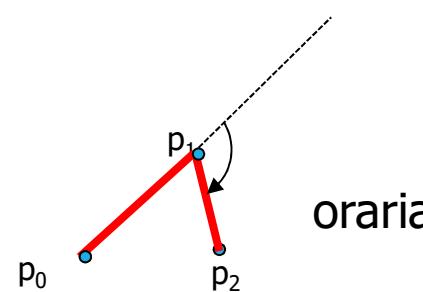
per i rimanenti punti con $i = 3$ a N

- fintanto che l'angolo formato dal punto sotto la cima dello stack, da quello in cima allo stack e da p_i non provoca una svolta antioraria (a sinistra), pop dallo stack
- push sullo stack di p_i

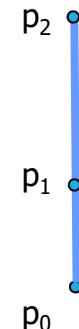
Determinazione della svolta



antioraria



oraria



allineata

$$p_0 = (x_0, y_0) \quad p_1 = (x_1, y_1) \quad p_2 = (x_2, y_2)$$

$$M = \begin{pmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{pmatrix}$$

Se $\det(M) = (x_1-x_0) \cdot (y_2-y_0) - (y_1-y_0) \cdot (x_2-x_0)$

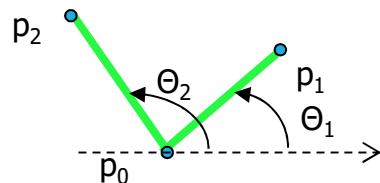
- >0 : svolta antioraria
- <0 : svolta oraria
- $=0$: allineamento

```
int xProd(Point p0, Point p1, Point p2){  
    int x1, x2, y1, y2, prod;  
    x1 = p1->x - p0->x;  
    x2 = p2->x - p0->x;  
    y1 = p1->y - p0->y;  
    y2 = p2->y - p0->y;  
    prod = (x1 * y2) - (x2 * y1);  
  
    if(prod > 0)  
        return 1;           ← svolta antioraria  
    else if(prod < 0)  
        return -1;          ← svolta oraria  
    return 0;  
}
```

allineamento

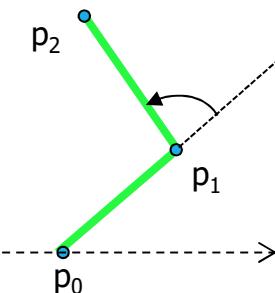
Confronto tra punti per angolo polare

$$p_0 = (x_0, y_0) \quad p_1 = (x_1, y_1) \quad p_2 = (x_2, y_2)$$



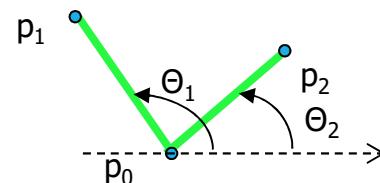
se $p_0-p_1-p_2$ è una svolta antioraria

`CrossProduct(P[0], p1, p2)` è 1



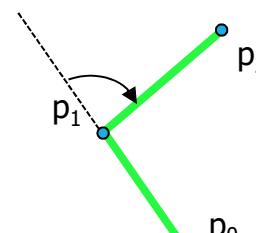
allora
 $\theta_2 > \theta_1$

`ComparePoints(p1, p2)` è -1



se $p_0-p_1-p_2$ è una svolta oraria

`CrossProduct(P[0], p1, p2)` è -1



allora
 $\theta_1 > \theta_2$

`ComparePoints(p1, p2)` è 1

```
int CmpPts(Point p1, Point p2){  
    int x = XProd(P[0], p1, p2);  
    if(x == 0) {  
        if(MoreDist(p1, p2)) {  
            p2->flag = 0;  
            return 1;  
        }  
        else {  
            p1->flag = 0;  
            return -1;  
        }  
    }  
    return -x;  
}
```

Punti allineati:
elimina quello più
vicino a P[0].

```

void GrahanScan(void) {
    int i, j;
    LookForP0(); MergeSort(1, N-1);
    STACKinit(N); STACKpush(P[0]);
    for(i = 1, j = 1; i < 3;)
        if(P[j]->flag) { i++; STACKpush(P[j++]); } else j++;
    for(i = j; i < N; i++){
        if(P[i]->flag) {
            while(xProd(STACKnext_to_top(),STACKtop(),STACKpop();P[i])!=1)
                STACKpush(P[i]);
        }
    }
    return;
}

```

complessità $T(N) = O(N \lg N) + O(N) = O(N \lg N)$



Problemi di ricerca e ottimizzazione

Paolo Camurati



Tipologie di problemi

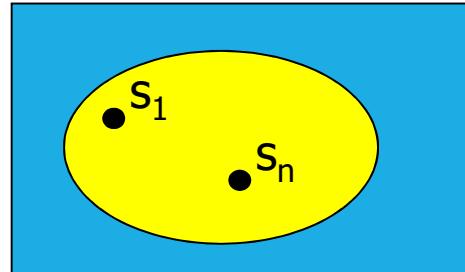
Problemi di calcolo:

- soluzione con procedimento matematico che porta, senza scelte e con un numero finito di passi, alla soluzione

- Esempi:
 - fattoriale
 - determinante
 - numeri di Fibonacci, di Catalan, di Bell etc.

Problemi di ricerca:

- dati:
 - S : spazio (insieme) delle soluzioni possibili
 - V : spazio delle soluzioni valide
 - in generale $V \subset S$
- appurare se $V = \emptyset$
- elencare gli elementi di V
 - almeno 1
 - tutti in caso di enumerazione.



S: spazio delle soluzioni

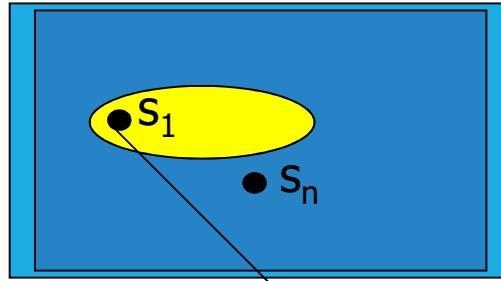
V: soluzioni valide

Esempi:

- insiemi delle parti che soddisfano condizione
- 8 regine
- Sudoku
- in grafo tutti i cammini semplici da un vertice

Problemi di ottimizzazione:

- $S \equiv V$: tutte le soluzioni sono valide
- data una funzione obiettivo f (costo o vantaggio), selezionare una o più soluzioni per cui f è minima o massima
- l'enumerazione è necessaria.



S: spazio delle soluzioni
≡
V: soluzioni valide

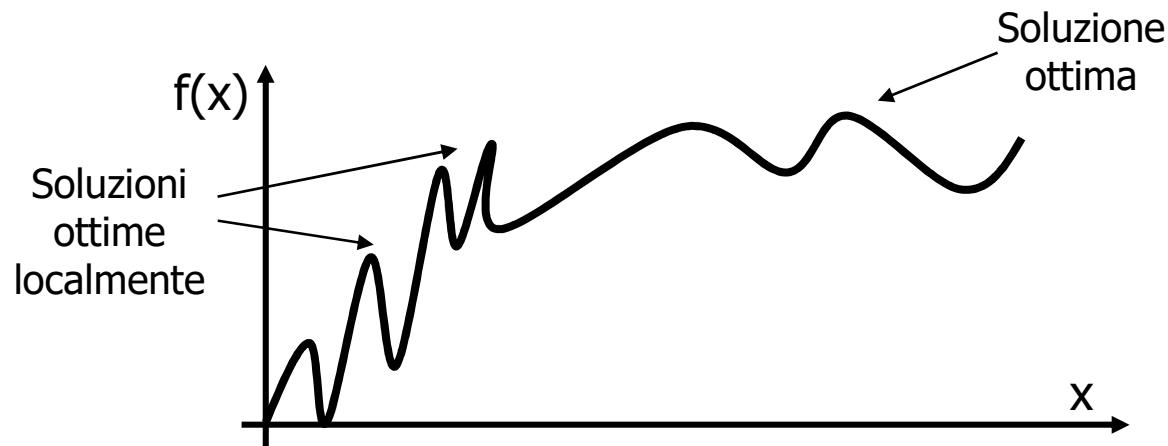
$f(s_1)$ minimo (massimo)

Esempi:

- massimizzare il valore di un insieme di oggetti compatibili con una capacità massima di un contenitore
- in grafo tutti i cammini semplici da un vertice a lunghezza massima

Minimo (massimo): assoluto o in un dominio contiguo:

- Soluzione *ottima*: min/max assoluto
- Soluzione *ottima localmente*: min/max locale



Risposte possibili:

- contare il numero di soluzioni valide
- trovare almeno una soluzione valida
- trovare tutte le soluzioni valide
- trovare tra le soluzioni valide quella (o quelle) ottima(e) secondo un criterio di ottimalità.

Esplorazione dello spazio delle soluzioni

Approccio incrementale:

- soluzione iniziale vuota
- estensione della soluzione mediante applicazione di scelte
- terminazione al raggiungimento di soluzione

Algoritmo generico che usa una struttura dati SD:

Ricerca():

- metti la soluzione iniziale in SD
- finché SD non diventa vuota:
 - estrai una soluzione parziale da SD;
 - se è una soluzione valida, Return Soluzione
 - applica le scelte lecite e metti le soluzioni parziali risultanti in SD
- Return fallimento.

Quando SD è:

- una **coda** (FIFO), la ricerca è in **ampiezza** (breadth-first)
- una **pila** (LIFO), la ricerca è in **profondità** (depth-first)
- una **coda a priorità**, la ricerca è **best-first**.

Se l'algoritmo:

- non conosce nulla del problema, si dice **non informato**
- ha conoscenza specifica (**euristica**), si dice **informato**

Se l'algoritmo è in grado di esplorare tutto lo spazio si dice **completo**.

Approccio seguito: algoritmo di ricerca

- in profondità
- non informato
- completo
- ricorsivo.

Rappresentazione

Spazio delle soluzioni rappresentato come **albero di ricerca**:

- di **altezza** n , dove n è la dimensione della soluzione
- di **grado** k , dove k è il massimo numero di scelte possibili
- la **radice** è la soluzione iniziale vuota
- i **nodi intermedi** sono etichettati con le soluzioni parziali
- le **foglie** sono le soluzioni. Una funzione determina se sono soluzioni valide

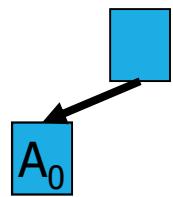
Esempio

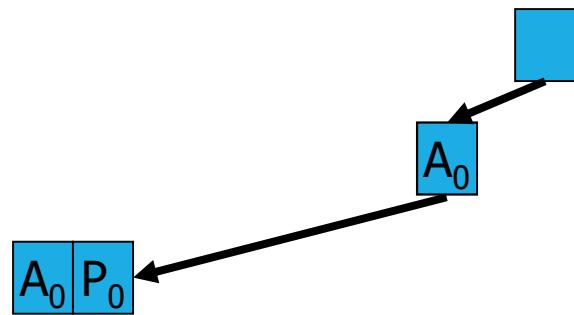
In un ristorante c'è un menu a prezzo fisso composto da antipasto, primo e secondo.

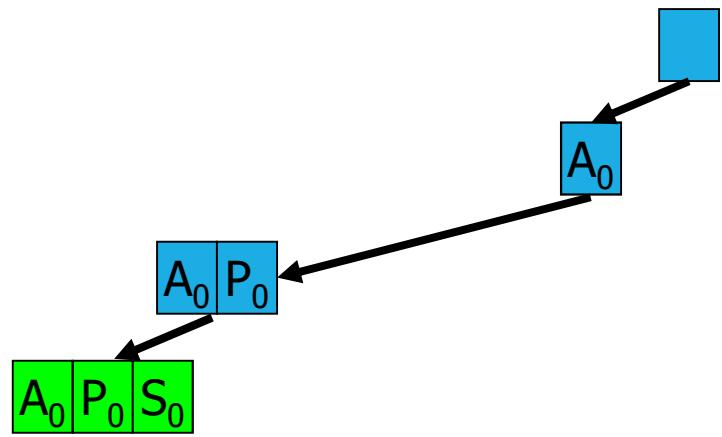
Il cliente può scegliere tra 2 antipasti A_0, A_1 , 3 primi P_0, P_1 e P_2 e 2 secondi S_0, S_1 .

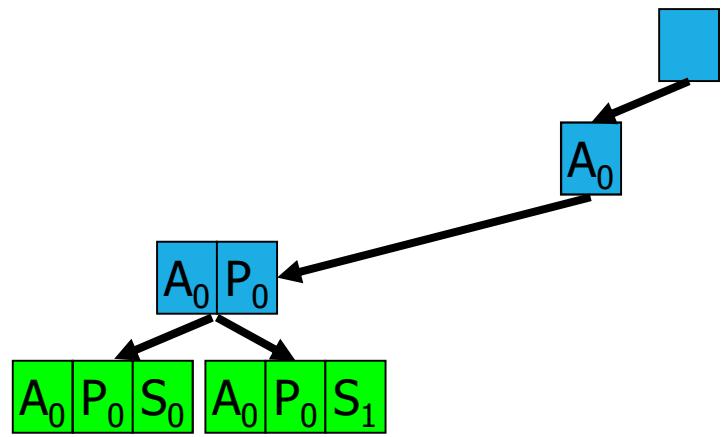
Quanti e quali pranzi diversi si possono scegliere con questo menu?

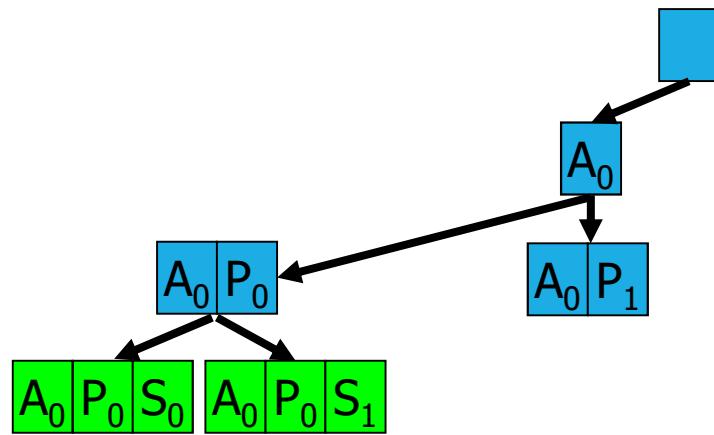


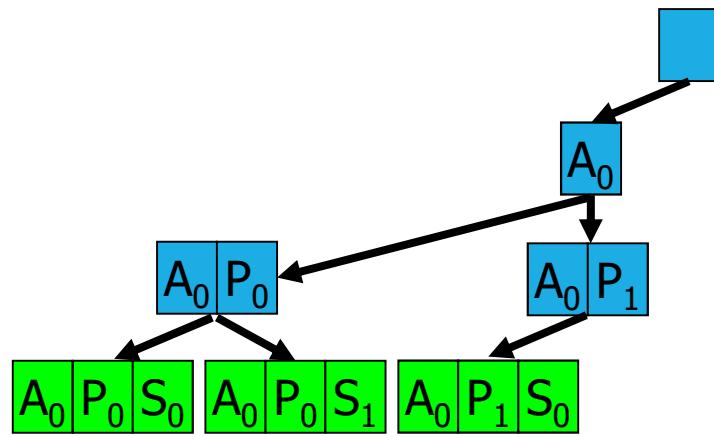


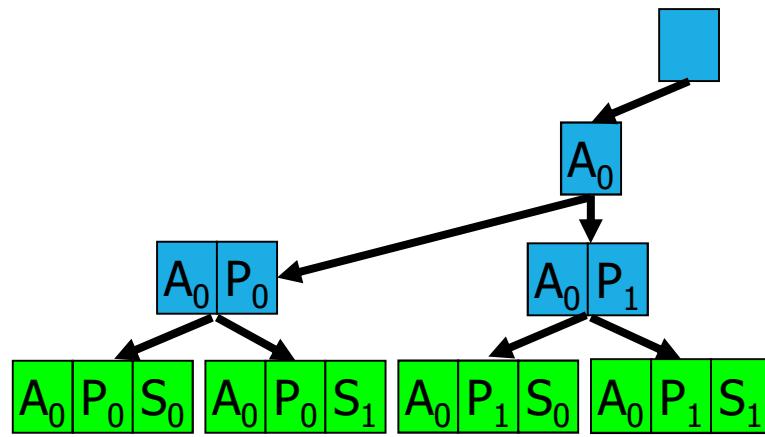


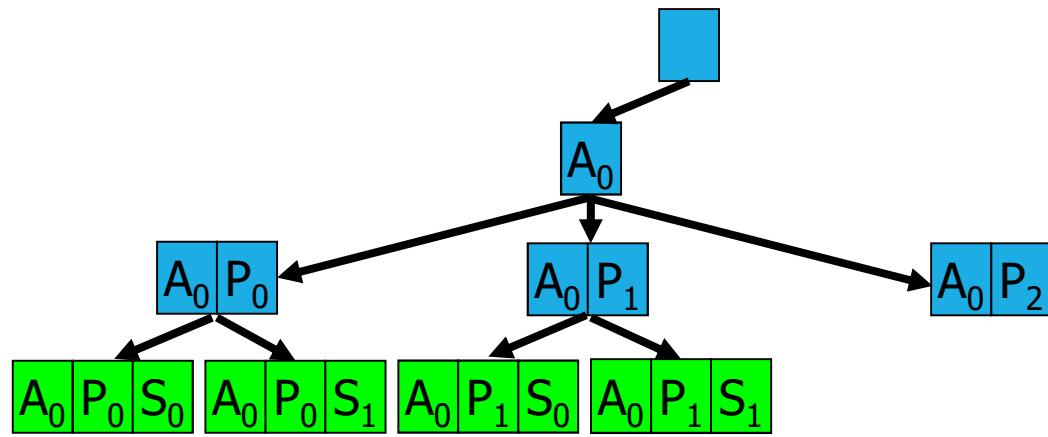


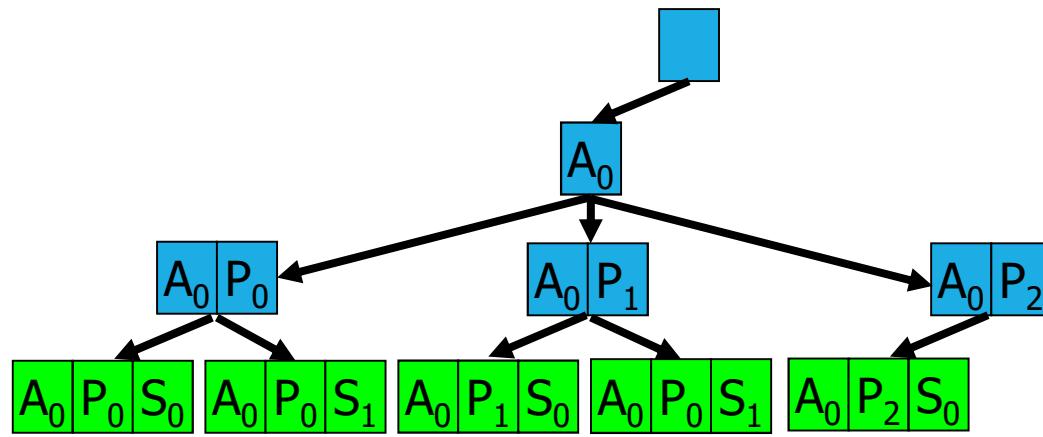


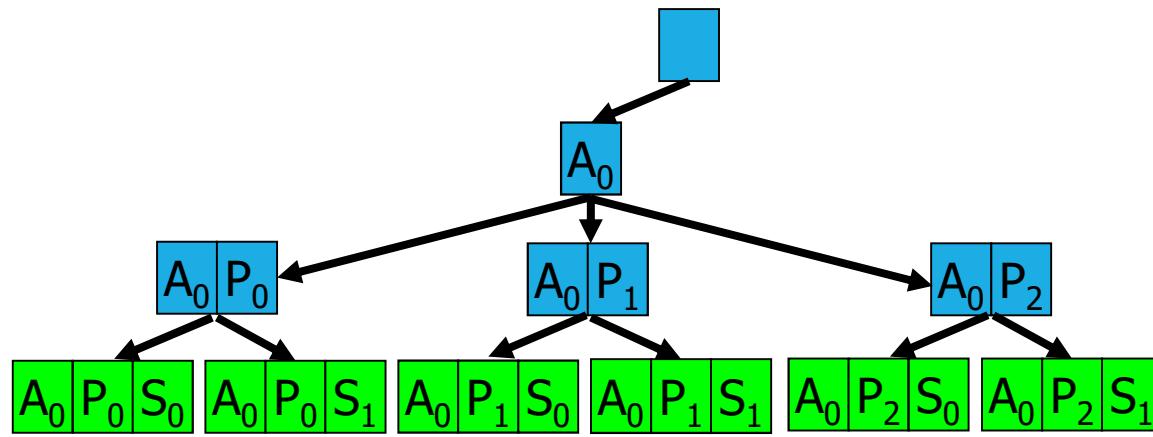


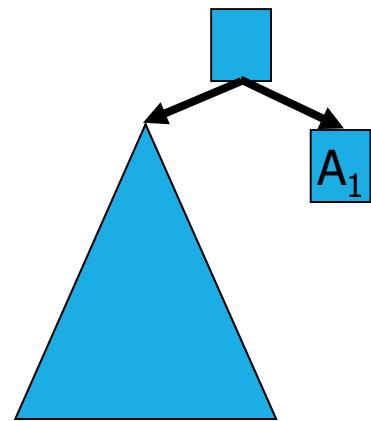












e così via

Calcolo Combinatorio e spazio delle soluzioni

- Scelte come elementi di un gruppo
- Spazio delle soluzioni caratterizzato dalle regole di associazione (raggruppamento) degli elementi
- Calcolo Combinatorio: regole di associazione
 - elementi **distinti** o no
 - elementi **ordinati** o no
 - elementi **ripetuti** o no

Il Calcolo Combinatorio: definizione

Il calcolo combinatorio:

- conta quanti sono i sottoinsiemi di un insieme dato che godono di una certa proprietà
- cioè determina il numero dei modi mediante i quali possono essere associati, secondo prefissate regole, gli elementi di uno stesso gruppo.

L'argomento sarà trattato in Metodi Matematici per l'Ingegneria.
Nel problem-solving serve enumerare questi modi, non solo contarli.

Principi-base: addizione

Se un insieme S di oggetti è diviso in sottoinsiemi $S_0 \dots S_{n-1}$ a 2 a 2 disgiunti tali che la loro unione sia S (definizione di partizione)

$$S = S_0 \cup S_1 \cup \dots \cup S_{n-1} \quad \& \quad \forall i \neq j \quad S_i \cap S_j = \emptyset$$

il numero degli oggetti in S può essere determinato sommando il numero degli oggetti in ciascuno degli insiemi $S_0 \dots S_{n-1}$

$$|S| = \sum_{i=0}^{n-1} |S_i|$$

Formulazione alternativa:

se un oggetto può essere scelto in p_0 modi da un gruppo S_0 , ... e
in p_{n-1} modi da un gruppo separato S_{n-1} , allora la selezione
dell'oggetto da uno qualunque degli n gruppi può essere fatta in
 $\sum_{i=0}^{n-1} |p_i|$ modi.

Esempio

Ci sono 4 corsi di Informatica e 5 di Matematica. Uno studente ne può seguire 1 solo. In quanti modi può scegliere?

Insiemi disgiunti \Rightarrow

Modello: principio di addizione

Numero di scelte = $4 + 5 = 9$

Principi-base: moltiplicazione

Dati n insiemi S_i ($0 \leq i < n$) ciascuno di cardinalità $|S_i|$, il numero di n -uple ordinate $(s_0 \dots s_{n-1})$ con $s_0 \in S_0 \dots s_{n-1} \in S_{n-1}$ è:

$$\prod_{i=0}^{n-1} |S_i|$$

Formulazione alternativa:

se un oggetto x_0 può essere scelto in p_0 modi da un gruppo, un oggetto x_1 può essere scelto in p_1 modi, un oggetto x_{n-1} può essere scelto in p_{n-1} modi, la scelta di una n -upla di oggetti $(x_0 \dots x_{n-1})$ può essere fatta in $p_0 \cdot p_1 \dots \cdot p_{n-1}$ modi.

Esempio

In un ristorante c'è un menu a prezzo fisso composto da antipasto, primo, secondo e dolce.

Il cliente può scegliere tra 2 antipasti, 3 primi, 2 secondi e 4 dolci.

Quanti pranzi diversi si possono scegliere con questo menu?

Modello: principio di moltiplicazione

Numero di scelte = $2 \times 3 \times 2 \times 4 = 48$

Criteri di raggruppamento

Si possono raggruppare k oggetti presi da un gruppo S di n elementi tenendo presente:

- **l'unicità** degli elementi: gli elementi del gruppo S sono tutti distinti, quindi S è un insieme? O è un multiinsieme (multiset)?
- **l'ordinamento**: 2 configurazioni sono le stesse a meno di un riordinamento?
- **le ripetizioni**: uno stesso oggetto del gruppo può o meno essere riusato più volte all'interno di uno stesso raggruppamento?

Disposizioni semplici

no ripetizioni

insieme

Una **disposizione semplice** $D_{n,k}$ di n oggetti distinti di classe k (a k a k) è un sottoinsieme ordinato composto da k degli n oggetti ($0 \leq k \leq n$).

l'ordinamento conta

Vi sono

$$D_{n,k} = \frac{n!}{(n-k)!} = n \cdot (n-1) \cdot \dots \cdot (n-k+1)$$

disposizioni semplici di n oggetti a k a k .

Si noti che:

- distinti \Rightarrow il gruppo su cui si opera è un insieme
- ordinato \Rightarrow l'ordinamento conta
- semplice \Rightarrow in ogni raggruppamento ci sono esattamente k oggetti non ripetuti

Due raggruppamenti sono diversi:

- o perché c'è almeno un elemento diverso
- o perché l'ordine è diverso.

Esempio

rappresentazione
posizionale: l'ordine conta!

Quanti e quali sono i numeri di 2 cifre distinte che si possono scrivere utilizzando i numeri 4, 9, 1 e 0?

n = 4

k = 2

no cifre
ripetute

Modello: disposizioni semplici

$$D_{4,2} = 4!/(4-2)! = 4 \cdot 3 = 12$$

Soluzione:

{49, 41, 40, 94, 91, 90, 14, 19, 10, 04, 09, 01 }

Disposizioni con ripetizione

ripetizioni

insieme

Una **disposizione con ripetizione** $D'_{n, k}$ di n oggetti distinti di classe k ($a \ k \ a \ k$) è un sottoinsieme ordinato composto da k degli n oggetti ($0 \leq k$) ognuno dei quali può essere preso sino a k volte.

Vi sono

nessun limite superiore!

l'ordinamento conta

$$D'_{n, k} = n^k$$

disposizioni con ripetizione di n oggetti a k a k .

Si noti che:

- distinti \Rightarrow il gruppo su cui si opera è un insieme
- ordinato \Rightarrow l'ordinamento conta
- assenza di «semplice» \Rightarrow in ogni raggruppamento uno stesso oggetto può figurare, ripetuto, fino ad un massimo di k volte
- k può essere $> n$

Due raggruppamenti sono diversi se uno di essi:

- contiene almeno un oggetto che non figura nell'altro oppure
- gli oggetti sono diversamente ordinati oppure
- gli oggetti che figurano in uno figurano anche nell'altro ma sono ripetuti un numero diverso di volte.

Esempio

rappresentazione
posizionale: l'ordine conta!

Quanti e quali sono i numeri binari puri su 4 bit?

n = 2

Ogni bit può assumere valore 0 o 1.

k = 4

Modello: disposizioni con ripetizione

$$D'_{2,4} = 2^4 = 16$$

Soluzione

{0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111
1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 }

Permutazioni semplici

no ripetizioni

insieme

Una disposizione semplice $D_{n,n}$ di n oggetti distinti di classe n (a n a n) si dice **permutazione semplice** P_n . Si tratta di un sottoinsieme ordinato composto dagli n oggetti.

l'ordinamento conta

Vi sono

$$P_n = D_{n,n} = n!$$

permutazioni semplici di n oggetti.

Si noti che:

- distinti \Rightarrow il gruppo su cui si opera è un insieme
- ordinato \Rightarrow l'ordinamento conta
- semplice \Rightarrow in ogni raggruppamento si sono esattamente n oggetti non ripetuti.

Due raggruppamenti sono diversi perché gli elementi sono gli stessi, ma l'ordine è diverso.

Esempio

Quanti e quali sono gli anagrammi di ORA (parola di 3 lettere distinte)?

n = 3

no ripetizioni

rappresentazione
posizionale: l'ordine conta!

Modello: permutazioni semplici

$$P_3 = 3! = 6$$

Soluzione

{ ORA, OAR, ROA, RAO, AOR, ARO }

Permutazioni con ripetizione

elementi ripetuti

Dato un multiinsieme di n oggetti di cui a uguali fra loro, b uguali fra loro, etc., il numero di **permute distinte con oggetti ripetuti** è:

l'ordine conta!

$$P_n^{(a, b, \dots)} = \frac{n!}{(a! \cdot b! \dots)}$$

Si noti che:

- assenza di «distinti» \Rightarrow il gruppo su cui si opera è un multiinsieme
- permutazioni \Rightarrow l'ordinamento conta

Due raggruppamenti sono diversi perché gli elementi sono gli stessi ma l'ordine è diverso.

Esempio

rappresentazione
posizionale: l'ordine conta!

Quanti e quali sono gli anagrammi distinti di ORO (parola di 3 lettere di cui 2 identiche)?

$n = 3$

$\alpha = 2$

Modello: permutazioni con ripetizione

$$P^{(2)}_3 = 3! / 2! = 3$$

Soluzione

{ OOR, ORO, ROO }

Combinazioni semplici

no ripetizioni

insieme

Una **combinazione semplice** $C_{n, k}$ di n oggetti distinti di classe k (a k a k) è un sottoinsieme non ordinato composto da k degli n oggetti ($0 \leq k \leq n$).

l'ordinamento non conta

Il numero di combinazioni di n elementi a k a k è al numero di disposizioni di n elementi a k a k diviso per il numero di permutazioni di k elementi.

Si noti che:

- distinti \Rightarrow il gruppo su cui si opera è un insieme
- non ordinato \Rightarrow l'ordinamento non conta
- semplice \Rightarrow in ogni raggruppamento ci sono esattamente k oggetti non ripetuti

Due raggruppamenti sono diversi perché c'è almeno un elemento diverso.

Vi sono:

$$C_{n,k} = \binom{n}{k} = \frac{D_{n,k}}{P_k} = \frac{n!}{k!(n-k)!}$$

combinazioni semplici di n oggetti a k a k.

coefficiente binomiale

Definizione ricorsiva del coefficiente binomiale:

$$\binom{n}{0} = \binom{n}{n} = 1$$

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Esempio

I'ordine non conta!

Quante terne si possono fare con i 90 numeri del gioco del Lotto?

k = 3

n = 90

Modello: combinazioni semplici

$$C_{90,3} = \frac{90!}{3!(90-3)!} = \frac{90 \cdot 89 \cdot 88 \cdot 87!}{3 \cdot 2 \cdot 87!} = 117480$$

In un torneo quadrangolare di calcio tra Juve, Toro, Inter e Milan di sola andata, quante e quali partite si disputano?

n= 4, k = 2

$$C_{4,2} = 4! / 2!(4-2)! = 6$$

Soluzione

{ Juve-Milan, Juve-Inter, Juve-Toro, Milan-Inter, Milan-Toro, Inter-Toro }

Combinazioni con ripetizione

ripetizioni

Una **combinazione con ripetizione** $C'_{n, k}$ di n oggetti distinti di classe k (a k a k) è un sottoinsieme non ordinato composto da k degli n oggetti ($0 \leq k$) ognuno dei quali può essere preso sino a k volte.

nessun limite superiore!

Vi sono

$$C'_{n,k} = \frac{(n+k-1)!}{k!(n-1)!}$$

insieme

l'ordinamento non conta

combinazioni con ripetizione di n oggetti a k a k .

Si noti che:

- distinti \Rightarrow il gruppo su cui si opera è un insieme
- non ordinato \Rightarrow l'ordinamento non conta
- assenza di «semplice» \Rightarrow in ogni raggruppamento uno stesso oggetto può figurare, ripetuto, fino ad un massimo di k volte
- k può essere $> n$.

Due raggruppamenti sono diversi se uno di essi:

- contiene almeno un oggetto che non figura nell'altro oppure
- gli oggetti che figurano in uno figurano anche nell'altro ma sono ripetuti un numero diverso di volte.

Esempio

$k = 2$

Lanciando contemporaneamente 2 dadi, quante sono le composizioni con cui si possono presentare le facce?

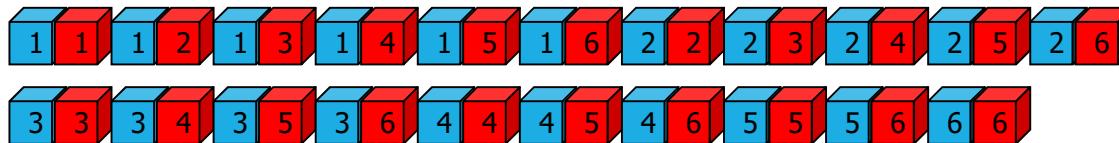
l'ordine non conta!

$n = 6$

Modello: combinazioni con ripetizione

$$C'_{6,2} = (6 + 2 - 1)! / 2!(6-1)! = 21$$

Soluzione



L'Insieme delle Parti

Dato un insieme S di k elementi ($k=|S|$), l'insieme delle parti (o powerset) $\wp(S)$ è l'insieme dei sottoinsiemi di S , incluso S stesso e l'insieme vuoto.

Esempio:

$S = \{1, 2, 3, 4\}$ e $k = 4$

$\wp(S) = \{\{\}, \{4\}, \{3\}, \{3,4\}, \{2\}, \{2,4\}, \{2,3\}, \{2,3,4\}, \{1\}, \{1,4\}, \{1,3\}, \{1,3,4\}, \{1,2\}, \{1,2,4\}, \{1,2,3\}, \{1,2,3,4\}\}$

Le Partizioni di un insieme

Dato un insieme I di n elementi, una collezione $S = \{S_i\}$ di blocchi forma una **partizione** di I se e solo se valgono tutte le seguenti condizioni:

- i blocchi sono non vuoti:

$$\forall S_i \in S \quad S_i \neq \emptyset$$

- i blocchi sono a coppie disgiunti:

$$\forall S_i, S_j \in S \text{ con } i \neq j \quad S_i \cap S_j = \emptyset$$

- la loro unione è I :

$$I = \bigcup_i S_i$$

Il numero di blocchi k varia da 1 (blocco = insieme I) a n (ogni blocco contiene un solo elemento di I).

Esempio

$I = \{1, 2, 3, 4\}$ $n = 4$

$k = 1$

1 partizione:

$\{1, 2, 3, 4\}$

$k = 2$

7 partizioni:

$\{1, 2, 3\}, \{4\}$

$\{1, 2, 4\}, \{3\}$

$\{1, 2\}, \{3, 4\}$

$\{1, 3, 4\}, \{2\}$

$\{1, 3\}, \{2, 4\}$

$\{1, 4\}, \{2, 3\}$

$\{1\}, \{2, 3, 4\}$

$k = 3$

6 partizioni:

$\{1, 2\}, \{3\}, \{4\}$

$\{1, 3\}, \{2\}, \{4\}$

$\{1\}, \{2, 3\}, \{4\}$

$\{1, 4\}, \{2\}, \{3\}$

$\{1\}, \{2, 4\}, \{3\}$

$\{1\}, \{2\}, \{3, 4\}$

$k = 4$

1 partizione:

$\{1\}, \{2\}, \{3\}, \{4\}$

Numero di partizioni

Il numero complessivo delle partizioni di un insieme I di n oggetti in k blocchi con $1 \leq k \leq n$ è dato dai numeri di Bell definiti dalla seguente ricorrenza:

$$B_0 = 1$$

$$B_{n+1} = \sum_{k=0}^n \binom{n}{k} \cdot B_k$$

I primi numeri di Bell sono: $B_0 = 1$, $B_1 = 1$, $B_2 = 2$, $B_3 = 5$, $B_4 = 15$, $B_5 = 52$,

Lo spazio di ricerca non è modellato tramite calcolo combinatorio.

Nota: l'ordine dei blocchi e degli elementi in ogni blocco non conta.

Di conseguenza le 2 partizioni:

$\{1, 3\}, \{2\}, \{4\}$ e $\{2\}, \{3, 1\}, \{4\}$

sono identiche.

Esplorazione esaustiva dello spazio delle soluzioni

Paolo Camurati



Scomposizione in sottoproblemi

È il passo più importante del progetto di una soluzione ricorsiva:

- bisogna identificare il problema risolto dalla singola ricorsione
- cioè suddividere il lavoro tra varie chiamate ricorsive.

Si opera in maniera distribuita, senza visione unitaria della soluzione.

Approcci:

- ogni ricorsione sceglie un elemento della soluzione.
Terminazione: la soluzione ha raggiunto la dimensione richiesta oppure non ci sono più scelte
- la ricorsione esamina uno degli elementi dell'insieme di partenza per decidere se e dove andrà aggiunto alla soluzione.

Si segue il primo approccio perché più intuitivo.

Strutture dati

- Strutture dati
 - globali, cioè comuni a tutte le istanze della funzione ricorsiva
 - locali, cioè locali a ciascuna delle istanze
- Strutture dati globali:
 - dati del problema (matrice, mappa, grafo), vincoli, scelte disponibili, soluzione
- Strutture dati locali:
 - indici di livello di chiamata ricorsiva, copie locali di strutture dati, indici o puntatori a parti di strutture dati globali

- Globale nell'accezione precedente non implica uso di variabili globali C
- Uso di variabili globali C per strutture dati globali:
 - sconsigliato ma non vietato quando le funzioni ricorsive operano su pochi e ben noti dati
 - vantaggio: pochi parametri passati alle funzioni ricorsive
- Soluzione adottata: tutti i dati (globali e locali) passati come parametri. Possibilità di racchiuderli in una **struct** per leggibilità.

Tipologie di strutture dati per oggetti interi

- Oggetti non interi: tavole di simboli per ricondursi ad interi
- insieme o insiemi di oggetti di partenza:
 - unico: vettore `val`
 - molteplici: sottovettori di tipo `Livello`
 - alternativa: liste
- soluzione: non si chiede di memorizzarle tutte, ma solo di elencarle:
 - vettore `sol`
 - variabile scalare (ad esempio `cnt`) passata come parametro `by value` e ritornata

- indici:
 - pos identifica il livello della ricorsione e serve per decidere quali caselle di scelta usare o soluzione riempire
 - n e k: indicano la dimensione del problema e della soluzione cercata
- vincoli: non tutte le scelte sono lecite. Quelle lecite soddisfano vincoli:
 - statici
 - dinamici, (ad esempio il vettore mark).

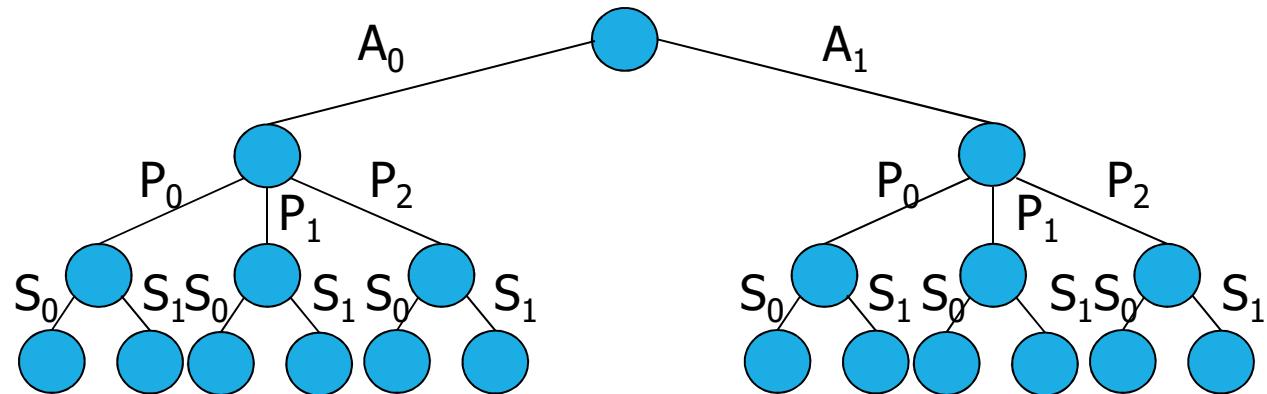
Principio di moltiplicazione

- Si effettuano n scelte in successione, rappresentate mediante un albero
- I nodi hanno un numero di figli variabile livello per livello. Ognuno dei figli può essere visto come una delle scelte possibili a quel livello
- Il massimo numero di figli determina il grado dell'albero
- L'altezza dell'albero è n . Le soluzioni sono le etichette degli archi che si incontrano in ogni cammino radice-foglia.

Esempio

Menu con scelta tra 2 antipasti (A_0, A_1), 3 primi (P_0, P_1, P_2) e 2 secondi (S_0, S_1) ($n=k=3$).

Albero di grado 3 e altezza 3, 12 percorsi radice-foglie.



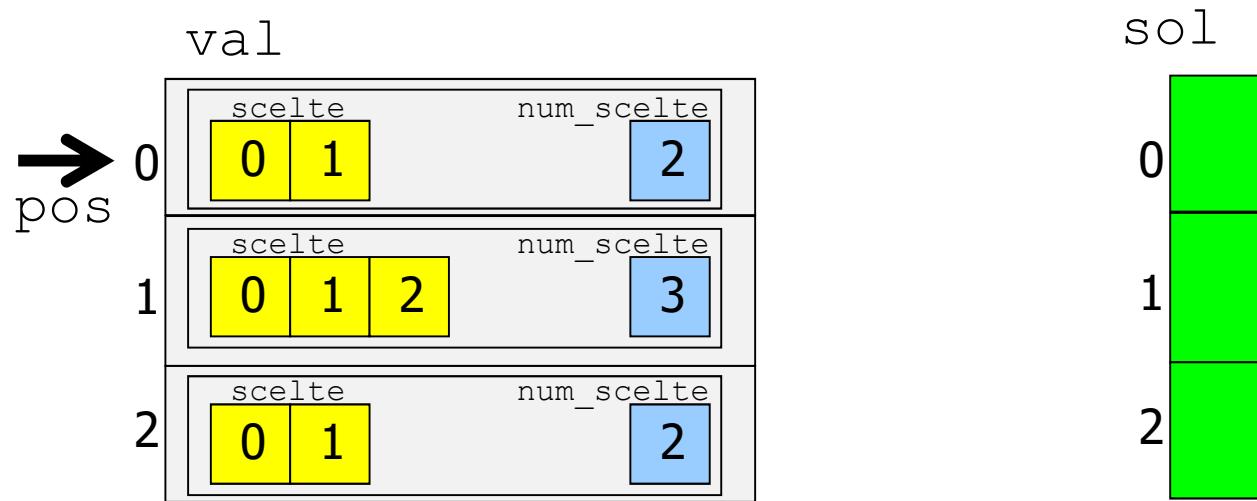
Soluzione:

$(A_0, P_0, S_0), (A_0, P_0, S_1), (A_0, P_1, S_0), (A_0, P_1, S_1), (A_0, P_2, S_0), (A_0, P_2, S_1),$
 $(A_1, P_0, S_0), (A_1, P_0, S_1), (A_1, P_1, S_0), (A_1, P_1, S_1), (A_1, P_2, S_0), (A_1, P_2, S_1)$

I principi-base dell'esplorazione

- Si prendono n *decisioni in sequenza*, ciascuna tra diverse *scelte*, il cui numero è fisso dato il livello di decisione, ma variabile di livello in livello
- le scelte sono in corrispondenza biunivoca con un sottoinsieme degli interi (non necessariamente contigui)
- le scelte possibili sono memorizzate in un vettore `val` di dimensione n di strutture `Livello`. Ogni struttura è un intero per il numero di scelte per quel livello `num_scelte` e un vettore `*scelte` di interi di quella dimensione per le scelte
- la soluzione è memorizzata in un vettore `sol` di interi di dimensione n .

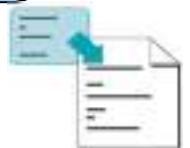
In riferimento all'esempio



- si vogliono enumerare tutte le soluzioni, esplorandone l'intero spazio
- tutte le soluzioni sono valide
- le chiamate ricorsive sono associate alla soluzione, la cui dimensione ad ognuna di esse cresce di 1
- una soluzione viene rappresentata come un vettore sol di n elementi che registra le scelte fatte ad ogni passo
- ad ogni passo pos indica la dimensione della soluzione parziale
- se $\text{pos} >= n$ si è trovata una soluzione

- il passo ricorsivo itera sulle scelte possibili per il valore corrente di `pos`, cioè il contenuto di `sol[pos]` è preso da `val[pos].scelte[i]` estendendo ogni volta la soluzione
- e ricorre sulla scelta `pos+1`-esima
- `cnt` è il valore di ritorno della ricorsione e conteggia il numero di soluzioni.

```
typedef struct {int *scelte; int num_scelte; } Livello;  
  
val = malloc(n*sizeof(Livello));  
  
for (i=0; i<n; i++)  
    val[i].scelte = malloc(val[i].n_scelte*sizeof(int));  
  
sol = malloc(n*sizeof(int));
```



01calcolocombinatorio

```
int princ_molt(int pos, Livello *val, int *sol, int n, int cnt) {
    int i;
    if (pos >= n) {
        for (i = 0; i < n; i++)
            printf("%d ", sol[i]);
        printf("\n");
        return cnt+1;
    }
    for (i = 0; i < val[pos].num_scelte; i++) {
        sol[pos] = val[pos].scelte[i];
        cnt = princ_molt(pos+1, val, sol, n, cnt);
    }
    return cnt;
}
```

Modelli

Lo spazio delle possibilità può essere modellato come quello delle:

- disposizioni semplici
- disposizioni con ripetizione
- permutazioni semplici
- permutazioni con ripetizione
- combinazioni semplici
- combinazioni con ripetizione
- insieme delle parti
- (partizioni).

I principi-base dell'esplorazione

- Si immagini di dovere prendere delle *decisioni in sequenza*, ciascuna tra diverse *scelte* senza avere informazioni che guidano la decisione
- le scelte possibili sono memorizzate in un vettore `val` di interi di dimensione n
- si vogliono enumerare tutte le soluzioni, esplorandone l'intero spazio, decidendo, una volta raggiunta una soluzione, se è valida/ottima

Alternative:

- le chiamate ricorsive sono associate alla soluzione, la cui dimensione ad ognuna di esse cresce di 1. Terminazione quando la dimensione della soluzione corrente è quella finale
- le chiamate ricorsive sono associate alle scelte. Ad ogni chiamata si effettua una scelta. Terminazione quando tutte le scelte sono state esaurite.

Nel seguito si adotta la prima alternativa.

- una soluzione viene rappresentata come un vettore `sol` di interi di k elementi che registra le scelte fatte ad ogni passo
- ad ogni passo `pos` indica la dimensione della soluzione parziale
 - se $pos >= k$, si è trovata una soluzione, di cui si appura la validità/ottimalità
 - altrimenti, se è possibile una scelta `pos+1`-esima, con essa si estende `sol` e si procede da questa ricorsivamente
 - altrimenti, si “annulla” l’ultima scelta `pos` (backtrack) e si ricomincia dalla `pos-1`-esima scelta
- iterazione: il contenuto di `sol[pos]` è preso da `val`.

Il Backtracking

Il backtracking non è un paradigma vero e proprio, come il divide et impera, il greedy o la programmazione dinamica in quanto non vi è uno schema generale.

È piuttosto una tecnica algoritmica per esaminare ordinatamente le possibili istanze (soluzioni ammissibili o valide) di uno spazio di ricerca.

Un'applicazione che dimostra l'importanza del backtracking è il Puzzle di Einstein.

Il Puzzle di Einstein



In una strada sono allineate 5 case:

- di 5 colori diversi (blu, verde, rosso, avorio, giallo)
- abitate 5 persone di diversa nazionalità, che:
- bevono 5 bevande diverse
- hanno 5 animali diversi
- fumano sigarette di 5 marche diverse.

Dato un insieme di fatti, determinare:

- chi beve acqua
- chi ha la zebra.

Secondo Einstein solo il 2% delle persone saprebbe rispondere.

The screenshot shows the homepage of the Corriere della Sera website. At the top, there are three tabs: "Politecnico di Torino", "Corriere della Sera - Ultime notizie", and "Nuova scheda". The main navigation bar includes links for "SEZIONI", "EDIZIONI LOCALI", "CORRIERE TV", "ARCHIVIO", "TROVOCASA", "TROVOLAVORO", "SERVIZI", "CERCA", "LOGIN", and "SCOPRI". A banner at the top left promotes "PostePay" with a "50%" discount. The main headline reads: "Manovra, Salvini e Di Maio aprono all'Ue Il piano: meno deficit. È derby sul reddito "Blockchain" e appalti: il dossier di Conte". Below the headline, there is a small bio for the author: "di Bruno Dazzi, Aro Croci, Merco Lamonica, Lorenzo Selvi". To the right of the main content, there are two columns of articles. The first column features a photo of Einstein writing on a chalkboard with the equation $R = \infty$. The second column has a heading: "Il test di Einstein, la prova regina per essere assunti: prova il quesito logico". This specific section is highlighted with a red oval. The footer contains links for "DATAROOM", "Scrivi qui per eseguire la ricerca", and various browser icons.

Fatti

1. La casa verde è immediatamente a destra della casa avorio
2. Le Kool sono fumate nella casa gialla
3. Le Kool sono fumate nella casa vicino a quella dove si tiene il cavallo
4. Il fumatore di Old Gold possiede lumache
5. Lo spagnolo è proprietario del cane
6. Il giapponese fuma Parliament
7. L'uomo che fuma Chesterfield vive nella casa accanto all'uomo con la volpe

Fatti

8. Il norvegese vive nella prima casa
9. Il norvegese vive vicino alla casa blu
10. L'inglese vive nella casa rossa
11. Il latte si beve nella casa in mezzo
12. Il caffè è bevuto nella casa verde
13. Chi fuma le Lucky Strike beve succo d'arancia
14. Il the è bevuto dall'ucraino

Quesiti

- Chi beve acqua?
- Chi ha la zebra?

Secondo Einstein solo il 2% delle persone saprebbe rispondere.

Fatto #1

	1	2	3	4	5
Nazione	NO 🇳🇴	UK UA JP ES	UK UA JP ES	UK UA JP ES	UK UA JP ES
Colore					
Animale					
Bevanda					
Sigarette					

Il norvegese vive nella prima casa



Nelle altre case non vive il norvegese

Fatto #2

	1	2	3	4	5
Nazione	NO 	UK UA JP ES	UK UA JP ES	UK UA JP ES	UK UA JP ES
Colore	R V A G	B	R V A G R V A G R V A G		
Animale					
Bevanda					
Sigarette					

Il norvegese vive vicino alla casa blu



Le altre case non sono blu

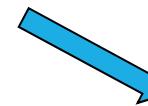
Fatto #3

	1	2	3	4	5
Nazione	NO	UA JP ES	UK UA JP ES	UK UA JP ES	UK UA JP ES
Colore	V A G	B	R V A G R V A G R V A G		
Animale					
Bevanda					
Sigarette					

L'inglese vive nella casa rossa



La casa del norvegese non è rossa



L'inglese non vive nella casa blu

Fatto #4

	1	2	3	4	5
Nazione	NO	UA JP ES	UK UA JP ES	UK UA JP ES	UK UA JP ES
Colore	V A G	B	R V A G R V A G R V A G		
Animale					
Bevanda	TCSA	TCSA	Latte	TCSA	TCSA
Sigarette					

Il latte si beve nella casa in mezzo



Nelle altre case non si beve latte

Fatto #5

	1	2	3	4	5
Nazione	NO 🇳🇴	UA JP ES	UK UA JP ES	UK UA JP ES	UK UA JP ES
Colore	V A G	B	R A G	R V A G R V A G	
Animale					
Bevanda	TCSA	TSA	Latte	TCSA	TCSA
Sigarette					

Il caffè è bevuto nella casa verde



La casa di mezzo non è verde



Il latte non si beve nella casa blu

Fatto #6

	1	2	3	4	5
Nazione	NO 🇳🇴	UA JP ES	UK UA JP ES	UK UA JP ES	UK UA JP ES
Colore	V A G	B	R A G	R V A G R V A G	
Animale					
Bevanda	TCSA	TCSA	Latte	TCSA	TCSA
Sigarette	OKCLP	OKCLP	OKCP	OKCLP	OKCLP

Chi fuma le Lucky Strike beve succo d'arancia



Chi beve latte non fuma Lucky Strike

Fatto #7

	1	2	3	4	5
Nazione	NO	UA JP ES	UK JP ES	UK UA JP ES	UK UA JP ES
Colore	V A G	B	R A G	R V A G R V A G	
Animale					
Bevanda	CSA	TCSA	Latte	TCSA	TCSA
Sigarette	OKCLP	OKCLP	OKCP	OKCLP	OKCLP

Il the è bevuto dall'ucraino



Il norvegese non beve the

L'ucraino non vive nella casa di mezzo

Fatti #8 e #9

	1	2	3	4	5
Nazione	NO	UA JP ES	UK JP ES	UK UA JP ES	UK UA JP ES
Colore	G	B	R A	R V A	R V A
Animale					
Bevanda	CSA	TCSA	Latte	TCSA	TCSA
Sigarette	Kool	OCLP	OCP	OCLP	OCLP

La casa verde è immediatamente a destra
della casa avorio
Le Kool sono fumate nella casa gialla

La prima casa non è né verde
né avorio, è gialla
Solo il norvegese fuma Kool

Fatti #8 e #9

	1	2	3	4	5
Nazione	NO 🇳🇴	UA JP ES	UK JP ES	UK UA JP ES	UK UA JP ES
Colore	G	B	R A	R V A	R V
Animale					
Bevanda	CSA	TCSA	Latte	TCSA	TCSA
Sigarette	Kool	OCLP	OCP	OCLP	OCLP

La casa verde è immediatamente a destra
della casa avorio
Le Kool sono fumate nella casa gialla



L'ultima casa non è avorio, in
quanto non ce n'è una verde a
destra

Fatto #10

	1	2	3	4	5
Nazione	NO	UA JP ES	UK JP ES	UK UA JP ES	UK UA JP ES
Colore	G	B	R A	R V A	R V
Animale	CLVZ		CLVZ	CLVZ	CLVZ
Bevanda	CSA	TCSA	Latte	TCSA	TCSA
Sigarette	Kool	OCLP	OCP	OCLP	OCLP

Le Kool sono fumate nella casa vicino a quella dove si tiene il cavallo

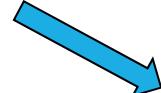
Il cavallo è nella casa blu

Nelle altre case non c'è il cavallo

Fatto #11

	1	2	3	4	5
Nazione	NO	UA JP ES	UK JP ES	UK UA JP ES	UK UA JP ES
Colore	G	B	R A	R V A	R V
Animale	CVZ		CLVZ	CLVZ	CLVZ
Bevanda	CSA	TCSA	Latte	TCSA	TCSA
Sigarette	Kool	CLP	OCP	OCLP	OCLP

Il fumatore di Old Gold possiede lumache



Il norvegese non possiede lumache Chi possiede il cavallo non fuma Old Gold

Fatto #12

	1	2	3	4	5
Nazione	NO	UA JP	UK JP ES	UK UA JP ES	UK UA JP ES
Colore	G	B	R A	R V A	R V
Animale	VZ		CLVZ	CLVZ	CLVZ
Bevanda	CSA	TCSA	Latte	TCSA	TCSA
Sigarette	Kool	CLP	OCP	OCLP	OCLP

Lo spagnolo è proprietario del cane



Il norvegese non possiede il cane Chi possiede il cavallo non è spagnolo

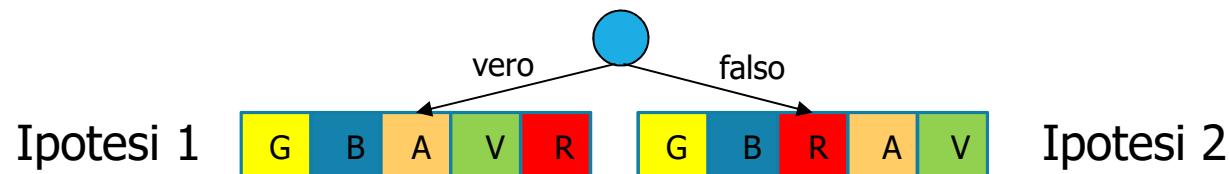
Fatto #5, fatto #6, fatto #7

	1	2	3	4	5
Nazione	NO 	UA JP	UK JP ES	UK UA JP ES	UK UA JP ES
Colore	G	B	R A	R V A	R V
Animale	VZ		CLVZ	CLVZ	CLVZ
Bevanda	Acqua	TCS	Latte	TCS	TCS
Sigarette	Kool	CLP	OCP	OCLP	OCLP

- Il caffè è bevuto nella casa verde
 - Il the è bevuto dall'ucraino
 - Chi fuma le Lucky Strike beve succo d'arancia
- } → Il Norvegese non beve né the, né caffè, né succo, beve acqua

Ipotesi

La casa verde è immediatamente a destra della casa avorio



Ip. #1: fatto #3, fatto #5

G	B	A	V	R
---	---	---	---	---

	1	2	3	4	5
Nazione	NO	UA JP	JP ES	UA JP ES	UK
Colore	G	B	A	V	R
Animale	VZ		CLVZ	CLVZ	CLVZ
Bevanda	Acqua	TS	Latte	Caffè	TS
Sigarette	Kool	CLP	OCLP	OCLP	OCLP

- L'inglese vive nella casa rossa
- Il caffè è bevuto nella casa verde

Ip. #1: fatto #7

	1	2	3	4	5
Nazione	NO	UA	JP ES	JP ES	UK
Colore	G	B	A	V	R
Animale	VZ		CLVZ	CLVZ	CLVZ
Bevanda	Acqua	The	Latte	Caffè	Succo
Sigarette	Kool	CL	OCLP	OCLP	OCL

Il the è bevuto dall'ucraino → L'inglese beve succo e non fuma Parliament

L'ucraino abita nella casa blu

L'ucraino non fuma Parliament

Ip. #1: fatto #6

	1	2	3	4	5
Nazione	NO	UA	JP ES	JP ES	UK
Colore	G	B	A	V	R
Animale	VZ		CLVZ	CLVZ	CLVZ
Bevanda	Acqua	The	Latte	Caffè	Succo
Sigarette	Kool	Chest.	OP	OP	Lucky

Chi fuma le Lucky Strike beve succo d'arancia



L'inglese fuma Lucky Strike



L'ucraino fuma Chesterfield

Ip. #1: fatto #12

	1	2	3	4	5
Nazione	NO	UA	JP ES	JP ES	UK
Colore	G	B	A	V	R
Animale	VZ		CLVZ	CLVZ	LVZ
Bevanda	Acqua	The	Latte	Caffè	Succo
Sigarette	Kool	Chest.	OP	OP	Lucky

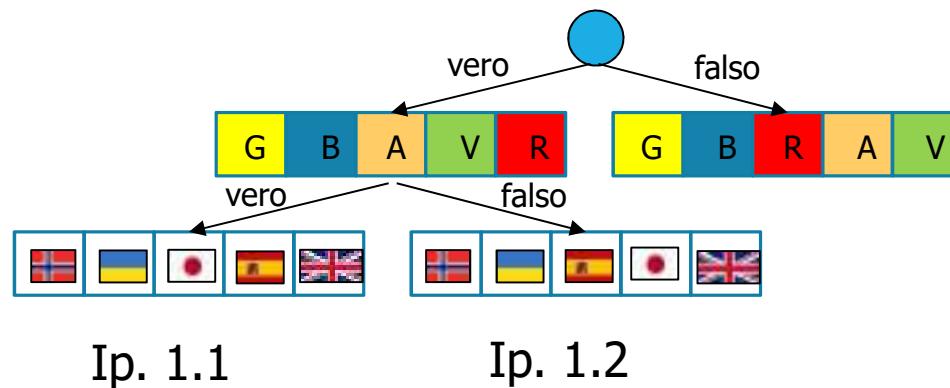
Lo spagnolo è proprietario del cane



L'inglese non possiede il cane

Ipotesi

Giapponese in casa 3, spagnolo in casa 4

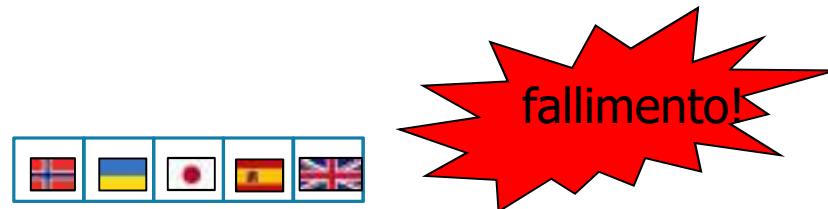


Ip. #1.1: fatto #12, fatto #13

1	2	3	4	5	
Nazione	NO	UA	JP	ES	UK
Colore	G	B	A	V	R
Animale	VZ		LVZ		LVZ
Bevanda	Acqua	The	Latte	Caffè	Succo
Sigarette	Kool	Chest.	Parl.	Old	Lucky

- Lo spagnolo possiede il cane
 - Il giapponese fuma Parliament
- } → Lo spagnolo fuma Old Gold

Ip. #1.1: fatto #11



	1	2	3	4	5
Nazione	NO	UA	JP	ES	UK
Colore	G	B	A	V	R
Animale	VZ		LVZ		LVZ
Bevanda	Acqua	The	Latte	Caffè	Succo
Sigarette	Kool	Chest.	Parl.	Old	Lucky

Il fumatore di Old Gold possiede lumache → **impossibile!** → **backtrack**

Ip. #1.2: fatto #12, fatto #13, deduzione

	1	2	3	4	5
Nazione	NO	UA	ES	JP	UK
Colore	G	B	A	V	R
Animale	VZ			LVZ	LVZ
Bevanda	Acqua	The	Latte	Caffè	Succo
Sigarette	Kool	Chest.	Old	Parl.	Lucky

- Il giapponese fuma Parliament
 - Lo spagnolo possiede il cane
- } → Lo spagnolo fuma Old Gold

Ip. #1.2: fatto #11

fallimento!

	1	2	3	4	5
Nazione	NO	UA	ES	JP	UK
Colore	G	B	A	V	R
Animale	VZ			LVZ	LVZ
Bevanda	Acqua	The	Latte	Caffè	Succo
Sigarette	Kool	Chest.	Old	Parl.	Lucky

Il fumatore di Old Gold possiede lumache → **impossibile!** → **backtrack**

Ip. #2: fatto #3, fatto #5



	1	2	3	4	5
Nazione	NO	UA JP	UK	UA JP ES	JP ES
Colore	G	B	R	A	V
Animale	VZ		CLVZ	CLVZ	CLVZ
Bevanda	Acqua	TS	Latte	TS	Caffè
Sigarette	Kool	CLP	OCP	OCLP	OCLP

- L'inglese vive nella casa rossa
- Il caffè è bevuto nella casa verde

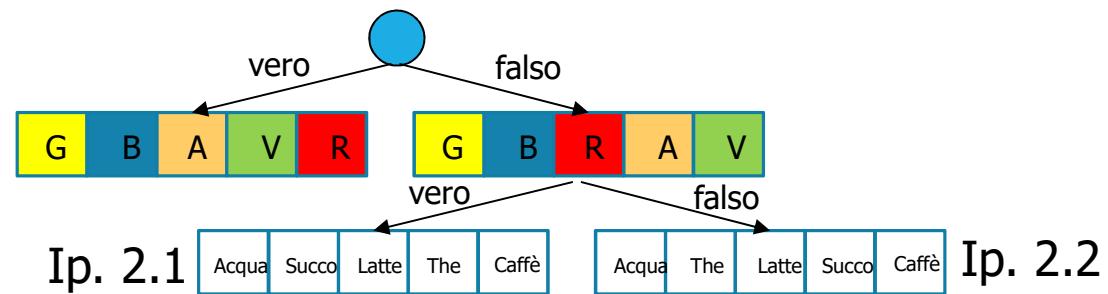
Ip. #2: fatto #5, fatto #7, fatto #12

	1	2	3	4	5
Nazione	NO 🇳🇴	UA JP	UK 🇬🇧	UA JP ES	JP ES
Colore	G	B	R	A	V
Animale	VZ		LVZ	CLVZ	CLVZ
Bevanda	Acqua	TS	Latte	TS	Caffè
Sigarette	Kool	CLP	OCP	OCLP	OCLP

- Il the è bevuto dall'ucraino
 - Il caffè è bevuto nella casa verde
 - Lo spagnolo è proprietario del cane
- } → L'ucraino non abita nella casa verde
- L'inglese non possiede il cane

Ipotesi

Succo in casa blu, the in casa avorio



Ip. #2.1: fatto #6, fatto #7

	Acqua	Succo	Latte	The	Caffè
Nazione	NO 🇳🇴	UA JP	UK 🇬🇧	UA 🇺🇦	JP ES
Colore	G	B	R	A	V
Animale	VZ		LVZ	CLVZ	CLVZ
Bevanda	Acqua	Succo	Latte	The	Caffè
Sigarette	Kool	Lucky	OCP	OCP	OCP

- Il the è bevuto dall'ucraino
- Chi fuma le Lucky Strike beve succo d'arancia

Ip. #2.1: deduzione

		Acqua	Succo	Latte	The	Caffè
	1	2	3	4	5	
Nazione	NO	JP	UK	UA	JP ES	
Colore	G	B	R	A	V	
Animale	VZ		LVZ	CLVZ	CLVZ	
Bevanda	Acqua	Succo	Latte	The	Caffè	
Sigarette	Kool	Lucky	OCP	OCP	OCP	

Il giapponese abita nella casa blu

Ip. #2.1: fatto #13



	1	2	3	4	5
Nazione	NO 🇳🇴	JP 🇯🇵	UK 🇬🇧	UA 🇺🇦	JP ES
Colore	G	B	R	A	V
Animale	VZ		LVZ	CLVZ	CLVZ
Bevanda	Acqua	Succo	Latte	The	Caffè
Sigarette	Kool	Lucky Parliament	OC	OC	OC

Il giapponese fuma Parliament → **impossibile!** → **backtrack**

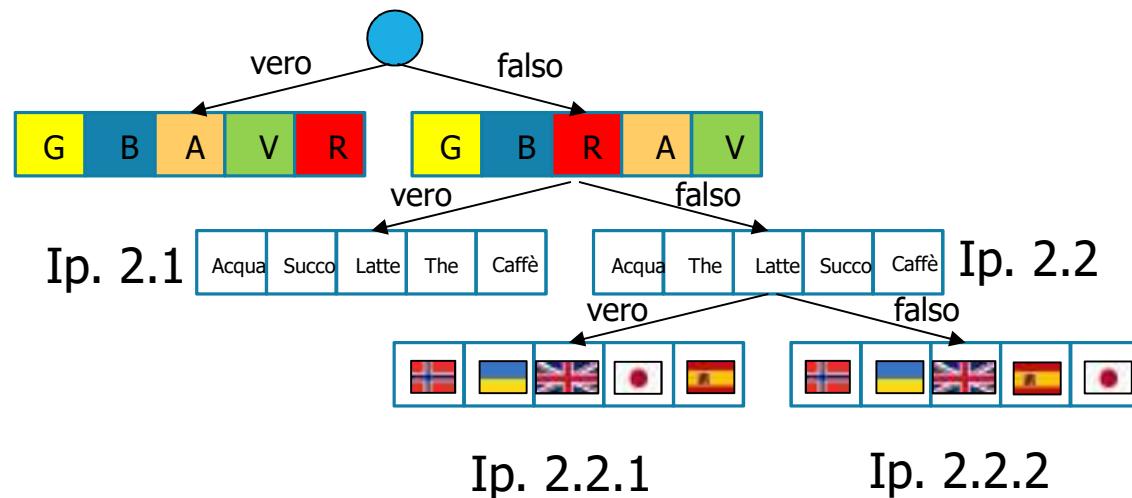
Ip. 2.2: fatto #6, fatto #7

	Acqua	The	Latte	Succo	Caffè
Nazione	NO	UA	UK	JP ES	JP ES
Colore	G	B	R	A	V
Animale	VZ		LVZ	CLVZ	CLVZ
Bevanda	Acqua	The	Latte	Succo	Caffè
Sigarette	Kool	CP	OCP	Lucky	OCP

- Il the è bevuto dall'ucraino
- Chi fuma le Lucky Strike beve succo d'arancia

Ipotesi

Spagnolo in casa 4, giapponese in casa 5



Ip. 2.2.1 : fatto #6, fatto #12, fatto #13



	1	2	3	4	5
Nazione	NO	UA	UK	JP	ES
Colore	G	B	R	A	V
Animale	VZ		LVZ	LVZ	
Bevanda	Acqua	The	Latte	Succo	Caffè
Sigarette	Kool	C	OC	Lucky Parliament	

- Lo spagnolo è proprietario del cane
 - Il giapponese fuma Parliament
 - Chi fuma le Lucky Strike beve succo d'arancia
- } → **impossibile!** → **backtrack**

Ip. 2.2.2 : fatto #12, fatto #13

	1	2	3	4	5
Nazione	NO	UA	UK	ES	JP
Colore	G	B	R	A	V
Animale	VZ		LVZ		LVZ
Bevanda	Acqua	The	Latte	Succo	Caffè
Sigarette	Kool	C	OC	Lucky	Parliament

- Lo spagnolo è proprietario del cane
- Il giapponese fuma Parliament

Ip. 2.2.2: deduzioni

	1	2	3	4	5
Nazione	NO	UA	UK	ES	JP
Colore	G	B	R	A	V
Animale	VZ		LVZ		LVZ
Bevanda	Acqua	The	Latte	Succo	Caffè
Sigarette	Kool	Chester.	Old G.	Lucky	Parliament

Il giapponese fuma Parliament → { L'ucraino fuma Chesterfield
L'inglese fuma Old Gold

Ip. #2.2.2 : fatto #11 e fatto #14

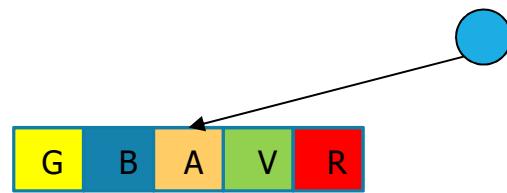


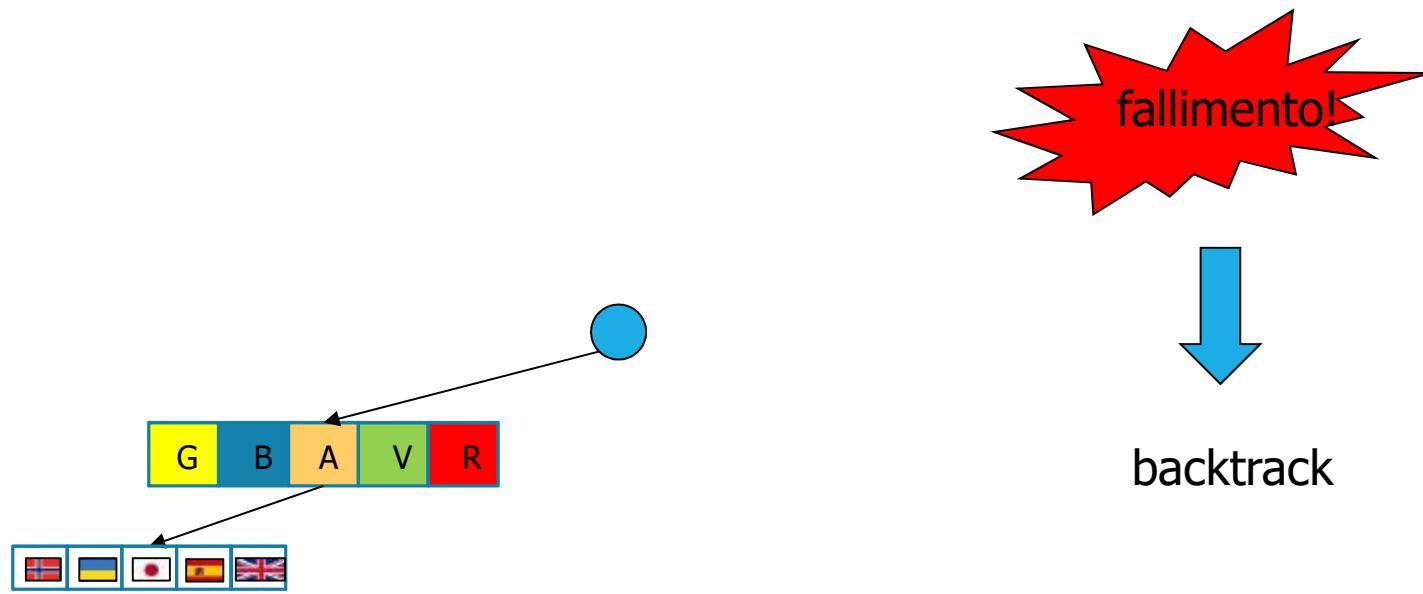
	1	2	3	4	5
Nazione	NO	UA	UK	ES	JP
Colore	G	B	R	A	V
Animale					
Bevanda	Acqua	The	Latte	Succo	Caffè
Sigarette	Kool	Chester.	Old G.	Lucky	Parliament

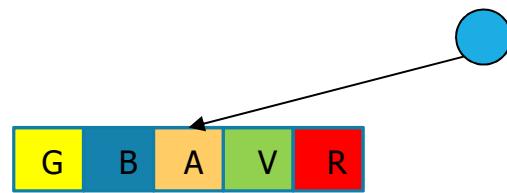
- Il fumatore di Old Gold possiede lumache
- L'uomo che fuma Chesterfield vive nella casa accanto all'uomo con la volpe

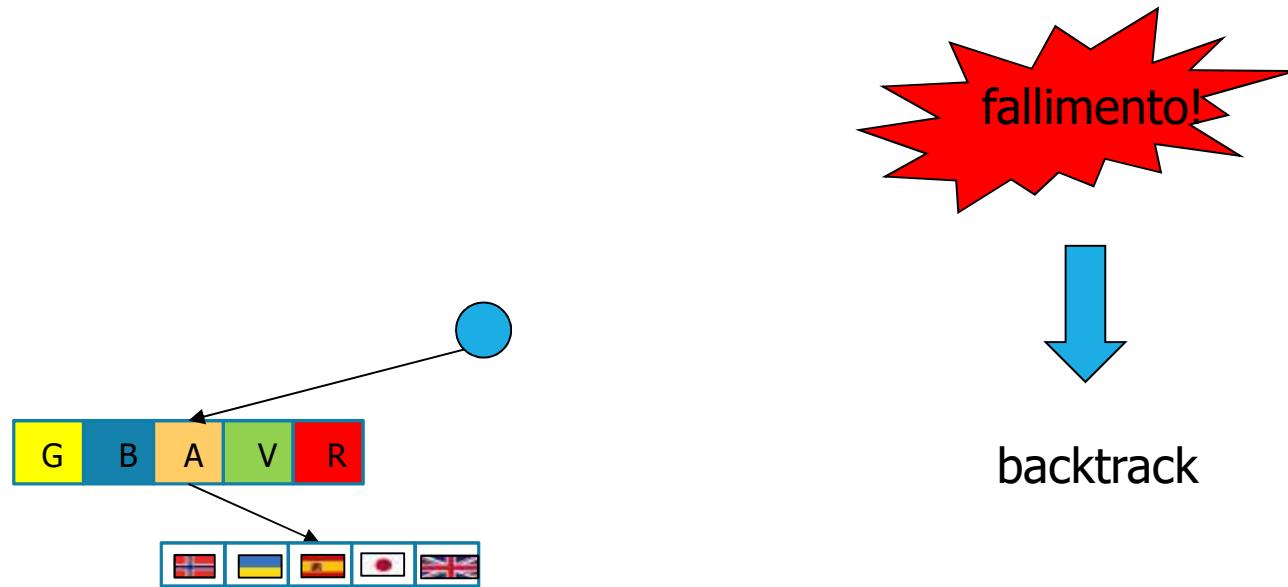
Osservazioni

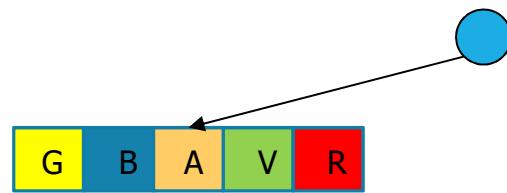
- Difficoltà:
 - non solo implicazioni dirette, ma anche esclusioni
 - struttura ad albero per tener traccia delle ipotesi
 - backtrack!

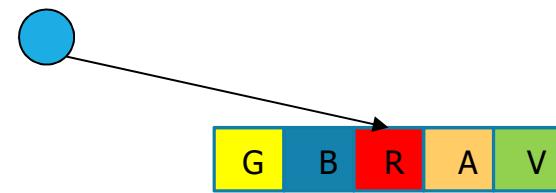


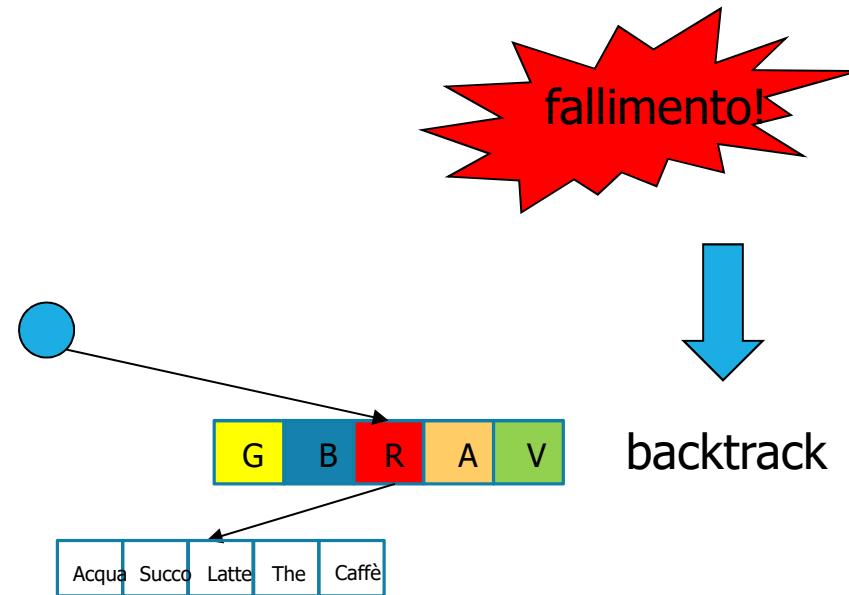


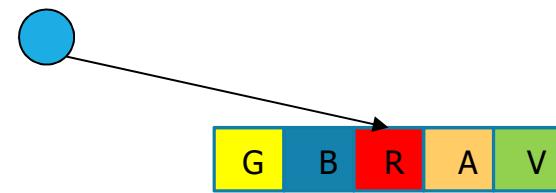


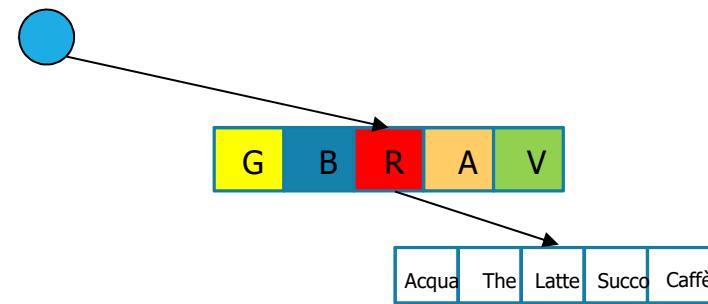


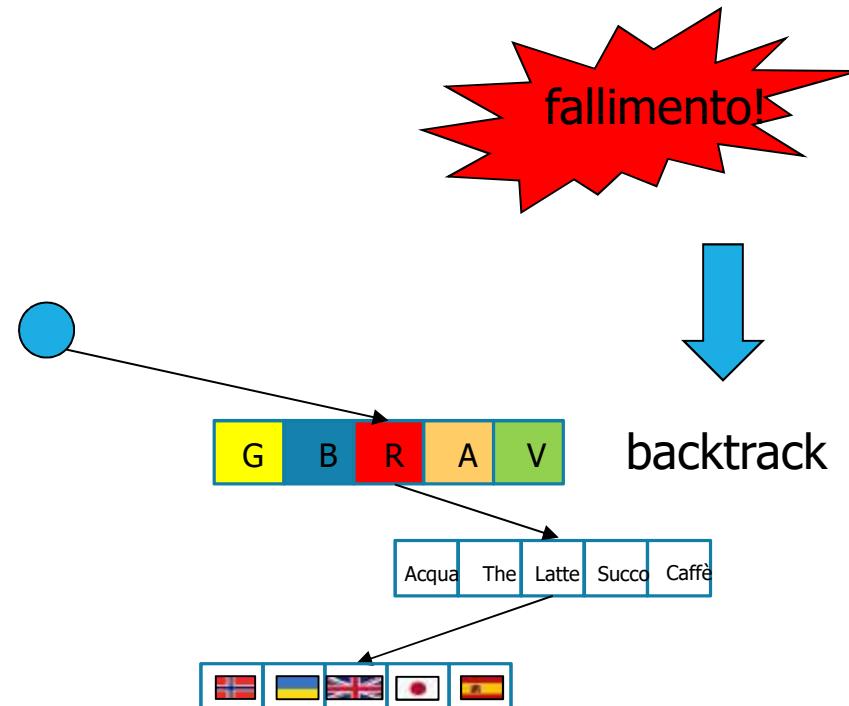


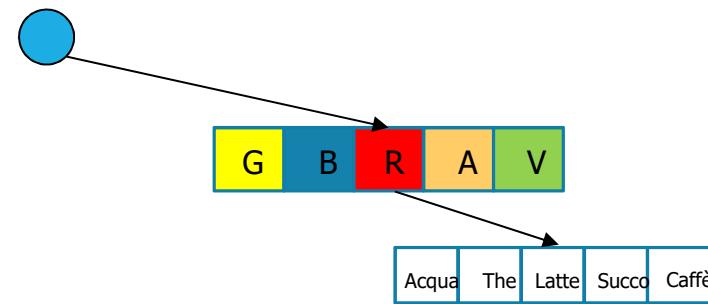




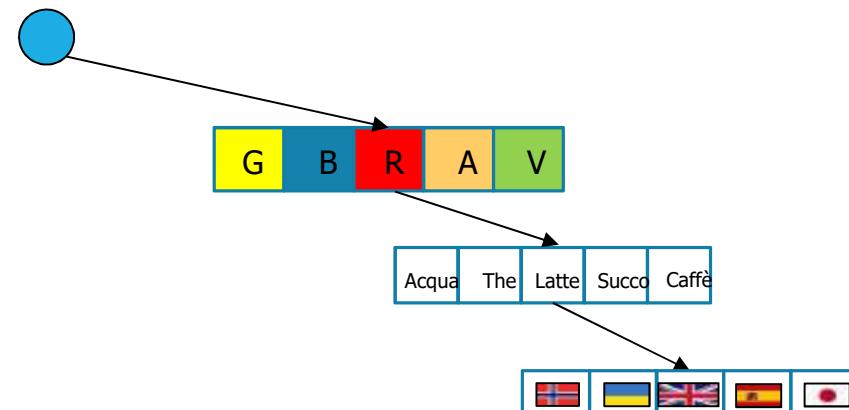








successo!



Disposizioni semplici

Per non generare elementi ripetuti:

- un vettore `mark` registra gli elementi già presi ($\text{mark}[i]==0 \Rightarrow$ elemento i -esimo non ancora preso, 1 altrimenti)
- la cardinalità di `mark` è pari al numero di elementi di `val` (tutti distinti, essendo un insieme)
- in fase di scelta l'elemento i -esimo viene preso solo se $\text{mark}[i]==0$, `mark[i]` viene assegnato con 1
- in fase di backtrack, `mark[i]` viene assegnato con 0
- `cnt` registra il numero di soluzioni.

```

int disp(int pos,int *val,int *sol,int *mark, int n, int k,int cnt){
    int i;
    if (pos >= k){ → terminazione
        for (i=0; i<k; i++) printf("%d ", sol[i]);
        printf("\n");
        return cnt+1; → iterazione sulle n scelte
    }
    for (i=0; i<n; i++){ → controllo ripetizione
        if (mark[i] == 0) {
            mark[i] = 1; → marcamento e scelta
            sol[pos] = val[i];
            cnt = disp(pos+1, val, sol, mark, n, k,cnt);
            mark[i] = 0; → ricorsione
        }
    } → smarcamento
    return cnt;
}

```

```

val = malloc(n * sizeof(int));
sol = malloc(k * sizeof(int));
mark = calloc(n, sizeof(int));

```

Esempio

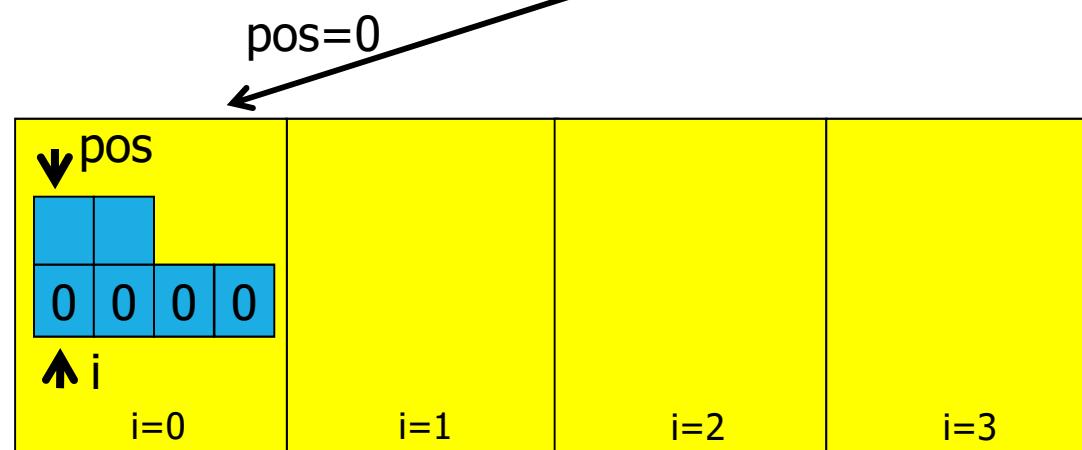
Quanti e quali sono i numeri di 2 cifre distinte che si possono scrivere utilizzando i numeri 4, 9, 1 e 0?

$$n = 4, k = 2, \text{val} = \{4, 9, 1, 0\}$$

$$D_{4,2} = 4!/(4-2)! = 4 \cdot 3 = 12$$

Soluzione:

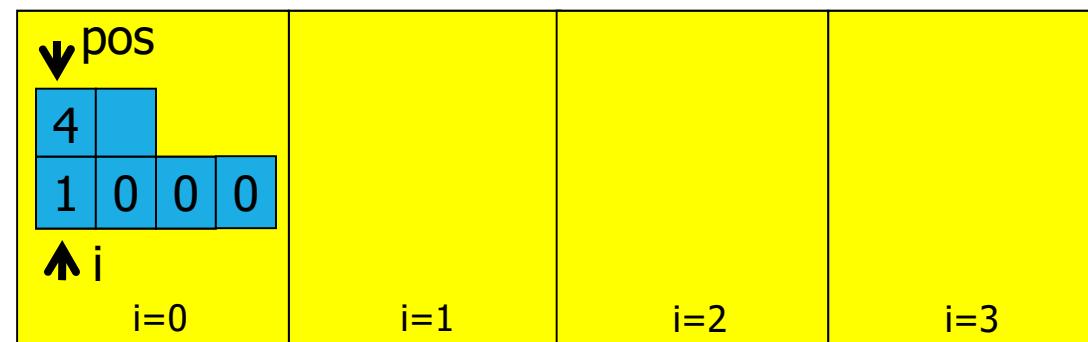
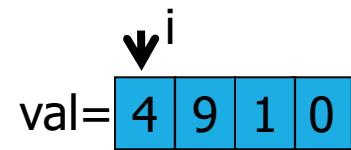
$$\{49, 41, 40, 94, 91, 90, 14, 19, 10, 04, 09, 01\}$$



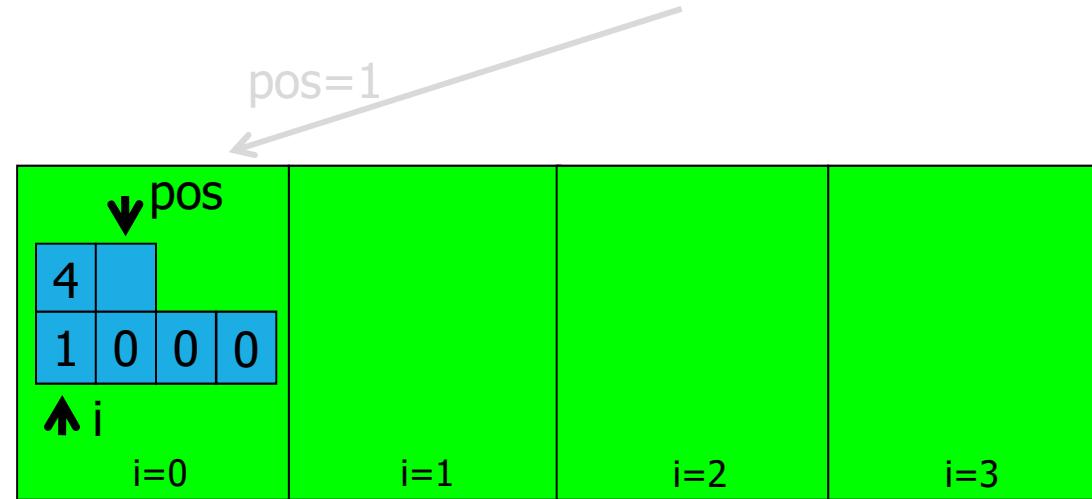
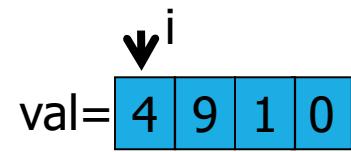
$\text{mark}[i] = 0$

$\text{sol}[\text{pos}] = \text{val}[\text{i}]$

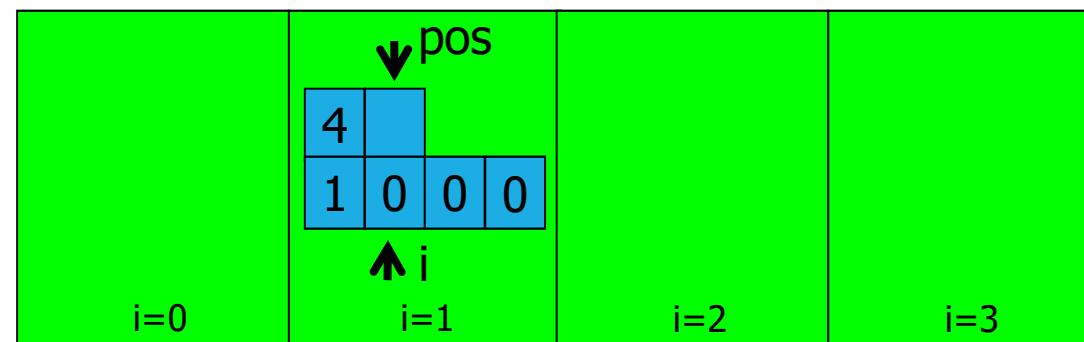
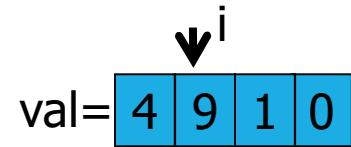
 $\text{mark}[i] = 1$



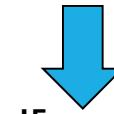
ricorsione
con pos=1



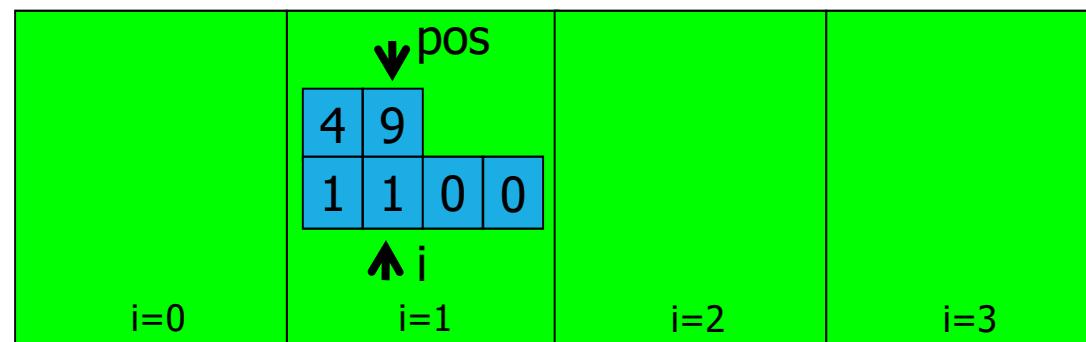
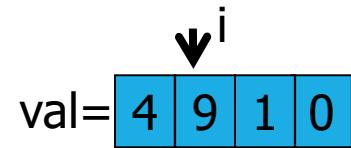
$mark[i] = 1$



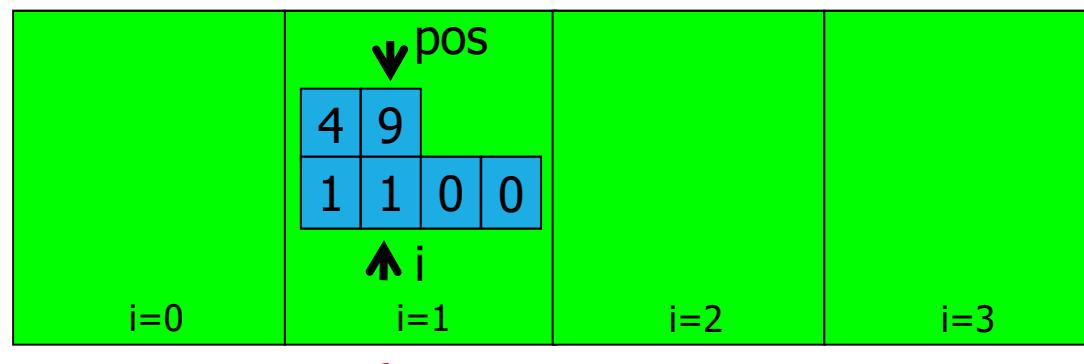
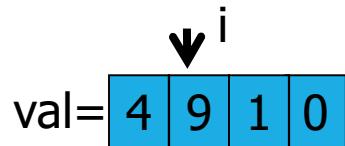
$\text{mark}[i] = 0$



$\text{sol}[\text{pos}] = \text{val}[i]$
 $\text{mark}[i] = 1$



ricorsione
con pos=2

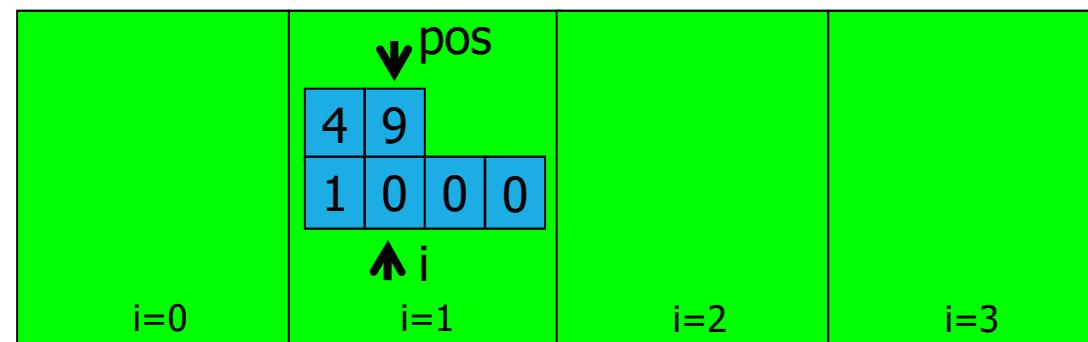
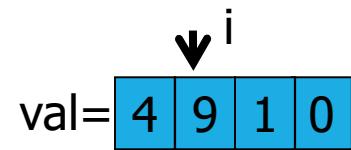


terminazione: visualizza, aggiorna cnt

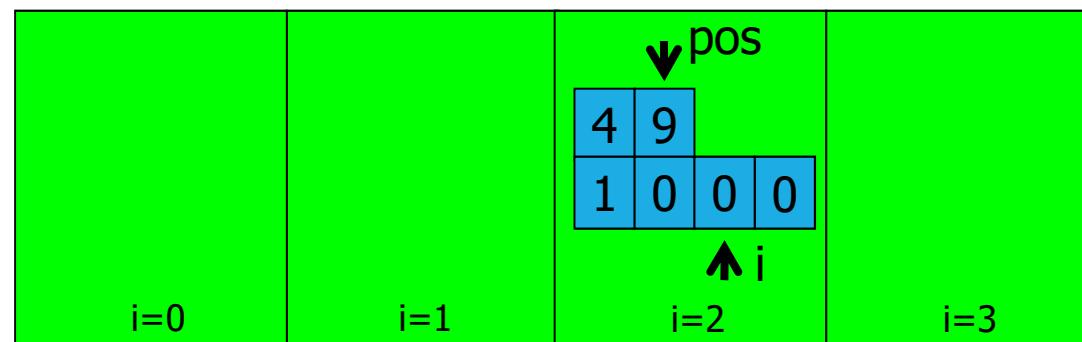
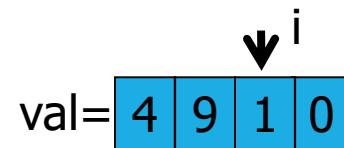
ritorna

sol =

4	9
---	---



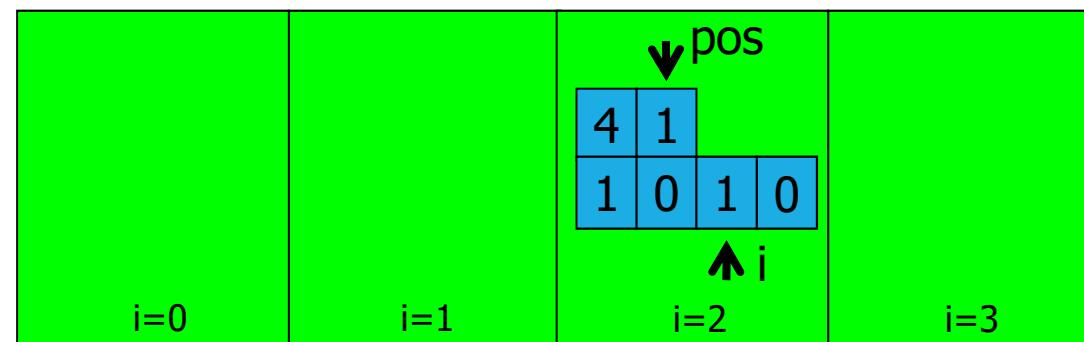
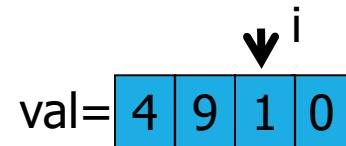
smarca mark[i]



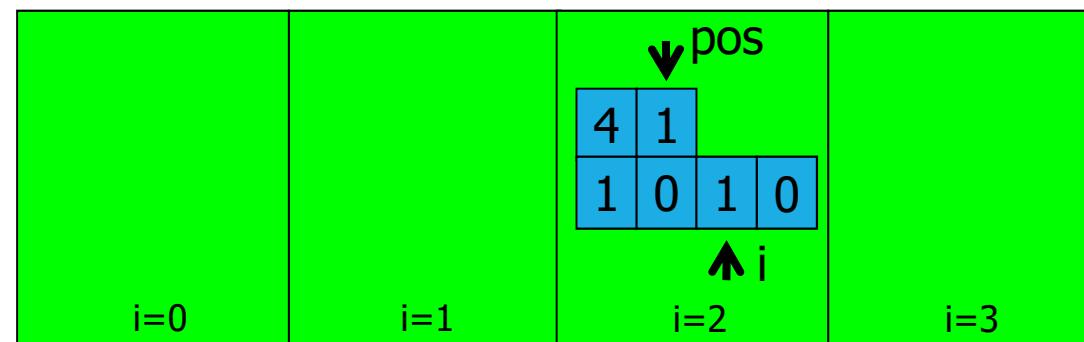
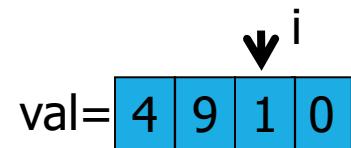
$mark[i] = 0$



$sol[pos] = val[i]$
 $mark[i] = 1$



ricorsione
con $pos=2$

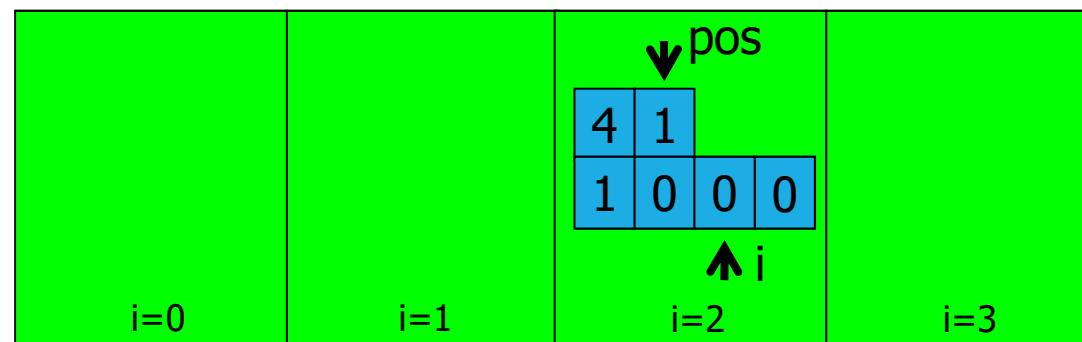
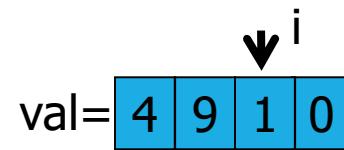


terminazione: visualizza, aggiorna cnt

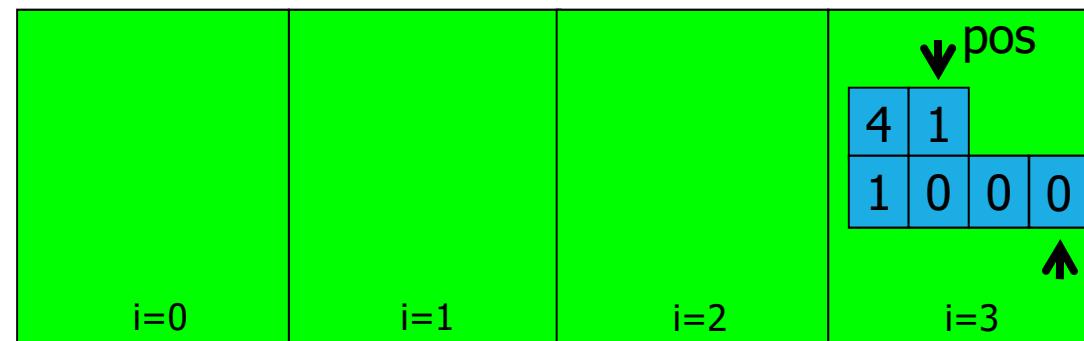
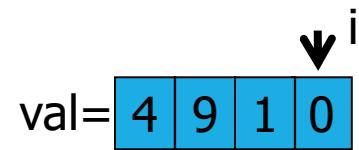
ritorna

$\downarrow sol$ =

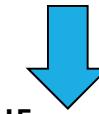
4	1
---	---



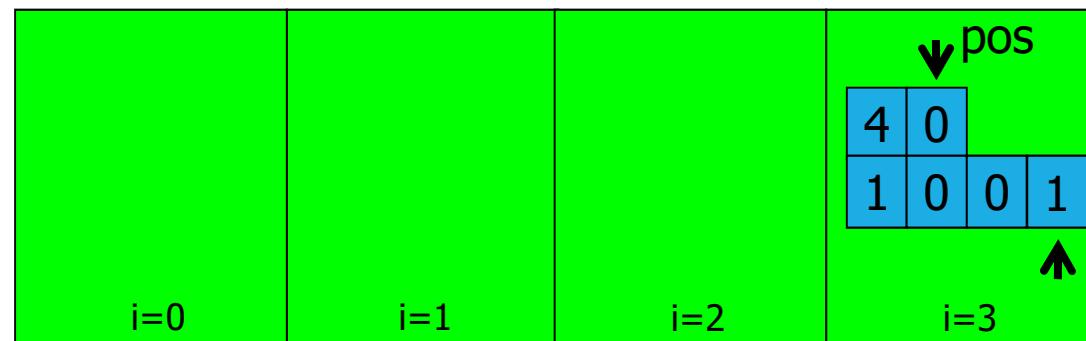
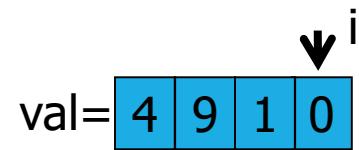
smarca mark[i]



$mark[i] = 0$

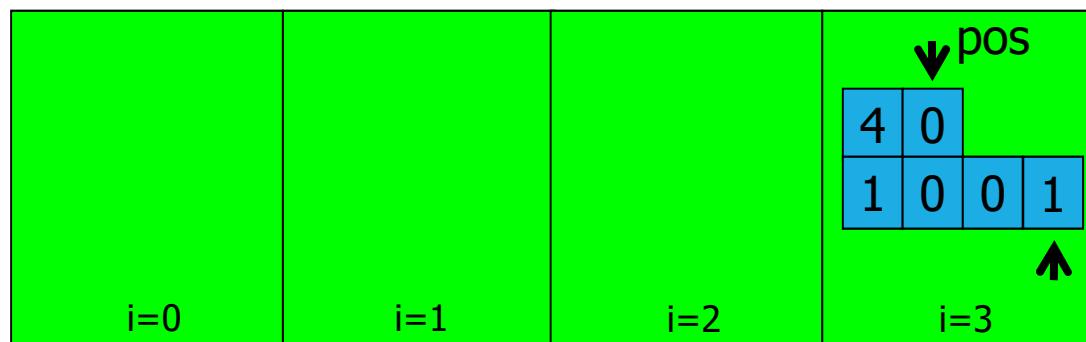


$sol[pos] = val[i]$
 $mark[i] = 1$



ricorsione
con $pos=2$

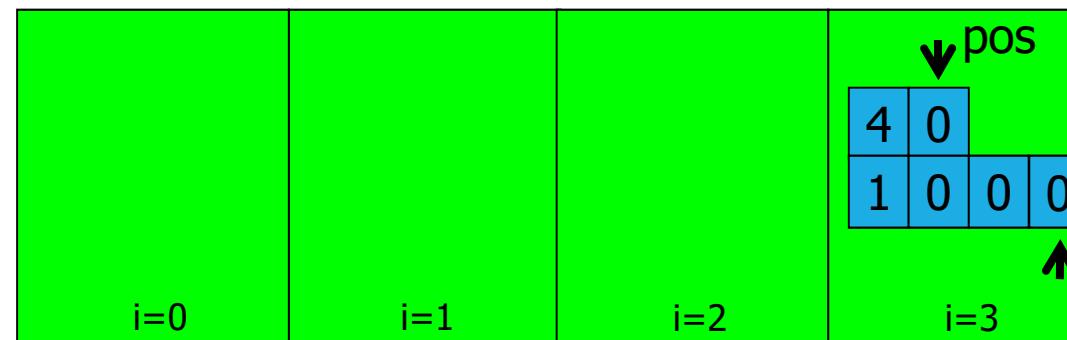
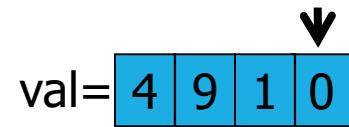
A handwritten note pointing to the $pos=2$ entry in the grid, indicating a recursive step or a specific position being processed.



terminazione: visualizza, aggiorna cnt

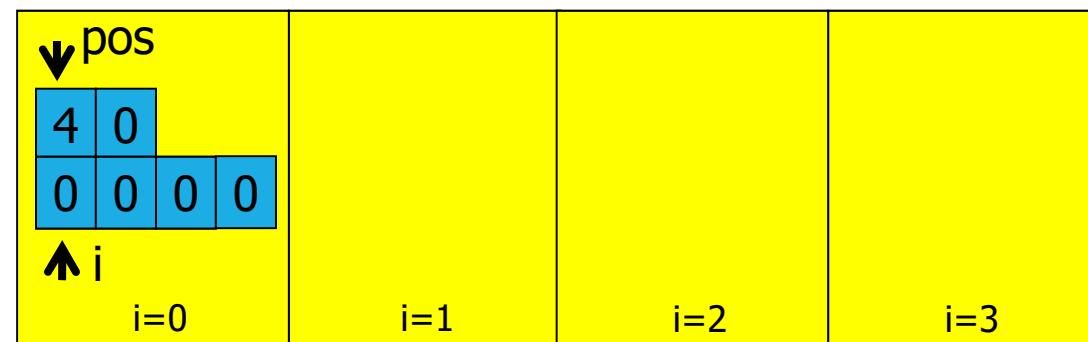
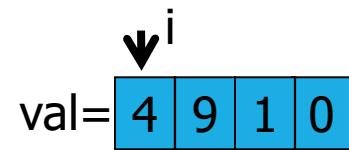


ritorna

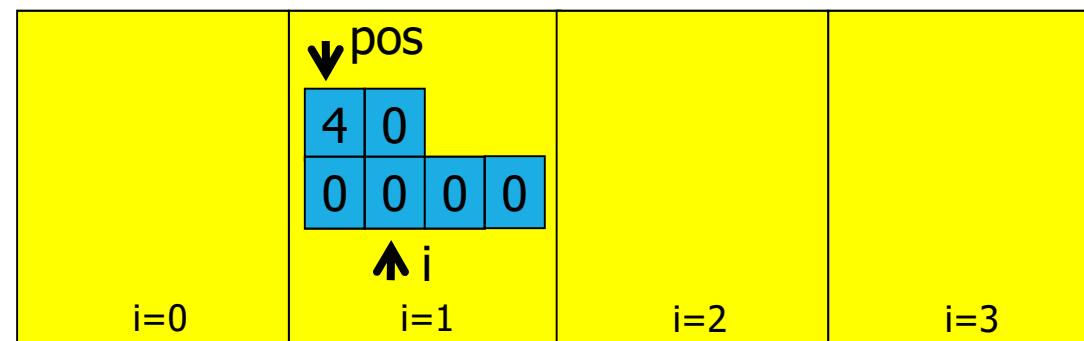
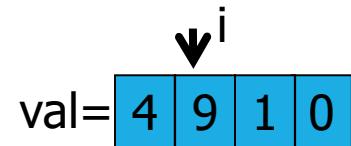


smarca mark[i]

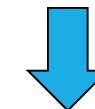
ciclo for terminato, ritorna



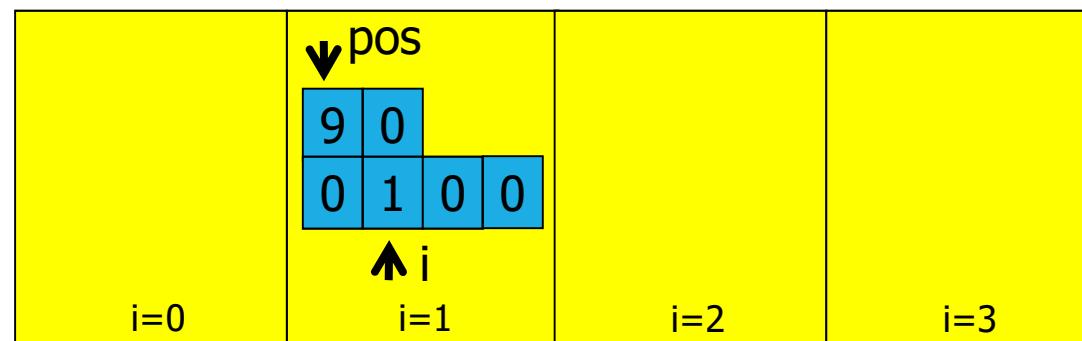
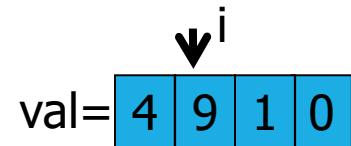
smarca mark[i]



$\text{mark}[i] = 0$



$\text{sol}[\text{pos}] = \text{val}[i]$
 $\text{mark}[i] = 1$



ricorsione
con $\text{pos}=1$

etc. etc.

Disposizioni ripetute

- Ogni elemento può essere ripetuto fino a k volte.
- Non c'è un vincolo imposto da n su k
- Per ognuna delle posizioni si enumerano esaustivamente tutte le scelte possibili
- cnt registra il numero di soluzioni.

```

int disp_rip(int pos,int *val,int *sol,int n,int k,int cnt){
    int i;
    if (pos >= k) { → terminazione
        for (i=0; i<k; i++)
            printf("%d ", sol[i]);
        printf("\n");
        return cnt+1;
    }
    for (i = 0; i < n; i++) { → iterazione sulle n scelte
        sol[pos] = val[i];
        cnt = disp_rip(pos+1, val, sol, n, k, cnt);
    }
    return cnt;
}

```

Permutazioni semplici

Per non generare elementi ripetuti:

- un vettore `mark` registra gli elementi già presi ($\text{mark}[i]==0 \Rightarrow$ elemento i-esimo non ancora preso, 1 altrimenti)
- la cardinalità di `mark` è pari al numero di elementi di `val` (tutti distinti, essendo un insieme)
- in fase di scelta l'elemento i-esimo viene preso solo se $\text{mark}[i]==0$, $\text{mark}[i]$ viene assegnato con 1
- in fase di backtrack, `mark[i]` viene assegnato con 0
- `cnt` registra il numero di soluzioni.

```

int perm(int pos,int *val,int *sol,int *mark, int n, int cnt){
    int i;
    if (pos >= n){ terminazione
        for (i=0; i<n; i++) printf("%d ", sol[i]);
        printf("\n");
        return cnt+1;
    }
    for (i=0; i<n; i++) {
        if (mark[i] == 0) { controllo ripetizione
            mark[i] = 1;
            sol[pos] = val[i]; marcamento e scelta
            cnt = perm(pos+1, val, sol, mark, n, cnt);
            mark[i] = 0; ricorsione
        }
    }
    return cnt; smarcamento
}

```

val = malloc(n * sizeof(int));
sol = malloc(n * sizeof(int));
mark = calloc(n, sizeof(int));

Esempio

Dato un insieme val di n interi, generare tutte le permutazioni di questi valori.

Il numero di permutazioni è $n!$.

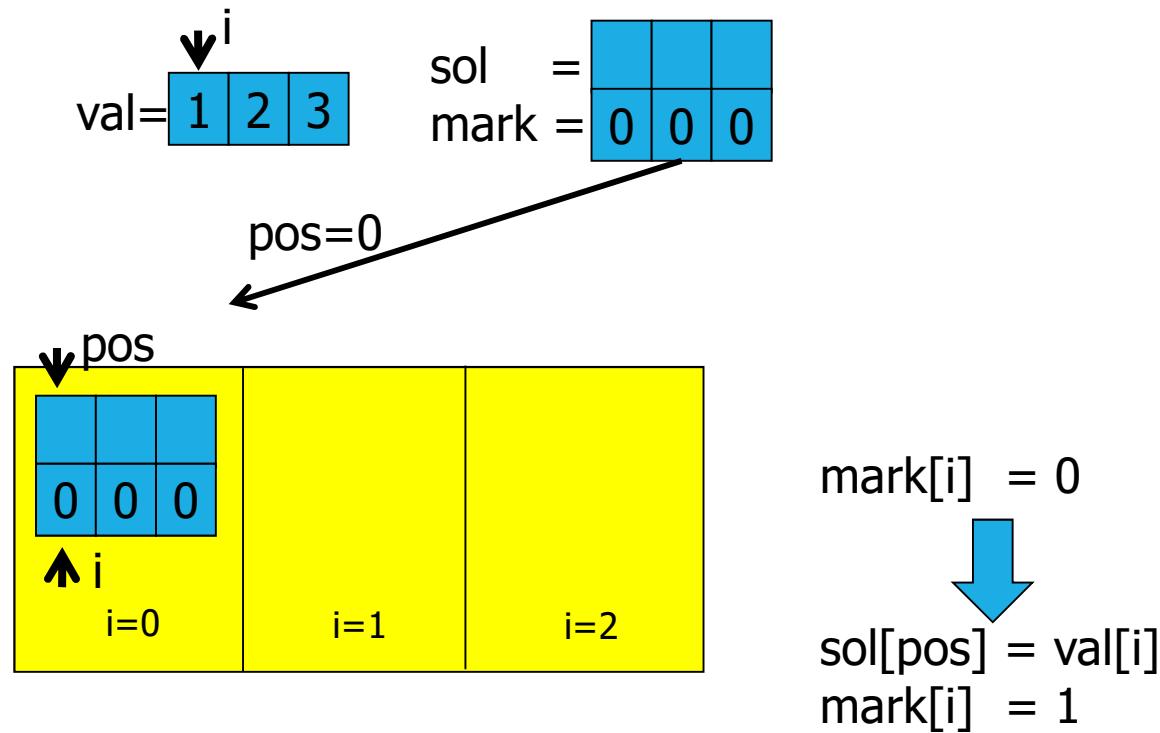
Esempio

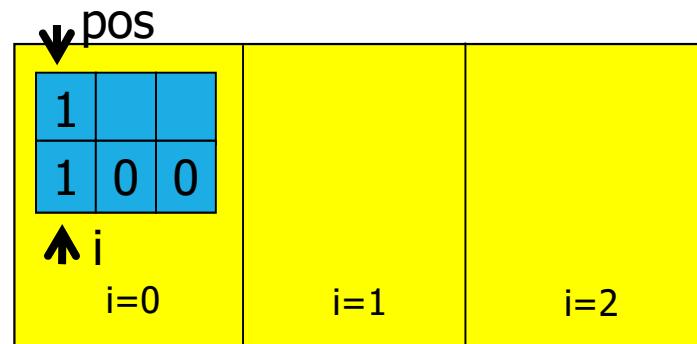
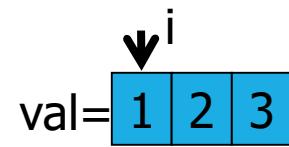
$\text{val} = \{1, 2, 3\}$ $n = 3$

$n! = 6$.

Le 6 permutazioni sono:

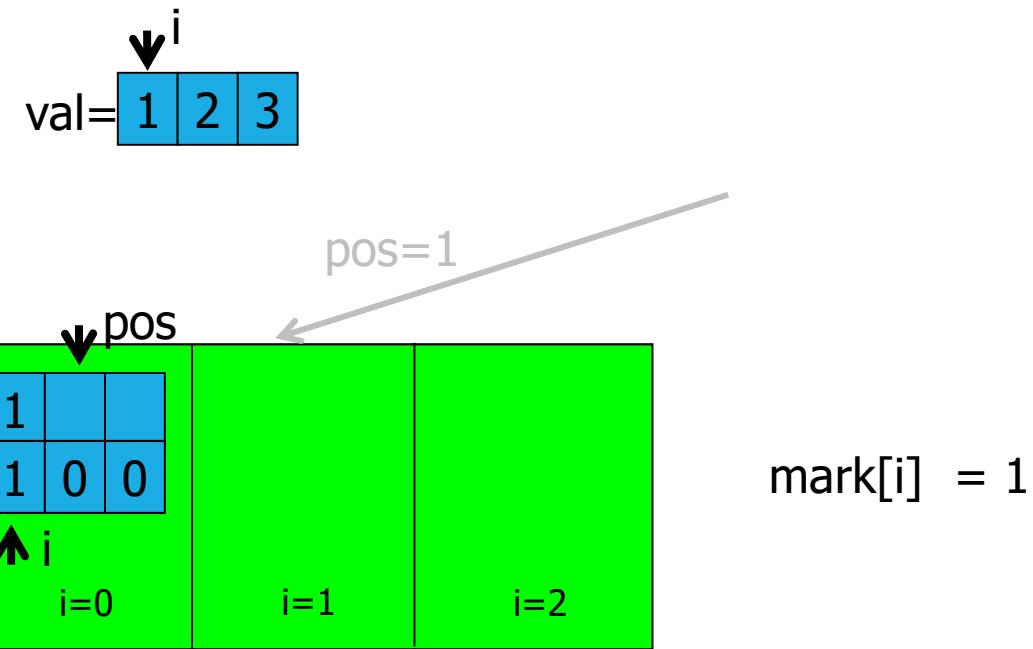
$\{1,2,3\} \{1,3,2\} \{2,1,3\} \{2,3,1\} \{3,1,2\} \{3,2,1\}$

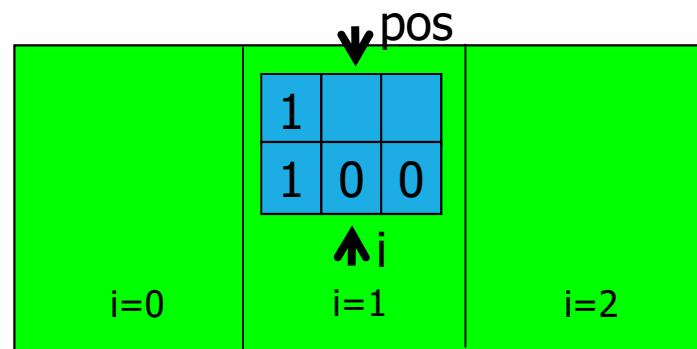
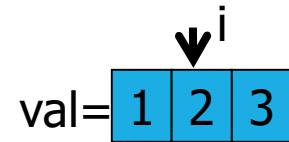




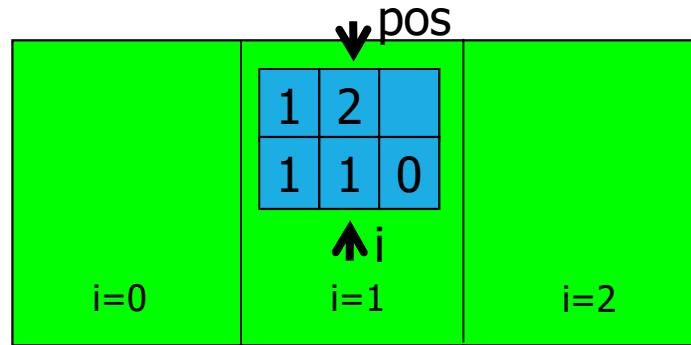
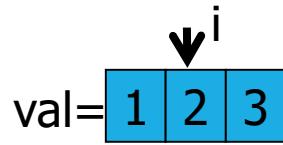
ricorsione
con $\text{pos}=1$

A diagram showing a recursive call with $\text{pos}=1$. An arrow points from the text "ricorsione con $\text{pos}=1$ " to the $\text{pos}=1$ entry in the grid.



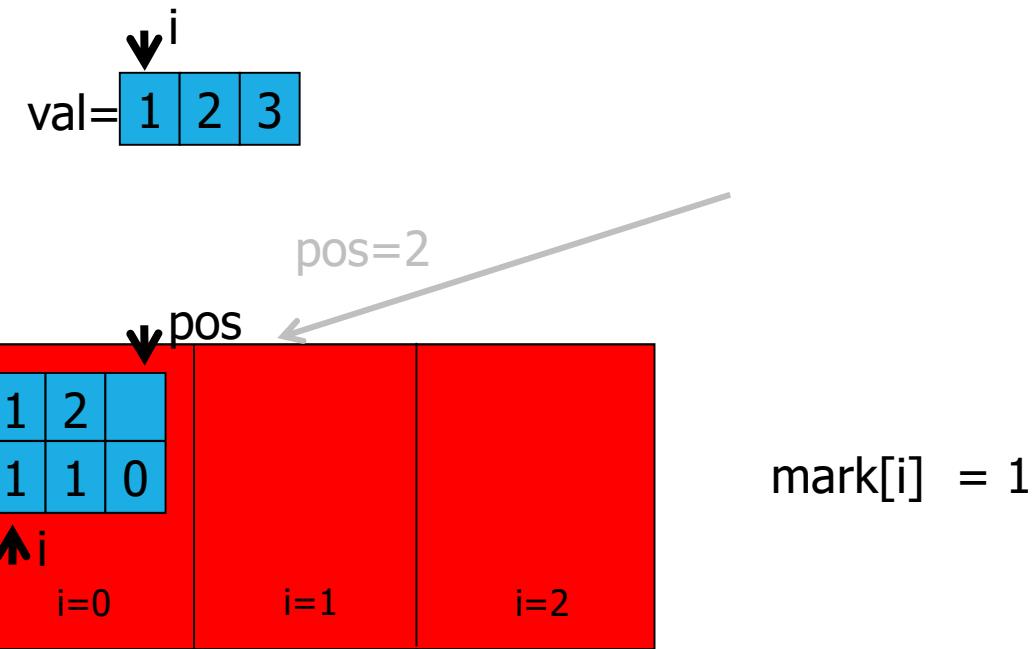


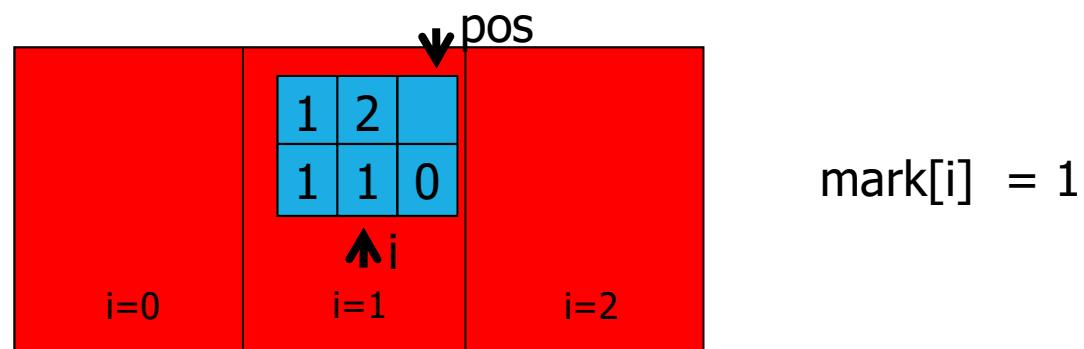
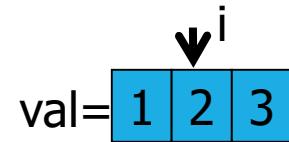
$\text{mark}[i] = 0$
 \downarrow
 $\text{sol}[\text{pos}] = \text{val}[i]$
 $\text{mark}[i] = 1$

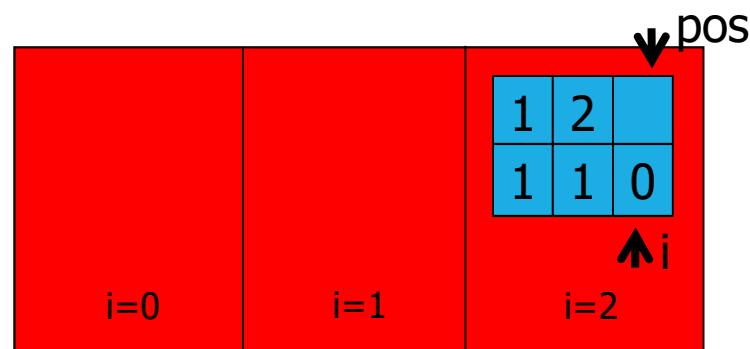
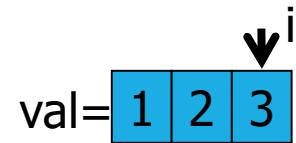


ricorsione
con $pos=2$

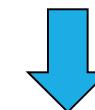
$mark[i] = 0$
 \downarrow
 $sol[pos] = val[i]$
 $mark[i] = 1$



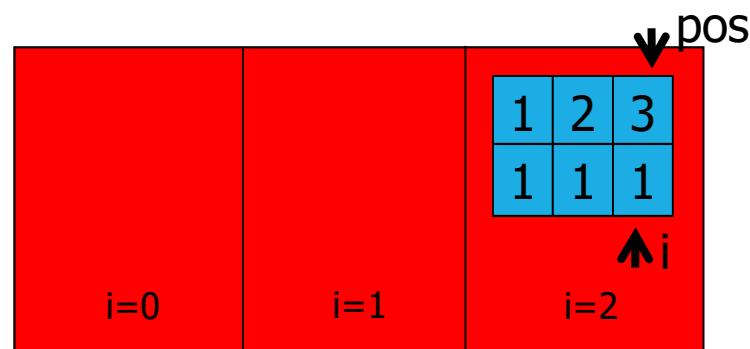
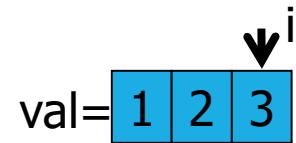




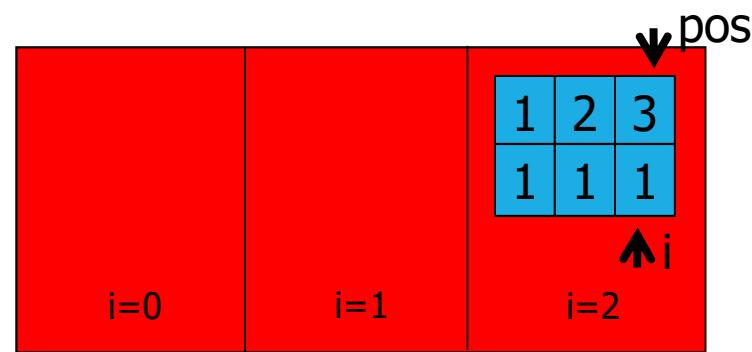
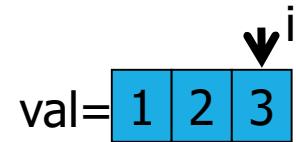
$\text{mark}[i] = 0$



$\text{sol}[\text{pos}] = \text{val}[i]$
 $\text{mark}[i] = 1$

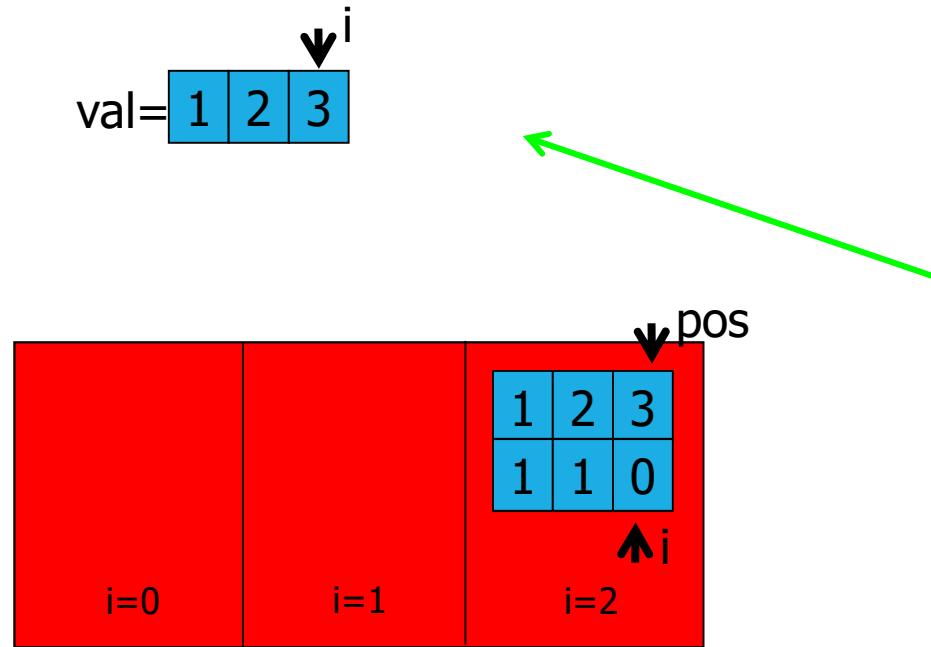


ricorsione
con $\text{pos}=3$

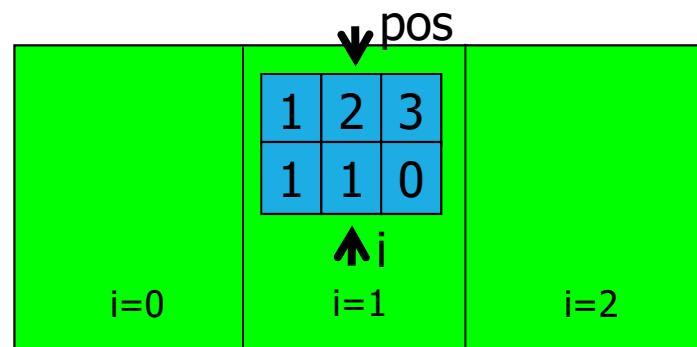
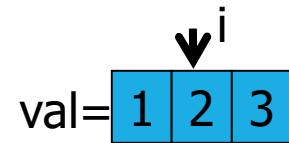


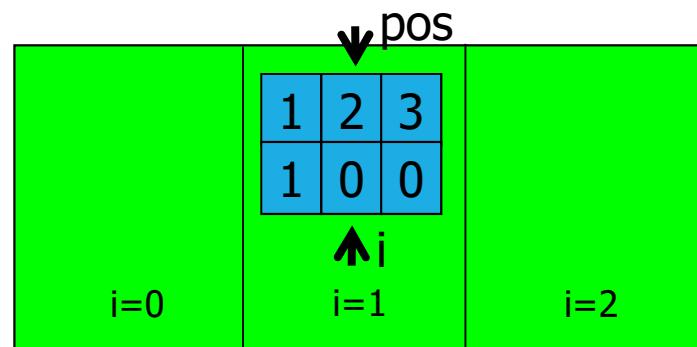
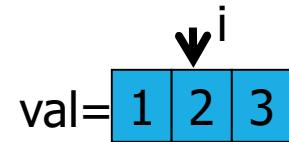
terminazione: visualizza, aggiorna cnt
ritorna

$\text{sol} = \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline\end{array}$

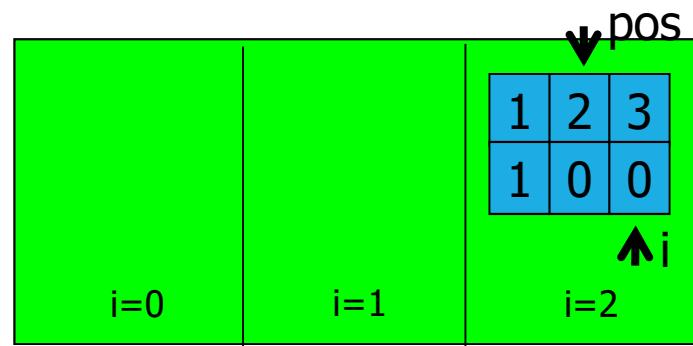
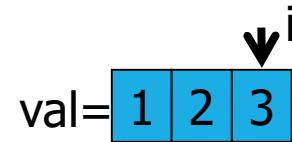


smarca `mark[i]`
 ciclo for terminato, ritorna

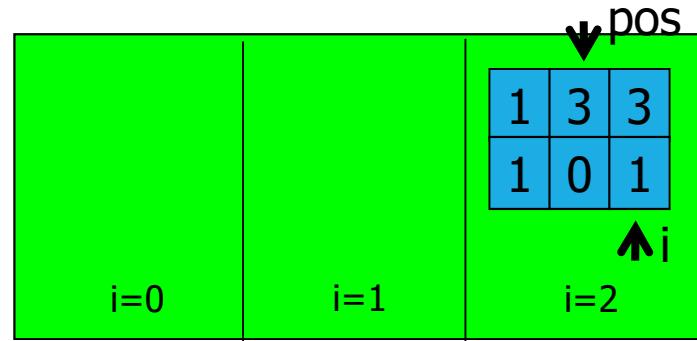
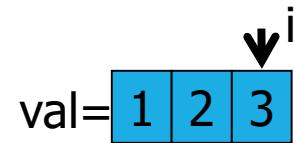




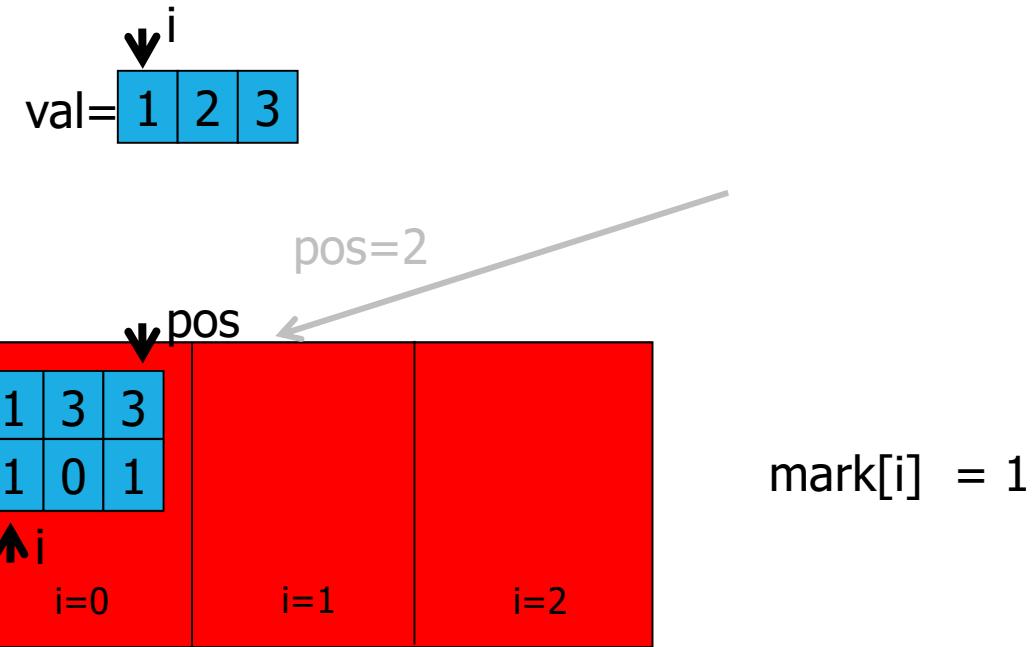
smarca $\text{mark}[i]$

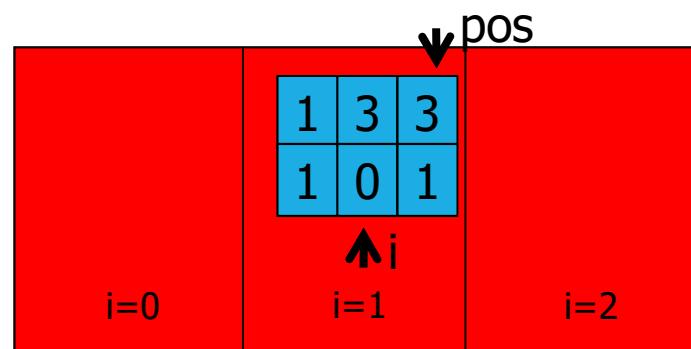
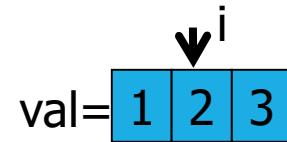


$\text{mark}[i] = 0$
 \downarrow
 $\text{sol}[pos] = \text{val}[i]$
 $\text{mark}[i] = 1$

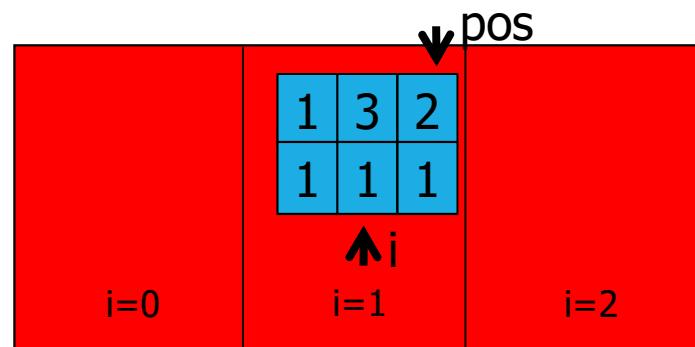
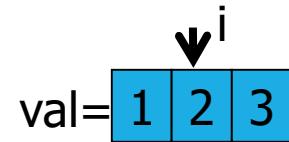


ricorsione con pos=

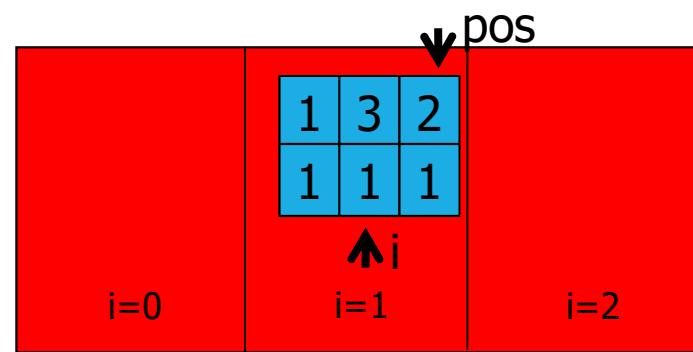
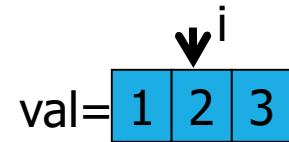




$\text{mark}[i] = 0$
 \downarrow
 $\text{sol}[\text{pos}] = \text{val}[i]$
 $\text{mark}[i] = 1$

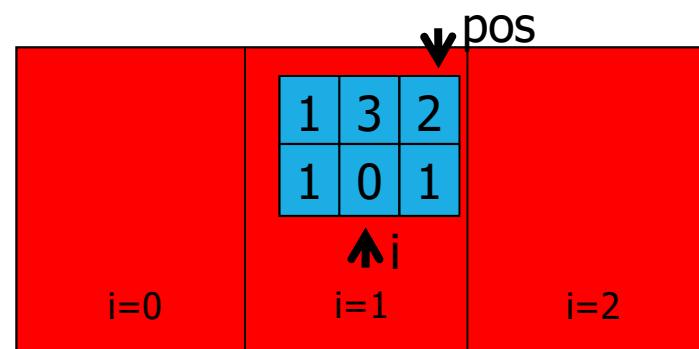
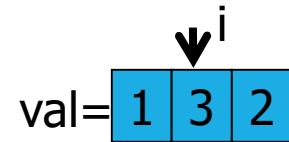


ricorsione
con pos=3

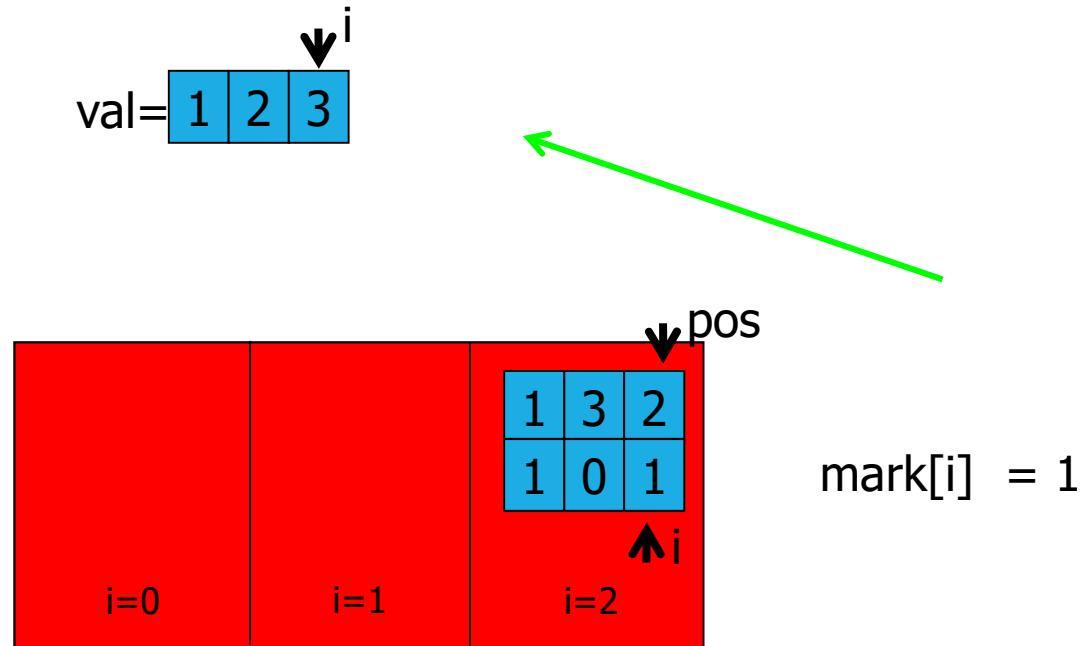


terminazione: visualizza, aggiorna cnt
ritorna

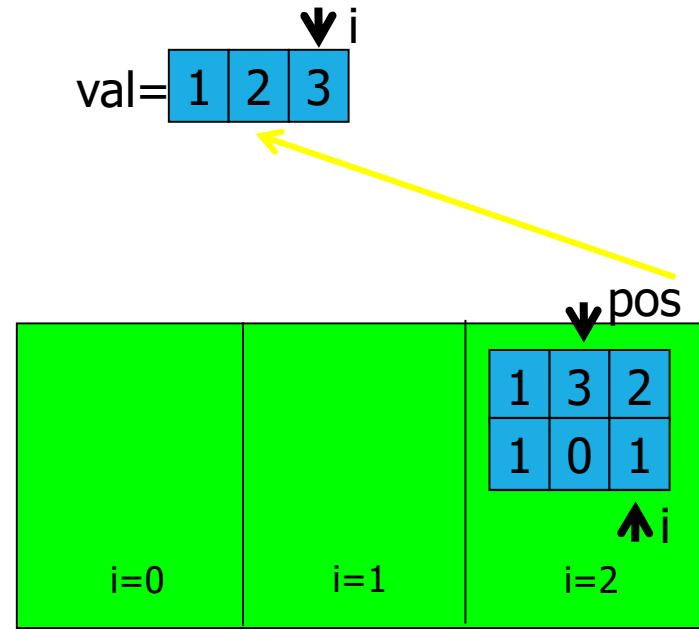
$\text{sol} = \begin{array}{|c|c|c|} \hline 1 & 3 & 2 \\ \hline \end{array}$



smarca $\text{mark}[i]$



ciclo for terminato, ritorna



ciclo for terminato, ritorna

etc. etc.

Esempio: anagrammi con ripetizioni

Si legga una stringa, se ne generino tutti gli anagrammi.

- Se tutte le lettere della stringa sono distinte, gli anagrammi sono distinti.
- Se le lettere della stringa si ripetono, anche gli anagrammi si ripetono.
- Ipotizzando che le lettere ripetute siano in qualche modo distinguibili, il problema si riconduce alle permutazioni semplici.
- Se la stringa è lunga n , il numero di anagrammi è $n!$

Esempio

stringa = ORO, n = 3

$$n! = 6.$$

I 6 anagrammi con ripetizione sono:

{ ORO, OOR, ROO, ROO, OOR, ORO }

```
int anagr(int pos, char *sol, char *val int *mark, int n, int cnt) {  
    int i;  
    if (pos >= n) {  
        sol[pos] = '\0';  
        printf("%s\n", sol);  
        return cnt+1;  
    }  
    for (i=0; i<n; i++)  
        if (mark[i] == 0) {  
            mark[i] = 1;  
            sol[pos] = val[i];  
            cnt = anagrammi(pos+1,sol,val,mark,n,cnt);  
            mark[i] = 0;  
        }  
    return cnt;  
}
```



02anagrammi_con Ripetizioni

Permutazioni ripetute

Si procede in maniera analoga alle permutazioni semplici, con le seguenti variazioni:

- n è la cardinalità del multiinsieme
- si memorizzano nel vettore `dist_val` di n_dist celle gli elementi distinti del multiinsieme
 - si ordina il vettore `val` con un algoritmo $O(n \log n)$
 - si «compatta» `val` eliminando gli elementi duplicati e lo si memorizza in `dist_val` con un algoritmo $O(n)$

- il vettore `mark` di `n_dist` elementi registra all'inizio il numero di occorrenze degli elementi distinti del multiset
- l'elemento `dist_val[i]` viene preso se `mark[i]>0`, `mark[i]` viene decrementato
- al ritorno dalla ricorsione `mark[i]` viene incrementato
- `cnt` registra il numero di soluzioni.

```

int perm_r(int pos, int *dist_val, int *sol, int *mark,
           int n, int n_dist, int cnt) {
    int i;
    if (pos >= n) { terminazione
        for (i=0; i<n; i++)
            printf("%d ", sol[i]);
        printf("\n");
        return cnt+1;
    }
    for (i=0; i<n_dist; i++) { iterazione sulle n_dist scelte
        if (mark[i] > 0) { controllo occorrenze
            mark[i]--;
            marcamento e scelta
            sol[pos] = dist_val[i];
            cnt=perm_r(pos+1,dist_val,sol,mark,n, n_dist,cnt);
            mark[i]++;
        }
        smarcamento
    }
    return cnt;
}
val = malloc(n*sizeof(int));
dist_val = malloc(n*sizeof(int));
sol = malloc(n*sizeof(int));

```

Esempio: anagrammi distinti

Data una stringa con lettere eventualmente ripetute, generare tutti i suoi anagrammi distinti.

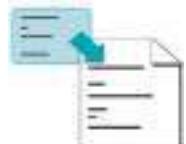
stringa ORO n = 3

dist_val = { O, R }, n_dist = 2

$$P^{(2)}_3 = 3! / 2! = 3$$

Soluzione

{ OOR, ORO, ROO }



03anagrammi_distinti

stringa ORO

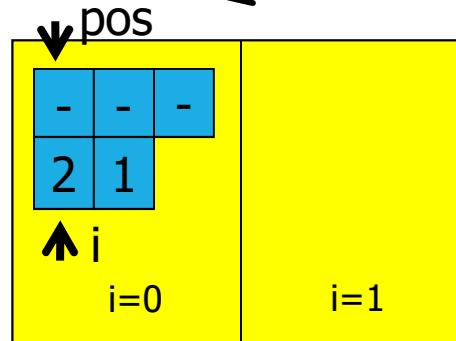
dist_val =

O R

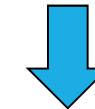
sol
mark =

-	-	-
2	1	

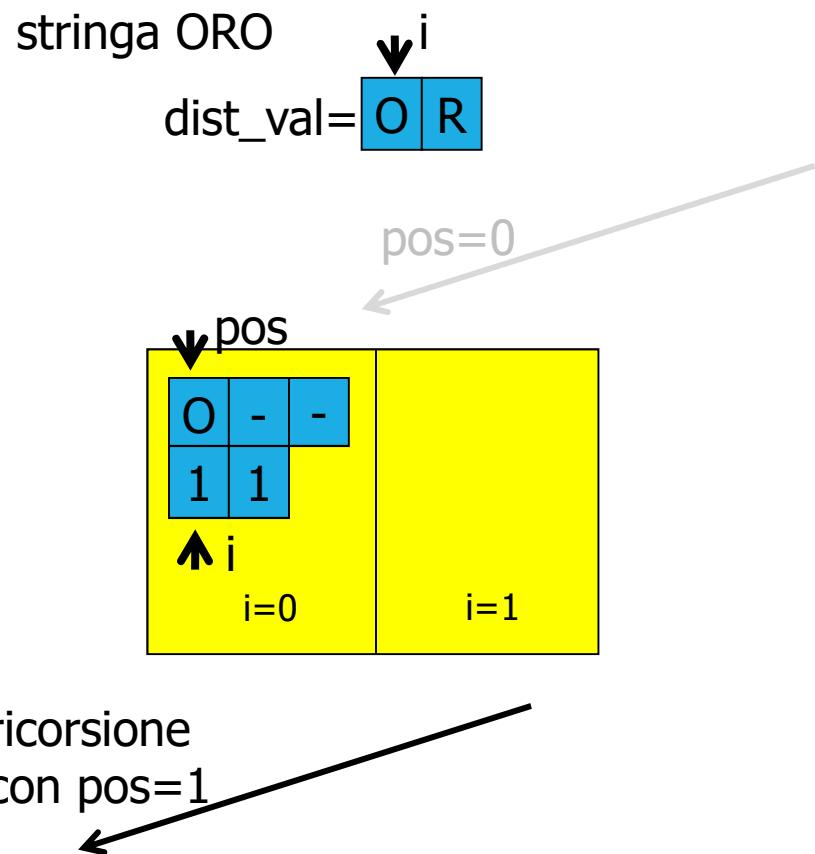
pos=0



$\text{mark}[i] > 0$

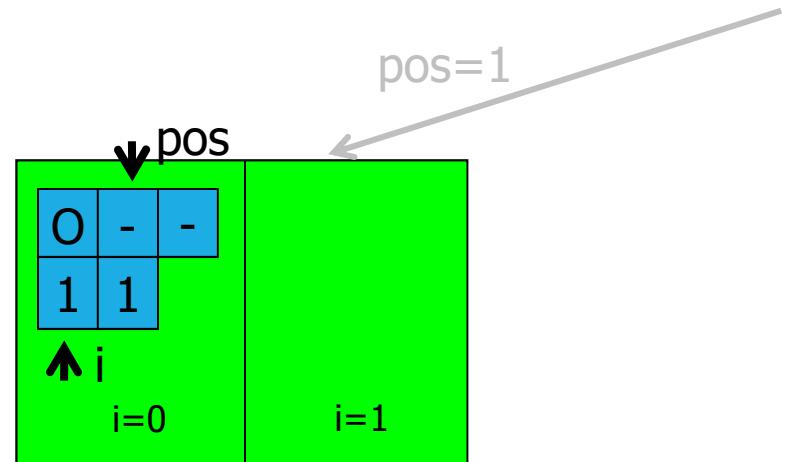


$\text{sol}[\text{pos}] = \text{val}[i]$
 $\text{mark}[i]--$

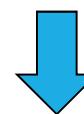


stringa ORO

dist_val= 

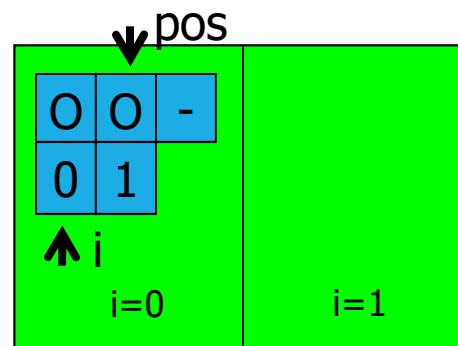


mark[i] > 0

 sol[pos] = val[i]
mark[i]--

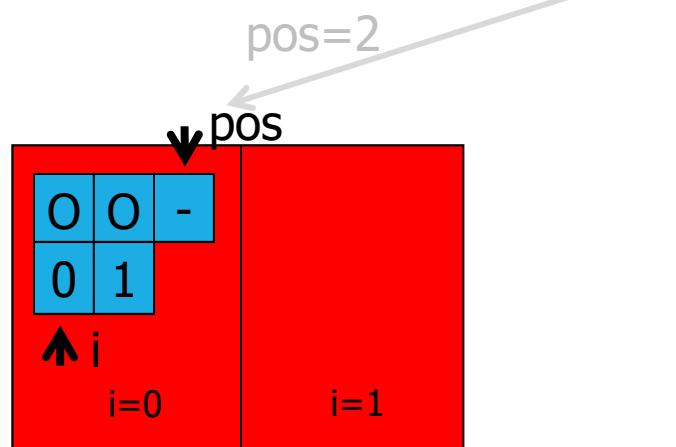
stringa ORO

dist_val= 



ricorsione
con pos=2

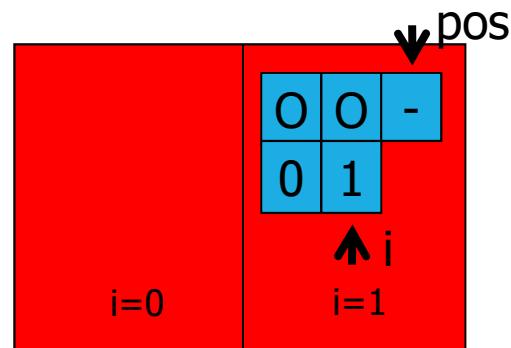
stringa ORO
dist_val= 



mark[i] non è > 0

stringa ORO

dist_val= 



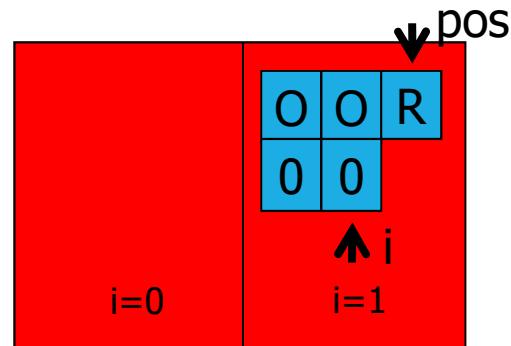
$\text{mark}[i] > 0$



$\text{sol}[\text{pos}] = \text{val}[i]$
 $\text{mark}[i]--$

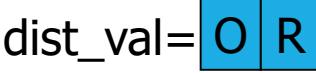
stringa ORO

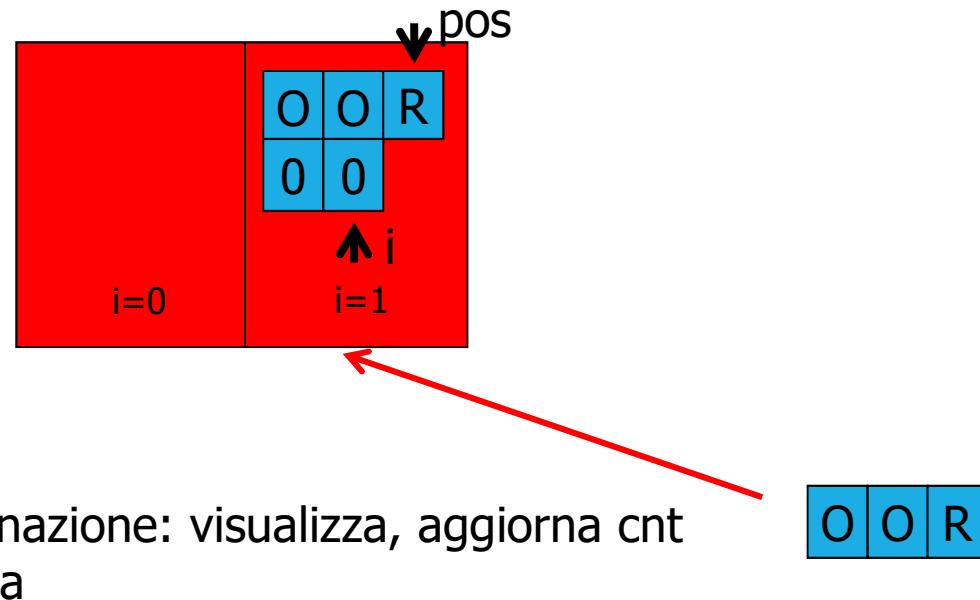
dist_val= 



ricorsione
con pos=3

stringa ORO

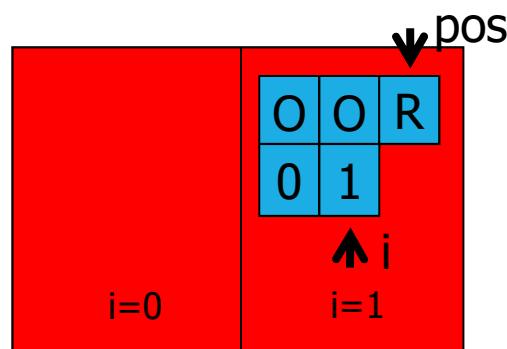
dist_val= 



stringa ORO

dist_val= 

i



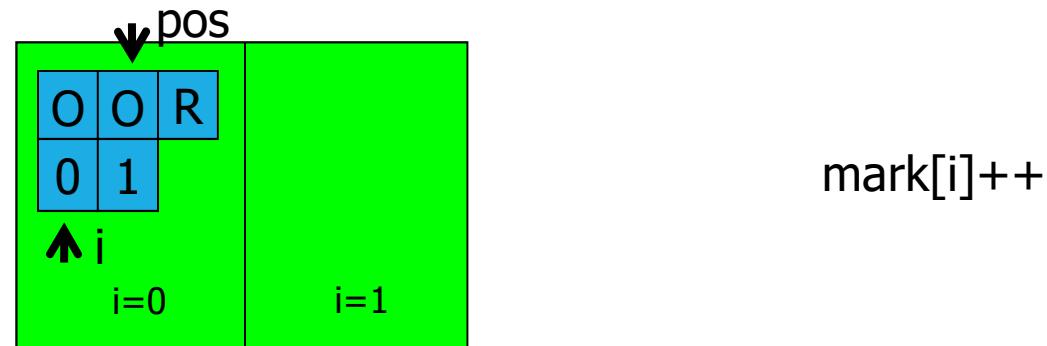
mark[i]++

backtrack

ciclo for terminato, ritorna

stringa ORO

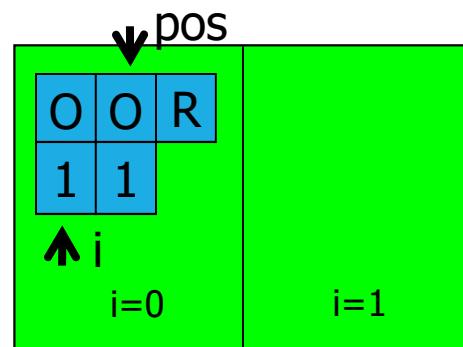
↓
dist_val= O R



backtrack

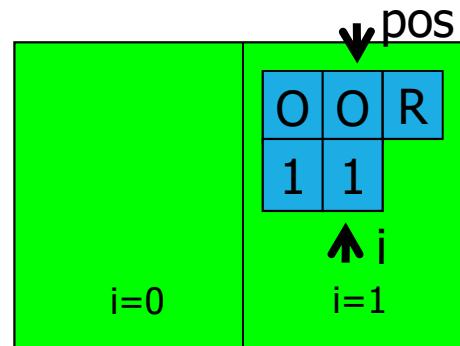
stringa ORO

dist_val= 



stringa ORO

dist_val= 



$\text{mark}[i] > 0$

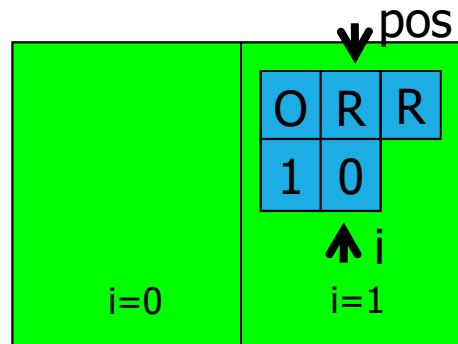


$\text{sol[pos]} = \text{val}[i]$
 $\text{mark}[i]--$

stringa ORO

dist_val= 

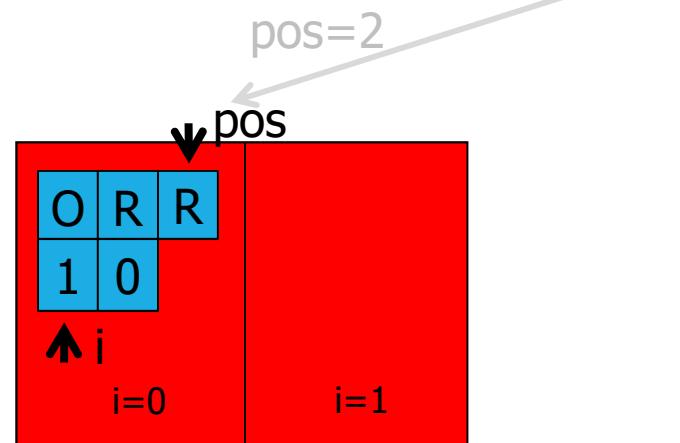
i



ricorsione
con $pos=2$

stringa ORO

dist_val= 



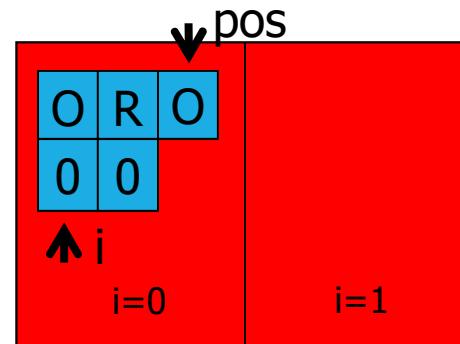
mark[i] > 0



sol[pos] = val[i]
mark[i]--

stringa ORO

dist_val= 

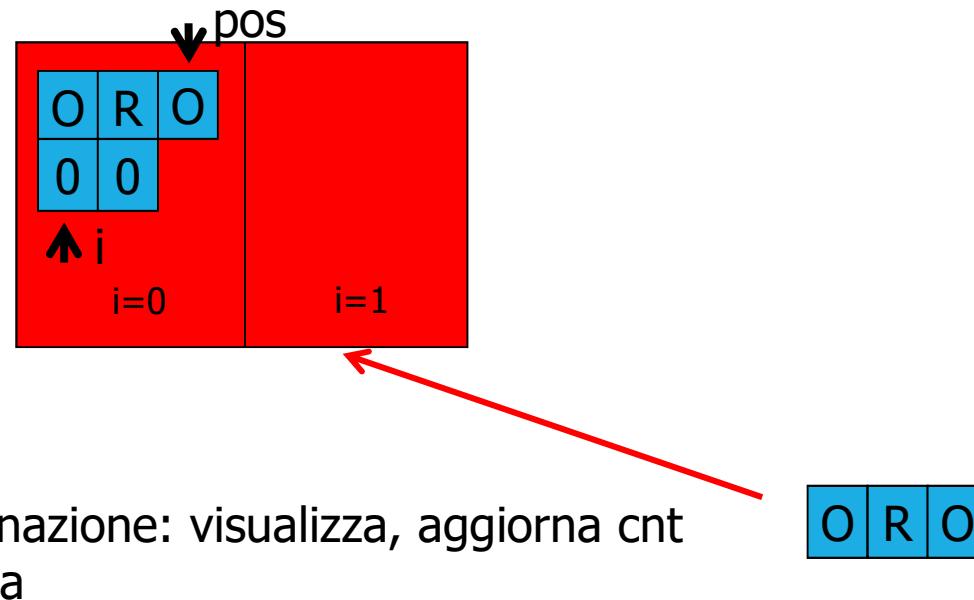


ricorsione
con pos=3

stringa ORO

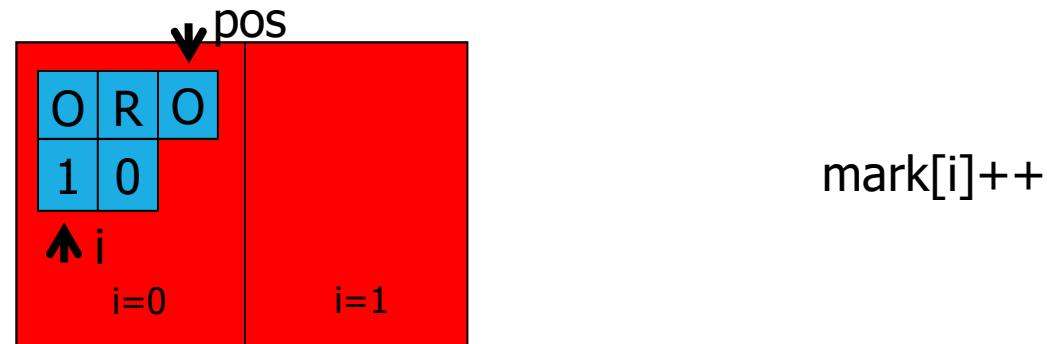
dist_val= 

\downarrow^i



stringa ORO

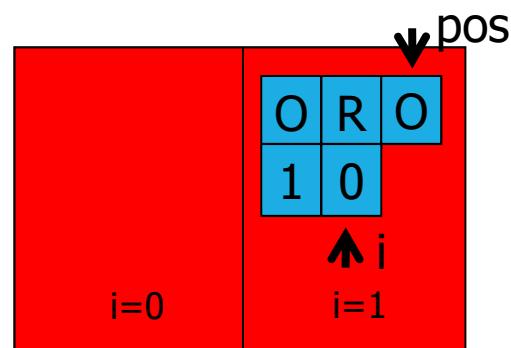
↓
dist_val= O R



backtrack

stringa ORO

dist_val = O R

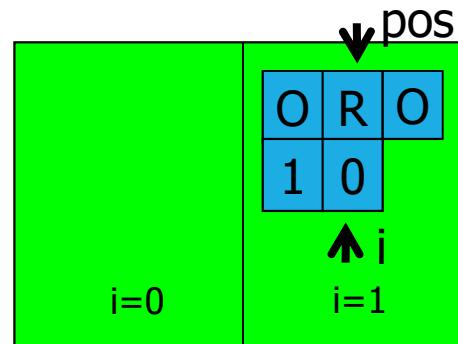


`mark[i]` non è > 0

ciclo for terminato, ritorna

stringa ORO

dist_val= 

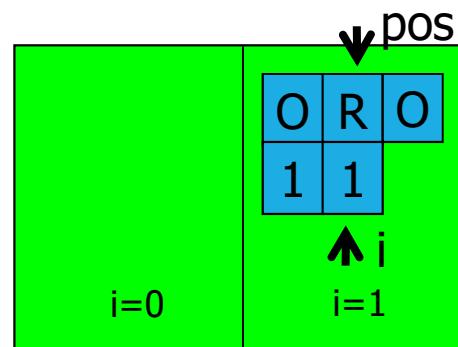


mark[i]++

backtrack

stringa ORO

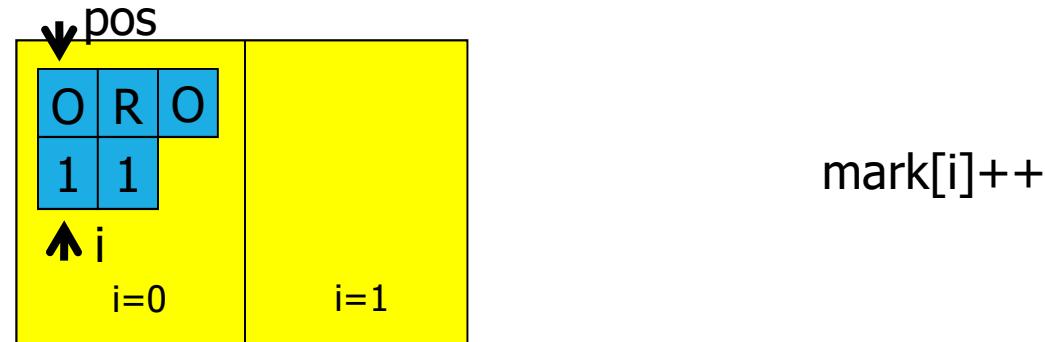
dist_val= 



ciclo for terminato, ritorna

stringa ORO

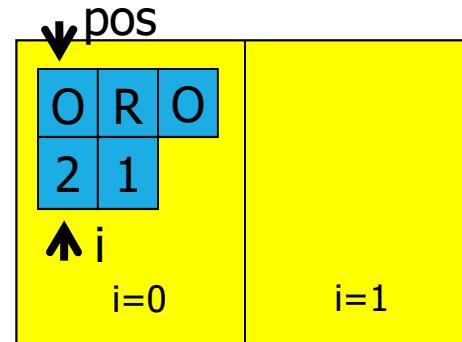
dist_val= 



backtrack

stringa ORO

dist_val= 



etc. etc.

Combinazioni semplici

Rispetto alle disposizioni semplici si tratta di «forzare» uno dei possibili ordinamenti:

- un indice `start` determina a partire da quale valore di `val` si inizia a riempire `sol`. Il vettore `val` viene scandito tramite indice `i` a partire da `start`
- il vettore `sol` viene riempito a partire dall'indice `pos` con i valori possibili di `val` da `start` in poi
- una volta assegnato a `sol` il valore `val[i]`, si ricorre con `i+1` e `pos+1`
- non serve il vettore `mark`
- `cnt` registra il numero di soluzioni.

```

int comb(int pos, int *val, int *sol, int n, int k, int start, int cnt) {
    int i, j;
    if (pos >= k) { terminazione
        for (i=0; i<k; i++)
            printf("%d ", sol[i]); iterazione sulle scelte
        printf("\n");
        return cnt+1;
    }
    for (i=start; i<n; i++) { scelta: sol[pos] riempito con i valori possibili di val da start in poi
        sol[pos] = val[i];
        cnt = comb(pos+1, val, sol, n, k, i+1, cnt);
    }
    return cnt;
}

```

ricorsione

prossima posizione

prossima scelta

`val = malloc(n * sizeof(int));`
`sol = malloc(k * sizeof(int));`

Esempio: combinazione di k tra n valori

Dato un insieme di n interi, generare tutte le combinazioni semplici di k di questi valori. Il numero di combinazioni è $n! / ((n-k)! * k!)$.

val = {7, 2, 0, 4, 1} :

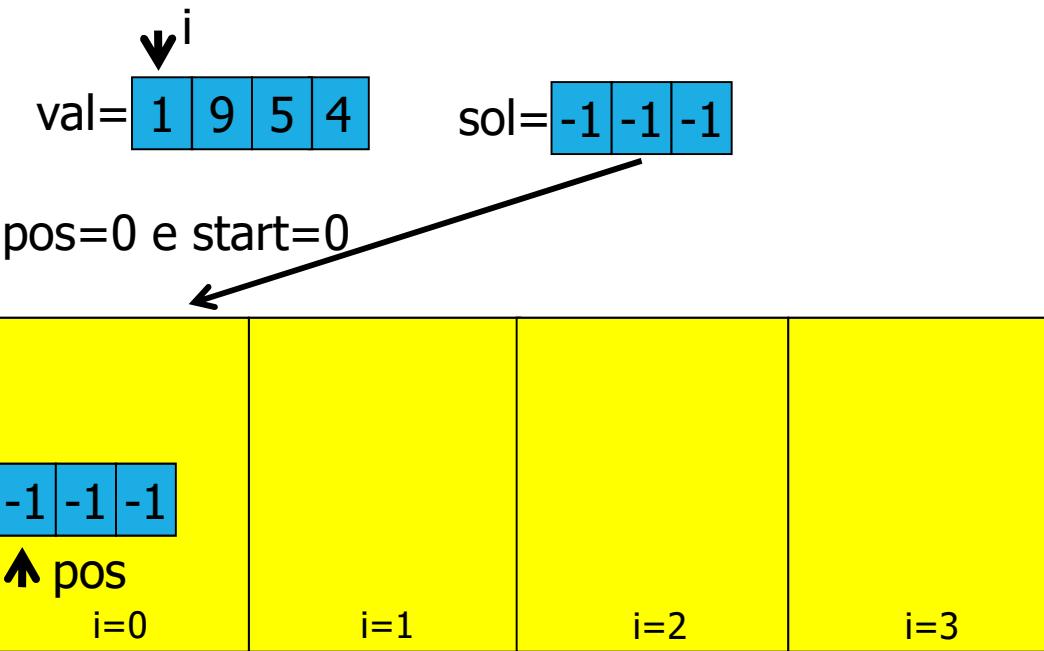
- $n = 5$ e $k = 4$: 5 combinazioni

{7,2,0,4} {7,2,0,1} {7,2,4,1} {7,0,4,1} {2,0,4,1}

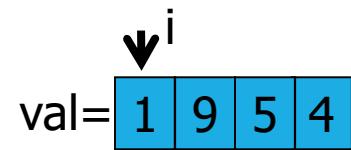
val = {1, 9, 5, 4} :

- $n = 4$ e $k = 3$: 4 combinazioni

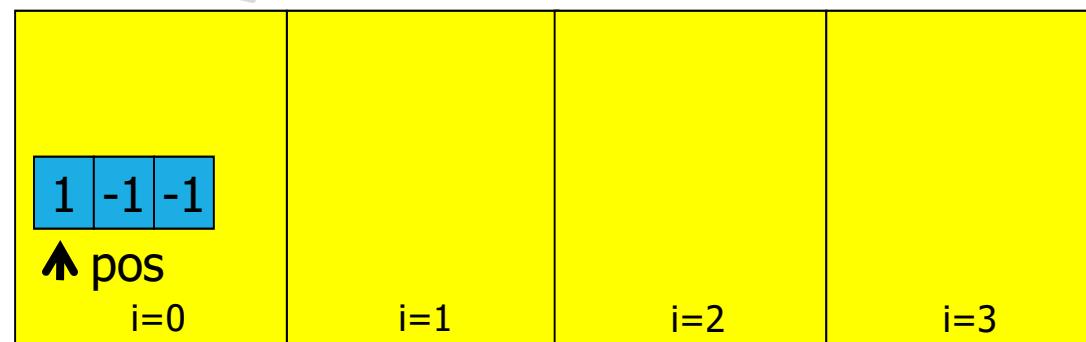
{1,9,5} {1,9,4} {1,5,4} {9,5,4}



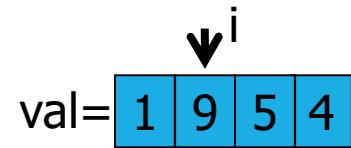
$$\text{sol}[\text{pos}] = \text{val}[i]$$



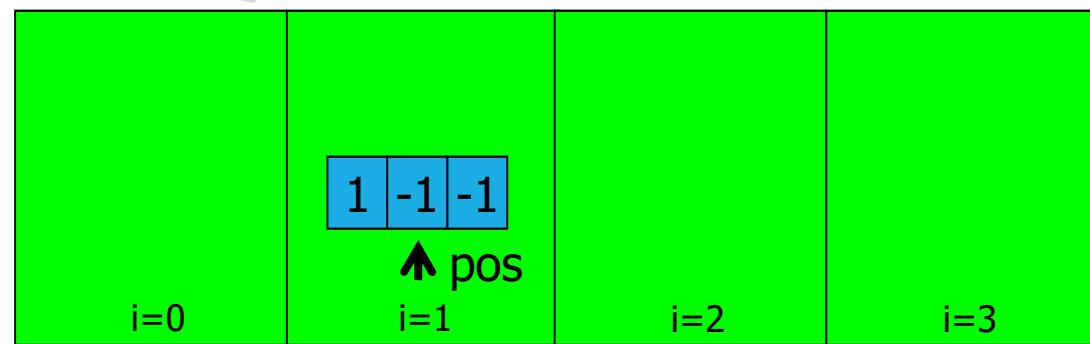
pos=0 e start=0



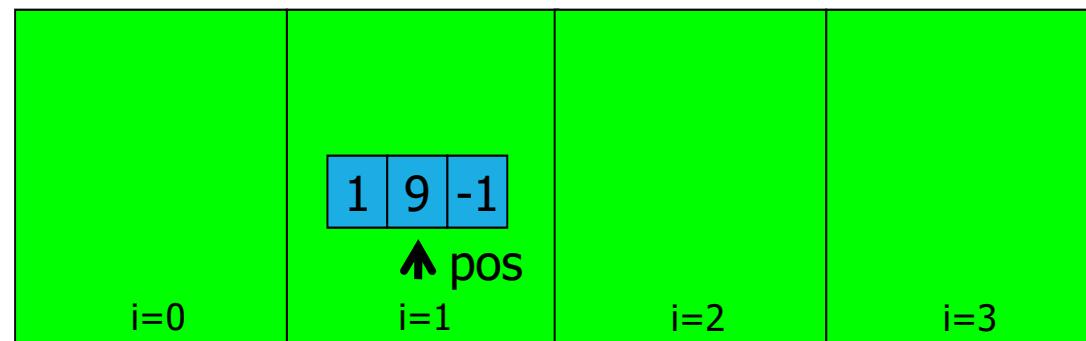
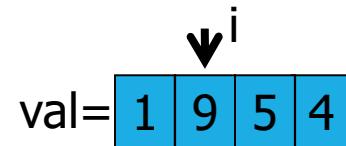
ricorsione
con pos=1 e start=1



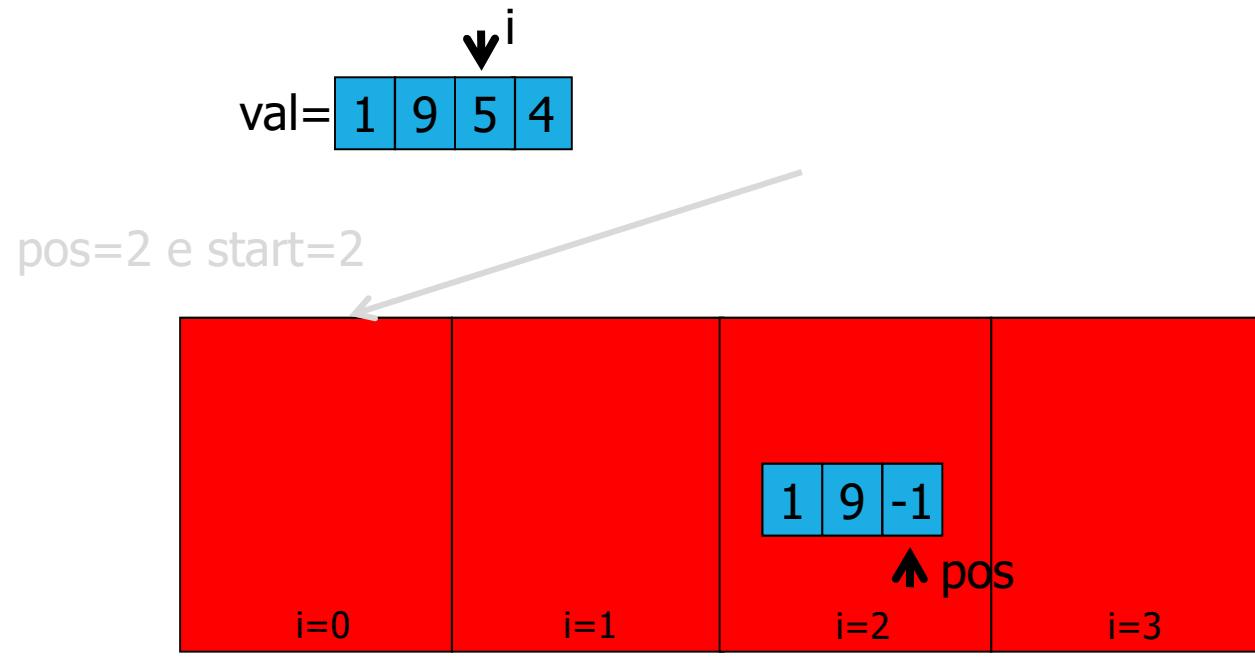
pos=1 e start=1



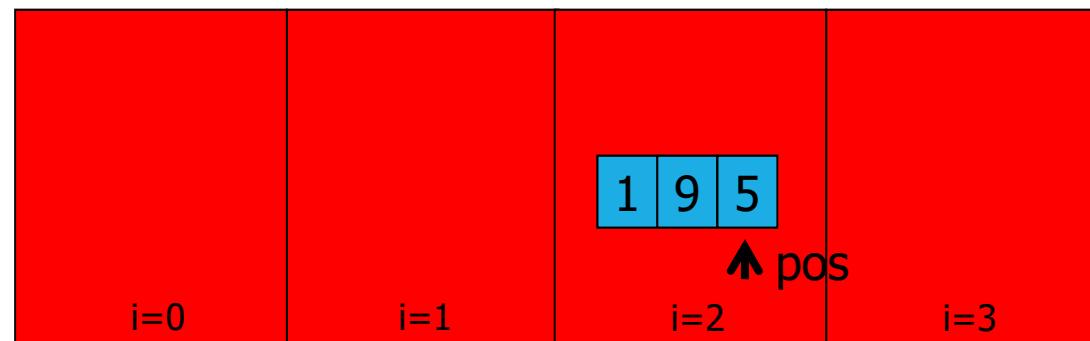
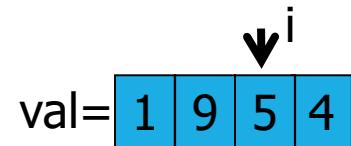
$$sol[pos] = val[i]$$



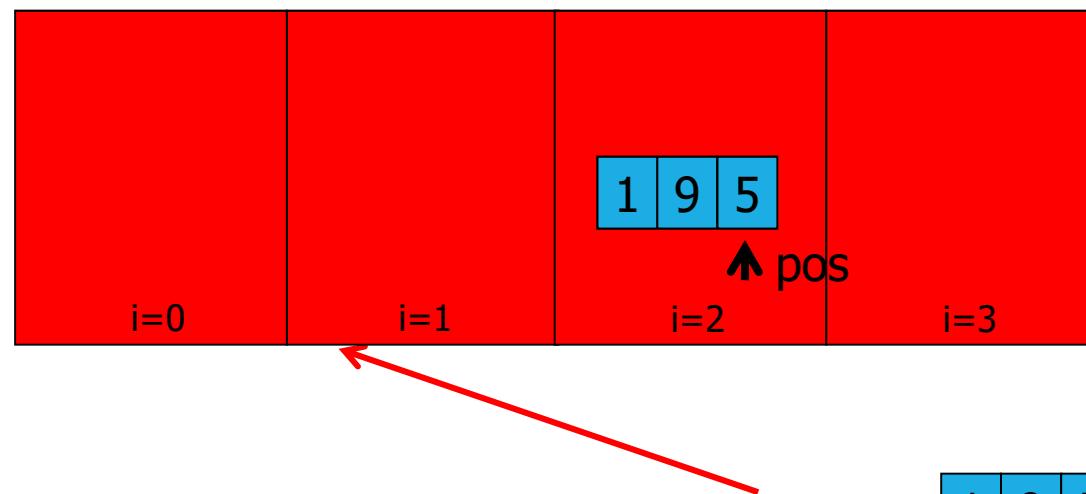
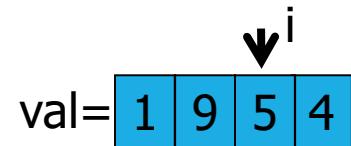
ricorsione
con pos=2 e start=2



$$\text{sol[pos]} = \text{val}[i]$$

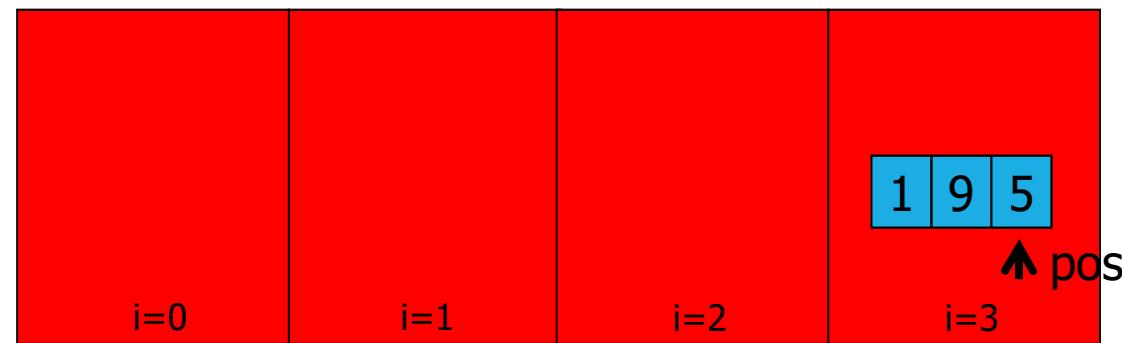
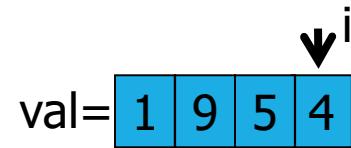


ricorsione
con pos=3 e start=3

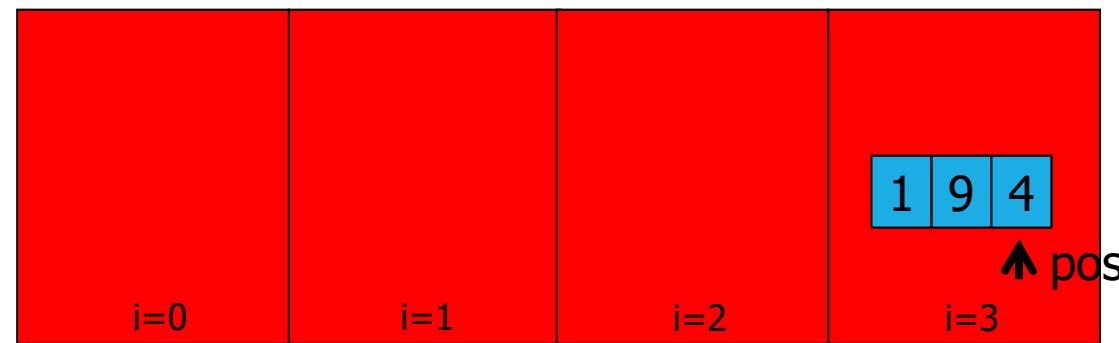
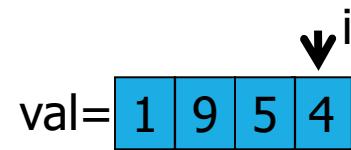


terminazione: visualizza, aggiorna cnt
ritorna

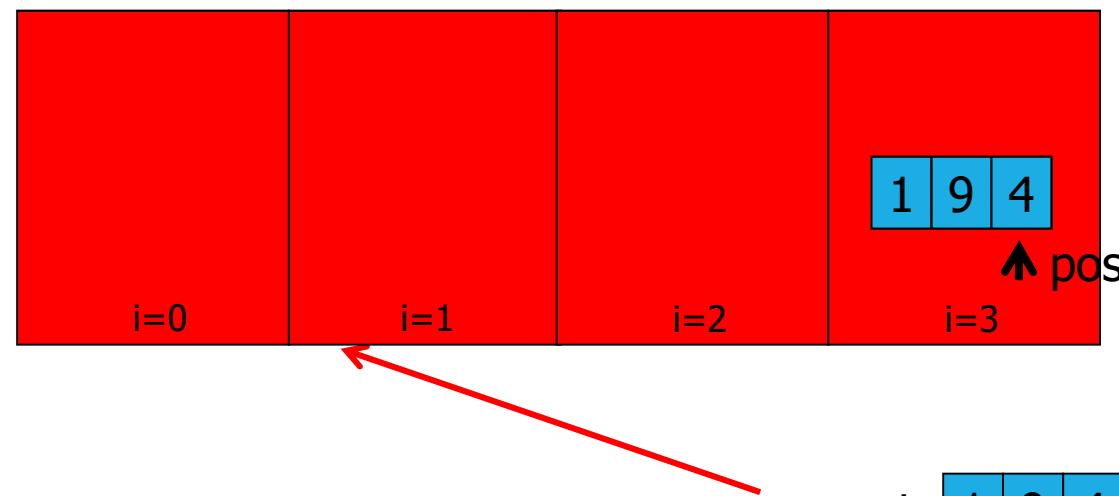
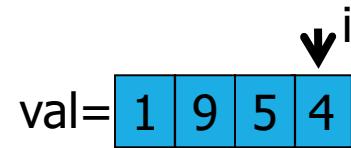
sol = 



$$sol[pos] = val[i]$$

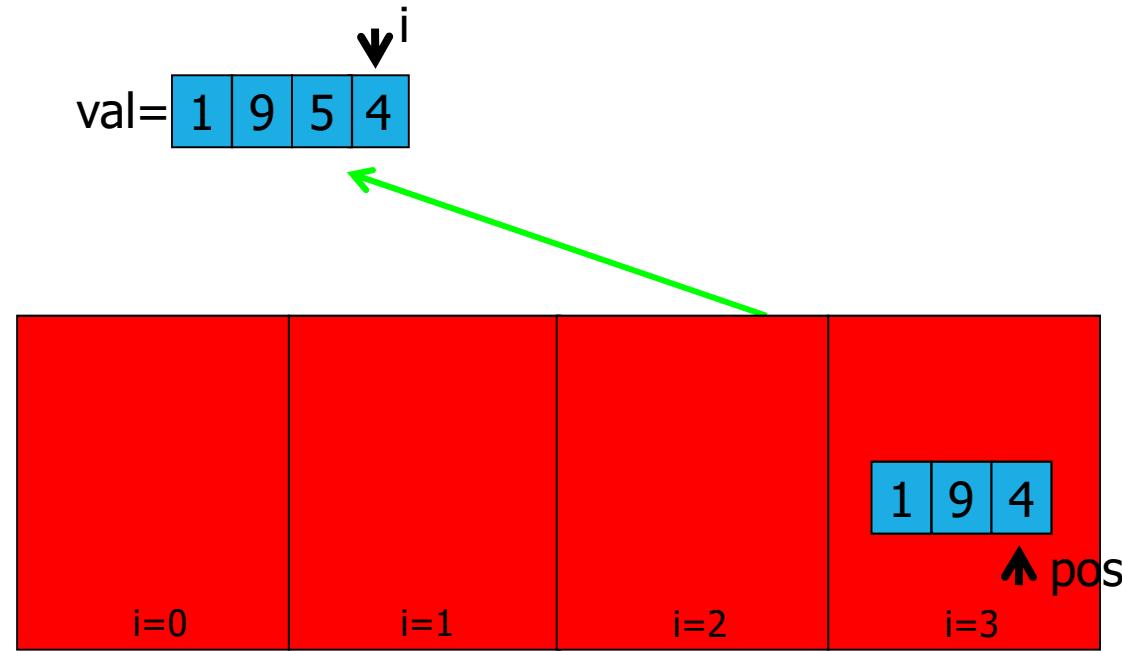


ricorsione
con pos=3 e start=4

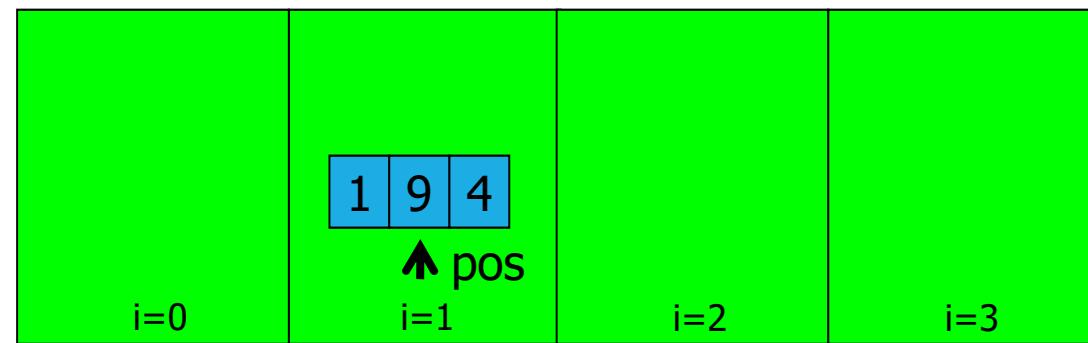
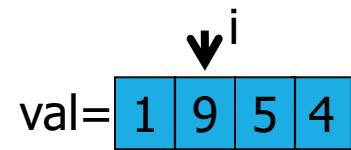


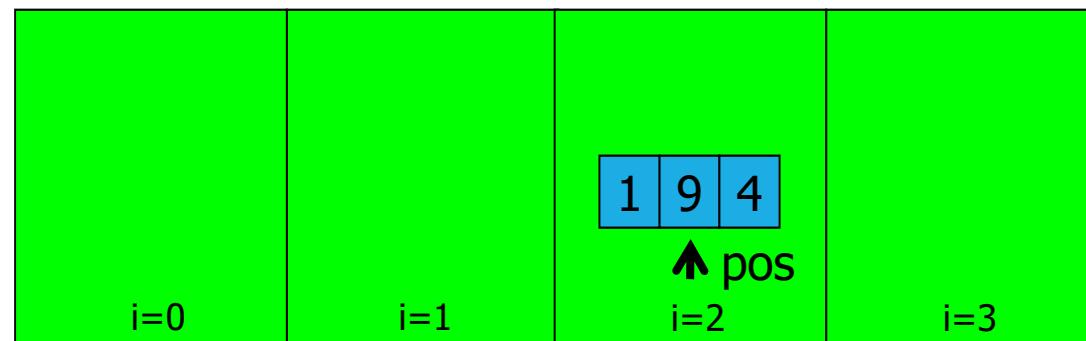
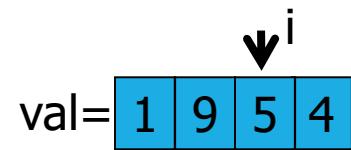
terminazione: visualizza, aggiorna cnt
ritorna

sol = 

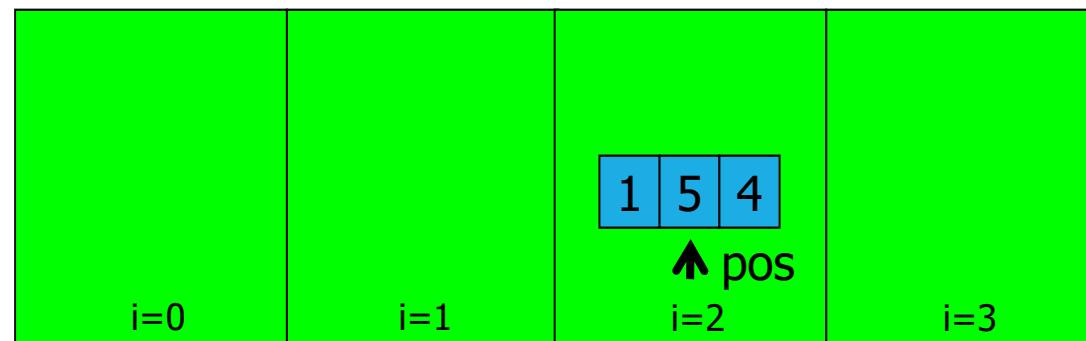
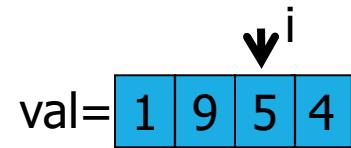


ciclo for terminato, ritorna

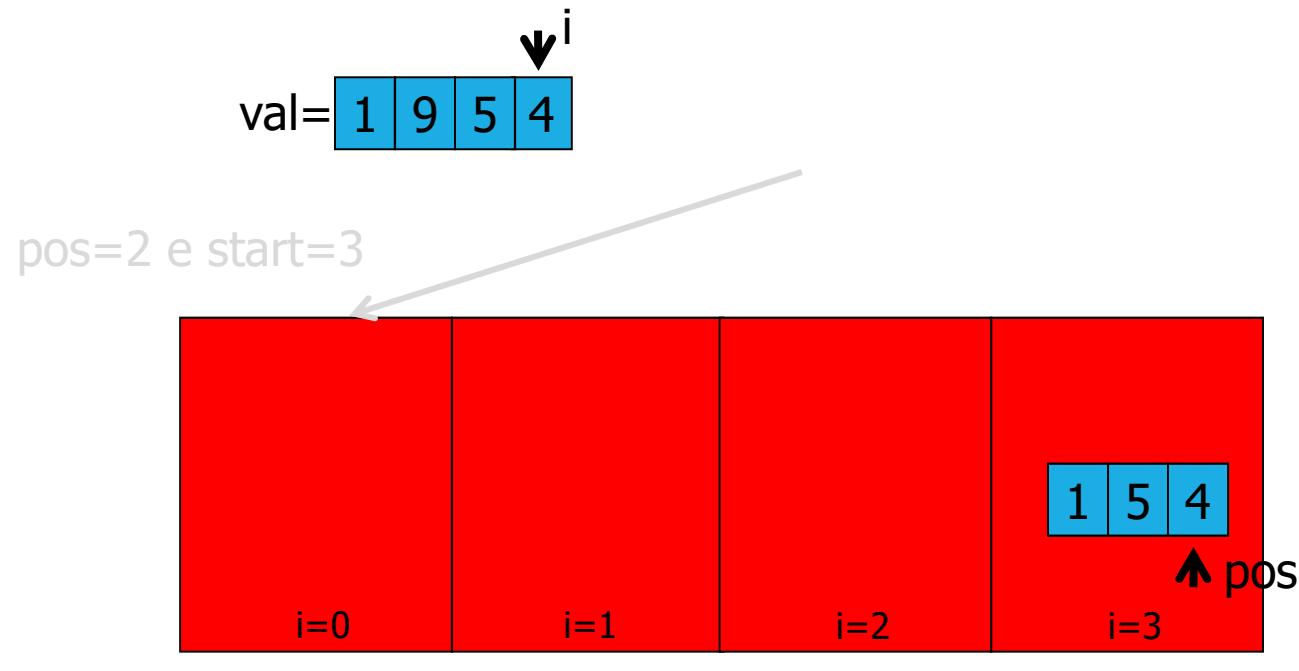




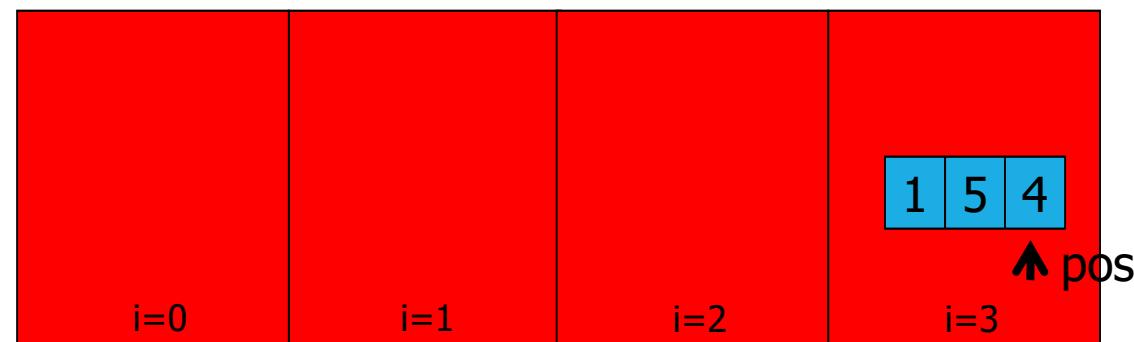
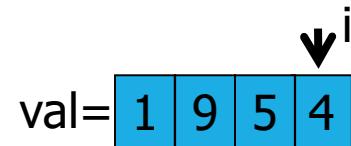
$$sol[pos] = val[i]$$



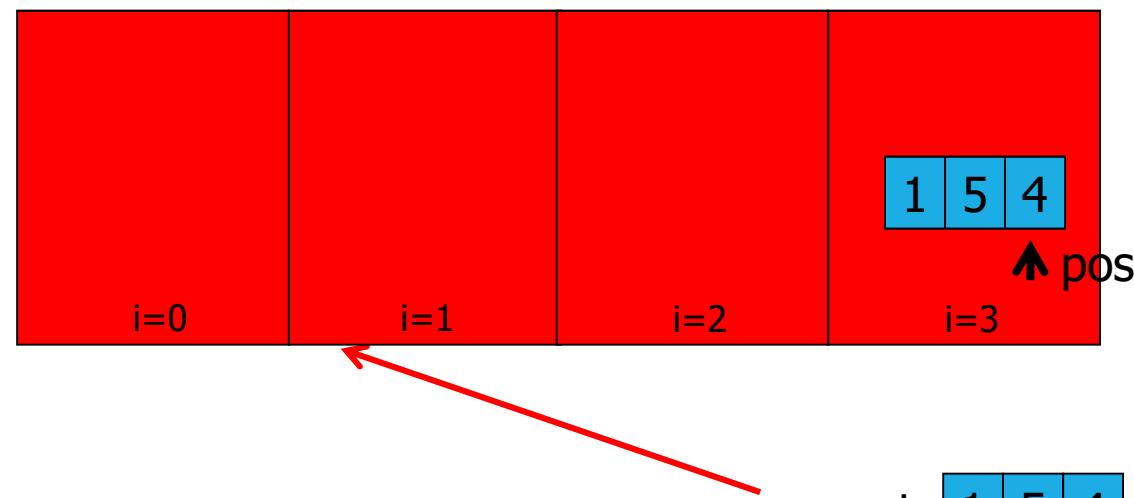
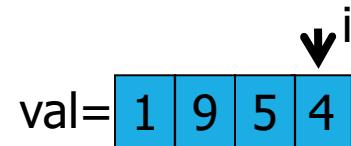
ricorsione
con pos=2 e start=3



$$\text{sol[pos]} = \text{val}[i]$$

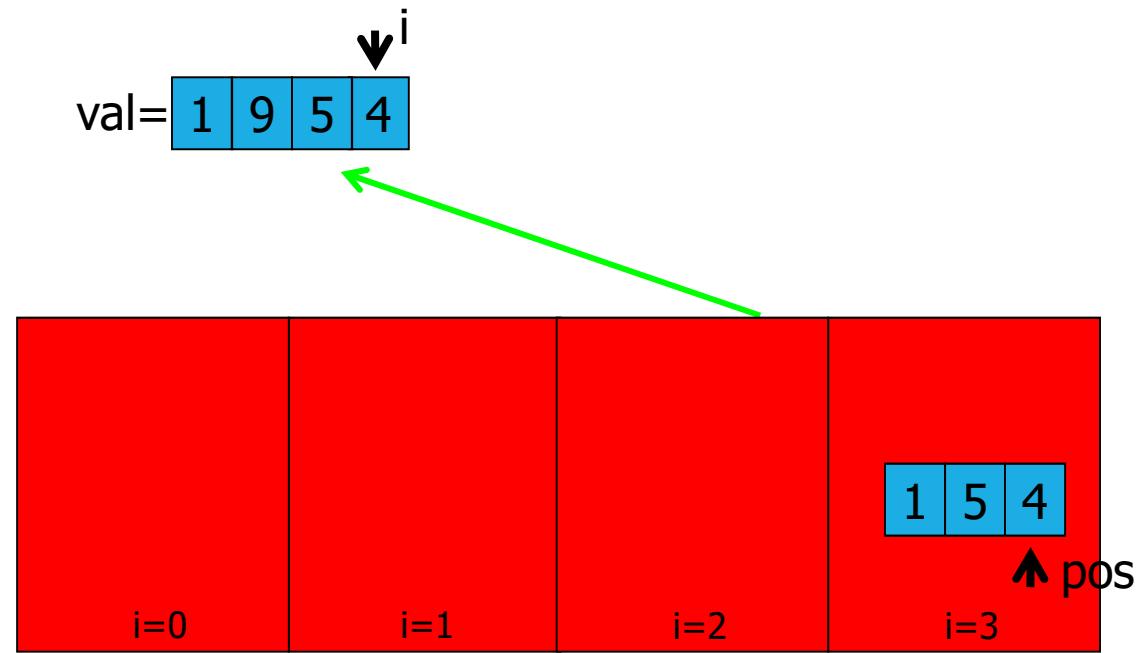


ricorsione
con pos=3 e start=4

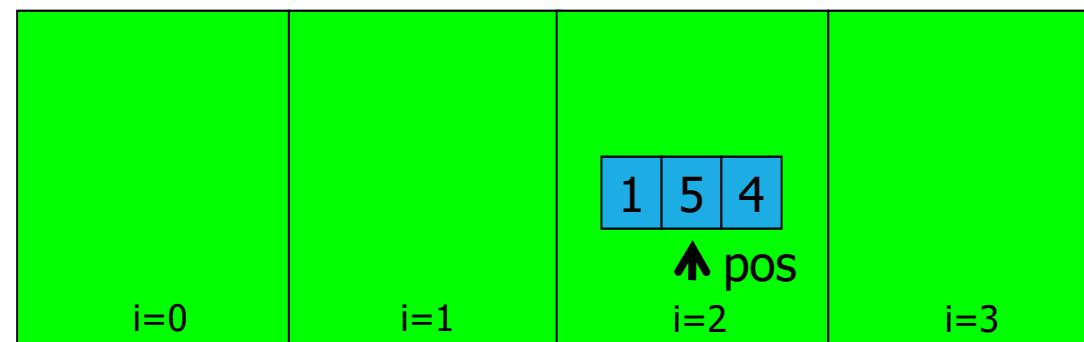
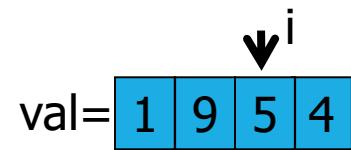


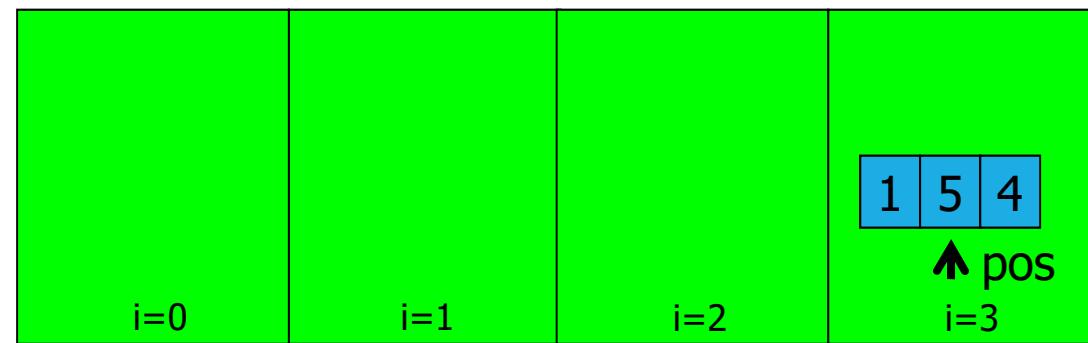
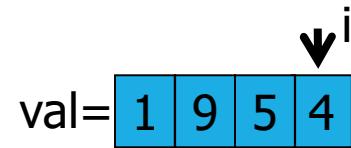
terminazione: visualizza, aggiorna cnt
ritorna

sol = 

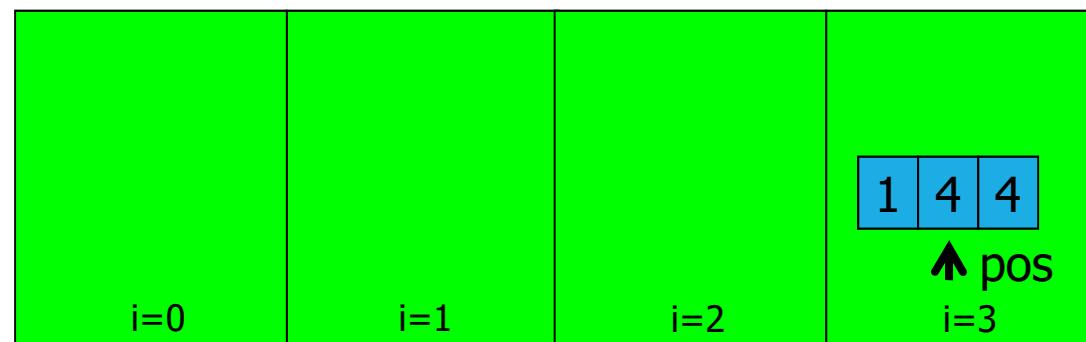
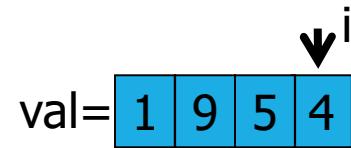


ciclo for terminato, ritorna

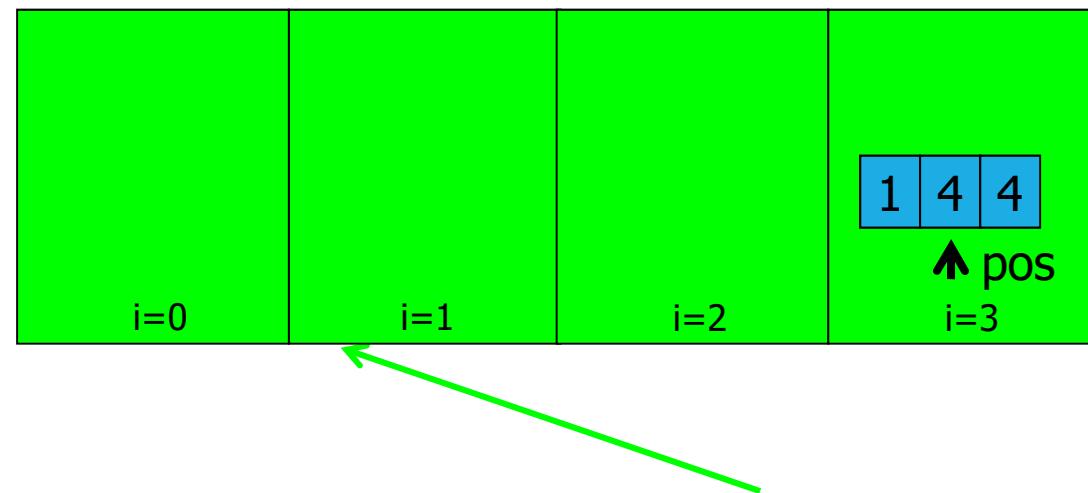
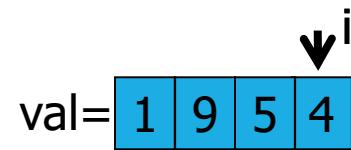




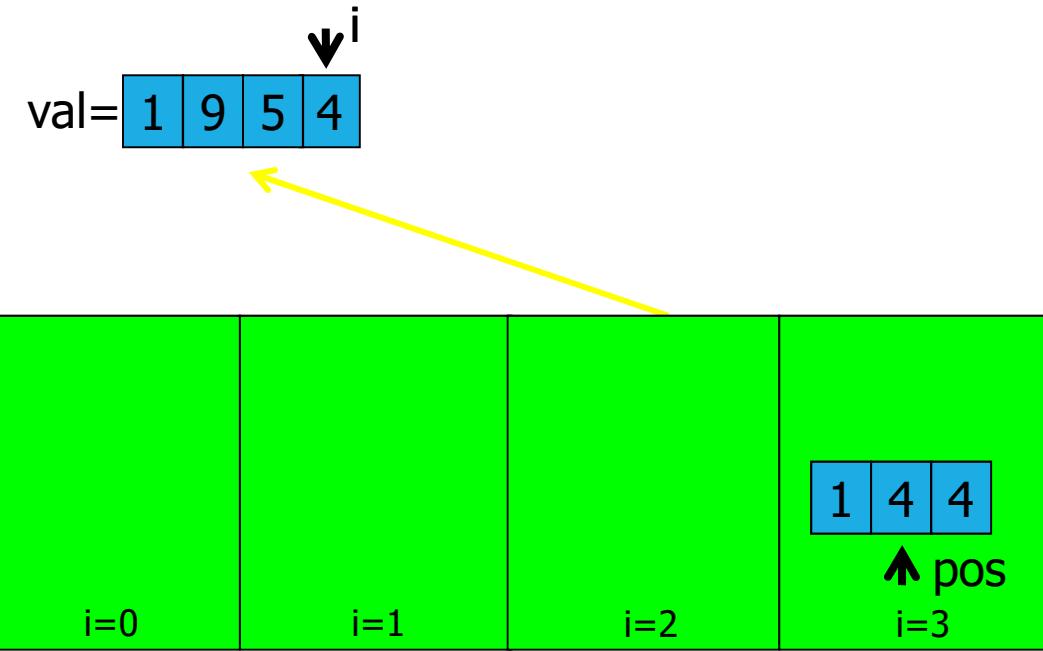
$$sol[pos] = val[i]$$



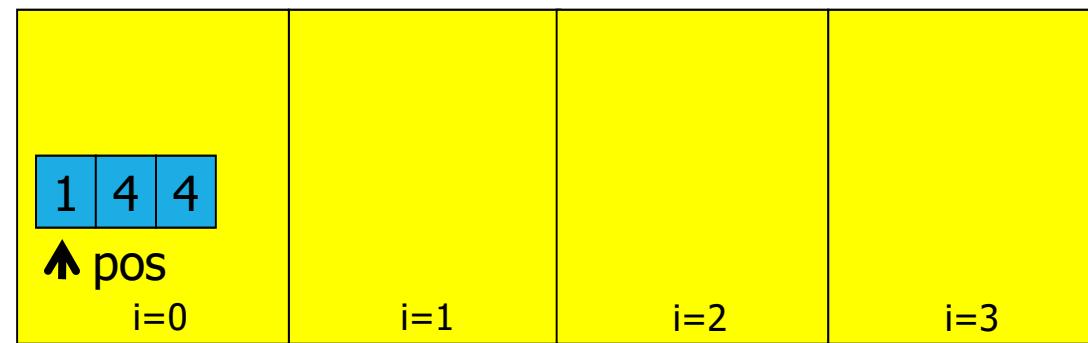
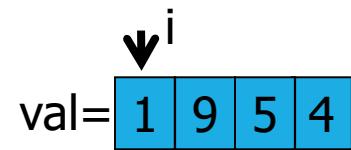
ricorsione
con pos=2 e start=4

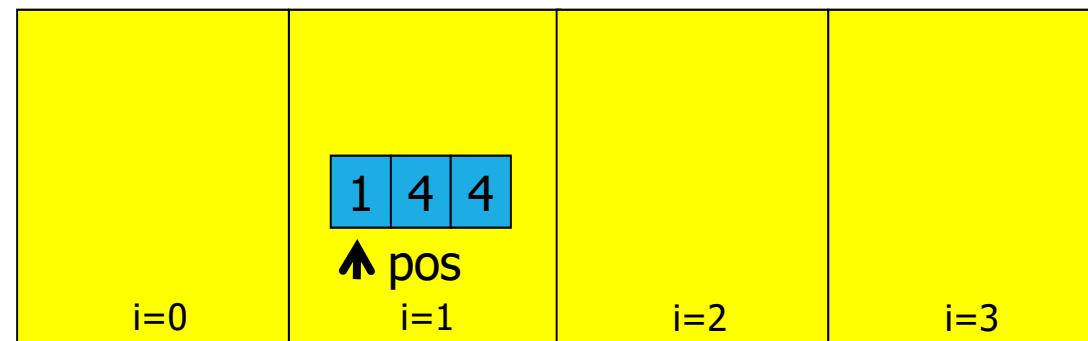
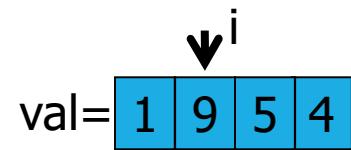


il ciclo for non inizia nemmeno
ritorna

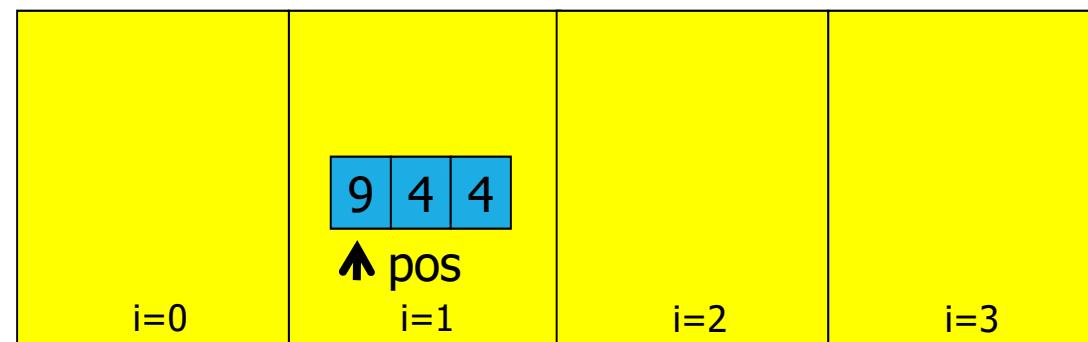
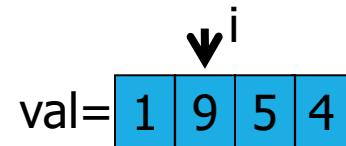


ciclo terminato, ritorna





$$\text{sol}[\text{pos}] = \text{val}[i]$$



ricorsione
con pos=1 e start=2

etc. etc.

Combinazioni ripetute

Come per le combinazioni semplici ma:

- la ricorsione avviene solo per pos+1 e non per i+1
- l'indice start viene incrementato quando termina la ricorsione
- cnt registra il numero di soluzioni.

```

int comb_r(int pos,int *val,int *sol,int n,int k,int start,int cnt) {
    int i, j;
    if (pos >= k) { terminazione
        for (i=0; i<k; i++)
            printf("%d ", sol[i]);
        printf("\n");
        return cnt+1;
    }
    for (i=start; i<n; i++) { iterazione sulle scelte
        sol[pos] = val[i];
        cnt = comb_r(pos+1, val, sol, n, k, start, cnt);
        start++;
    }
    return cnt;
}

```

scelta: sol[pos] riempito con i valori possibili di val da start in poi

ricorsione su prossima posizione

aggiornamento di start

val = malloc(n * sizeof(int));
 sol = malloc(k * sizeof(int));

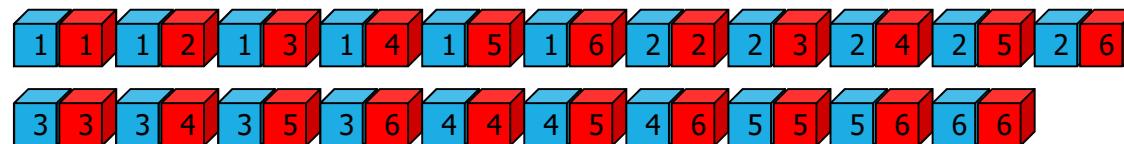
Esempio

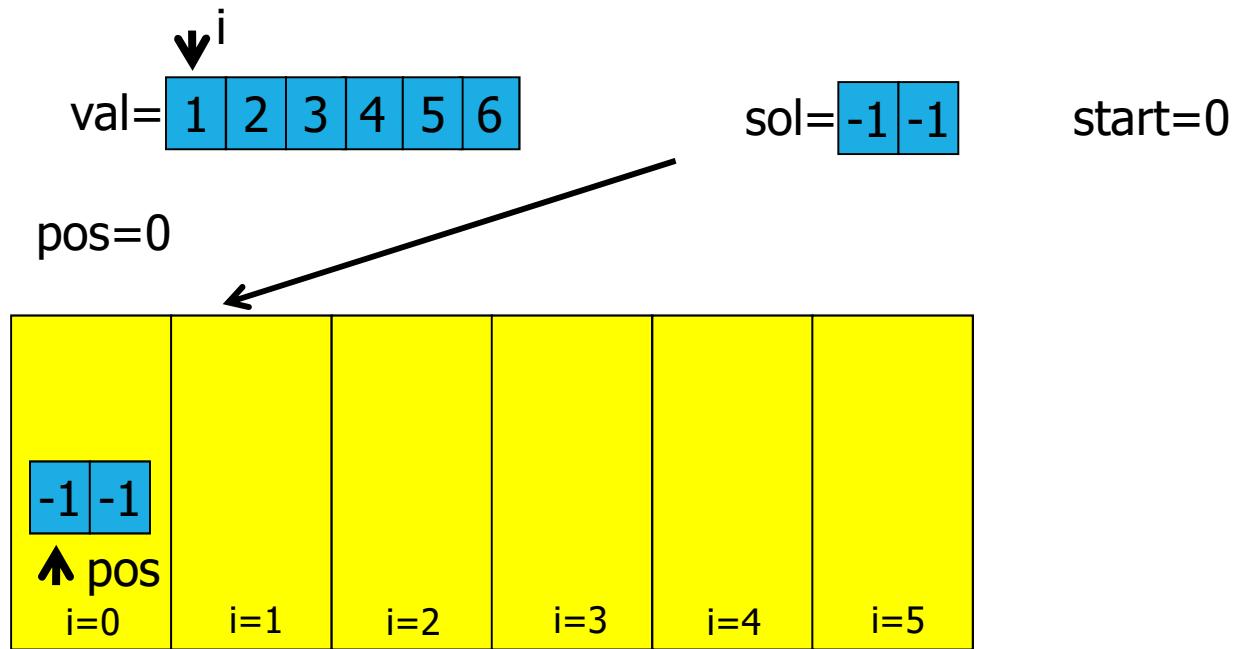
Lanciando contemporaneamente 2 dadi , quante sono le composizioni con cui si possono presentare le facce?

Modello: combinazioni con ripetizione

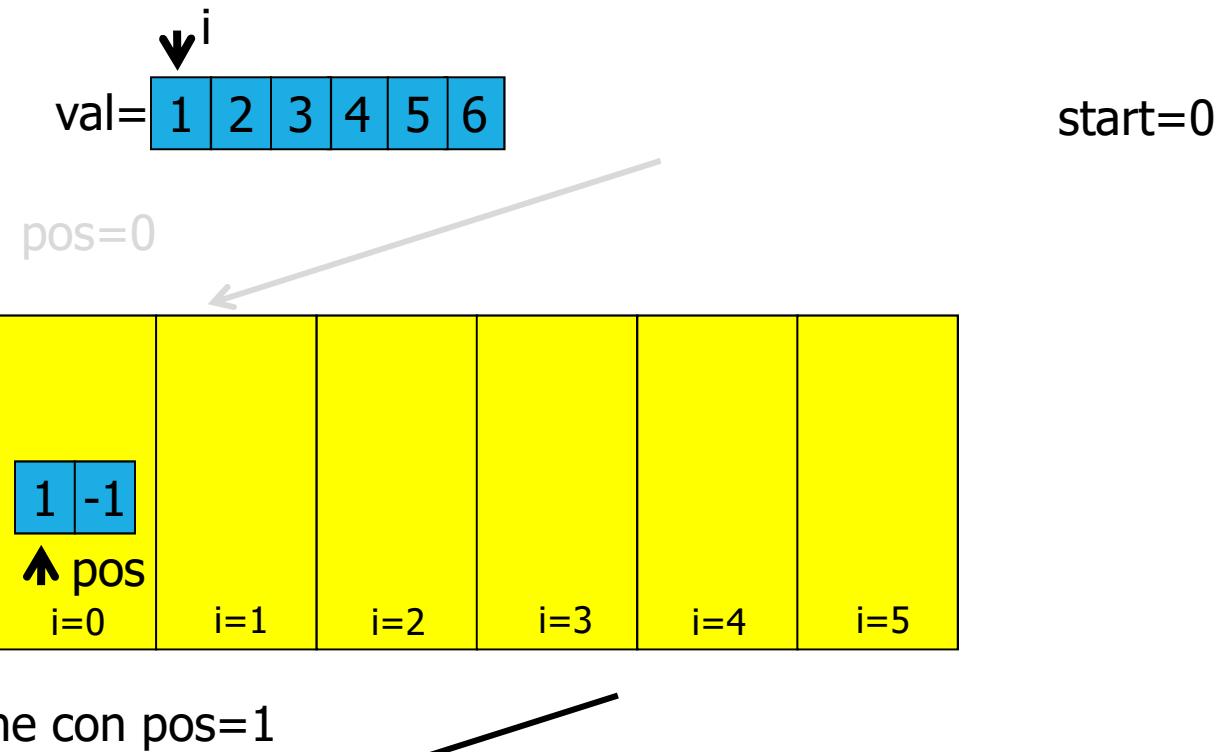
$$C'_{6,2} = (6 + 2 - 1)! / 2!(6-1)! = 21$$

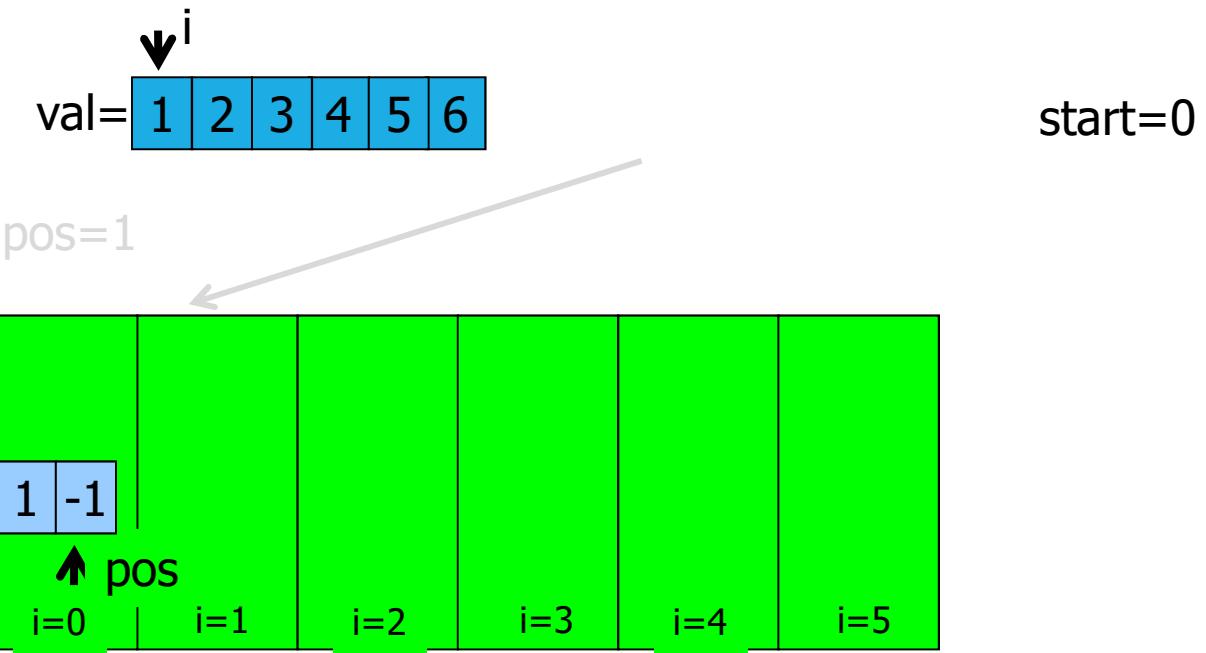
Soluzione





$$\text{sol}[\text{pos}] = \text{val}[i]$$





$$sol[pos] = val[i]$$

\downarrow^i
val=

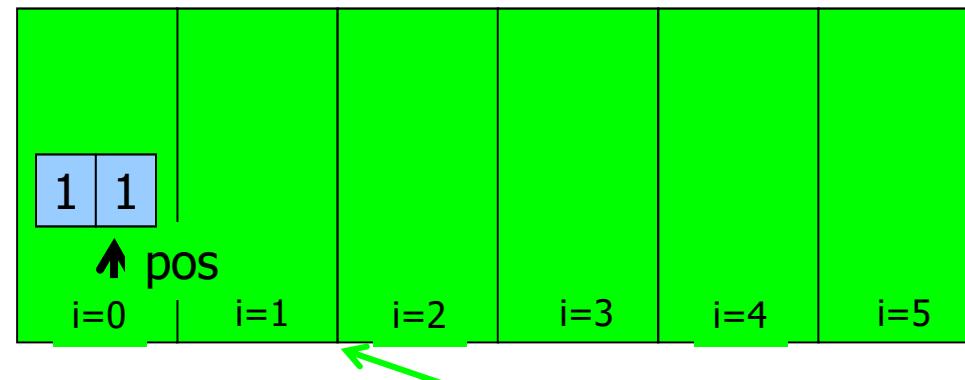
1	2	3	4	5	6
---	---	---	---	---	---

start=0

pos=1



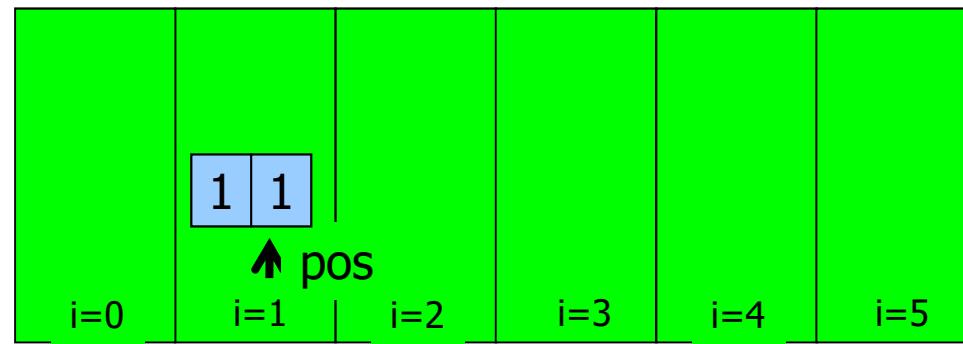
ricorsione con pos=2



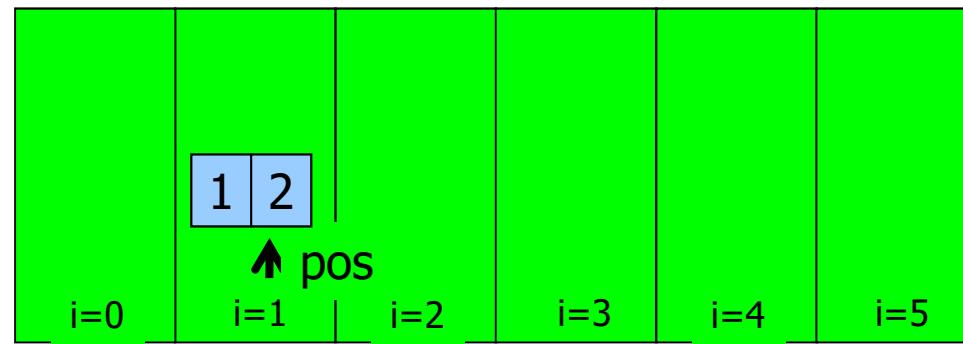
terminazione: visualizza, aggiorna cnt
 ritorna e aggiorna start

sol =

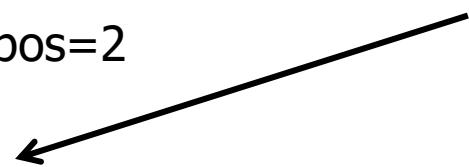
1	1
---	---

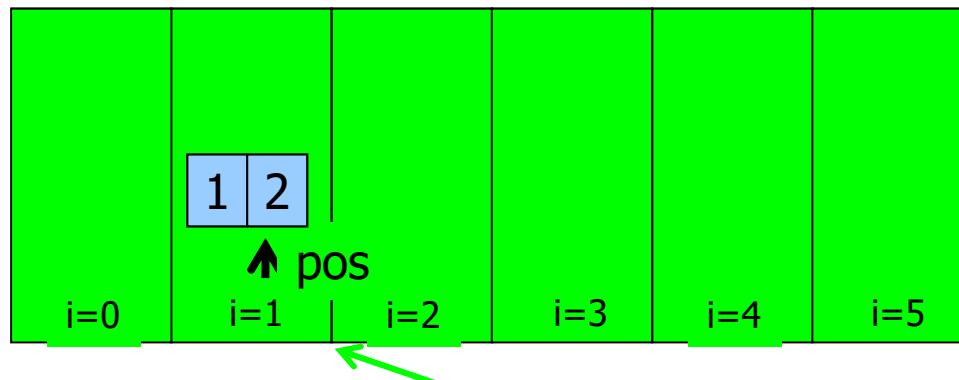
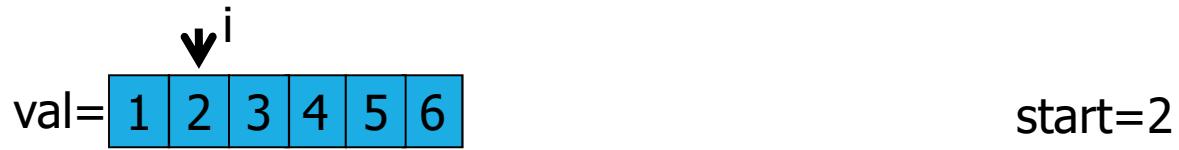


$$sol[pos] = val[i]$$



ricorsione con pos=2

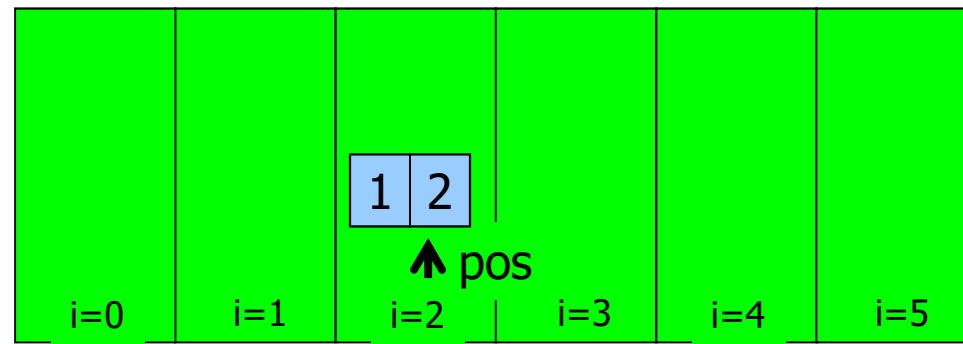
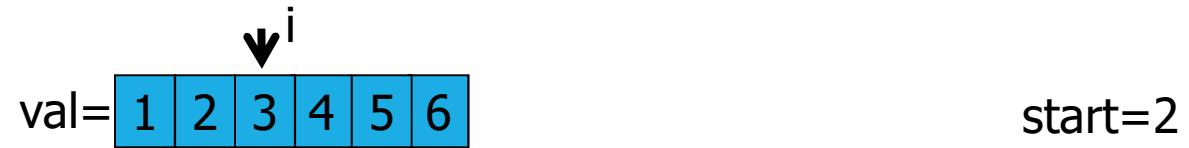




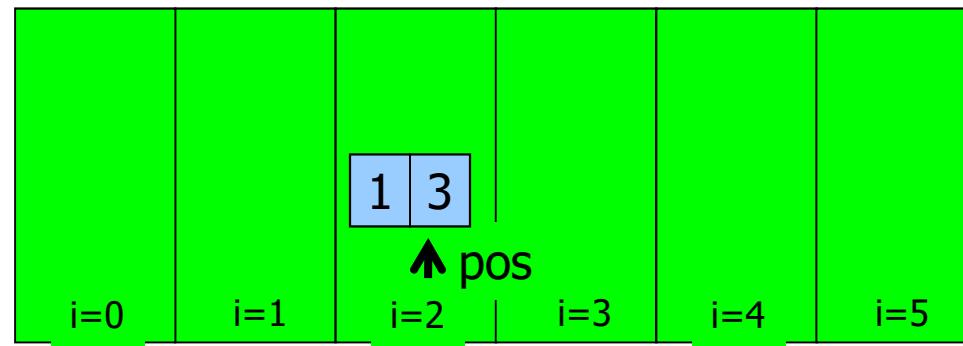
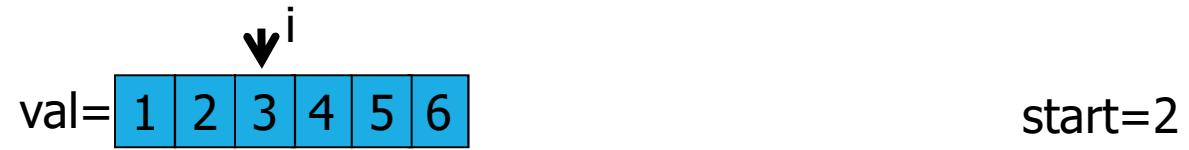
terminazione: visualizza, aggiorna cnt
 ritorna e aggiorna start

sol =

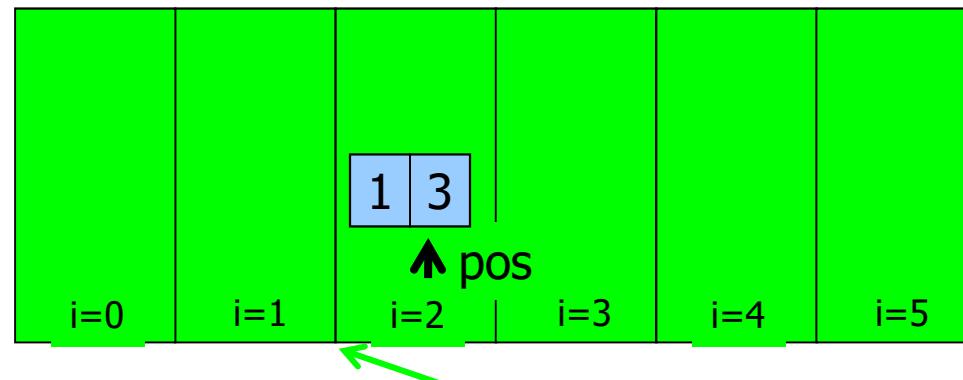
1	2
---	---



$$sol[pos] = val[i]$$



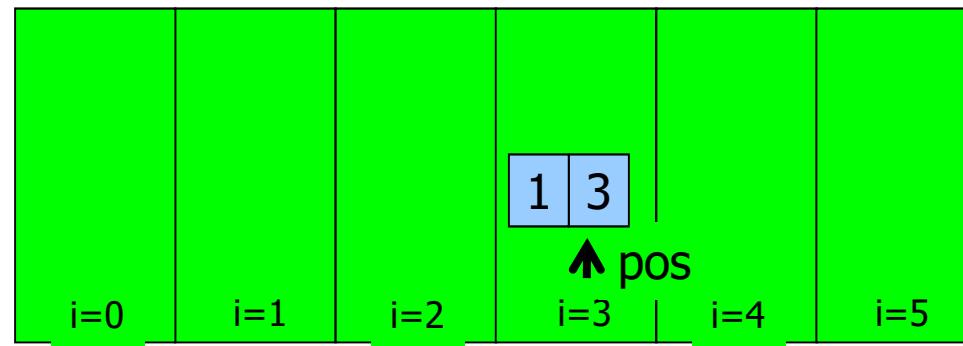
ricorsione con pos=2



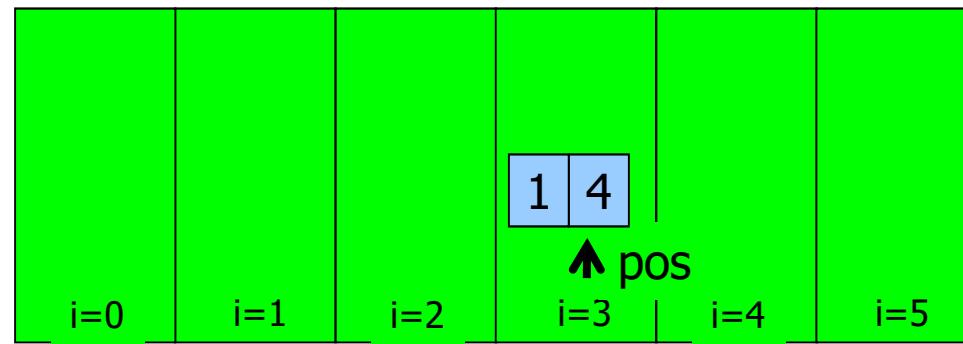
terminazione: visualizza, aggiorna cnt
 ritorna e aggiorna start

sol =

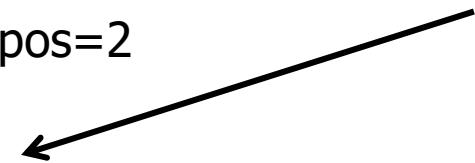
1	3
---	---

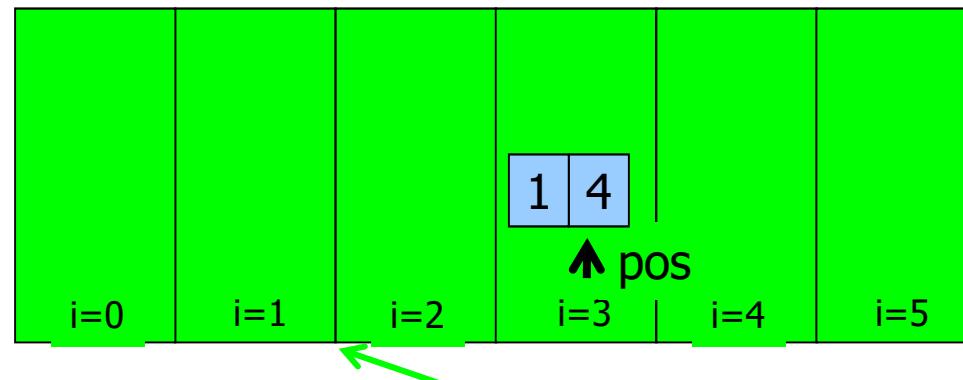


$$\text{sol}[\text{pos}] = \text{val}[i]$$



ricorsione con pos=2

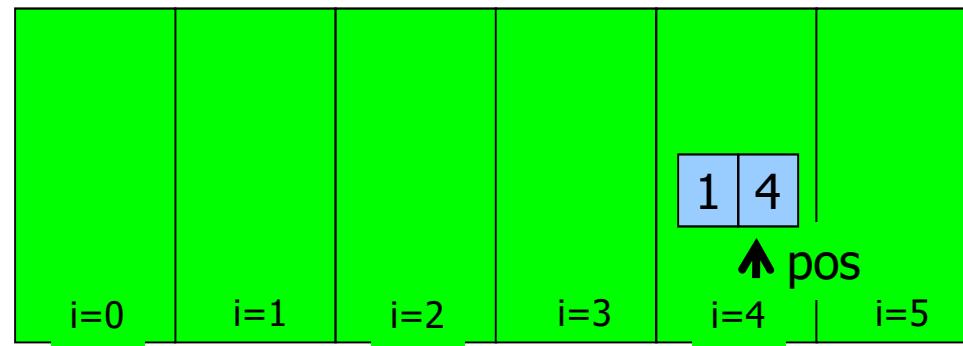




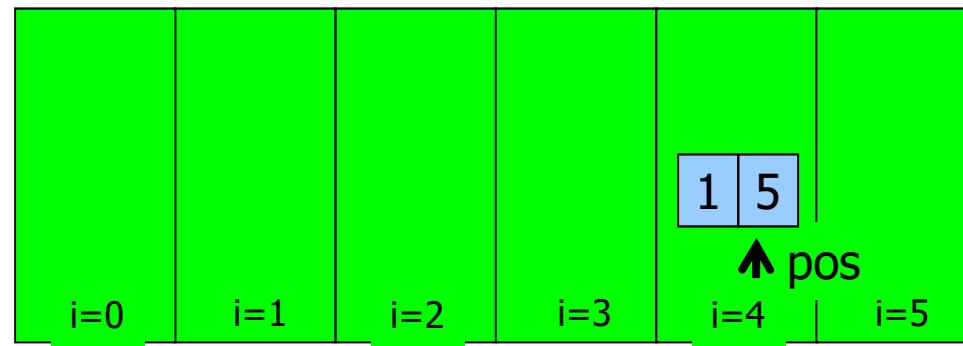
terminazione: visualizza, aggiorna cnt
 ritorna e aggiorna start

sol =

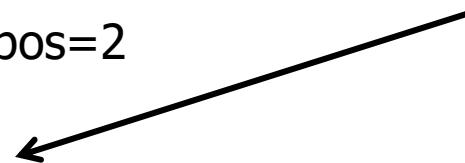
1	4
---	---

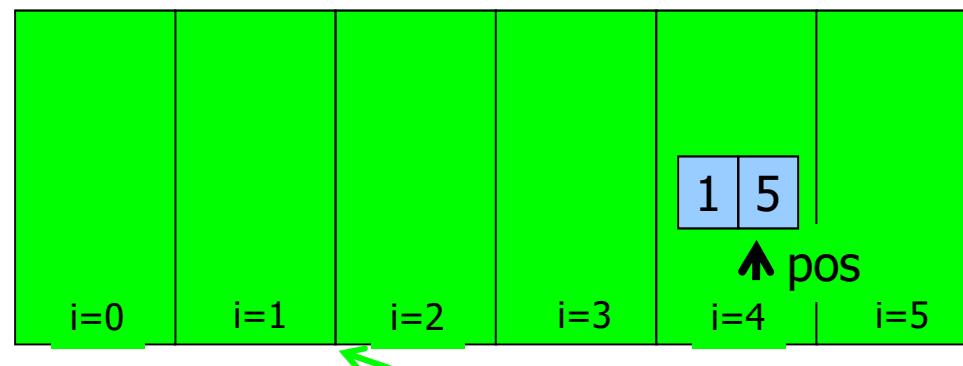


$$\text{sol[pos]} = \text{val}[i]$$



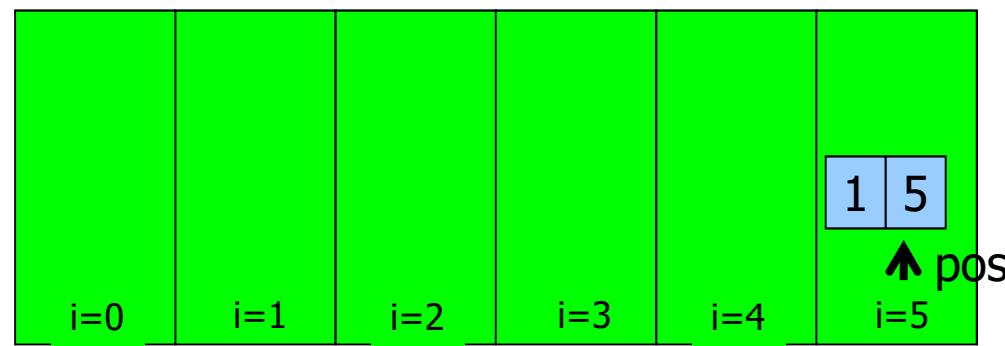
ricorsione con pos=2



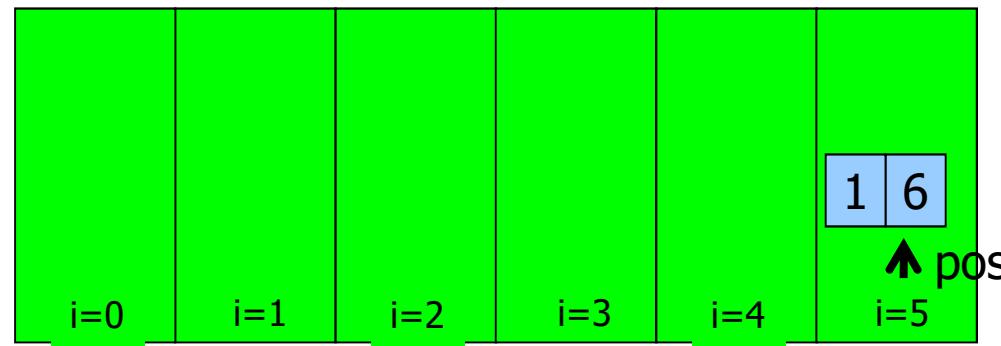


terminazione: visualizza, aggiorna cnt
ritorna e aggiorna start

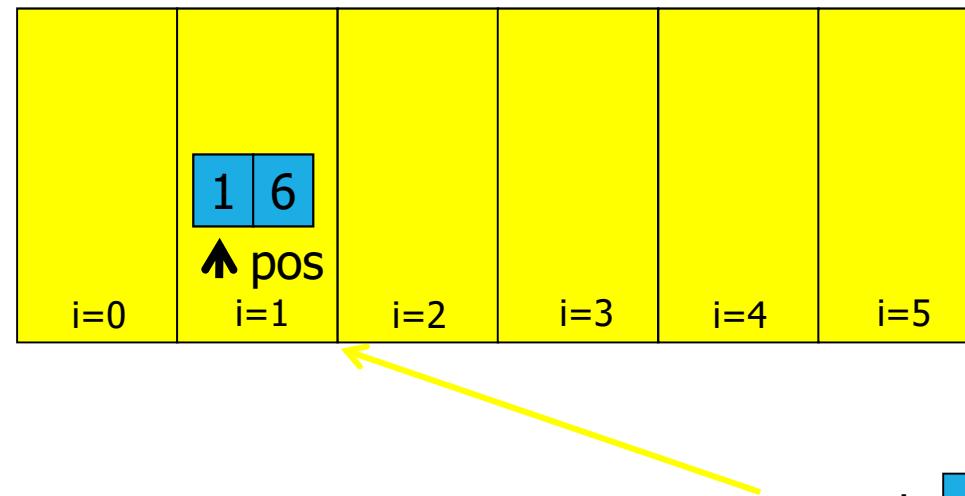
sol = 



$$\text{sol[pos]} = \text{val}[i]$$



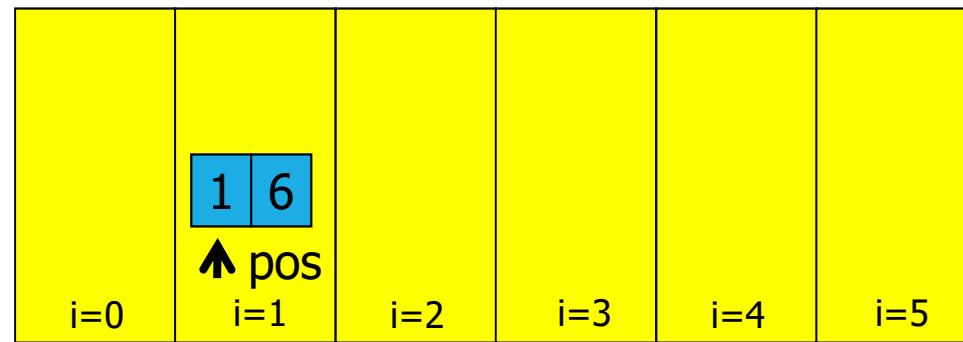
ricorsione con pos=2



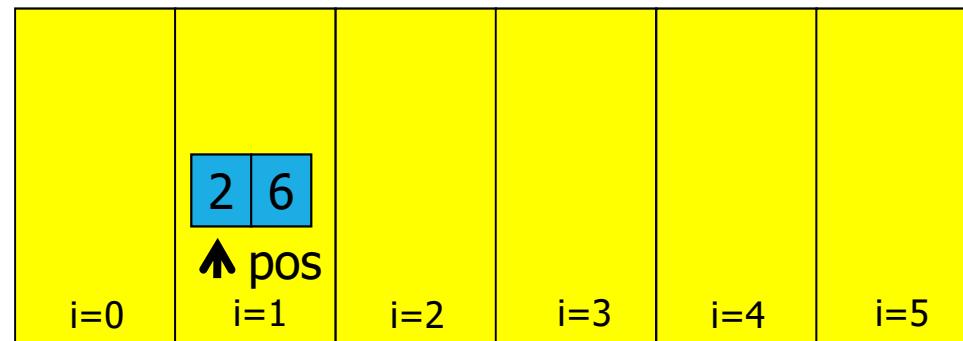
terminazione: visualizza, aggiorna cnt
 ritorna e aggiorna start

sol =

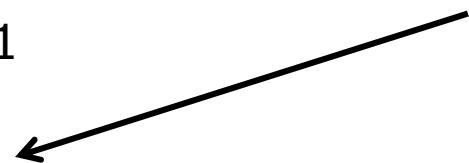
1	6
---	---



$$sol[pos] = val[i]$$



ricorsione con 1



etc. etc.

L'Insieme delle Parti

Dato un insieme S di n elementi, si può calcolare l'insieme delle parti ricorrendo a 3 modelli:

1. paradigma divide et impera
2. disposizioni ripetute
3. combinazioni semplici

Divide et impera

- caso terminale: insieme il cui unico elemento è l'insieme vuoto $\{\emptyset\}$
- caso ricorsivo: insieme formato dall'unione:
 - dall'insieme delle parti $\wp(S_{n-1})$ per n-1 elementi
 - $\forall i$ con gli insiemi che risultano dall'unione di ognuno degli insiemi \wp_i che appartengono all'insieme delle parti per n-1 elementi $\wp(S_{n-1})$ con l'insieme che contiene l'elemento i-esimo $\{s_i\}$

$$\wp(S_n) = \begin{cases} \{\emptyset\} & \text{se } n = 0 \\ \wp(S_{n-1}) \cup \{\wp_i \cup \{s_i\} \mid \wp_i \in \wp(S_{n-1})\} & \text{se } n > 0 \end{cases}$$

- Si usano 2 rami ricorsivi distinti, a seconda che l'elemento corrente sia incluso o meno nella soluzione
- in sol si memorizza direttamente l'elemento, non un flag di presenza/assenza
- l'indice start serve per escludere soluzioni simmetriche (quindi già calcolate)
- il valore di ritorno cnt rappresenta il numero totale di insiemi.

```

int powerset(int pos,int *val,int *sol,int n,int start,int cnt) {
    int i;
    if (start >= n) {
        for (i = 0; i < pos; i++)
            printf("%d ", sol[i]);
        printf("\n");
        return cnt+1;
    }
    for (i = start; i < n; i++) {
        sol[pos] = val[i];
        cnt = powerset(pos+1, val, sol, n, i+1, cnt);
    }
    cnt = powerset(pos, val, sol, n, n, cnt);
    return cnt;
}

```

terminazione: non ci sono più elementi

per tutti gli elementi da start in poi

includi elemento e ricorri

non aggiungere nulla e ricorri



04powerset

Disposizioni ripetute

Ogni sottoinsieme è rappresentato dal vettore della soluzione sol di n elementi:

- l'insieme delle scelte possibili per ogni posizione del vettore è {0, 1}, quindi k = 2. Il ciclo for è sostituito da 2 assegnazioni esplicite
 - $\text{sol}[\text{pos}] = 0$ se l'oggetto pos-esimo non appartiene al sottoinsieme
 - $\text{sol}[\text{pos}] = 1$ se l'oggetto pos-esimo appartiene al sottoinsieme
 - nella stessa soluzione 0 e 1 possono comparire più volt
- È scambiato il ruolo di n e k rispetto alla definizione di disposizioni ripetute (dove n era il numero di scelte e k la dimensione della soluzione).

```

int powerset(int pos,int *val,int *sol,int n,int cnt) {
    int j;
    if (pos >= n) {
        printf("{ \t");
        for (j=0; j<n; j++)
            if (sol[j]!=0)
                printf("%d \t", val[j]);
        printf("} \n");
        return cnt+1;
    }
    sol[pos] = 0;
    cnt = powerset(pos+1, val, sol, n, cnt);
    sol[pos] = 1;
    cnt = powerset(pos+1, val, sol, n, cnt);
    return cnt;
}

```

terminazione:
stampa soluzione

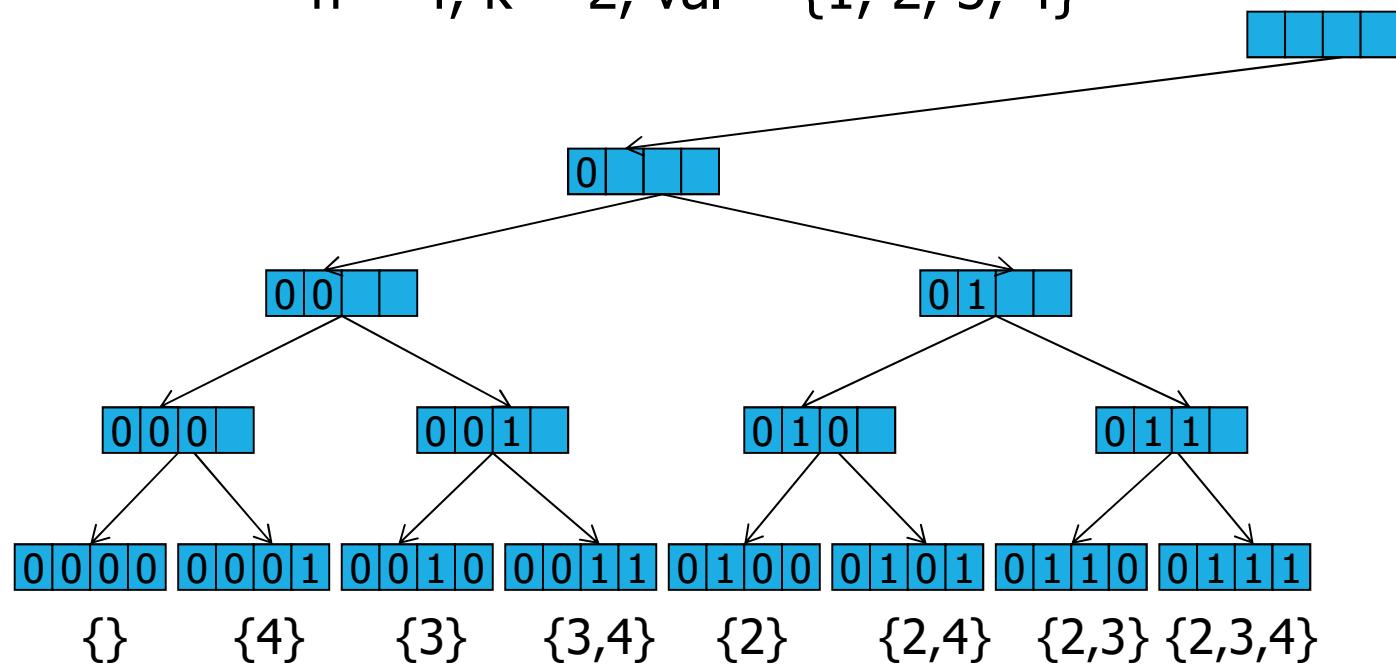
non prendere
l'elemento pos

ricorri su pos+1

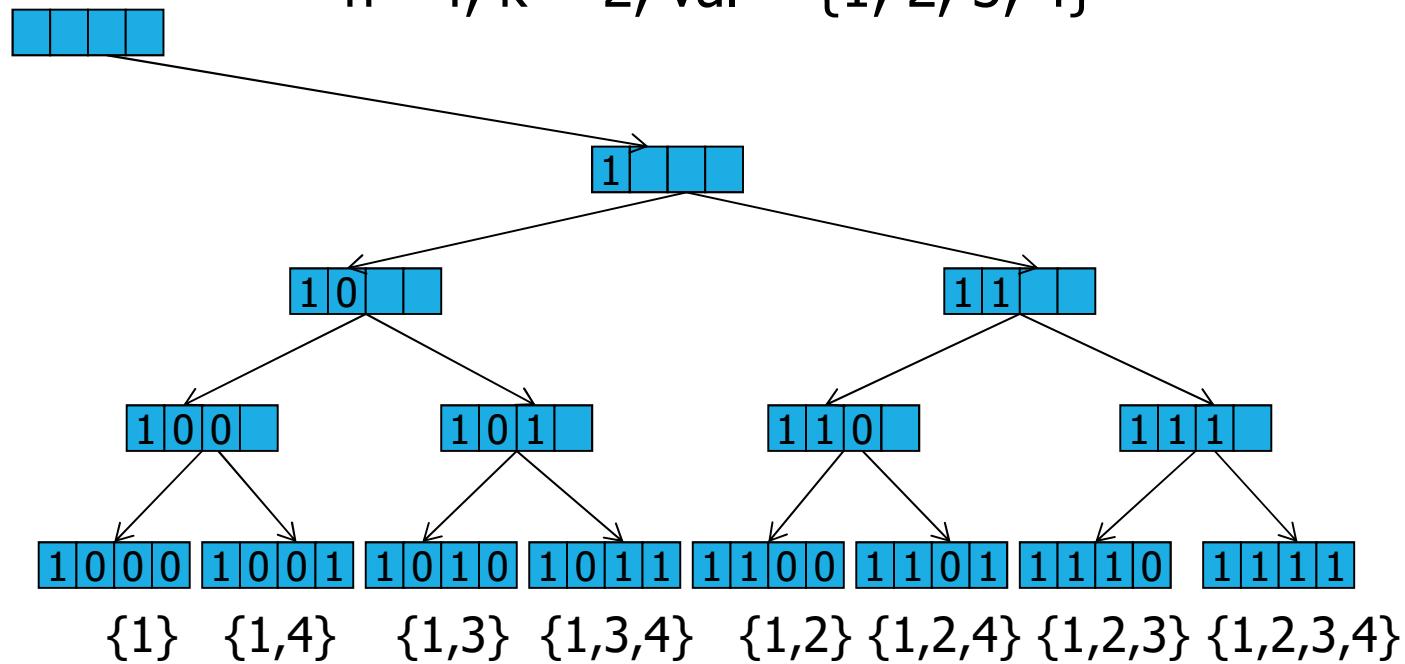
backtrack: prendi
l'elemento pos

ricorri su pos+1

$n = 4, k = 2, \text{val} = \{1, 2, 3, 4\}$



$n = 4, k = 2, \text{val} = \{1, 2, 3, 4\}$



Combinazioni semplici

- Unione di insieme vuoto e insieme delle parti degli insiemi con $j = 1, 2, 3, \dots, n$ elementi
- Trattandosi di insiemi l'ordine non conta
- Modello: combinazioni semplici di n elementi presi a gruppi di j

$$\wp(S) = \{ \emptyset \} \cup \bigcup_{j=1}^n \left\{ \binom{n}{j} \right\}$$

- il wrapper si occupa dell'unione dell'insieme vuoto (non generato dalle combinazioni) e dell'iterare la chiamata alla funzione ricorsiva delle combinazioni.

wrapper

```
int powerset(int *val, int n, int *sol){  
    int cnt = 0, j;  
    printf("{ }\n"); → insieme vuoto  
    cnt++;  
    for(j = 1; j <= n; j++){  
        cnt += powerset_r(val, n, sol, j, 0, 0);  
    }  
    return cnt;  
}
```

iterazione delle
chiamate ricorsive

```
int powerset_r(int* val, int n, int *sol, int j, int pos,int start){  
    int cnt = 0, i;  
    if (pos >= j){  
        printf("{ ");  
        for (i = 0; i < j; i++)  
            printf("%d ", sol[i]);  
        printf(" }\n");  
        return 1;  
    }  
    for (i = start; i < n; i++){  
        sol[pos] = val[i];  
        cnt += powerset_r(val, n, sol, j, pos+1, i+1);  
    }  
    return cnt;  
}
```

caso terminale: raggiunto
numero prefissato di elementi

per tutti gli elementi
da start in poi

Partizioni di un insieme S

Rappresentazione delle partizioni:

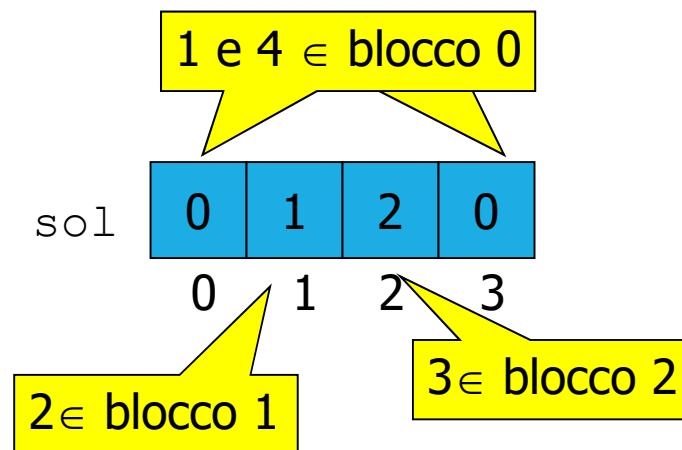
- dato l'elemento, si indica il blocco a cui appartiene univocamente
- dato il blocco, si elencano gli elementi (anche più d'uno) che vi appartengono.

La prima soluzione è preferita in quanto permette di usare vettori di interi.

I	1	2	3	4	val
	0	1	2	3	

Esempio

Se $I = \{1, 2, 3, 4\}$, $n = \text{card}(I)=4$ e si richiedono partizioni in $k = 3$ blocchi (i blocchi hanno indice 0, 1 e 2), la partizione $\{1, 4\}, \{2\}, \{3\}$ è rappresentata come:



Problemi

Dati I e $n = \text{card}(I)$, determinare:

- una partizione qualsiasi
- tutte le partizioni in k blocchi con k tra 1 e n
- tutte le partizioni in k blocchi.

disposizioni ripetute

algoritmo di Er

Disposizioni ripetute

- Il numero di oggetti memorizzati nel vettore val è n
- Il numero di decisioni da prendere è n, quindi il vettore sol contiene n celle
- Il numero delle scelte possibili per ogni oggetto è il numero di blocchi, che varia tra 1 e k
- Ogni blocco è identificato da un indice i compreso tra 0 e k-1
- sol[pos] contiene l'indice i del blocco cui appartiene l'oggetto di indice corrente pos.

- È scambiato il ruolo di n e k rispetto agli altri esempi (dove n era il numero di scelte e k la dimensione della soluzione)
- Si tratta di una generalizzazione del powerset rimuovendo il vincolo della scelta limitata a 0 oppure 1
- Necessità di un controllo nella condizione di terminazione per evitare blocchi vuoti (calcolo delle occorrenze di ciascun blocco)
- La funzione calcola tutte le partizioni. In seguito si vedrà come fermarsi alla prima.

```

void disp_ripet(int pos,int *val,int *sol,int n,int k) {
    int i, j, ok=1, *occ;
    if (pos >= n) {
        occ = calloc(k, sizeof(int));
        for (j=0; j<n; j++)
            occ[sol[j]]++;
        i=0;
        while ((i < k) && ok) {
            if (occ[i]==0) ok = 0;
            i++;
        }
        free(occ);
        if (ok == 0) return;
        else { /*STAMPA SOLUZIONE */ }
    }
    for (i = 0; i < k; i++) {
        sol[pos] = i; disp_ripet(pos+1, val, sol, n, k);
    }
}

```

vettore delle
occorrenze dei blocchi

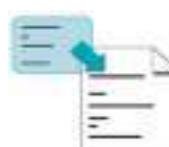
calcolo occorrenze

controllo occorrenze

val = malloc(n*sizeof(int));
sol = malloc(n*sizeof(int));

soluzione scartata

ricorsione



05part_simplif

Algoritmo di Er (1987)

Calcolo di tutte le partizioni di n oggetti memorizzati nel vettore val in k blocchi con k compreso tra 1 e n :

- indice pos per scorrere gli n oggetti e terminare la ricorsione quando $pos \geq n$
- indice i per scorrere gli m blocchi utilizzabili in quel passo
- vettore sol di n elementi per la soluzione

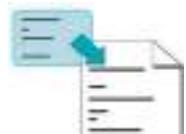
- 2 ricorsioni:
 - si attribuisce l'oggetto corrente a uno dei blocchi utilizzabili nel passo corrente (indice i tra 0 e $m-1$) e si ricorre sul prossimo oggetto ($pos+1$)
 - si attribuisce l'oggetto corrente al blocco m e si ricorre sul prossimo oggetto ($pos+1$) e su un numero di blocchi utilizzabili incrementato di 1 ($m+1$).

```

void SP_rec(int n,int m,int pos,int *sol,int *val) {
    int i, j;
    if (pos >= n) { condizione di terminazione
        printf("partizione in %d blocchi: ", m);
        for (i=0; i<m; i++)
            for (j=0; j<n; j++)
                if (sol[j]==i)
                    printf("%d ", val[j]);
        printf("\n");
        return;
    }
    for (i=0; i<m; i++) { ricorsione sugli oggetti
        sol[pos] = i;
        SP_rec(n, m, pos+1, sol, val);
    }
    sol[pos] = m; ricorsione su oggetti e blocchi
    SP_rec(n, m+1, pos+1, sol, val);
}

```

`val = malloc(n*sizeof(int));`
`sol = calloc(n, sizeof(int));`



06part_Er

Calcolo di tutte le partizioni di n oggetti memorizzati nel vettore `val` esattamente in k blocchi:

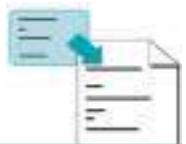
- come prima, passando il parametro k usato nella condizione di terminazione per “filtrare” le soluzioni accettate.

```

void SP_rec(int n,int k,int m,int pos,int *sol,int *val){
    int i, j;
    if (pos >= n) {           condizione di terminazione
        if (m == k) {          filtro
            for (i=0; i<m; i++)
                for (j=0; j<n; j++)
                    if (sol[j]==i)
                        printf("%d ", val[j]);
            printf("\n");
        }
        return;
    }
    for (i=0; i<m; i++) {
        sol[pos] = i;          ricorsione sugli oggetti
        SP_rec(n, k, m, pos+1, sol, val);
    }
    sol[pos] = m;             ricorsione su oggetti e blocchi
    SP_rec(n, k, m+1, pos+1, sol, val);
}

```

`val = malloc(n*sizeof(int));`
`sol = calloc(n, sizeof(int));`



07part_k_blocchi_Er

Esplorazione esaustiva dello spazio delle soluzioni: singola soluzione

Paolo Camurati



Singola soluzione

La ricorsione è particolarmente utile quando si vogliono elencare tutte le soluzioni e questo è obbligatorio nei problemi di ottimizzazione.

Se ne basta una sola, bisogna far sì che tutte le ricorsioni si chiudano, ricordando che ognuna torna a quella che l'ha chiamata.

È infatti errato pensare di poter «forare» la catena delle ricorsioni, tornando subito a quella iniziale.

Soluzioni:

1. uso di un flag:
 - variabile globale (soluzione sconsigliata)
 - parametro passato by reference
2. funzione ricorsiva che ritorna un valore di successo o fallimento che viene testato.

Uso di flag come parametro by reference:

- si definisce un flag `stop` (inizializzato a 0), il puntatore al quale è passato come parametro alla funzione ricorsiva:
 - in caso di terminazione con successo, `stop` è messo a 1
 - il ciclo sulle scelte ha nella condizione di esecuzione `stop==0`

```
/* main */
int stop = 0;
.....
funz_ric(....., &stop);

void funz_ric(....., int *stop_ptr) {
    .....
    if (condizione di terminazione) {
        .....
        (*stop_ptr) = 1;
        return;
    }
    for (i=0; condizione su i && (*stop_ptr)==0; i++) {
        .....
        funz_ric(....., stop_ptr);
        .....
    }
    return;
}
```

Funzione ricorsiva che ritorna un valore intero di successo/fallimento (versione senza pruning):

- nella condizione di terminazione, se la condizione di accettazione è verificata si ritorna 1, altrimenti si ritorna 0
- nel ciclo di scelta:
 - si effettua la scelta
 - si testa il risultato della chiamata ricorsiva: se c'è successo si ritorna 1
- terminato il ciclo di scelta: si ritorna 0.

nel main

```
if (funz_ric(.....)==0)
    printf("soluzione non trovata\n");
int funz_ric(.....) {
    if (condizione di terminazione)
        if (condizione di accettazione) {
            .....
            return 1;
        }
        return 0;
    }
    for (ciclo sulle scelte) {
        scelta;
        if (funz_ric(.....))
            return 1;
        .....
    }
    return 0;
}
```

Problemi di ottimizzazione

Paolo Camurati



Esplorando esaustivamente lo spazio delle soluzioni, si tiene traccia della soluzione fino al momento ottima, che si aggiorna eventualmente ad ogni passo.

È necessario generare tutte le soluzioni.

Il conto corrente

Input: vettore di interi di lunghezza nota n. Ogni intero rappresenta un movimento distinto su un conto bancario:

- >0: entrata
- <0: uscita.

Dato un ordine per i movimenti, il saldo corrente è il valore ottenuto sommando algebricamente al saldo precedente (inizialmente 0) l'importo del movimento.

Per ogni ordinamento dei movimenti ci sarà un saldo corrente massimo e un saldo corrente minimo, mentre il saldo finale sarà ovviamente lo stesso, qualunque sia l'ordine.

Determinare l'ordine dei movimenti che minimizza la differenza tra saldo corrente massimo e saldo corrente minimo.

Esempio

Dati $n=10$ e $\text{val}=\{-1,-6,3,14,-5,16,7,8,-9,120\}$

- ordinamento $\{3,-1,14,-6,-5,16,7,8,-9,120\}$

corr.	0	3	2	16	10	5	21	28	36	27	147
max	0	3	3	16	16	16	21	28	36	36	147
min	0	3	2	2	2	2	2	2	2	2	2
Δ	0	0	1	14	14	14	19	26	34	34	145

- ordinamento $\{120,3,-1,14,-6,-5,16,-9,7,8\}$

corr.	0	120	123	122	136	130	125	141	132	139	147
max	0	120	123	123	136	136	136	141	141	141	147
min	0	120	120	120	120	120	120	120	120	120	120
Δ	0	0	3	3	16	16	16	21	21	21	27

minimo

- Modello: permutazioni semplici per enumerare gli ordinamenti
- Enumerazione di tutte le soluzioni
- Funzione **check** per valutare l'ottimalità di ogni soluzione.

Algoritmo:

- algoritmo ricorsivo per le permutazioni semplici
- quando si è raggiunta la condizione di terminazione:
 - si calcola il saldo max e min e della differenza corrente
 - si confronta la differenza corrente con la minima sinora trovata
 - eventualmente si aggiorna la soluzione.
- Ipotesi:
- è noto un limite superiore alla massima differenza minima (= INT_MAX).

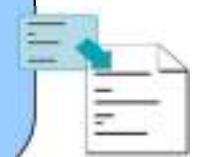
```

// #include anche di <limits.h>
int min_diff = INT_MAX;
void perm(int pos,int *val,int *sol,int *mark,int *fin,int n);
void check(int *sol, int *fin, int n);

int main(void) {
    int i, n, *val, *sol, *mark, *fin;
    printf("Inserisci n: "); scanf("%d", &n);
    // allocazione di val, sol, mark e fin di n interi
    for (i=0; i < n; i++) { sol[i]=-1; mark[i]= 0; }
    // leggi valori in val
    perm(0, val, sol, mark, fin, n);
    // stampa risultato da fin
    // free di val, sol, mark e fin
    return 0;
}

```

variabile globale



08contocorrente

non serve il numero delle permutazioni

```
void perm(int pos, int *val, int *sol, int *mark, int *fin, int n) {  
    int i;  
    if (pos >= n) {  
        check(sol, fin, n);  
        return;  
    }  
    for (i=0; i<n; i++)  
        if (mark[i] == 0) {  
            mark[i] = 1;  
            sol[pos] = val[i];  
            perm(pos+1, val, sol, mark, fin, n);  
            mark[i] = 0;  
        }  
    return;  
}
```

condizione di terminazione

controllo ottimalità soluzione

generazione delle permutazioni

```

void check(int *sol, int *fin, int n) {
    int i, saldo=0, max_curr=0, min_curr=INT_MAX, diff_curr;
    for (i=0; i<n; i++) {
        saldo += sol[i]; calcolo del saldo
        if (saldo > max_curr)
            max_curr = saldo; aggiornamento massimo e minimo
        if (saldo < min_curr)
            min_curr = saldo;
    } calcolo della differenza
    diff_curr = max_curr - min_curr;
    if (diff_curr < min_diff) {
        min_diff = diff_curr; controllo di ottimalità
        for (i=0; i<n; i++)
            fin[i] = sol[i]; aggiornamento soluzione
    }
    return;
}

```

Lo zaino (discreto)

Dato un insieme di N oggetti ciascuno dotato di peso w_j e di valore v_j e dato un peso massimo cap , determinare il sottoinsieme S di oggetti tali che:

- $\sum_{j \in S} w_j x_j \leq \text{cap}$
- $\sum_{j \in S} v_j x_j = \text{MAX}$
- $x_j \in \{0,1\}$

Ogni oggetto o è preso ($x_j = 1$) o lasciato ($x_j = 0$). Ogni oggetto esiste in una sola instanziazione.

Esempio

N= 4

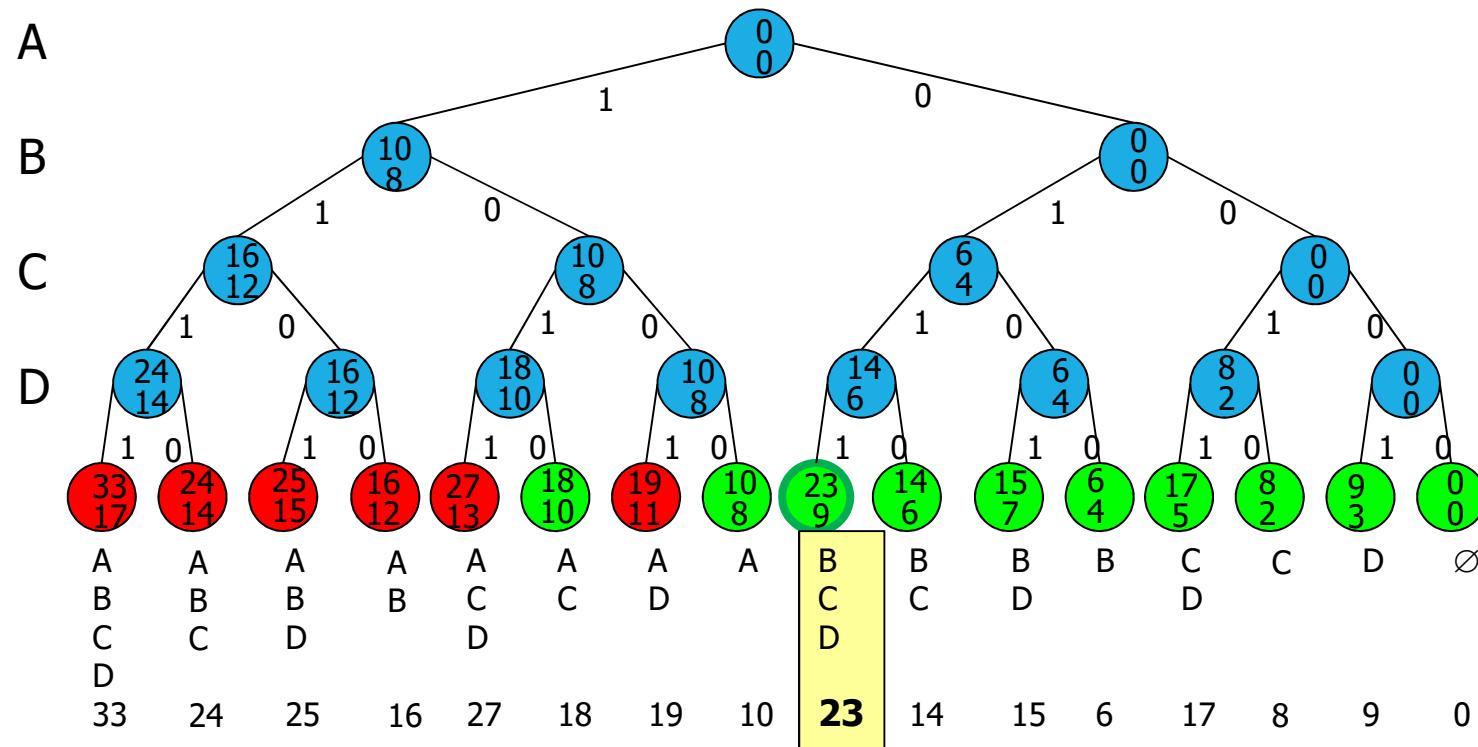
cap = 10

	Nome	A	B	C	D
Valore v _i	10	6	8	9	
Peso w _i	8	4	2	3	

Soluzione:

insieme {B, C, D} con valore massimo 23

Modello: powerset (disposizioni ripetute)



Strutture dati

- Strutture già viste per le disposizioni ripetute e inoltre:
- Variabili intere:
 - `cap` per la capacità dello zaino
 - `c_val` per il valore corrente (`c` = current)
 - `c_cap` per la capacità usata correntemente
 - `b_val` valore ottimo corrente (`b` = best)
- Vettore di interi `b_sol` per la soluzione ottima corrente.

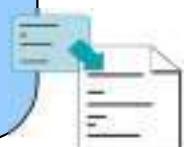
```

void powerset(int pos, Item *items, int *sol, int k, int cap,
    int c_cap, int c_val, int *b_val, int *b_sol) {

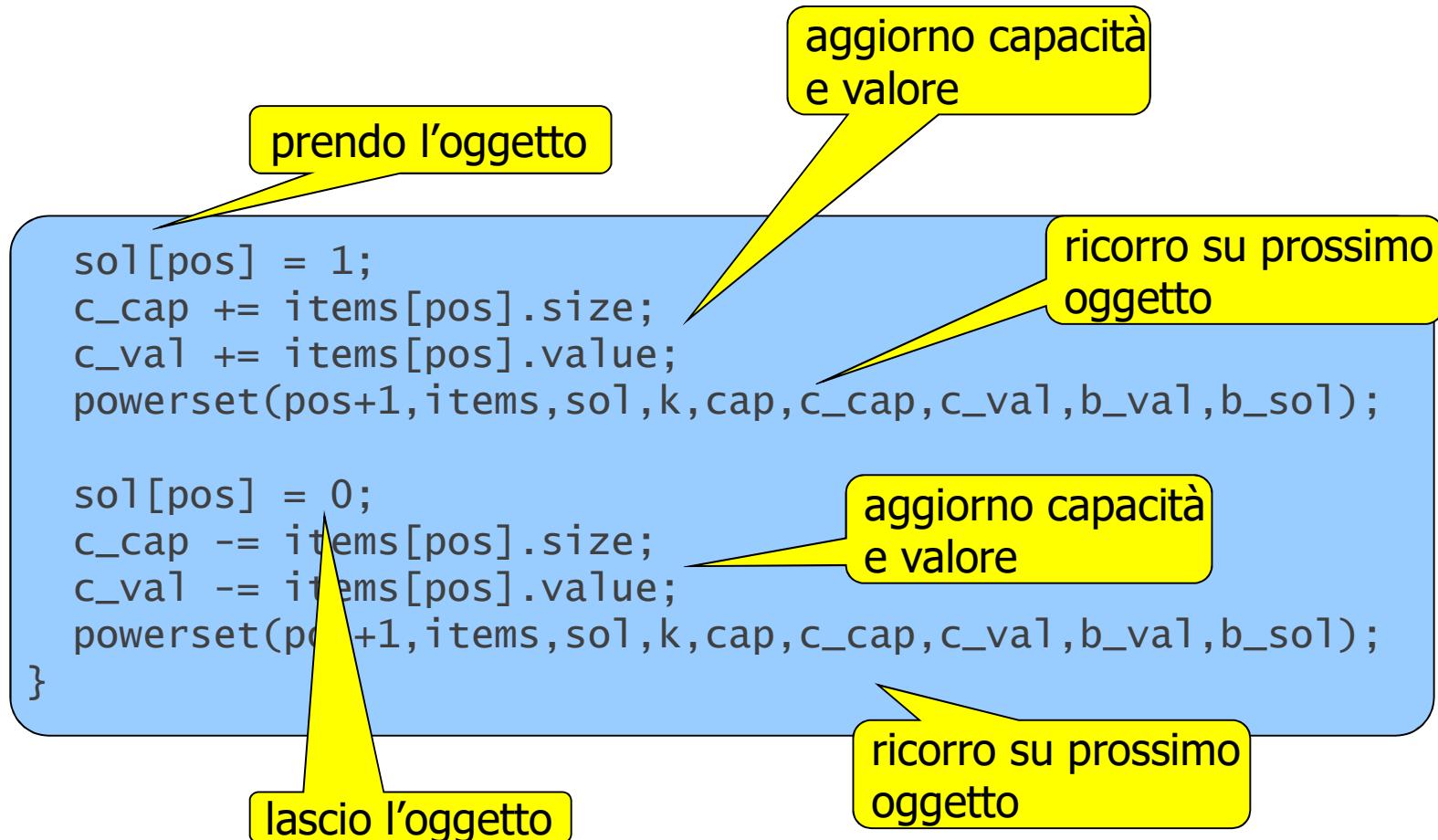
    int j;           condizione di terminazione

    if (pos >= k) { controllo accettabilità
        if (c_cap <= cap) {
            if (c_val > *b_val) { controllo ottimalità
                for (j=0; j<k; j++)
                    b_sol[j] = sol[j];
                *b_val = c_val;
            }
        }
    }
    return;
}

```



09knapsack_no_pruning



Il pruning dello spazio delle soluzioni

Paolo Camurati



- Criterio di accettazione della soluzione espresso mediante vincoli
- Vincoli valutati:
 - direttamente nei casi terminali, senza specifica struttura dati
 - ad ogni chiamata ricorsiva, mediante struttura dati aggiornata dinamicamente
- Crescita molto rapida dello spazio delle soluzioni \Rightarrow inapplicabilità dell'approccio enumerativo

Puzzle di Einstein:

- modello: principio di moltiplicazione
- numero di decisioni da prendere $n=5$
- scelte disponibili per ogni decisione: permutazioni semplici di 5 alternative ($5!$ scelte)
- dimensione dello spazio di ricerca: $(5!)^5 = 24.883.200.000$
- impossibile valutare i vincoli solo nel caso terminale!

- Osservazioni:
- delle $5!$ scelte sulla nazionalità, solo $(5-1)!$ hanno il norvegese nella prima casa
- perché considerare anche quelle che certamente porteranno a soluzioni inaccettabili (ad esempio il norvegese non nella prima casa)?
- scartarle non inficia la completezza della ricerca!



PRUNING

Pruning:

- riduzione dello spazio di ricerca
- nessuna perdita di completezza
- scarto a priori dei rami dell'albero che non possono portare a soluzioni valide/ottime.

I vincoli permettono di:

- escludere a priori strade che non portano a soluzioni accettabili
- anticipare il test di accettazione fatto nella condizione di terminazione in modo da subordinare ad esso la discesa ricorsiva.

La somma di sottoinsiemi

Dato un insieme S di numeri interi positivi distinti e un intero X , determinare tutti i sottoinsiemi Y di S tale che la somma degli elementi di Y sia uguale a X .

Esempio: $S = \{2, 1, 6, 4\}$ $X = 7$

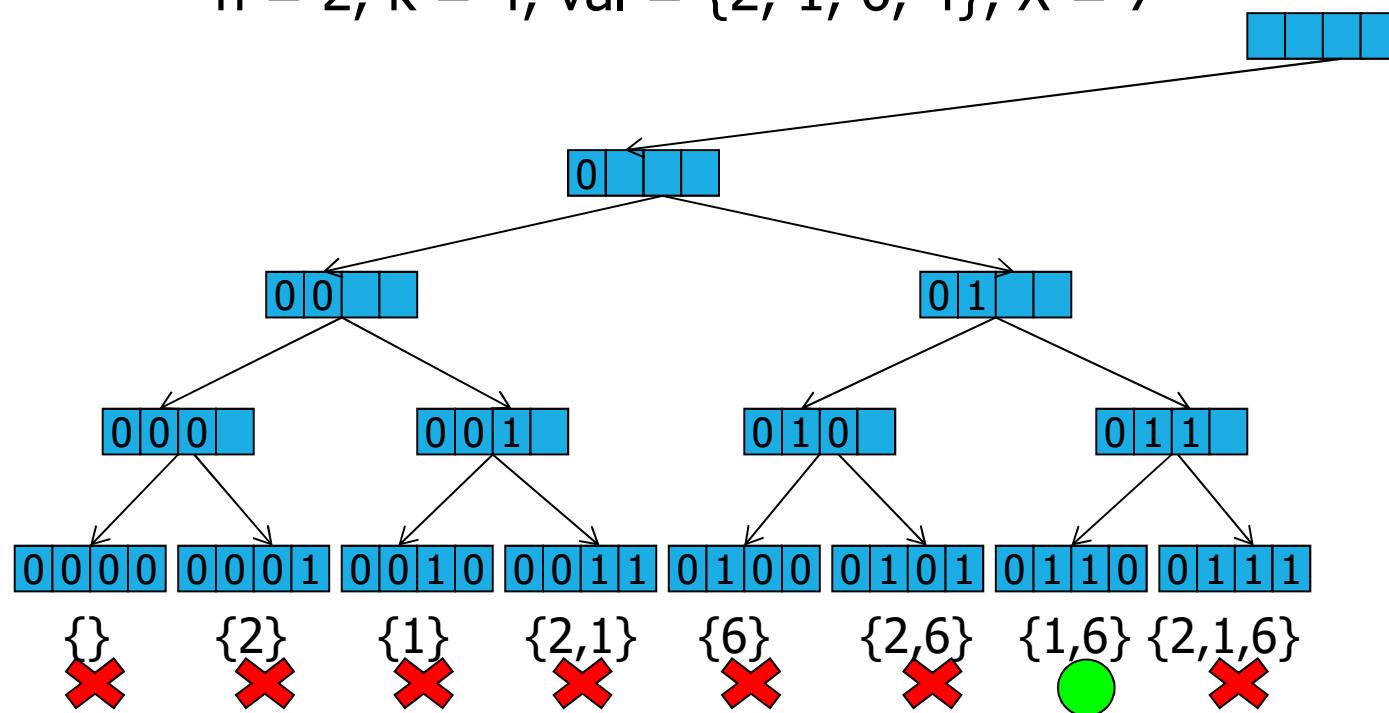
Soluzione:

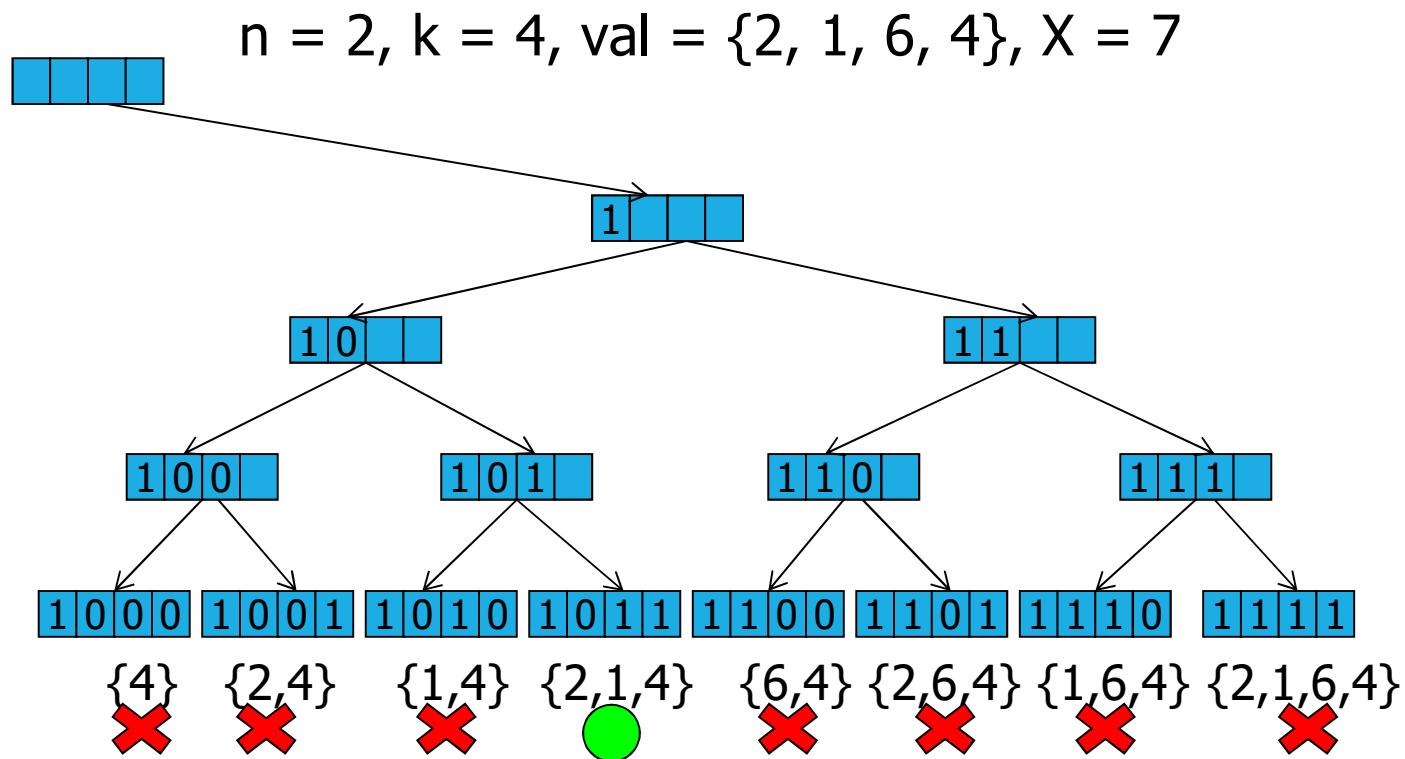
$$Y = \{ \{1,2,4\}, \{1,6\} \}$$

Approccio enumerativo (senza pruning):

- calcolare il powerset $\wp(\text{val})$ (disposizioni ripetute)
- per ogni sottoinsieme (condizione di terminazione), verificare se la somma dei suoi elementi è X.

$n = 2, k = 4, \text{val} = \{2, 1, 6, 4\}, X = 7$





```

void powerset(int pos,int *val,int *sol,int k,int x){
    int j, out;
    if (pos >= k) { terminazione
        out = check(sol, val, x, k); verifica soluzione
        if (out==1) { stampa soluzione
            etc.etc.
        }
    }
}

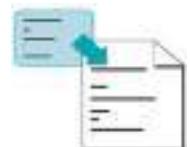
```

```

int check(int *sol, int *val, int x, int k) {
    int j, tot=0;
    for (j=k-1; j>=0; j--)
        if (sol[j]!=0)
            tot += val[k-j-1];
    if (tot==x)
        return 1;
    return 0;
}

```

`val = malloc(k * sizeof(int));
sol = calloc(k, sizeof(int));`



10simple_sum_of_subsets

Il Pruning

- Anticipazione della valutazione dei vincoli in uno stato intermedio
- Non c'è una metodologia generale
- Casi tipici:
 - filtro **statico** sulle scelte: condizioni di accettazione che non dipendono dalle scelte precedenti, ma solo dal problema (ad esempio condizioni ai bordi in una mappa)
 - filtro **dinamico** sulle scelte: condizioni di accettazione che dipendono dalle scelte precedenti e dal problema (ad esempio la posizione di altri pezzi nel gioco)
 - **validazione** di una soluzione parziale: valutazione della speranza di raggiungere una soluzione o condizione sufficiente per decidere che la soluzione non può essere raggiunta.

La somma di sottoinsiemi

Approccio con pruning: strategia basata sulla valutazione della speranza:

- si ordina in modo crescente val
- p è la somma corrente ($p = \text{partial sum}$), inizialmente 0
- r , inizialmente pari alla somma di tutti i valori di val , contiene la somma dei valori non ancora presi, quindi ancora disponibili ($r = \text{remaining sum}$)
- ad ogni passo si prende in considerazione un elemento di val solo se “promettente”.

Un elemento di val è promettente se:

- la soluzione parziale + i valori che restano sono \geq della somma cercata

$$p + r \geq X \\ \&&$$

- la soluzione parziale + il valore di val è \leq della somma cercata

$$p + val[pos] \leq X$$

Se l'elemento è promettente:

- lo si prende ($\text{sol}[\text{pos}] = 1$)
- si ricorre sul prossimo ($\text{pos}+1$), aggiornando p ($p+\text{val}[\text{pos}]$) e r ($r-\text{val}[\text{pos}]$)
- in fase di backtrack, non lo si prende ($\text{sol}[\text{pos}] = 0$)
- si ricorre sul prossimo ($\text{pos}+1$), p resta invariato, r viene aggiornato ($r-\text{val}[\text{pos}]$)

deciso di non prendere l'elemento

Se l'elemento non è promettente, essendo il vettore `val` ordinato, anche tutti gli elementi che lo seguono sono «a fortiori» non promettenti:

- se $p + r < X$ tale somma non potrà aumentare considerando il prossimo elemento, indipendentemente dall'ordine
- se $p + val[pos] > X$, poiché l'elemento successivo è $>$ di quello corrente, la condizione $\leq X$ non potrà mai essere soddisfatta.

val= 

pos=0 val[pos]=1
p=0 r=13
0+13>=7 && 0+1<=7

sol= 



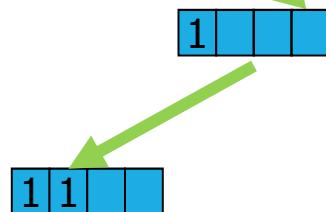


Promising: $p + r \geq x \text{ && } p + val[pos] \leq x$

val= 

pos=1 val[pos]=2
p=1 r=12
1+12>=7 && 1+2<=7

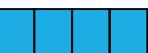
sol= 

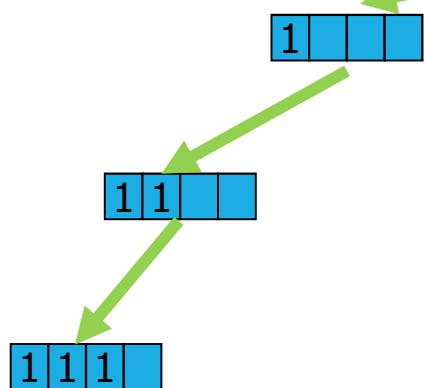


Promising: $p + r \geq x \text{ && } p + \text{val}[\text{pos}] \leq x$

val= 

pos=2 val[pos]=4
p=3 r=10
3+10>=7 && 3+4<=7

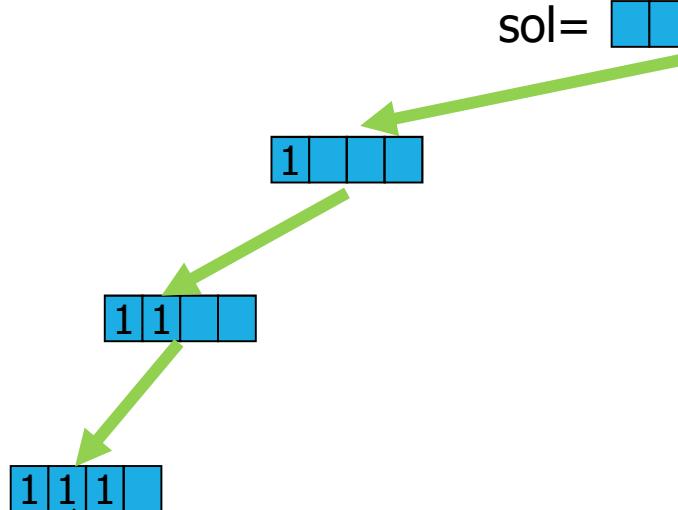
sol= 



Promising: $p + r \geq x \ \&\& \ p + val[pos] \leq x$

val= 

sol= 

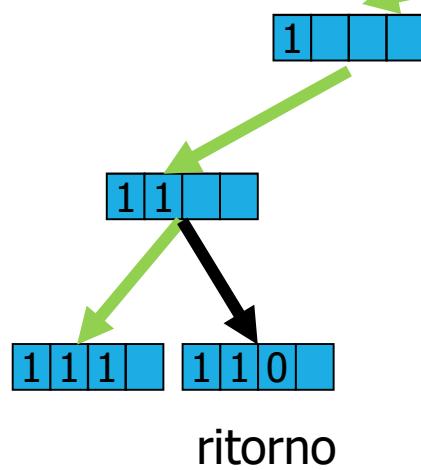


Promising: $p + r \geq x \text{ && } p + val[pos] \leq x$

val= 

pos=3 val[pos]=6
p=3 r=6
3+6>=7 && 3+6>7

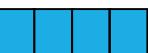
sol= 

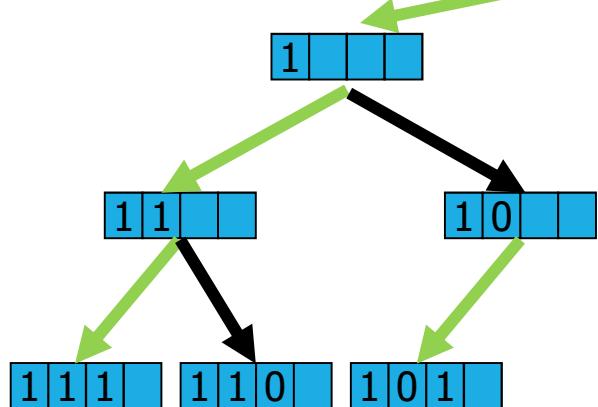


Promising: $p + r \geq x \text{ && } p + \text{val}[\text{pos}] \leq x$

val= 

pos=2 val[pos]=4
p=1 r=10
1+10>=7 && 1+4<=7

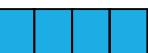
sol= 

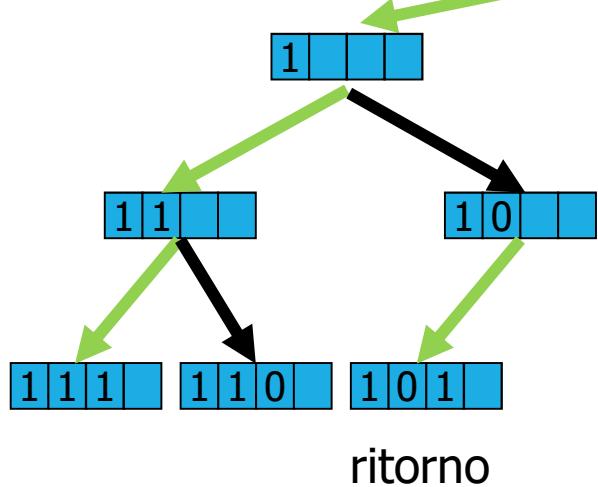


Promising: $p + r \geq x \ \&\& \ p + val[pos] \leq x$

val= 

pos=3 val[pos]=6
p=5 r=6
5+6 >= 7 && 5+6>7

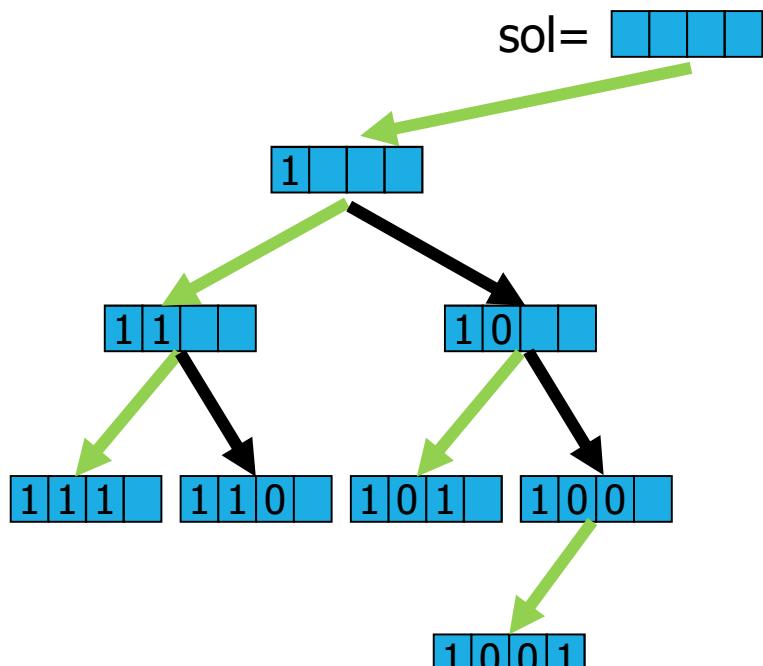
sol= 



Promising: $p + r \geq x \text{ && } p + \text{val}[\text{pos}] \leq x$

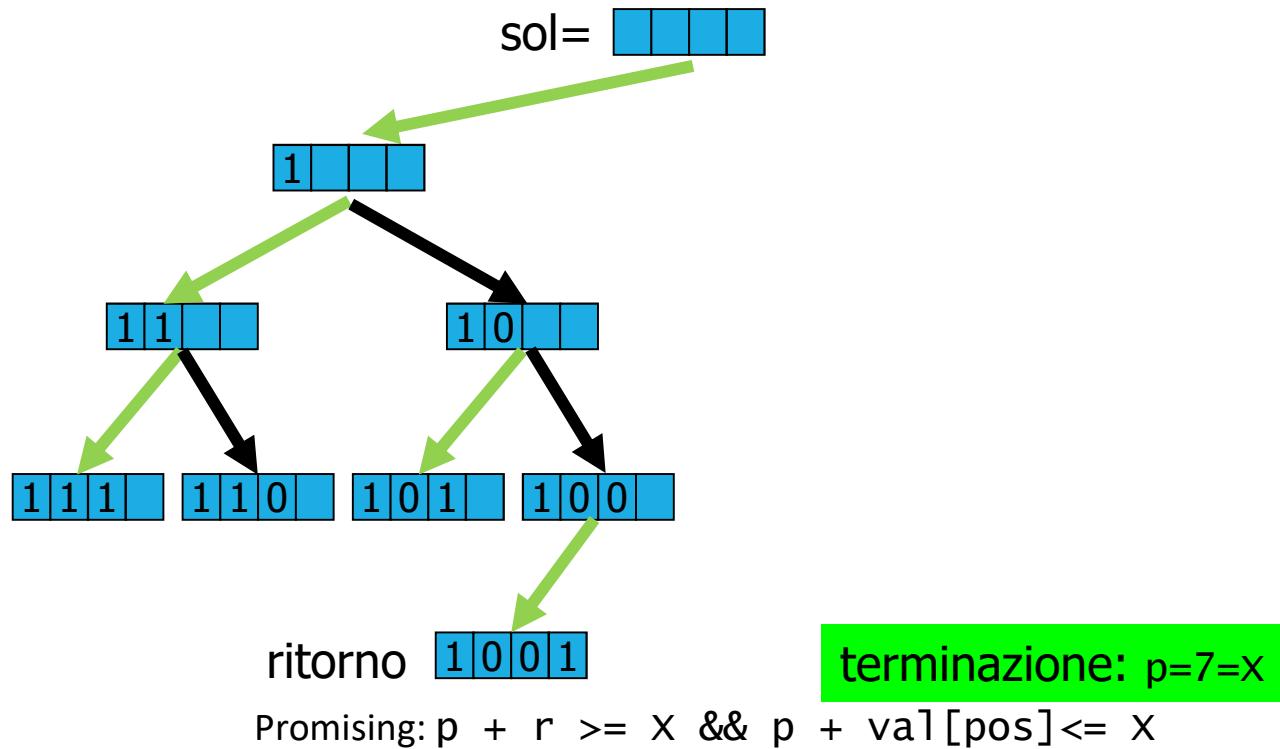
val= 

pos=3 val[pos]=6
p=1 r=6
1+6>=7 && 1+6<=7

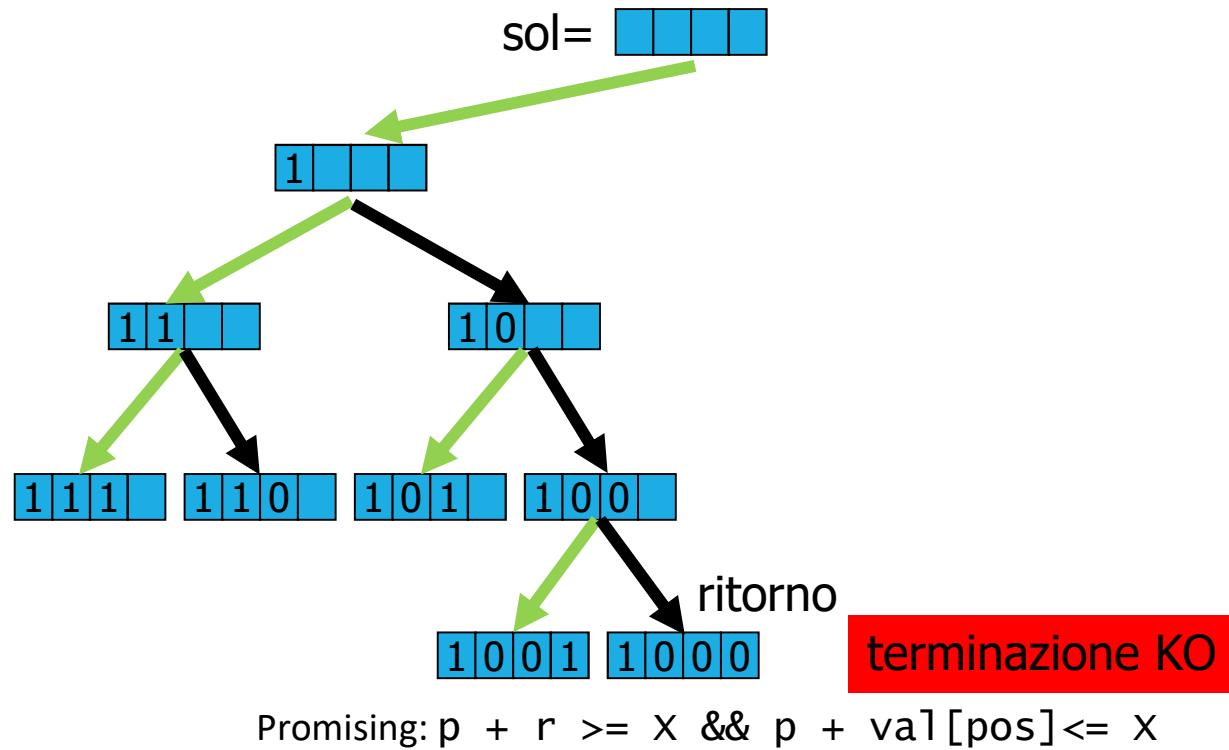


Promising: $p + r \geq x \text{ && } p + \text{val}[pos] \leq x$

val= 

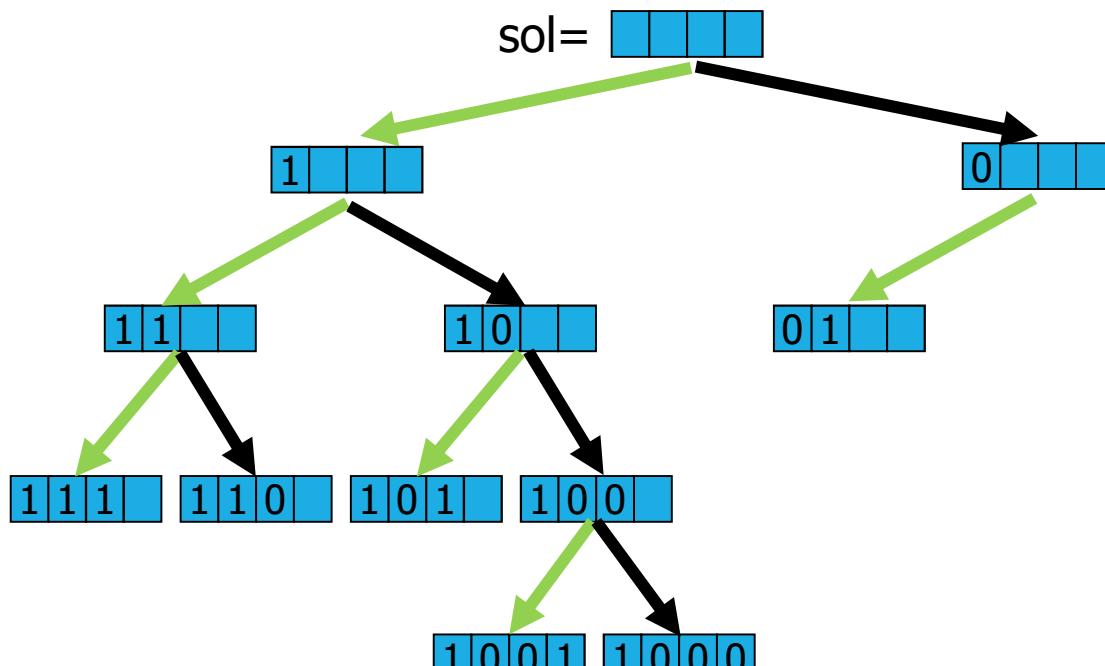


val= 



val= 

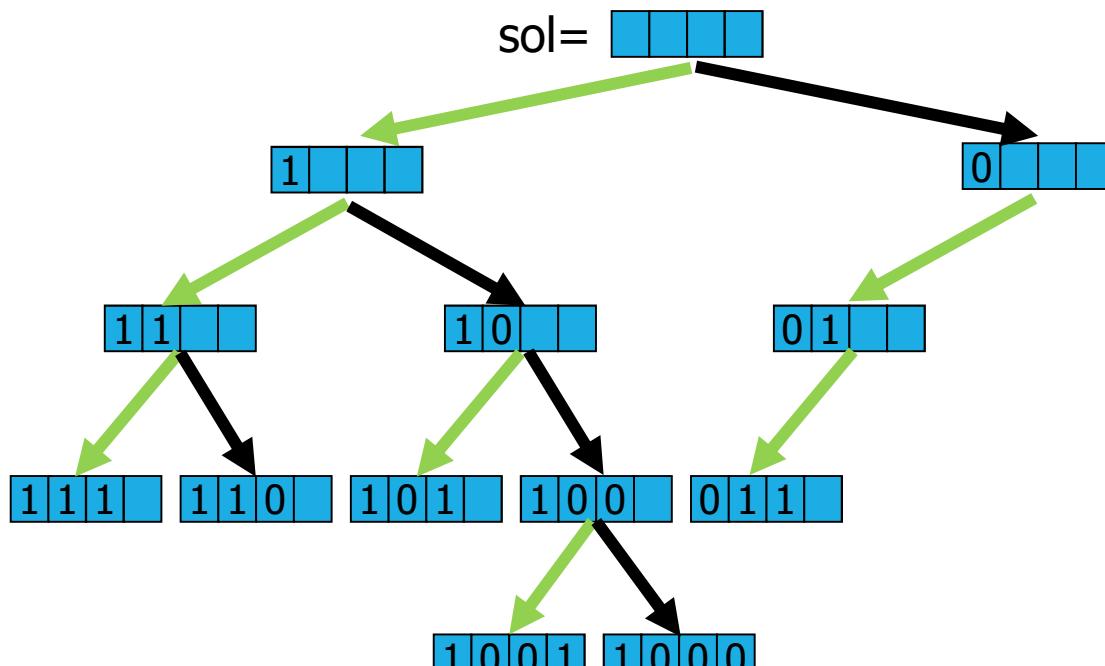
pos=1 val[pos]=2
p=0 r=12
0+12 >= 7 && 0+2<=7



Promising: $p + r \geq x \text{ && } p + \text{val}[\text{pos}] \leq x$

val= 

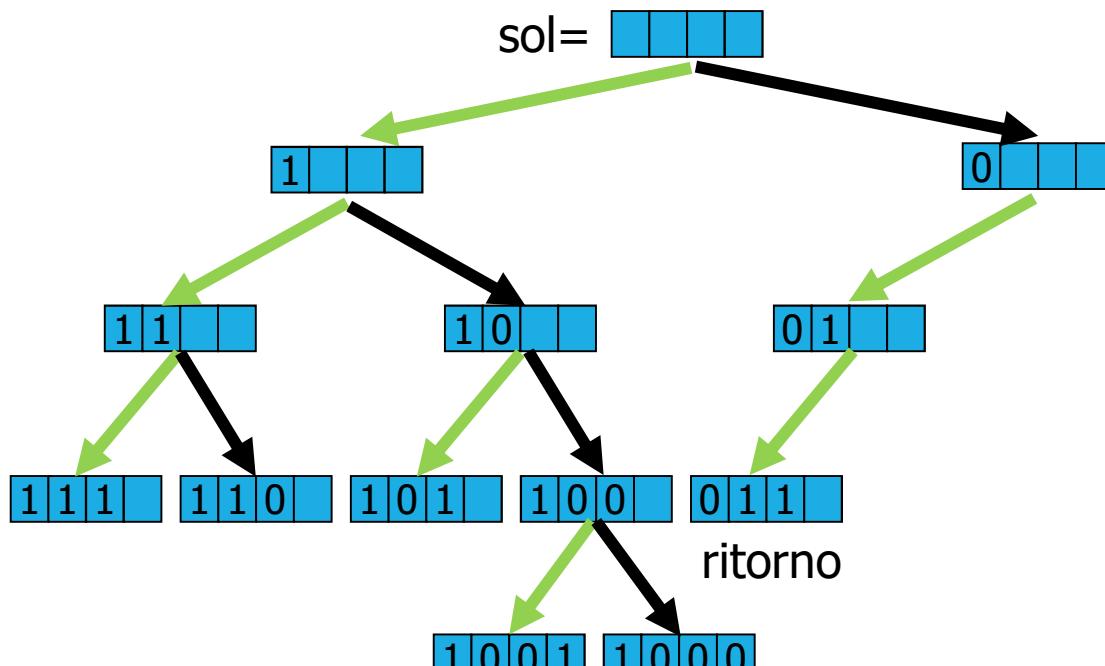
pos=2 val[pos]=4
p=2 r=10
2+10 >= 7 && 2+4<=7



Promising: $p + r \geq x \text{ && } p + \text{val}[pos] \leq x$

val= 

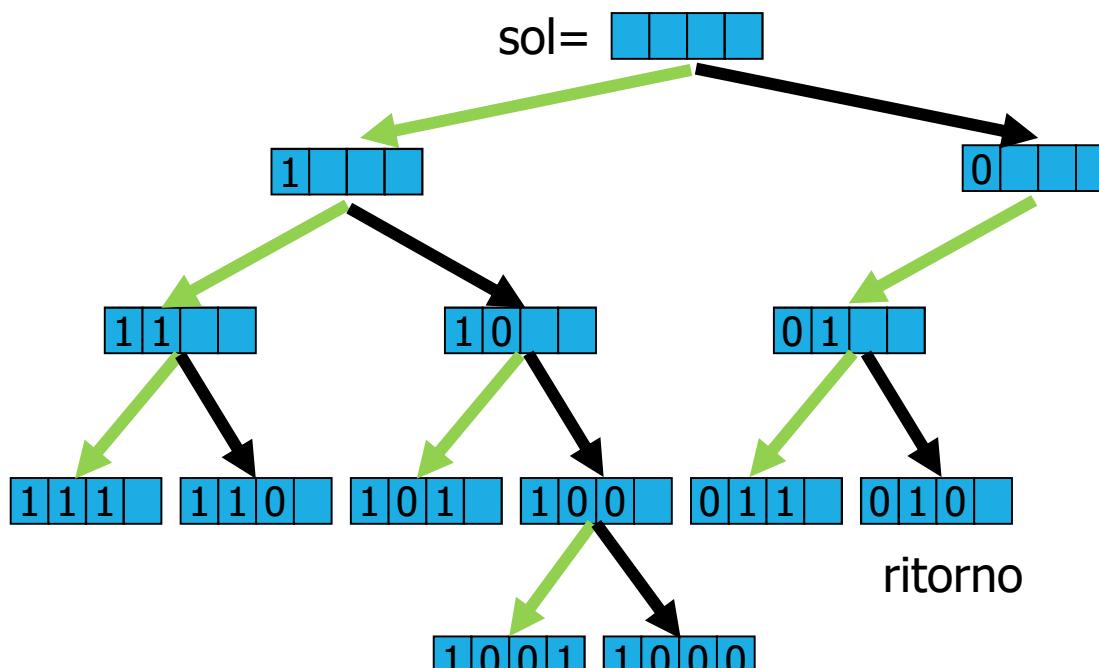
pos=3 val[pos]=6
p=6 r=6
6+6 >= 7 && 6+6>7



Promising: $p + r \geq x \text{ && } p + \text{val}[\text{pos}] \leq x$

val= 

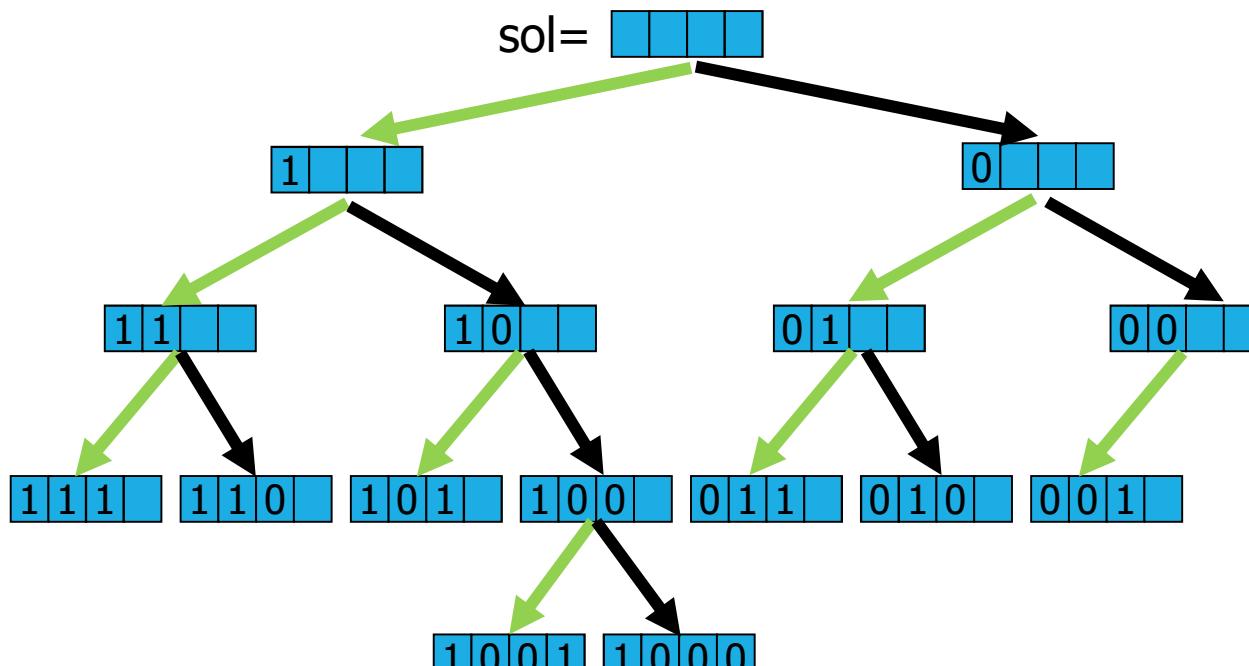
pos=3 val[pos]=6
p=2 r=6
2+6 >= 7 && 2+6>7



Promising: $p + r \geq x \text{ && } p + \text{val}[pos] \leq x$

val= 

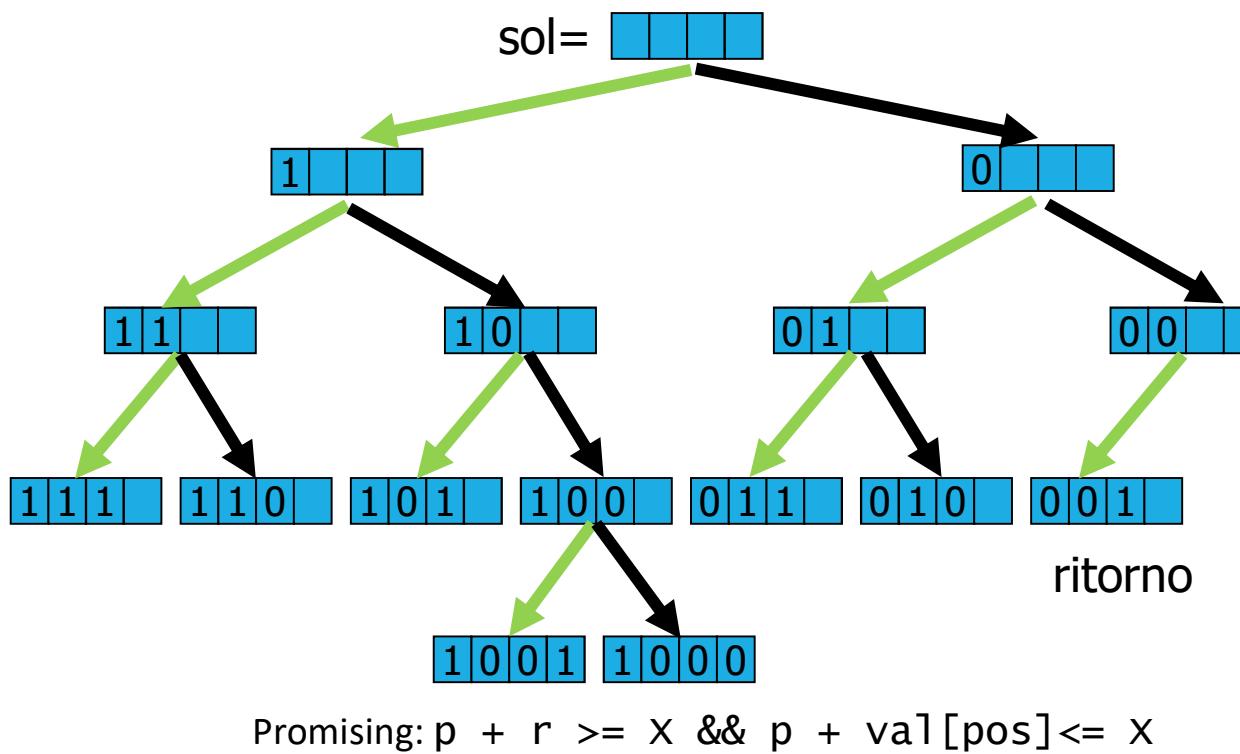
pos=2 val[pos]=4
p=0 r=10
0+10>=7 && 0+4<=7



Promising: $p + r \geq x \text{ & } p + \text{val}[\text{pos}] \leq x$

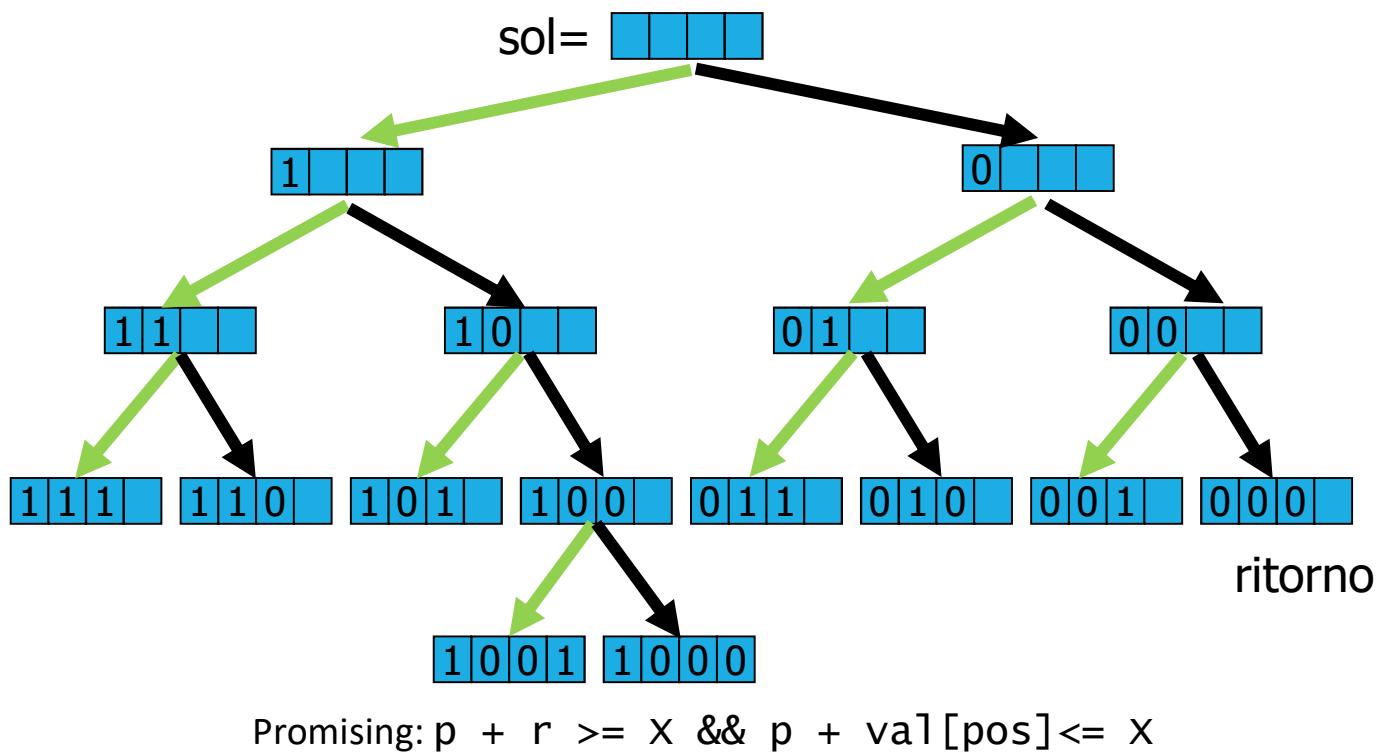
val= 

pos=3 val[pos]=6
p=4 r=6
4+6>=7 && 4+6>7



val= 

pos=3 val[pos]=6
p=0 r=6
0+6<7 && 0+6<=7



```

void sumset(int pos,int *val,int *sol,int p,int r,int x) {
    int j;
    if (p==x) {                                terminazione
        printf("\n{\t");
        for(j=0;j<pos;j++)
            if(sol[j])
                printf("%d\t",val[j]);
        printf("}\n");
        return;
    }
    if(promising(val,pos,p,r,x)){              stampa soluzione
        sol[pos]=1;                            controlla se promettente
        sumset(pos+1,val,sol,p+val[pos],r-val[pos],x);
        sol[pos]=0;                            prendi
        sumset(pos+1,val,sol,p, r-val[pos],x);  ricorri
    }
}

```

val = malloc(k*sizeof(int));
sol = malloc(k*sizeof(int));



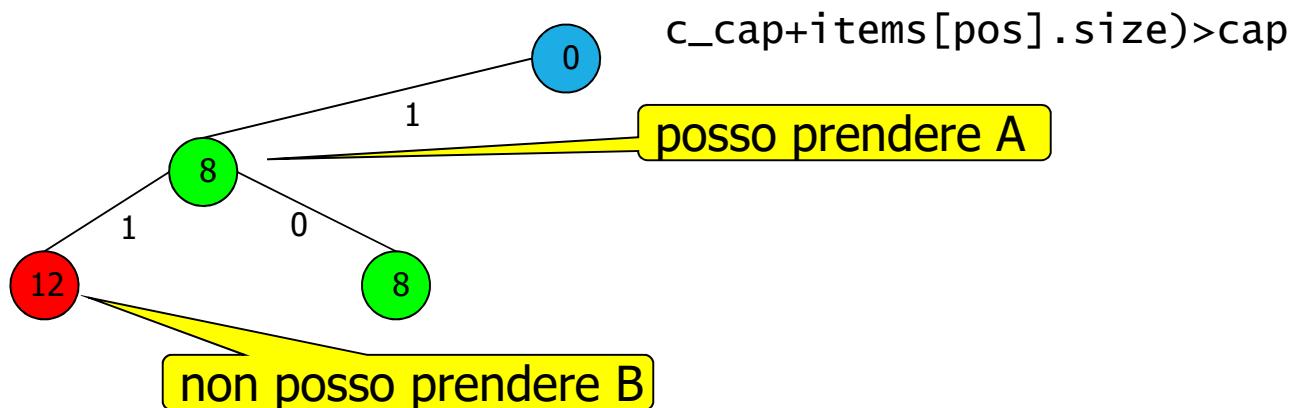
11sum_of_subsets

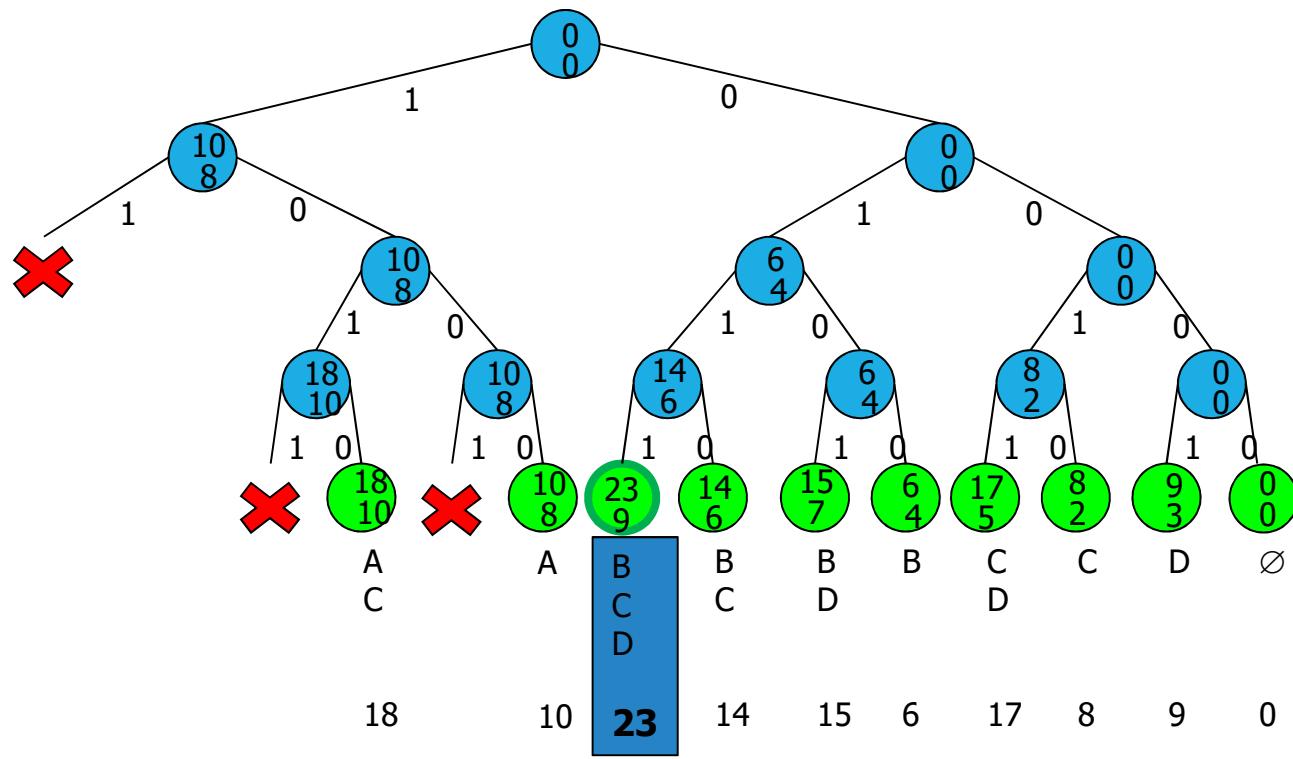
```
int promising(int *val,int pos,int p,int r,int x) {  
    return (p+r >=x)&&(p+val[pos]<=x);  
}
```

Lo zaino (discreto)

Approccio con pruning: disposizioni ripetute (powerset)

funzione di pruning: se, prendendo un oggetto, la capacità utilizzata eccede quella massima, l'oggetto non viene scelto





```

void powerset(int pos, Item *items, int *sol, int k, int cap,
    int c_cap, int c_val, int *b_val, int *b_sol) {
    int j;
    if (pos >= k) { terminazione
        if (c_val > *b_val) { controllo ottimalità
            for (j=0; j<k; j++)
                b_sol[j] = sol[j];
            *b_val = c_val;
        }
        return; controllo pruning
    }
    if ((c_cap + items[pos].size) > cap) {
        sol[pos] = 0; lascio oggetto
        powerset(pos+1, items, sol, k, cap, c_cap, c_val, b_val, b_sol);
        return;
    }
} ritorno
ricorro su prossimo oggetto

```



12knapsack_pruning

```
    sol[pos] = 1;
    c_cap += items[pos].size;
    c_val += items[pos].value;
    powerset(pos+1,items,sol,k, cap,c_cap,c_val,b_val,b_sol);
```

```
    sol[pos] = 0;
    c_cap -= items[pos].size;
    c_val -= items[pos].value;
    powerset(pos+1,items,sol,k, cap,c_cap,c_val,b_val,b_sol);
```

```
}
```

lascio oggetto

prendo oggetto

aggiorno capacità e valore

ricorro su prossimo
oggetto

aggiorno capacità
e valore

ricorro su prossimo
oggetto

Riferimenti

- Backtracking
 - Bertossi 16
- Permutazioni e sottoinsiemi
 - Bertossi 16.3

Esempi di problemi di ricerca e ottimizzazione

Paolo Camurati



Controllo di lampadine

Specifiche:

- n interruttori e m lampadine
- inizialmente tutte le lampadine sono spente
- ogni interruttore comanda un sottoinsieme delle lampadine:
 - un elemento $[i,j]$ di una matrice di interi $n \times m$ indica se vale 1 che l'interruttore i controlla la lampadina j , 0 altrimenti
- se un interruttore è premuto, tutte le lampadine da esso controllate **commutano** di stato

problema di ottimizzazione

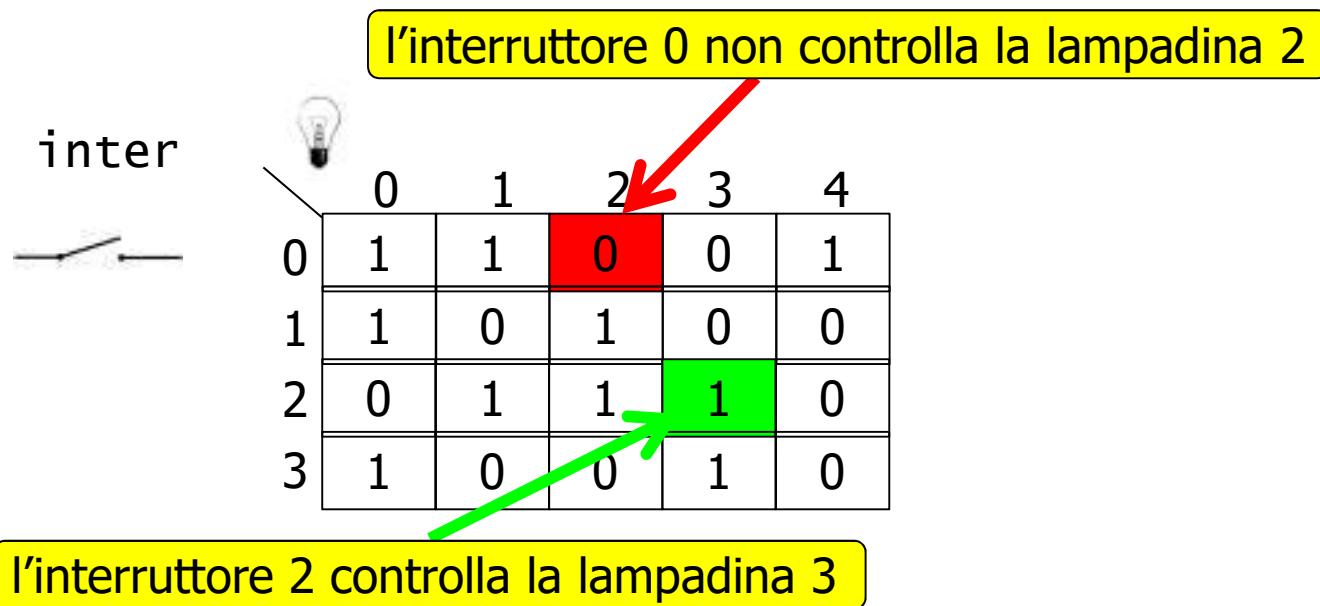
Scopo:

- determinare l'insieme **minimo** di interruttori da premere per accendere tutte le lampadine

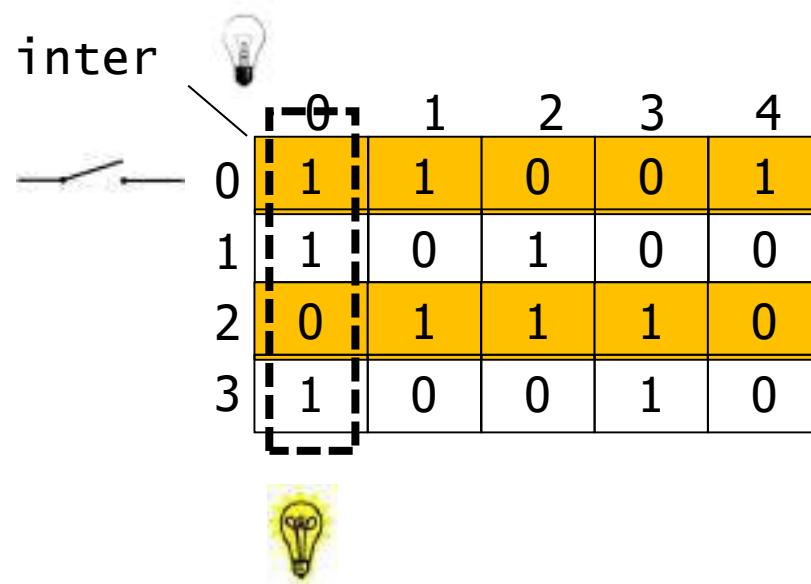
Condizione di accensione:

- una lampadina è accesa se e solo se il numero di interruttori premuti tra quelli che la controllano è dispari.

Esempio: $n=4$ $m=5$



Effetto degli interruttori 0 e 2 premuti:



int0 controlla lamp0

int2 non controlla lamp0

interruttori premuti
che controllano lamp0

1

Effetto degli interruttori 0 e 2 premuti:

int0
int2

inter	int0	int2	3	4
0	1	1	0	0
1	1	0	1	0
2	0	1	1	1
3	1	0	0	1



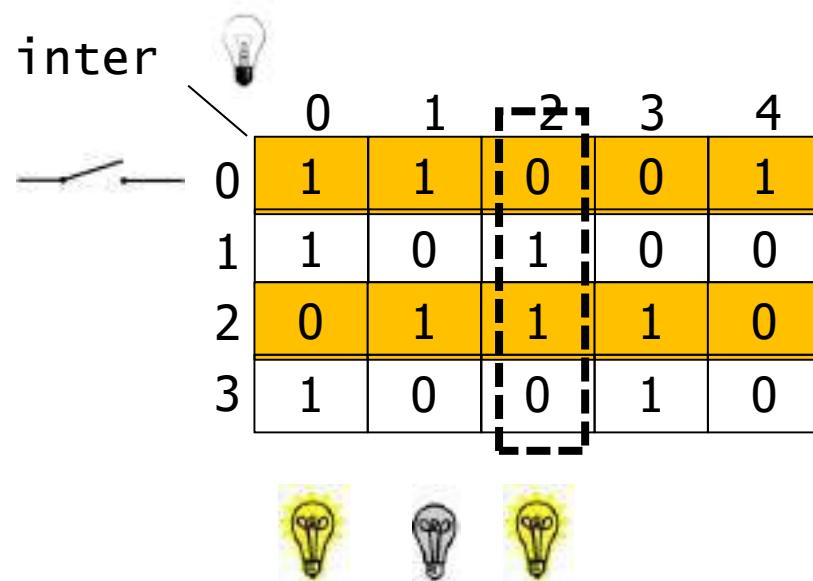
int0 controlla lamp1

int2 controlla lamp1

interruttori premuti
che controllano lamp1

2

Effetto degli interruttori 0 e 2 premuti:



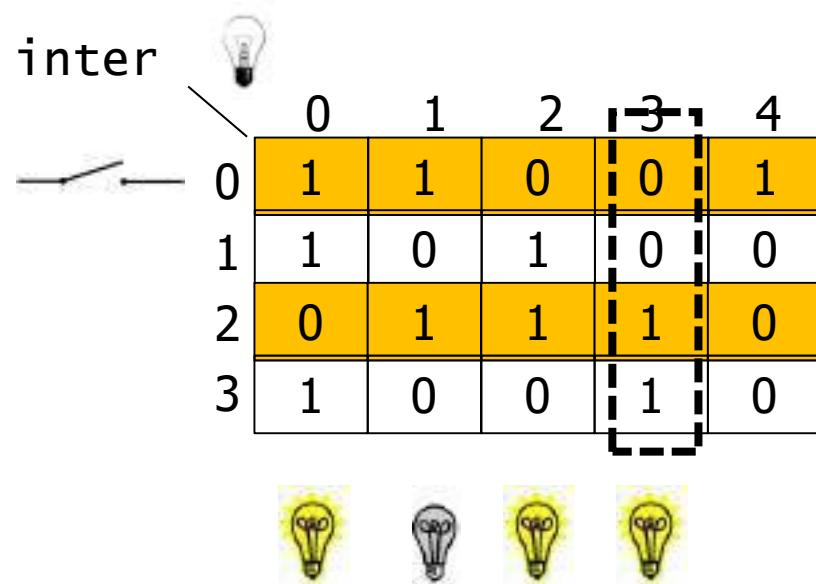
int0 non controlla lamp2

int2 controlla lamp2

interruttori premuti
che controllano lamp2

1

Effetto degli interruttori 0 e 2 premuti:



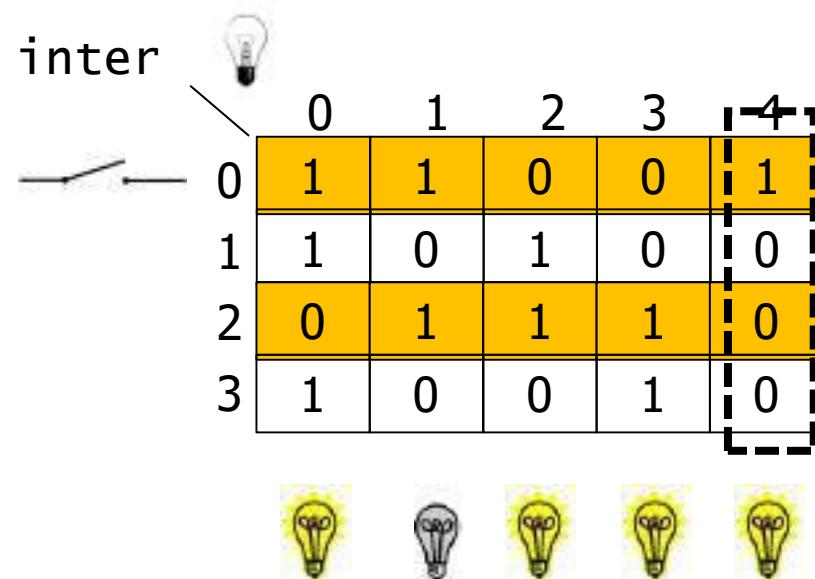
int0 non controlla lamp3

int2 controlla lamp3

interruttori premuti
che controllano lamp3

1

Effetto degli interruttori 0 e 2 premuti:



int0 controlla lamp4

int2 non controlla lamp4

interruttori premuti
che controllano lamp4

1

SOLUZIONE NON VALIDA

Effetto degli interruttori 0, 1 e 3 premuti:

inter

inter	0	1	2	3	4
0	1	1	0	0	1
1	1	0	1	0	0
2	0	1	1	1	0
3	1	0	0	1	0



Controllo:

lamp0: 3 interruttori
lamp1: 1 interruttore
lamp2: 1 interruttore
lamp3: 1 interruttore
lamp4: 1 interruttore

SOLUZIONE VALIDA

Algoritmo:

- generare tutti i sottoinsiemi di interruttori (non necessario l'insieme vuoto)
- per ogni sottoinsieme applicare una funzione di verifica di validità
- tra le soluzioni valide, scegliere la prima tra quelle a minima cardinalità.

Modello:

- insieme delle parti generato con combinazioni semplici di n elementi a k a k
- k cresce da 1 a n (non necessario l'insieme vuoto)
- la prima soluzione che si trova è anche quella a cardinalità minima.

Strutture dati:

- matrice inter di interi $n \times m$
- vettore sol di n interi
- non serve il vettore val (gli interruttori sono numerati da 0 a $n-1$)

```

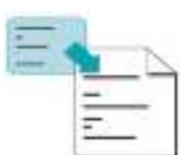
int main(void) {
    int n, m, k, i, trovato=0;
    FILE *in = fopen("switches.txt", "r");
    int **inter = leggiFile(in, &n, &m);
    int *sol = calloc(n, sizeof(int));
    printf("Powerset mediante combinazioni semplici\n\n");
    for (k=1; k <= n && trovato==0; k++) {
        if(powerset(0, sol, n, k, 0, inter, m))
            trovato = 1;
    }
    free(sol);
    for (i=0; i < n; i++)
        free(inter[i]);
    free(inter);
    return 0;
}

```

cardinalità sottoinsieme crescente da 1 a n

non serve l'insieme vuoto

stop appena trovata soluzione a cardinalità minima



01interruttori_comb_sempl

```

int powerset(int pos,int *sol,int n,int k,int start,int **inter,int m) {
    int i;
    if (pos >= k) {
        if (verifica(inter, m, k, sol)) {
            stampa(k, sol);
            return 1;
        }
        return 0;
    }
    for (i = start; i < n; i++) {
        sol[pos] = i;
        if (powerset(pos+1, sol, n, k, i+1, inter, m))
            return 1;
    }
    return 0;
}

```

verifica di validità

stop appena trovata
soluzione valida

soluzione non valida

nessuna soluzione
valida trovata

Verifica:

- dato un sottoinsieme di k interruttori premuti
 - per ogni lampadina contare quanti interruttori la controllano
 - registrare se pari o dispari (calcolando il resto della divisione intera per 2)
- soluzione valida se per ogni lampadina il numero di interruttori premuti che la controlla è dispari.

```

int verifica(int **inter, int m, int k, int *sol) {
    int i, j, ok = 1, *lampadine;
    lampadine = calloc(m, sizeof(int));
    forall lampadina
    for (j=0; j<m && ok; j++)
        for(i=0; i<k; i++)
            lampadine[j] += inter[sol[i]][j];
        forall interruttore del sottoinsieme
        if (lampadine[j]%2 == 0)
            ok = 0;
    }
    free(lampadine);
    return ok;
}
conta quanti interruttori del  
sottoinsieme la controllano
se pari KO

```

Verifica alternativa:

- vettore delle lampadine (inizialmente tutte spente)
- per ciascuna delle lampadine
 - per ciascuno degli interruttori del sottoinsieme

		interruttore	
		non controlla	controlla
lampadina	spenta	spenta	accesa
	accesa	accesa	spenta
stato della lampadina			

		interruttore	
		0	1
lampadina	0	0	1
	1	1	0
lampadina EXOR interruttore			

```

int verifica(int **inter, int m, int k, int *sol) {
    int i, j, ok = 1, *lampadine;
    lampadine = calloc(m, sizeof(int));
    ∀ lampadina

    for (j=0; j<m && ok; j++) {
        for (i=0; i<k; i++)
            ∀ interruttore del sottoinsieme
            lampadine[j] ^= inter[sol[i]][j];
        if (lampadine[j]==0)
            ok = 0;
    }
    lampadina EXOR interruttore
    free(lampadine);
    se pari KO
    return ok;
}

```

Longest Increasing Sequence

Data una sequenza di N interi

$$X = (x_0, x_1, \dots, x_{N-1})$$

si definisce **sottosequenza** di X di lunghezza k ($k \leq N$) una qualsiasi n-upla Y di k elementi di X con indici crescenti i_0, i_1, \dots, i_{k-1} non necessariamente contigui.

Esempio:

$X = 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15$

$Y = 0, 8, 12, 10, 14, 1, 7, 15$ è una sottosequenza di lunghezza $k=8$ di X (indici $0, 1, 3, 5, 7, 8, 14, 15$).

Si ricordi:

- **sottosequenza**: indici non necessariamente contigui
- **sottostringa/sottovettore**: indici contigui

Problema:

data una sequenza, identificare una qualsiasi delle sue LIS (Longest Increasing Subsequence), cioè una delle sottosequenze:

- strettamente crescenti && a lunghezza massima.

Esempio:

per $X=0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15$

esistono 4 LIS con $k=6$:

0, 2, 6, 9, 11, 15 0, 4, 6, 9, 11, 15

0, 2, 6, 9, 13, 15 0, 4, 6, 9, 13, 15

Algoritmo:

- generare tutti i sottoinsiemi di elementi di X (non necessario l'insieme vuoto)
- per ogni sottoinsieme applicare una funzione di verifica di validità (controllo di monotonia stretta)
- problema di ottimizzazione: tenere traccia della soluzione ottima corrente e confrontarla con ciascuna soluzione generata
- tra le soluzioni valide, scegliere una tra quelle a massima cardinalità.

Modello:

- insieme delle parti generato con disposizioni ripetute di n elementi
a k a k
- k cresce da 1 a n

Strutture dati:

- vettore v di n interi per i valori
- vettore s e bs di n interi per soluzione corrente e soluzione migliore
- interi l e bl per lunghezza corrente e lunghezza migliore.

```

void ps(int pos, int *v, int *s, int k, int *b1, int *bs) {
    int j, l=0, ris;
    if (pos >= k) { caso terminale
        for (j=0; j<k; j++)
            if (s[j]!=0) l++;
        ris = check(val, k, s, l);
        if (ris==1) {
            if (l >= *b1) { verifica di validità
                for (j=0; j<k; j++) bs[j] = s[j];
                *b1 = l;
            }
        }
        return;
    }
    s[pos] = 0; ps(pos+1, v, s, k, b1, bs);
    s[pos] = 1; ps(pos+1, v, s, k, b1, bs);
    return;
}

```



02LIS

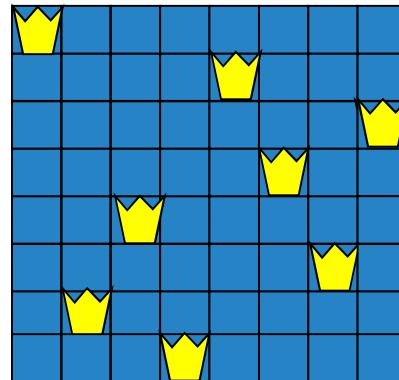
```
int check(int *v, int k, int *s int l) {  
    int i=0,j, t, ok=1;  
  
    for (t=0; t<l-1; t++){  
        while ((s[i]==0) && (i < k-1))  
            i++;  
        j=i+1;  
        while ((s[j]==0) && (j < k))  
            j++;  
        if (v[i] >= v[j])  
            ok = 0;  
        i = j;  
    }  
    return ok;  
}
```

Le 8 regine (Max Bezzel 1848)

Data una scacchiera 8×8 , disporvi 8 regine in modo che non si diano scacco reciprocamente:

- 92 soluzioni
- 12 fondamentali (tenendo conto di rotazioni e simmetrie)

Esempio:



Generalizzabile a N regine, con $N \geq 4$:

- N=4: 2 soluzioni
- N=5: 10 soluzioni
- N=6: 4 soluzioni
- etc.

Problema di ricerca per cui si vuole:

- 1 soluzione qualsiasi
- tutte le soluzioni

NB: le regine sono di per sè indistinte. I modelli che le considerano distinte generano soluzioni identiche a meno di permutazioni, rotazioni e simmetrie.

Modello 0:

- ogni cella può contenere o no una regina indistinta (il numero di regine varia da 0 a 64)
- **powerset con disposizioni ripetute**
- pruning opportuno
- filtro le soluzioni imponendo di avere esattamente 8 regine
- $D'_{n,k} = 2^{64} \approx 1.84 \cdot 10^{19}$ casi (senza pruning)!
- variabili globali s[N][N], num_sol
- variabile q che svolge il ruolo della variabile pos.

```

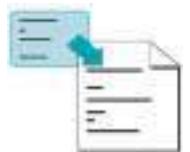
void powerset (int r, int c, int q) {
    if (c>=N) {
        c=0; r++;
    }
    if (r>=N) {           scacchiera finita!
        if (q!=N)
            return;
        if (controlla())
            stampa();
        return;
    }
    s[r][c] = q+1;
    powerset (r,c+1,q+1);   prova a mettere la regina su r,c
    s[r][c] = 0;             ricorri
    powerset (r,c+1,q);     backtrack
    return;
}

```

```

#define N 4
int s[N][N];
int num_sol=0;

```

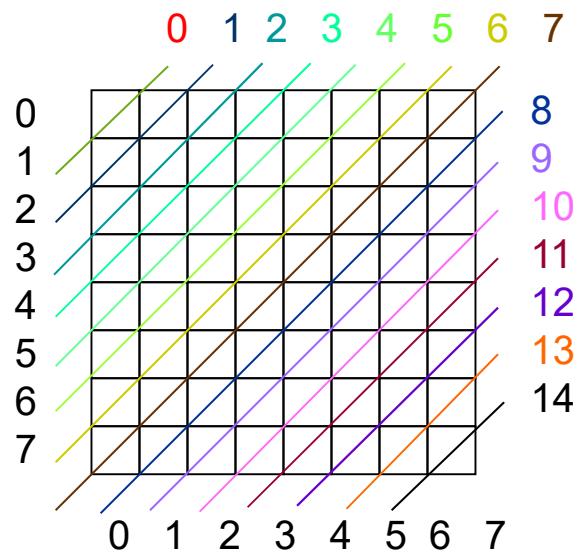


03otto-regine-powerset

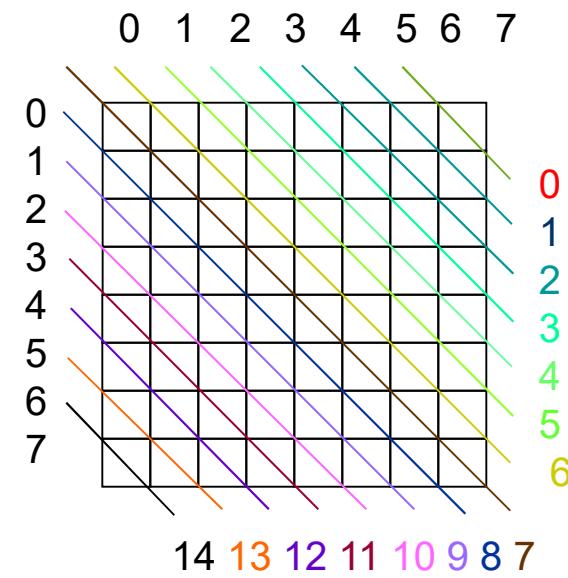
Funzione controlla:

- righe , colonne, diagonali e antidiagonali: conteggiare per ognuna il numero di celle della scacchiera diverse da 0. Se tale numero è >1 , la soluzione è inaccettabile
- diagonali:
 - 15 diagonali individuate dalla somma degli indici di riga e di colonna
 - 15 antidiagonali individuate dalla differenza degli indici di riga e di colonna (+ 7 per non avere valori negativi)

diagonali



antidiagonali



```
int controlla (void) {  
    int r, c, n;  
    for (r=0; r<N; r++) {  
        for (c=n=0; c<N; c++)  
            if (s[r][c] !=0)  
                n++;  
        if (n>1)  
            return 0;  
    }  
    for (c=0; c<N; c++) {  
        for (r=n=0; r<N; r++)  
            if (s[r][c] !=0)  
                n++;  
        if (n>1)  
            return 0;  
    }  
    .....
```

controlla righe

controlla colonne

controlla diagonali

```
for (d=0; d<2*N-1; d++) {  
    n=0;  
    for (r=0; r<N; r++) {  
        c = d-r;  
        if ((c>=0)&& (c<N))  
            if (s[r][c] !=0) n++;  
    }  
    if (n>1) return 0;  
}
```

controlla antidiagonali

```
for (d=0; d<2*N-1; d++) {  
    n=0;  
    for (r=0; r<N; r++) {  
        c = r-d+N-1;  
        if ((c>=0)&& (c<N))  
            if (s[r][c] !=0) n++;  
    }  
    if (n>1) return 0;  
}  
return 1;
```

l'ordinamento conta

Modello 1:

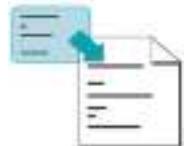
- piazza 8 distinte regine ($k = 8$) in 64 caselle ($n = 64$)
- **disposizioni semplici**
- $D_{n,k} = \frac{n!}{(n-k)!} \approx 1,78 \cdot 10^{14}$ casi!
- variabili **globali** num_sol e s[N] [N] che svolge il ruolo del vettore mark
- variabile q che svolge il ruolo della variabile pos.

```

void disp_sempl(int q) {
    int r,c;
    if (q >= N) { piazzate tutte le regine
        if(controlla()) {
            num_sol++;
            stampa();
        }
        return;
    }
    for (r=0; r<N; r++)
        for (c=0; c<N; c++) controllo se cella vuota
        if (s[r][c] == 0) {
            s[r][c] = q+1;
            disp_sempl(q+1); prova a mettere la regina su r,c
            s[r][c] = 0;
        }
    return;
}

```

#define N 4
int s[N][N];
int num_sol=0;



04otto-regine-disp-sempl

Modello 2:

- piazzo 8 indistinte regine ($k = 8$) in 64 caselle ($n = 64$)
 - **combinazioni semplici**  l'ordinamento non conta
 - $C_{n,k} = \frac{n!}{k!(n-k)!} \approx 4,42 \cdot 10^9$ casi!
-
- variabile globale $s[N][N]$ per la scacchiera
 - variabile q che svolge il ruolo della variabile pos
 - variabili $r0$ e $c0$ per forzare un ordinamento.

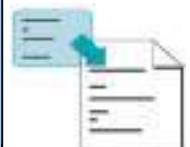
```

void comb_sempl(int r0, int c0, int q) {
    int r,c;
    if (q >= N) { piazzate tutte le regine
        if(controlla()) {
            num_sol++; stampa(); iterazione sulle scelte
        }
        return;
    }
    for (r=r0; r<N; r++)
        for (c=0; c<N; c++)
            if (((r>r0)||((r==r0)&&(c>=c0)))&&s [r] [c]==0) {
                s[r][c] = q+1; scelta
                comb_sempl (r,c,q+1); ricorri
                s[r][c] = 0;
            }
    return;
}

```

controllo sulla fattibilità della scelta

backtrack

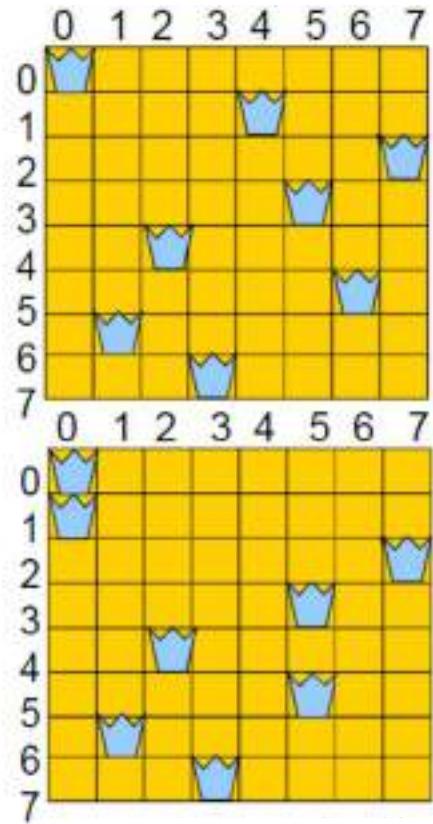


05otto-regine-comb-sempl

Modello 3:

- struttura dati monodimensionale:
 - ogni riga contiene una e una sola regina distinta in una delle 8 colonne ($n = 8$)
- ci sono 8 righe ($k = 8$)
- **disposizioni con ripetizione**
- $D'_{n,k} = n^k = 8^8 = 16.777.216$ casi!
- non serve più il controllo sulle righe, basta quello su colonne, diagonali e antidiagonali
- variabile `riga [N]`
- variabile `q` che svolge il ruolo della variabile `pos`.

l'ordinamento conta



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

controlloa()=1

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

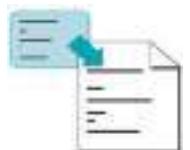
controlloa()=0

```

void disp_ripet(int q) {
    int i;
    if (q >= N) { _____ scacchiera finita!
        if(controlla()) {
            num_sol++;
            stampa();
        }
        return;
    }
    for (i=0; i<N; i++) {
        riga[q] = i; _____ prova a mettere la regina sulla riga
        disp_ripet(q+1);
    }
    return;
}

```

ricorri



06otto-regine-disp-ripet

```

int controlla (void) {
    int r, n, d, occ[N];
    vettore delle occorrenze

    for (r=0; r<N; r++) occ[r]=0;

    for (r=0; r<N; r++)
        occ[riga[r]]++;
    for (r=0; r<N; r++)
        if (occ[r]>1)
            return 0;

    for (d=0; d<2*N-1; d++) {
        n=0;
        for (r=0; r<N; r++) {
            if (d==r+riga[r]) n++;
        }
        if (n>1) return 0;
    }
}

```

```
for (d=0; d<2*N-1; d++) {  
    n=0;  
    for (r=0; r<N; r++) {  
        if (d==(r-riga[r]+N-1))  
            n++;  
    }  
    if (n>1) return 0;  
}  
return 1;
```

controlla antidiagonali

Modello 4:

- ogni riga e ogni colonna contengono una e una sola regina distinta in una delle 8 colonne ($n = 8$)
- ci sono 8 righe ($k = 8$)
- **permutazioni semplici**  l'ordinamento conta
- $P_n = D_{n,n} = n! = 40320$ casi possibili!

- variabili globali `riga[N]` e `mark[N]`
- variabile `q` che svolge il ruolo della variabile `pos`
- controllo solo su diagonali e antidiagonali.

```

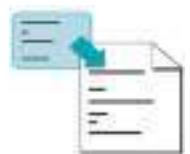
void perm_semp1(int q) { scacchiera finita!
    int c;
    if (q >= N) {
        if (controlla()) {
            num_sol++; stampa();
            return;
        }
        return;
    }
    for (c=0; c<N; c++)
        if (mark[c] == 0) {
            mark[c] = 1; riga[q] = c;
            perm_semp1(q+1); mark[c] = 0;
        }
    return;
}

```

ricorri

prova a mettere la regina sulla riga

backtrack



07otto-regine-perm-semp1

```
int controlla (void) {  
    int r, n, d;  
    for (d=0; d<2*N-1; d++) {  
        n=0;  
        for (r=0; r<N; r++)  
            if (d==r+riga[r])  
                n++;  
        if (n>1) return 0;  
    }  
    for (d=0; d<2*N-1; d++) {  
        n=0;  
        for (r=0; r<N; r++)  
            if (d==(r-riga[r]+N-1))  
                n++;  
        if (n>1) return 0;  
    }  
    return 1;  
}
```

controlla diagonali

controlla antidiagonali

trade-off tempo/spazio

Modello 4 ottimizzato:

- uso di 2 vettori $d[2*N-1]$ e $ad[2*N-1]$ per marcare le diagonali e le antidiagonali messe sotto scacco da una regina
- pruning: controllo di ammissibilità prima di procedere ricorsivamente.

```

void perm_sempl(int q) {
    int c;
    if (q >= N) {num_sol++; stampa(); return;}
    for (c=0; c<N; c++)
        if (((mark[c]==0)&&(d[q+c]==0)&&(ad[q-c+(N-1)]==0)){
            mark[c] = 1;
            d[q+c] = 1;
            ad[q-c+(N-1)] = 1;
            riga[q] = c;
            ricorri(q+1);
            mark[c] = 0;
            d[q+c] = 0;
            ad[q-c+(N-1)] = 0;
        }
    return;
}

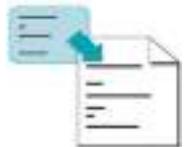
```

scacchiera finita!

controllo

prova a mettere la regina sulla riga

backtrack



08otto-regine-perm-sempl-ott

Aritmetica verbale

Specifiche:

input: 3 stringhe, 1 operazione (addizione)

Esempio:

$$\begin{array}{r} \text{S E N D} \\ + \text{M O R E} \\ \hline \text{M O N E Y} \end{array}$$

Interpretazione:

le stringhe sono interi “criptati”, cioè ogni lettera rappresenta una e una sola cifra decimale.

Output: decriptare le stringhe, cioè identificare la corrispondenza lettere – cifre decimali che soddisfa l’addizione data.

Assunzioni:

- solo caratteri alfabetici tutti maiuscoli o tutti minuscoli
- la cifra più significativa diversa da 0
- le prime due stringhe di lunghezza massima 8, non necessariamente uguale
- la terza stringa ha lunghezza coerente con quella delle prime 2
- si opera in base 10: nelle stringhe non compaiono più di 10 lettere distinte (`lett_dist <=10`)

Soluzione:

O=0, M=1, Y=2, E=5, N=6, D=7, R=8 e S=9

$$\begin{array}{r} \text{S E N D +} \\ \text{M O R E =} \\ \hline \text{M O N E Y} \end{array} \qquad \begin{array}{r} 9 5 6 7 + \\ 1 0 8 5 = \\ \hline 1 0 6 5 2 \end{array}$$

A ogni lettera distinta va associata una e una sola cifra decimale 0..9

Modello: disposizioni semplici di n elementi a k a k,
dove n = 10 e k = lett_dist

Strutture dati

tabella di simboli

- Variabile globale intera lett_dist
- Vettore lettere[10] di struct di tipo alpha con campo car (carattere distinto) e val (cifra decimale corrispondente)
- Vettore mark [10] per marcare le cifre già considerate

Algoritmo

- allocare e inizializzare il vettore `lettere` (funzione `init_alpha`)
- leggere le 3 stringhe
- riempire `lettere` con `lett_dist` caratteri distinti (funzione `setup` che usa la funzione di servizio `trova_indice`)
- calcolare le disposizioni semplici delle n=10 cifre decimali a k a k, dove k=`lett_dist` (funzione `disp`):
 - nella condizione di terminazione sostituire le lettere con le cifre, convertire ad intero (funzione `w2n`) , controllare la validità della soluzione (funzione `c_sol`) e, se valida, stamparla (funzione `stampa`)
 - ricorsione sulla posizione successiva nel vettore `lettere`.

Funzioni

accesso alla tabella di simboli

int trova_indice(alpha *lettere, char c)

dato il carattere C, trova e ritorna il suo indice nel vettore lettere, se non c'è ritorna -1

alpha * init_alpha()

alloca lettere[10] e inizializzalo (valore = -1, carattere = \0)

**void setup(alpha *lettere, char *str1,
char *str2, char *str3)**

date le 3 stringhe, metti i caratteri distinti in lettere e conta quanti sono (lett_dist)

```
int disp(alpha *lettere, int *mark, int pos, char *str1, char *str2, char *str3)
```

calcola le disposizioni delle n=10 cifre decimali a k a k, dove
k=lett_dist

```
int w2n(alpha *lettere, char *str)
```

rimpiazza nella stringa str le lettere con le cifre, sulla base
della corrispondenza memorizzata in lettere, converti a
intero, ritornando -1 nei casi in cui la cifra più significativa
della stringa è 0

```
int c_sol(alpha *lettere, char *str1,  
char *str2, char *str3)
```

controlla che le 3 stringhe convertite a intero soddisfino la somma

```
void stampa(alpha *lettere)
```

stampa la corrispondenza lettere-cifre memorizzata nei campi car e val di lettere

SEND MORE MONEY

lettere

car	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
val	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

lettere

car	S	E	N	D	M	O	R	Y	\0	\0
val	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

lettere

car	S	E	N	D	M	O	R	Y	\0	\0
val	9	5	6	7	1	0	8	2	-1	-1

dopo init_alpha

dopo setup

lett_dist = 8

dopo disp: esempio
di possibile disposizione

```

int main(void) {
    char str1[LUN_MAX], str2[LUN_MAX], str3[LUN_MAX+1];
    int mark[base] = {0};
    int i;

    // Lettura delle 3 stringhe

    alpha *lettere = init_alpha();
    setup(lettere, str1, str2, str3);

    disp(lettere, mark, 0, str1, str2, str3);

    free(lettere);
    return 0;
}

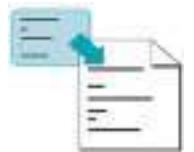
```

```

#define LUN_MAX 8+1
#define n 10
#define base 10
int lett_dist = 0;

```

variabile globale



09aritmetica_verba

```
typedef struct { char car; int val; } alpha;

int trova_indice(alpha *lettere, char c) {
    int i;
    for(i=0; i < lett_dist; i++)
        if (lettere[i].car == c) return i;
    return -1;
}
alpha *init_alpha() {
    int i; alpha *lettere;
    lettere = malloc(n * sizeof(alpha));
    if (lettere == NULL) exit(-1);
    for(i=0; i < n; i++) {
        lettere[i].val = -1; lettere[i].car = '\0';
    }
    return lettere;
}
```

```
void setup(alpha *lettere,char *st1,char *st2,char *st3){  
    int i, l1=strlen(st1), l2= strlen(st2), l3=strlen(st3);  
  
    for(i=0; i<l1; i++) {  
        if (trova_indice(lettere, st1[i]) == -1)  
            lettere[lett_dist++].car = st1[i];  
    }  
    for(i=0; i<l2; i++) {  
        if (trova_indice(lettere, st2[i]) == -1)  
            lettere[lett_dist++].car = st2[i];  
    }  
    for(i=0; i<l3; i++) {  
        if (trova_indice(lettere, st3[i]) == -1)  
            lettere[lett_dist++].car = st3[i];  
    }  
}
```

```

int w2n(alpha *lettere, char *st) {
    int i, v = 0, l=strlen(st);
    if (lettere[trova_indice(lettere, st[0])].val == 0)
        return -1;
    for(i=0; i < l; i++)
        v = v*10 + lettere[trova_indice(lettere, st[i])].val;
    return v;
}
int c_sol(alpha *lettere,char *st1,char *st2,char *st3) {
    int n1, n2, n3;
    n1 = w2n(lettere, st1);
    n2 = w2n(lettere, st2);
    n3 = w2n(lettere, st3);
    if (n1 == -1 || n2 == -1 || n3 == -1)
        return 0;
    return ((n1 + n2) == n3);
}

```

```
int disp(alpha *lettere, int *mark, int pos, char *st1,
         char *st2, char *st3) {
    int i = 0, risolto;
    if (pos == lett_dist) {
        risolto = contr_sol(lettere, st1, st2, st3);
        if (risolto) stampa(lettere);
        return risolto;
    }
    for(i=0;i < base; i++) {
        if (mark[i]==0) {
            lettere[pos].val = i; mark[i] = 1;
            if (disp(lettere, mark, pos+1, st1, st2, st3))
                return 1;
            lettere[pos].val = -1; mark[i] = 0;
        }
    }
    return 0;
}
```

I 36 ufficiali di Eulero

Ci sono 36 ufficiali provenienti da 6 reggimenti (colori) e appartenenti a 6 ranghi diversi (pezzi degli scacchi).



Disporre gli ufficiali in un quadrato 6×6 in modo che in ogni riga e in ogni colonna compaia un ufficiale di ogni rango e un ufficiale di ogni reggimento.

Il Quadrato Latino

Quadrato latino di ordine n: quadrato n per n in cui le n^2 caselle sono occupate da n simboli distinti in modo che ogni simbolo compare una e una sola volta in ogni riga e in ogni colonna del quadrato.

A	B	C	D
B	A	D	C
C	D	A	B
D	C	B	A

Il Quadrato Greco-latino

Quadrato greco-latino di ordine n su 2 insiemi S e T di n elementi: quadrato n per n in cui le n^2 caselle sono occupate da n^2 coppie ordinate di simboli distinti di S e T in modo che ogni coppia compaia una e una sola volta in ogni riga e in ogni colonna del quadrato.

Numero di coppie ordinate: principio di moltiplicazione $n \times n$.

A	B	C	D
B	A	D	C
C	D	A	B
D	C	B	A

α	β	γ	δ
γ	δ	α	β
δ	γ	β	α
β	α	δ	γ

A; α	B; β	C; γ	D; δ
B; γ	A; δ	D; α	C; β
C; δ	D; γ	A; β	B; α
D; β	C; α	B; δ	A; γ

36 ufficiali di Eulero: esiste un quadrato greco-latino con $n=6$?

I quadrati greco-latini esistono per ordine $n \geq 3$ con l'eccezione di $n=6$.

Eulero aveva ipotizzato che non esistessero per $n=6$ e per tutti i numeri pari che, divisi per 2, danno un numero dispari.

Falsa in generale la congettura di Eulero, ma vera per $n=6$.

Il Sudoku

Deriva dai quadrati latini di Eulero.

Input:

- griglia di 9×9 celle
- cella o vuota o con numero da 1 a 9
- 9 righe orizzontali, 9 colonne verticali
- da bordi doppi 9 regioni, di 3×3 celle contigue
- inizialmente da 20 a 35 celle riempite

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

Scopo del gioco è quello di riempire le caselle bianche con numeri da 1 a 9, in modo tale che in ogni riga, colonna e regione siano presenti tutte le cifre da 1 a 9 senza ripetizioni.

Una soluzione:

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

Si può generalizzare il Sudoku a griglie $n \times n$ purché n sia un quadrato perfetto (un numero intero che può essere espresso come il quadrato di un altro numero intero).

Modello:

- disposizioni con ripetizione
- k = numero di celle non preassegnate, n è il numero di scelte
- dimensione dello spazio: n^k .

Ricerca di tutte o di una soluzione.

Ricorsione di 2 tipi:

- casella già piena: non c'è scelta
- casella vuota: c'è scelta. In fase di ritorno si annulla la scelta (altrimenti si ricadrebbe nel caso precedente)
- pruning: controllo prima di ricorrere.

```

int main() {
    char nomefile[20];
    printf("Inserire il nome del file: ");    scanf("%s", nomefile);
    acquisisci(nomefile);
    disp_ripet(0);
    printf("\n Numero di soluzioni = %d\n", num_sol);
    return 0;
}
void acquisisci(char *nomefile) {
    int i, j; FILE *fp;
    fp = fopen(nomefile, "r"); /* inserire controllo di errore */
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++) fscanf(fp, "%d", &schema[i][j]);
    }
    fclose(fp);
    return;
}

```

variabili globali

```

#define n 9
int schema[n][n], num_sol=0;

```



10sudoku

```

void disp_ripet(int pos) {
    int i, j, k;
    if (pos >= n*n) { terminazione
        num_sol++; stampa(schema); return; indici casella corrente
    }
    i = pos / n; j = pos % n;
    if (schema[i][j] != 0) { cella già piena
        disp_ripet(pos+1);
        return;
    }
    for (k=1; k<=n; k++) {
        schema[i][j] = k; scelta
        if (controlla(pos, k)) controlla
            disp_ripet(pos+1);
        schema[i][j] = 0;
    }
    return;
}

```

ricorri su cella successiva

ricorri su cella successiva

smarca la cella, altrimenti si considera fissata a priori

```

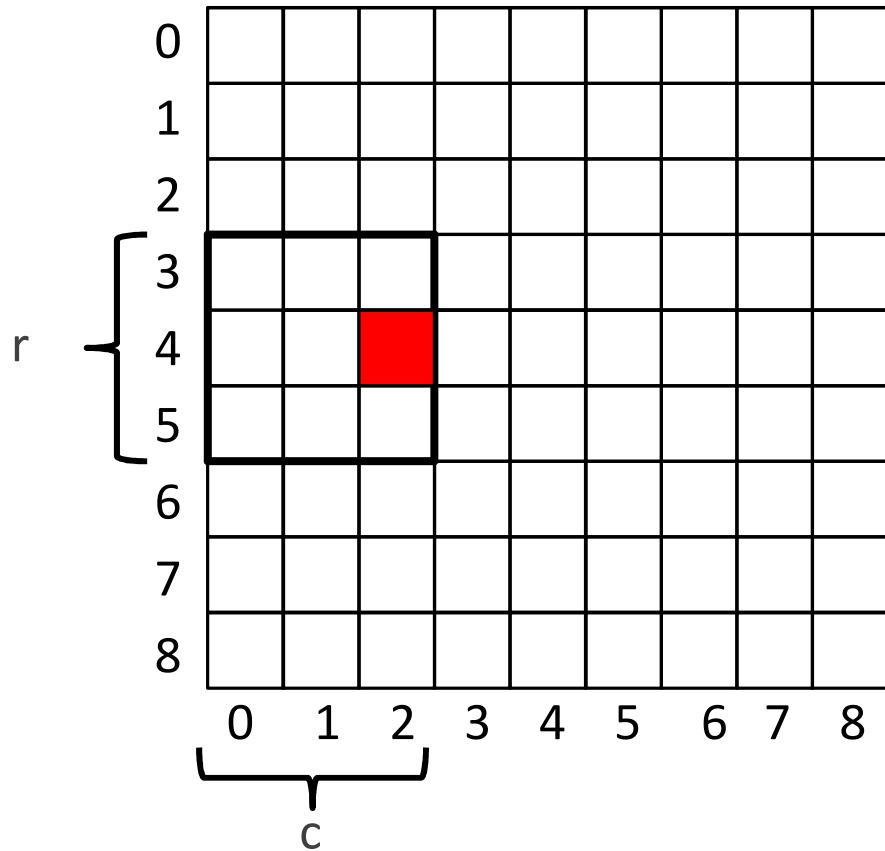
int controlla(int pos, int val){
    int i, j, r, c, dim=floor(sqrt(n));
    i = pos/n;
    j = pos % n;
    for (c=0; c<n; c++) {
        if (c!=j)
            if (schema[i][c]==val)
                return 0;
    }
    for (r=0; r<n; r++) {
        if (r!=i)
            if (schema[r][j]==val)
                return 0;
    }
}

```

i = pos/n; j = pos % n; indici casella corrente

for (c=0; c<n; c++) {
if (c!=j)
if (schema[i][c]==val)
return 0;
} data la riga i, ciclo sulle colonne:
controllo che il valore val inserito
in i, j non sia già presente, ad
esclusione della colonna j

for (r=0; r<n; r++) {
if (r!=i)
if (schema[r][j]==val)
return 0;
} data la colonna j, ciclo sulle righe:
controllo che il valore val inserito
in i, j non sia già presente, ad
esclusione della riga i



Identificazione del blocco:

$$\text{dim} = 3$$

$$n = 9$$

$$i = 4$$

$$j = 2$$

$$(i/\text{dim}) * \text{dim} \leq r < (i/\text{dim}) * \text{dim} + \text{dim}$$

$$3 \leq r < 6$$

$$(j/\text{dim}) * \text{dim} \leq c < (j/\text{dim}) * \text{dim} + \text{dim}$$

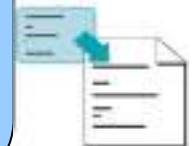
$$0 \leq c < 3$$

```
for (r=(i/dim)*dim; r<(i/dim)*dim+dim; r++)  
    for (c=(j/dim)*dim; c<(j/dim)*dim+dim; c++) {  
        if ((r!=i) || (c!=j))  
            if (schema[r][c]==val)  
                return 0;  
    }  
return 1;  
}
```

ciclo sui blocchi: controllo che il valore val inserito in i, j non sia presente nel blocco ad esclusione della cella i,j

Ricerca di una sola soluzione

```
int disp_ripet(int pos) {  
    int i, j, k;  
    if (pos >= n*n) {stampa(schema); return 1;}  
    i = pos / n;  
    j = pos % n;  
    if (schema[i][j] != 0) terminazione  
        return (disp_ripet(pos+1));  
  
    for (k=1; k<=n; k++) {  
        schema[i][j] = k; scelta  
        if (controlla(pos, k)) controllo  
            if (disp_ripet(pos+1)==1) terminazione  
                return 1;  
        schema[i][j] = 0; ricorri su cella successiva  
    } smarca  
    return 0; successo  
}  
fallimento
```



11sudoku1soluz

Scacchiere e grafi

Una scacchiera NxN può essere interpretata come *grafo隐式* dove:

- i vertici sono le caselle
- gli archi rappresentano la raggiungibilità di coppie di vertici. La definizione di raggiungibilità dipende dal problema in esame.

Per questa tipologia di grafo non è necessaria una rappresentazione esplicita di vertici ed archi, in quanto li si può ricavare direttamente dalla scacchiera.

In generale i problemi si riconducono alla ricerca di cammini per la quale non sono necessarie conoscenze di Teoria dei Grafi.

Il Tour del cavallo

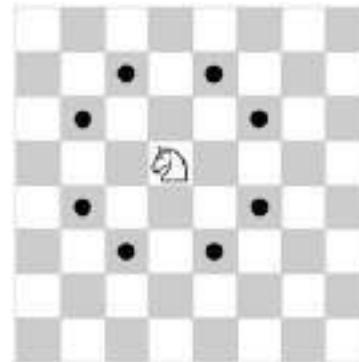
Si consideri una scacchiera NxN , trovare un “*giro di cavallo*”, cioè una sequenza di mosse valide del cavallo tale che ogni casella venga visitata una sola volta (visitata = casella su cui il cavallo si ferma, non casella attraverso cui transita).

Se il cavallo si ferma su una casella da cui si può raggiungere con una mossa la casella da cui si è partiti, il tour si dice chiuso, altrimenti si dice aperto.

Il tour del cavallo è un caso particolare di cammino di Hamilton se aperto o ciclo di Hamilton se chiuso.

Il più antico riferimento al tour del cavallo si trova in un testo poetico sanscrito del XI secolo dC. Fu studiato da Eulero nel XVIII secolo. La prima soluzione (euristica) risale al 1823 ed è la regola di von Warnsdorff.

Le mosse del cavallo lecite sono al massimo 8 a partire da ogni casella:



Interpretando la scacchiera come grafo non orientato:

- vertici = caselle
- archi: tra coppie di vertici mutuamente raggiungibili con le mosse lecite del cavallo

il problema si riconduce al Cammino di Hamilton (cammino semplice che tocca tutti i vertici).

Modello: principio di moltiplicazione: a partire da ogni cella si considerano le 8 possibili mosse. La dimensione dello spazio di ricerca è quindi $O(8^{NxN})$.

Pruning:

- si vincola la discesa ricorsiva a quelle, tra le 8 mosse possibili, che portano a caselle nella scacchiera. I vincoli sono quindi statici.

Si attribuisce a ogni cella un numero di mossa. Si termina quando sono state etichettate tutte le $N \times N$ caselle.

Esempio di tour del cavallo aperto per una scacchiera 8 x 8 a partire dalla cella (0,0):

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

```

int main(void) {
    int dx[8], dy[8], **scacc, x, y, N;
    dx[0]=2; dy[0]=1; dx[1]=1; dy[1]=2;
    dx[2]=-1;dy[2]=2; dx[3]=-2;dy[3]=1;
    dx[4]=-2;dy[4]=-1;dx[5]=-1;dy[5]=-2;
    dx[6]=1; dy[6]=-2;dx[7]=2; dy[7]=-1;
    printf("Dimensione: "); scanf("%d", &N);
    scacc = malloc2d(N);
    /* inizializzazione a -1 delle celle di scacc */
    printf("Partenza: "); scanf("%d %d", &x, &y);
    scacc[x][y] = 0;
    if (mv(1, x, y, dx, dy, scacc, N)==1) {
        printf("Mosse del cavallo\n"); stampa(scacc, N);
    } else
        printf("Soluzione non trovata\n");
    return 0;
}

```



12cavallo

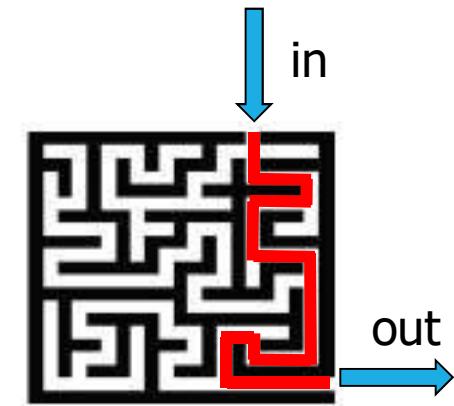
```
int mv(int m,int x,int y,int *dx,int *dy,int **s,int N){  
    int i, newx, newy;  
    if (m == N*N)  
        return 1;  
    for (i=0; i<8; i++) {  
        newx = x + dx[i];  
        newy = y + dy[i];  
        if ((newx<N) && (newx>=0) && (newy<N)&&(newy>=0)) {  
            if (s[newx][newy] == 0) {  
                s[newx][newy] = m;  
                if (mv(m+1, newx, newy, dx, dy, s, N) == 1)  
                    return 1;  
                s[newx][newy] = 0;  
            }  
        }  
    }  
    return 0;  
}
```

Il Labirinto

Un labirinto può essere rappresentato come una scacchiera NxM dove ogni cella o è vuota o è piena per rappresentare un muro.

Data una cella di ingresso e una di uscita, il problema consiste nel trovare, se esiste, un cammino semplice che le connette.

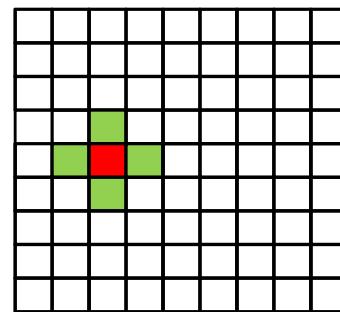
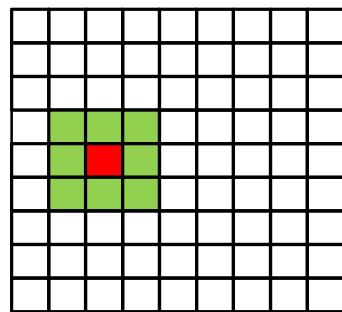
Una cella piena non può mai essere attraversata.



Relazione di raggiungibilità:

- da ogni cella si possono raggiungere al massimo le 8 celle a distanza 1 che appartengono alla scacchiera. Il movimento è secondo le 4 direzioni N, S, E, W e in obliquo
- da ogni cella si possono raggiungere al massimo le 4 celle che appartengono alla scacchiera in direzione N, S, E, W ma non in obliquo.

Nel problema in esame si considera la seconda scelta.



Interpretando la scacchiera come grafo non orientato:

- vertici = caselle
- archi: tra vertice corrente e vertici adiacenti secondo le direzioni N, S, E, W

il problema si riconduce all'enumerazione dei cammini semplici a partire dalla cella di ingresso con condizione di terminazione di aver raggiunto la cella di uscita.

Modello: principio di moltiplicazione: a partire da ogni cella si considerano le 4 possibili mosse. La dimensione dello spazio di ricerca è quindi $O(4^{NxN})$.

Un punto viene rappresentato mediante il tipo `punto_t`, una `struct` avente come campi le coordinate di riga e colonna.

Pruning:

- si vincola la discesa ricorsiva a quelle, tra le 4 mosse possibili, che portano a caselle nella scacchiera, che non siano muri e che siano diverse dalla casella da cui si parte.

```

int main (int argc, char *argv[]){
    /* dichiarazioni varie, tra cui punto_t ingresso, uscita; */
    /* apertura del file e lettura di labirinto e ingresso/uscita */

    L[ingresso.r][ingresso.c] = 'I';
    L[uscita.r][uscita.c] = 'U';
    printf("configurazione iniziale\n");
    stampa();

    if (mossa(ingresso, uscita)){
        printf("soluzione trovata\n");
        stampa();
    }
    else
        printf("soluzione NON trovata\n");
    return 0;
}

```



12cavalo

```

int mossa (punto_t corrente, punto_t uscita) {
    int i;
    punto_t nuovo;
    if (corrente.r == uscita.r && corrente.c == uscita.c){
        L[corrente.r][corrente.c] = 'U';
        return 1;
    }
    for (i=0; i < 4; i++) {
        nuovo = sposta(corrente,i);
        if (nuovo.r!=corrente.r || nuovo.c!=corrente.c) {
            L[nuovo.r][nuovo.c] = '*';
            if (mossa(nuovo,uscita)==1)
                return 1;
            L[nuovo.r][nuovo.c] = '.';
        }
    }
    return 0;
}

```

backtrack

```
punto_t sposta(punto_t punto, int i) {
    int r, c;
    int spr[4] = { 0,-1, 0, 1};
    int spc[4] = {-1, 0, 1, 0};

    r = punto.r+spr[i];
    c = punto.c+spc[i];
    if (r >= 0 && c >= 0 && r < nr && c < nc)
        if ((L[r][c])=='.' || (L[r][c])=='U'){
            punto.r = r;
            punto.c = c;
        }
    return punto;
}
```

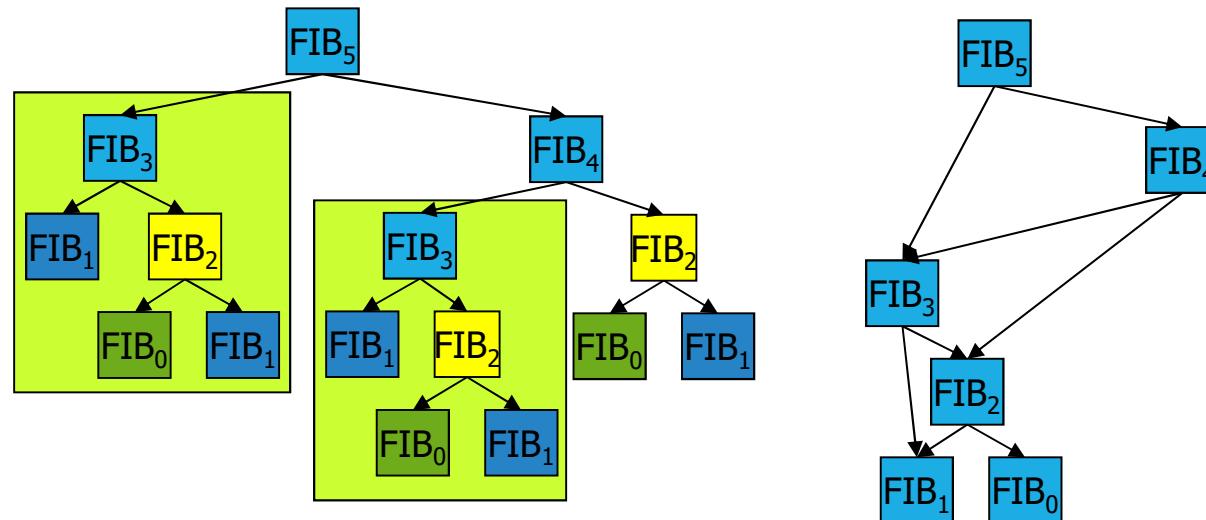
Il paradigma della Programmazione Dinamica

Paolo Camurati



Limiti della ricorsione

- Ipotesi di indipendenza dei sottoproblemi
- Memoria occupata



Paradigma alternativo: **Programmazione Dinamica**:

- memorizza le soluzioni ai sottoproblemi man mano che vengono trovate
- prima di risolvere un sottoproblema, controlla se è già stato risolto
- riusa le soluzioni ai sottoproblemi già risolti
- meglio del divide et impera per sottoproblemi condivisi

- procede:
 - bottom-up, mentre il divide et impera è top-down
 - top-down e si dice ricorsione con memorizzazione o **memoization**
- applicabile:
 - a problemi di ottimizzazione
 - solo se sono verificate certe condizioni
- passi:
 - verifica di applicabilità
 - soluzione ricorsiva come «ispirazione»
 - costruzione bottom-up iterativa della soluzione.

Esempio: le catene di montaggio

- Problema di ottimizzazione
 - Risolvibile con:
 - i modelli del Calcolo Combinatorio
 - il paradigma divide et impera ricorsivo
 - la programmazione dinamica bottom-up
 - Dall'esempio si indurrà la metodologia.
- $\Theta(2^n)$
 $\Theta(n)$

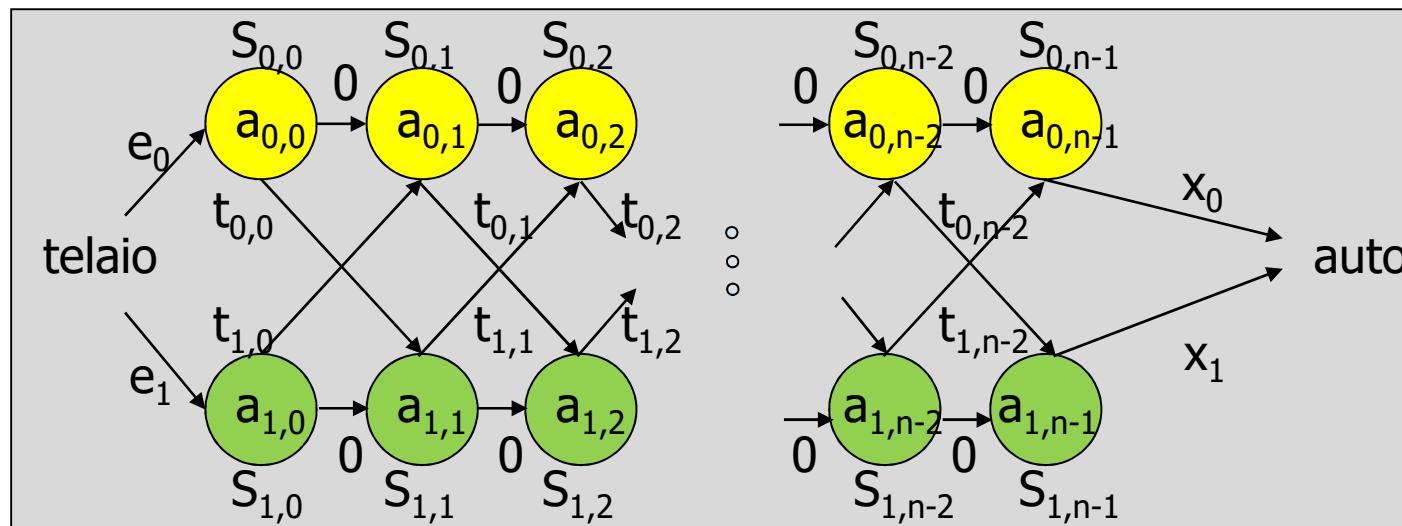
Formulazione del problema

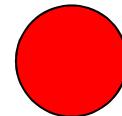
- Da **telaio** \Rightarrow ad **automobile**: passaggio per n stazioni di montaggio
- Fabbrica con:
 - 2 catene di n stazioni $S_{i,j}$ ciascuna ($0 \leq i \leq 1$, $0 \leq j < n$)
 - coppie di stazioni corrispondenti $S_{0,j}$ e $S_{1,j}$ svolgono la stessa funzione ma con tempi di lavorazione $a_{0,j}$ e $a_{1,j}$ diversi
 - tempo nullo per trasferimento da una stazione $S_{i,j-1}$ alla successiva nella stessa catena $S_{i,j}$

catena

stazione

- tempo $t_{i,j-1}$ per passare da una stazione $S_{i,j-1}$ di una catena alla stazione successiva dell'altra catena $S_{(i+1)\%2,j}$
- tempi di entrata/uscita e_0/e_1 o x_0/x_1 in o da ciascuna catena

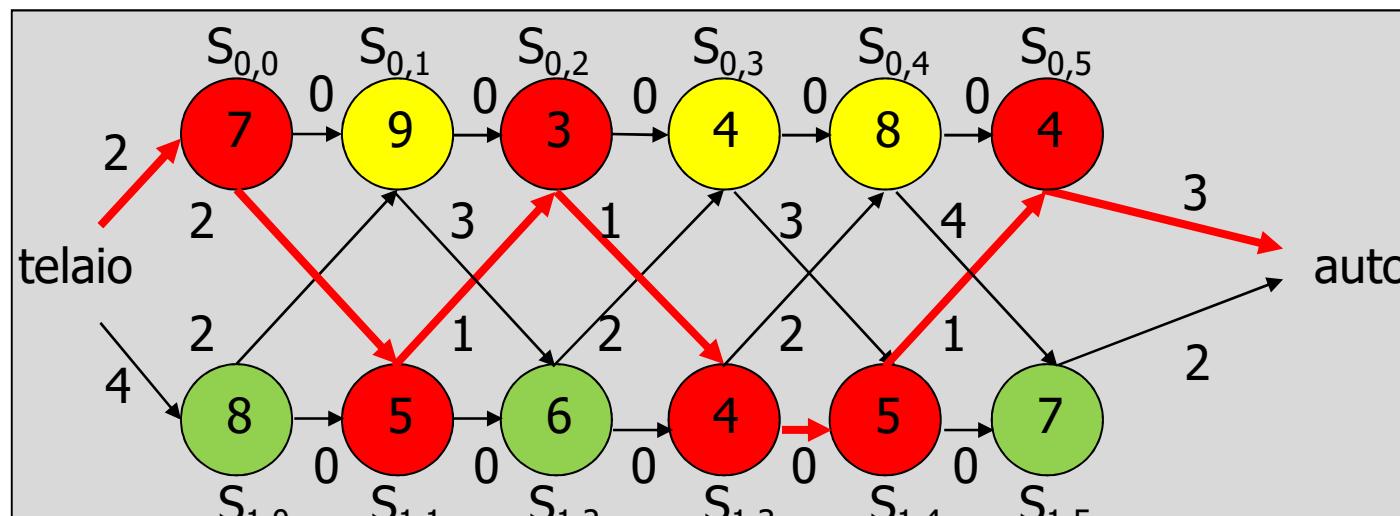




stazione usata

Scopo: costruire un'auto nel tempo minimo

Esempio:



Tempo minimo: 38

Soluzione «forza bruta»:

- modello: principio di moltiplicazione
- albero binario completo di altezza $n+1$
- DAG considerando lo scolo
- E
- enumerazione dei cammini con complessità $T(n) = \Theta(2^n)$

La Programmazione Dinamica (Bellman, 1957)



- Applicata a problemi di ottimizzazione
- Passi:
 - verifica di applicabilità: caratterizzazione della **struttura** di una soluzione ottima
 - ispirazione: definizione **ricorsiva** del **valore** di una soluzione ottima
 - soluzione:
 - tempo minimo = 38
 - calcolo **bottom-up** del **valore** di una soluzione ottima
 - costruzione di una **soluzione** ottima.

stazioni: $S_{0,0}, S_{1,1}, S_{0,2}, S_{1,3}, S_{1,4}, S_{0,5}$

Struttura della soluzione ottima

Supponiamo di raggiungere la j-esima stazione $S_{0,j}$ della catena 0 con costo (tempo) minimo $f_0[j]$ dall'entrata fino all'uscita da essa:

- se $j=0$: si entra nella catena con costo e_0 e si somma il costo $a_{0,j}$ della stazione $S_{0,j}$

$$f_0[j] = e_0 + a_{0,j}$$

costo di entrata

costo della stazione $S_{0,j}$

- se $1 \leq j < n$:
 - o si proviene dalla stessa catena (stazione $S_{0,j-1}$) con costo $f_0[j-1]$, costo di trasferimento nullo e costo $a_{0,j}$ della stazione $S_{0,j}$
$$f_0[j] = f_0[j-1] + a_{0,j}$$
 - o si proviene dall'altra catena (stazione $S_{1,j-1}$) con costo $f_1[j-1]$, costo di trasferimento $t_{1,j-1}$ e costo $a_{0,j}$ della stazione $S_{0,j}$
$$f_0[j] = f_1[j-1] + t_{1,j-1} + a_{0,j}$$

costo della stazione $S_{0,j-1}$

costo della stazione $S_{0,j}$

costo della stazione $S_{1,j-1}$

costo del trasferimento $t_{1,j-1}$

Ipotesi:

$f_0[j]$ minimo && la stazione precedente appartiene alla stessa catena ($S_{0,j-1}$)

Tesi:

il costo $f_0[j-1]$ deve essere minimo.

Dimostrazione (per assurdo):

se $f_0[j-1]$ non fosse minimo, $\exists f'_0[j-1] < f_0[j-1]$. Allora $f'_0[j] = f'_0[j-1] + a_{0,j} < f_0[j]$ e si **contraddirebbe** l'ipotesi di $f_0[j]$ minimo.

Ipotesi:

$f_0[j]$ minimo && la stazione precedente appartiene all'altra catena ($S_{1,j-1}$)

Tesi:

il costo $f_1[j-1]$ deve essere minimo.

Dimostrazione (per assurdo):

se $f_1[j-1]$ non fosse minimo, $\exists f'_1[j-1] < f_1[j-1]$. Allora $f'_0[j] = f'_1[j-1] + t_{1,j-1} + a_{0,j} < f_0[j]$ e si **contraddirebbe** l'ipotesi di $f_0[j]$ minimo.

Analogamente per $S_{1,j}$.

Conclusione: la soluzione ottima del problema comporta che siano ottime le soluzioni ai suoi sottoproblemi \Rightarrow
sottostruttura ottima.

La Programmazione Dinamica è applicabile solo a quei problemi di ottimizzazione che hanno una sottostruttura ottima.

Soluzione ricorsiva

Valore della soluzione ottima:

$$\text{soluzione} = \min(f_0[n-1] + x_0, f_1[n-1] + x_1)$$

costo di uscita dall'ultima stazione della catena 1

costo di uscita dall'ultima stazione della catena 0

costo di uscita dalla catena 0

costo di uscita dalla catena 1

$$f_0[j] = \begin{cases} f_0[0] = e_0 + a_{0,0} & j=0 \\ \min(f_0[j-1]+a_{0,j}, f_1[j-1]+t_{1,j-1}+a_{0,j}) & 1 \leq j < n \end{cases}$$
$$f_1[j] = \begin{cases} f_1[0] = e_1 + a_{1,0} & j=0 \\ \min(f_1[j-1]+a_{1,j}, f_0[j-1]+t_{0,j-1}+a_{1,j}) & 1 \leq j < n \end{cases}$$

```

int mCostR(int **a, int **t, int *e, int *x, int j, int i) {
    int ris;
    if (j==0)           solo calcolo del costo minimo
        return e[i] + a[i][j];
    ris = min(
        mCostR(a,t,e,x,j-1,i)+a[i][j],           i: catena corrente
        mCostR(a,t,e,x,j-1,(i+1)%2)+t[(i+1)%2][j-1]+a[i][j]
    );
    return ris;          (i+1)%2: altra catena
}

int assembly_line(int **a, int **t, int *e, int *x, int j){
    return min(
        mCostR(a,t,e,x,j, 0) + x[0],
        mCostR(a,t,e,x,j, 1) + x[1]
    );
}

```



01assembly_line

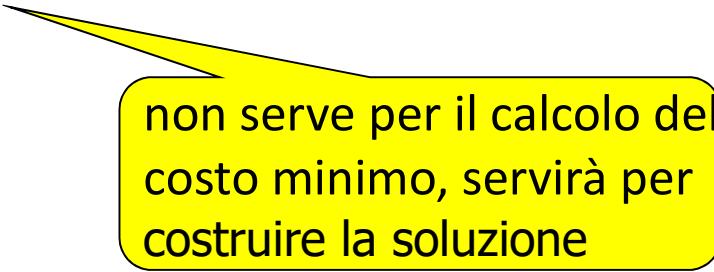
Limiti della soluzione ricorsiva

- complessità: $T(n) = \Theta(2^n)$

Soluzione ottima: calcolo bottom-up del valore

Strutture dati:

- tabella $f[0...1, 0...n-1]$ per memorizzare i costi $f[i, j]$ e identificare il costo minimo
- tabella $I[0...1, 0...n]$ per tenere traccia della catena di montaggio in cui la stazione $j-1$ è usata nel percorso a costo minimo per raggiungere la stazione j



non serve per il calcolo del costo minimo, servirà per costruire la soluzione

Passi:

- iniziale: calcolare i costi (minimi per definizione) di $f_0[0]$ e $f_1[0]$
- intermedi: per ogni stazione intermedia di ciascuna catena decidere se costa meno raggiungerla dalla precedente:
 - rimanendo nella stessa catena
 - proveniendo dall'altra catena
 - e calcolare il costo minimo
- finale: decidere se uscire dalla stazione $n-1$ esima della catena 0 o 1.

calcolo del costo minimo

```
int assembly_1lineDP(int **a, int **t, int *e, int *x,
                     int **f, int **l, int n){
    int j, res;

    f[0][0] = e[0] + a[0][0]; // passo iniziale catena 0
    f[1][0] = e[1] + a[1][0]; // passo iniziale catena 1
```

passo iniziale catena 0

passo iniziale catena 1

```
for (j=1; j<n; j++) {  
    if (f[0][j-1]+a[0][j]<=f[1][j-1]+t[1][j-1]+a[0][j]) {  
        f[0][j]=f[0][j-1]+a[0][j];  
        l[0][j]=0;  
    }  
    else {  
        f[0][j]=f[1][j-1]+t[1][j-1]+a[0][j];  
        l[0][j]=1;  
    }  
    if (f[1][j-1]+a[1][j]<=f[0][j-1]+t[0][j-1]+a[1][j]) {  
        f[1][j]=f[1][j-1]+a[1][j];  
        l[1][j]=1;  
    }  
    else {  
        f[1][j]=f[0][j-1]+t[0][j-1]+a[1][j];  
        l[1][j]=0;  
    }  
}
```

passi intermedi

catena 0: vengo da catena 0

catena 0: vengo da catena 1

catena 1: vengo da catena 1

catena 1: vengo da catena 0

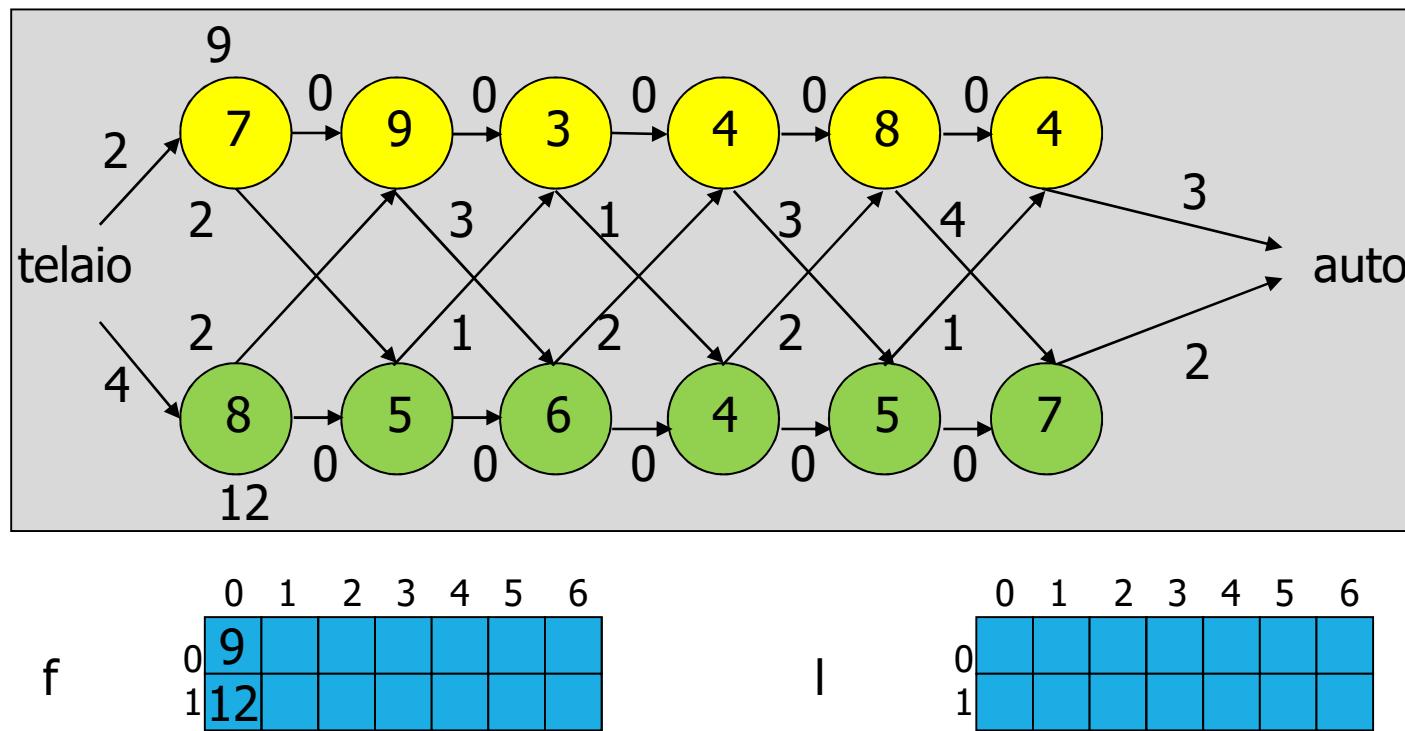
passo finale

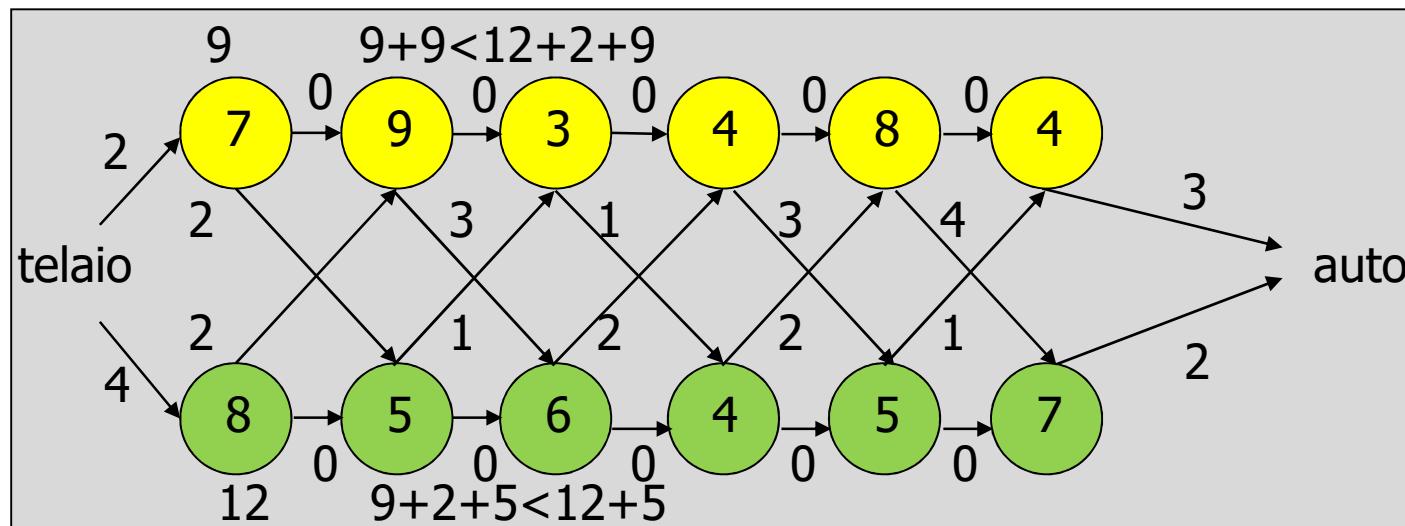
```
if (f[0][n-1] + x[0] <= f[1][n-1] + x[1]) {  
    res = f[0][n-1] + x[0];  
    l[0][n] = 0; l[1][n] = 0;  
}  
else {  
    res = f[1][n-1] + x[1];  
    l[1][n] = 1; l[0][n] = 1;  
}  
return res;  
}
```

esco dalla catena 0

esco dalla catena 1

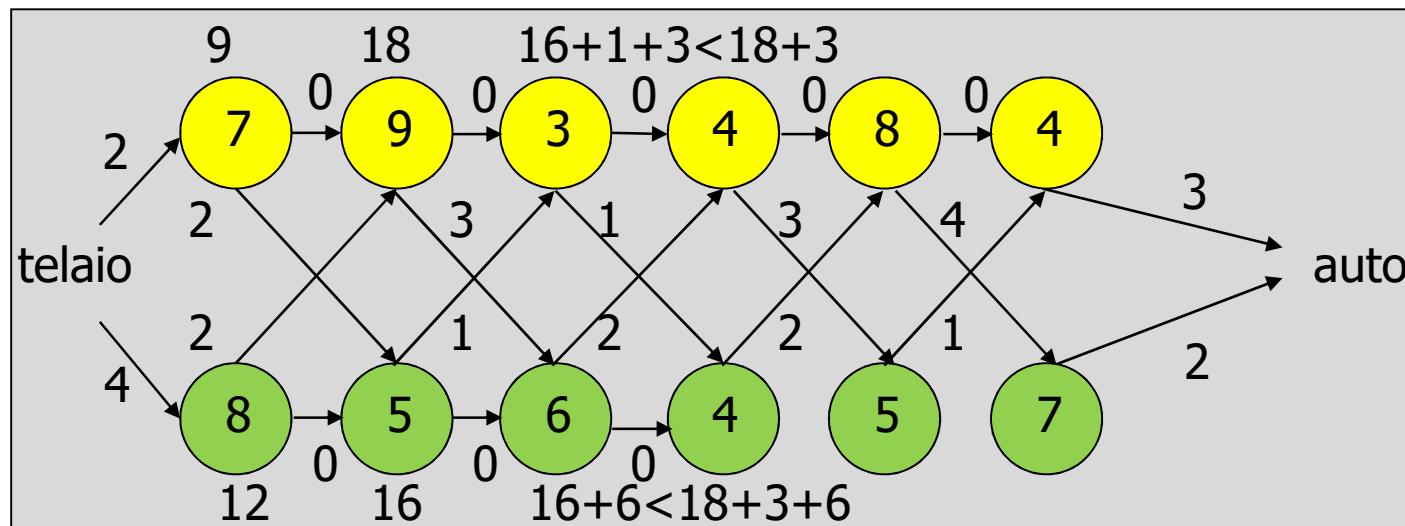
Esempio





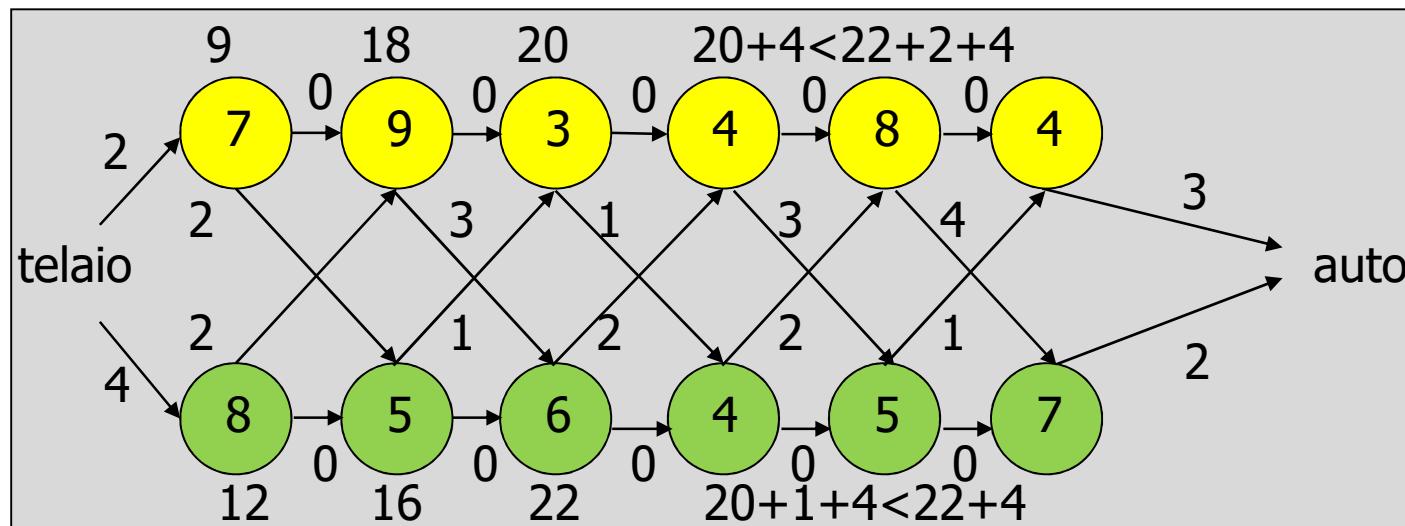
	0	1	2	3	4	5	6
f	9	18					
	12	16					

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



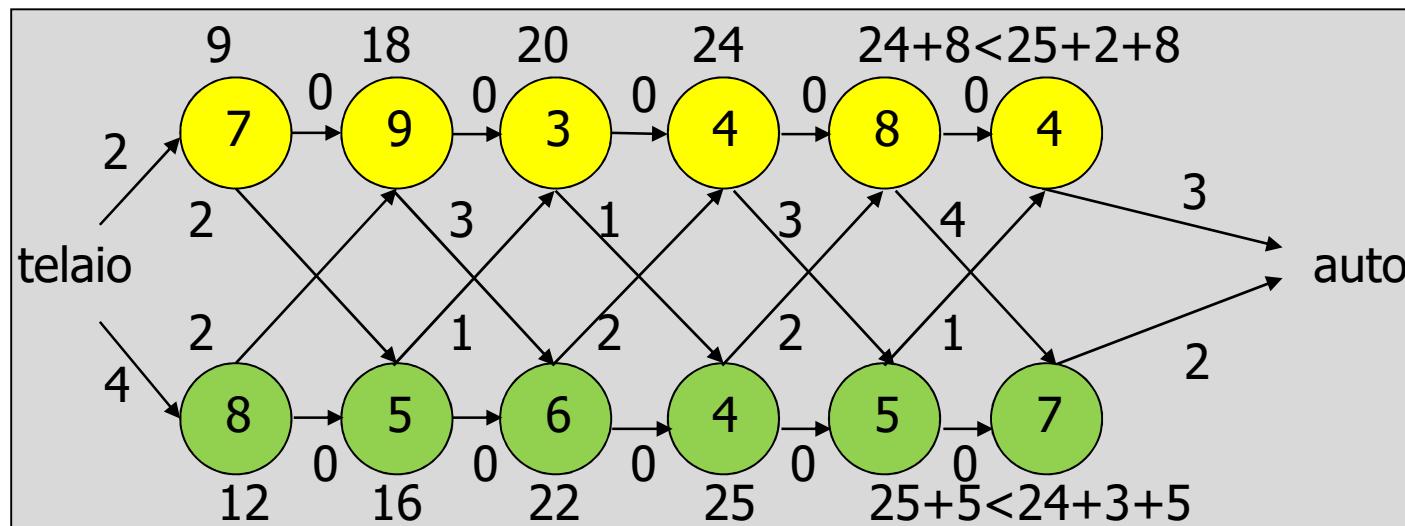
	0	1	2	3	4	5	6
f	9	18	20				
	12	16	22				

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



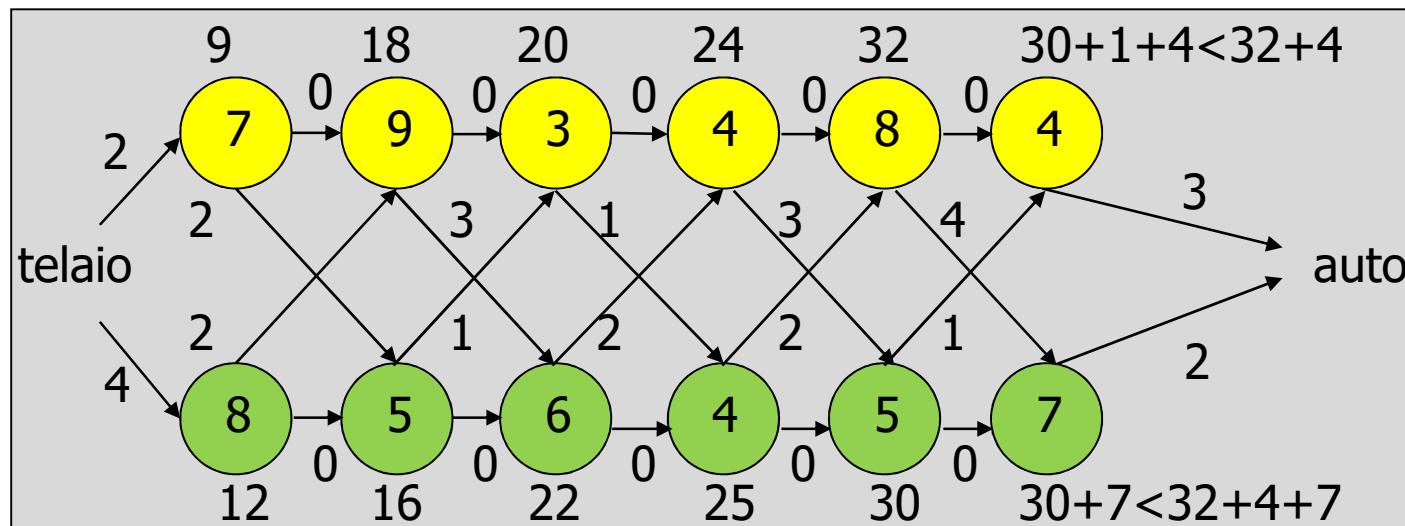
	0	1	2	3	4	5	6
f	9	18	20	24			
	12	16	22	25			

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



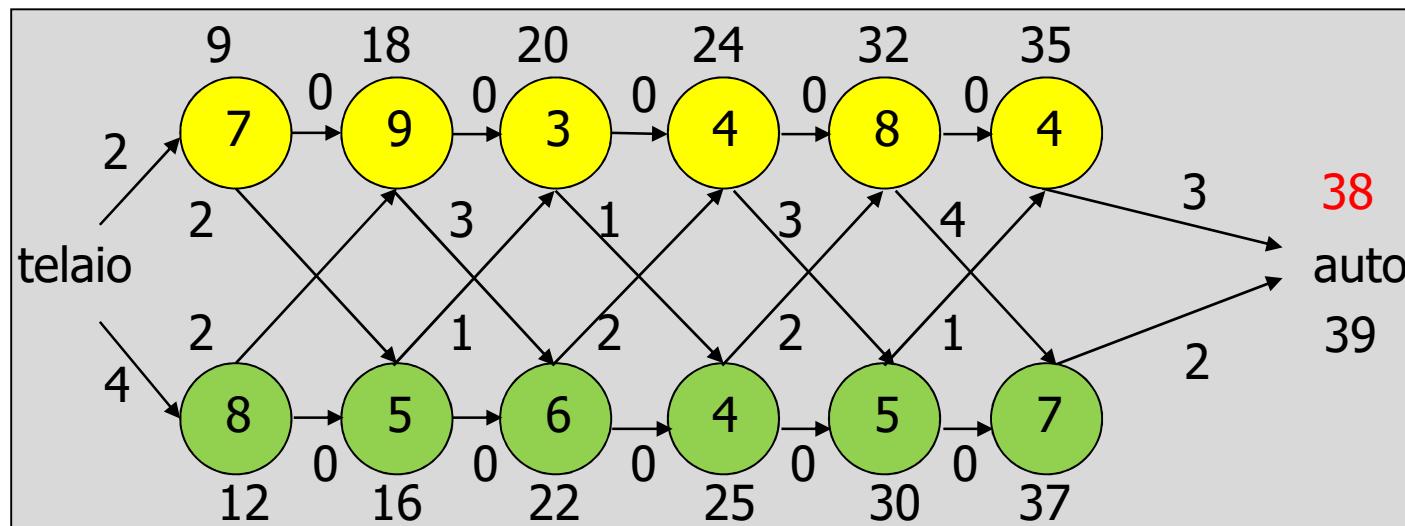
	0	1	2	3	4	5	6
f	9	18	20	24	32		
	12	16	22	25	30		

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



	0	1	2	3	4	5	6
f	9	18	20	24	32	35	
	12	16	22	25	30	37	

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



	0	1	2	3	4	5	6
f	9	18	20	24	32	35	38
	12	16	22	25	30	37	39

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

Complessità

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

rispetto al costo esponenziale nel tempo della soluzione ricorsiva.

Per la Programmazione Dinamica in generale:

$T(n) = \Theta(\text{numero di sottoproblemi} \times \text{costo di ricombinazione delle loro soluzioni ottime})$.

In questo caso: n sottoproblemi, costo di ricombinazione unitario.

Costruzione di una soluzione ottima

```
void displaySol(int **l, int i, int n) {  
    if (n==0)  
        return;  
    displaySol(l, l[i][n-1], n-1);  
    printf("line %d station %d\n", i, n-1);  
}
```

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

chiamata di `displaySol` con $i=0$

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

Soluzione = $\{S_{0,5}\}$

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

Soluzione = $\{S_{1,4}, S_{0,5}\}$

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

Soluzione = $\{S_{1,3}, S_{1,4}, S_{0,5}\}$

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

Soluzione = $\{S_{0,2}, S_{1,3}, S_{1,4}, S_{0,5}\}$

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

Soluzione = $\{S_{1,1}, S_{0,2}, S_{1,3}, S_{1,4}, S_{0,5}\}$

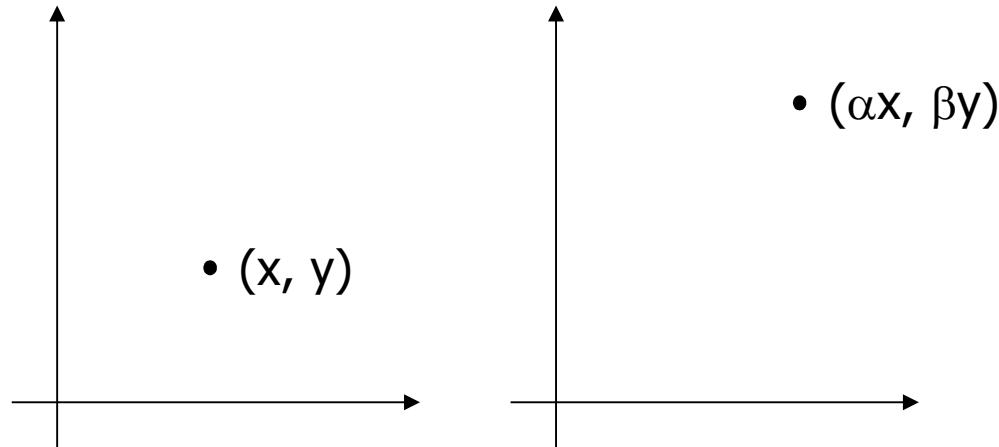
	0	1	2	3	4	5	6
0	0	1	0	0	1	0	0
1	0	1	0	1	1	0	0

Soluzione = $\{S_{0,0}, S_{1,1}, S_{0,2}, S_{1,3}, S_{1,4}, S_{0,5}\}$

Grafica e moltiplicazione di matrici

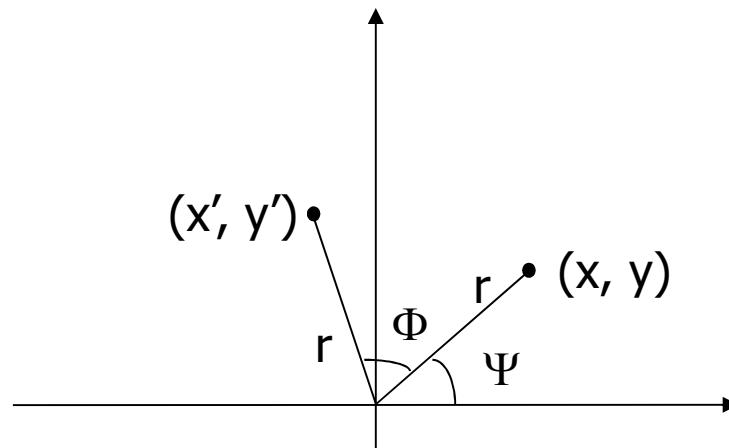
- Scena tridimensionale come insieme di triangoli nello spazio
- Triangolo individuato da 4 coordinate: assi x, y e z e dimensione fittizia (per scalamento)
- Operazioni grafiche elementari: scalamento, rotazione e traslazione di figure geometriche

Scalamento



$$[x, y, 1] \cdot \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{bmatrix} = [\alpha x, \beta y, 1]$$

Rotazione



$$x = r \cos \Psi$$

$$y = r \sin \Psi$$

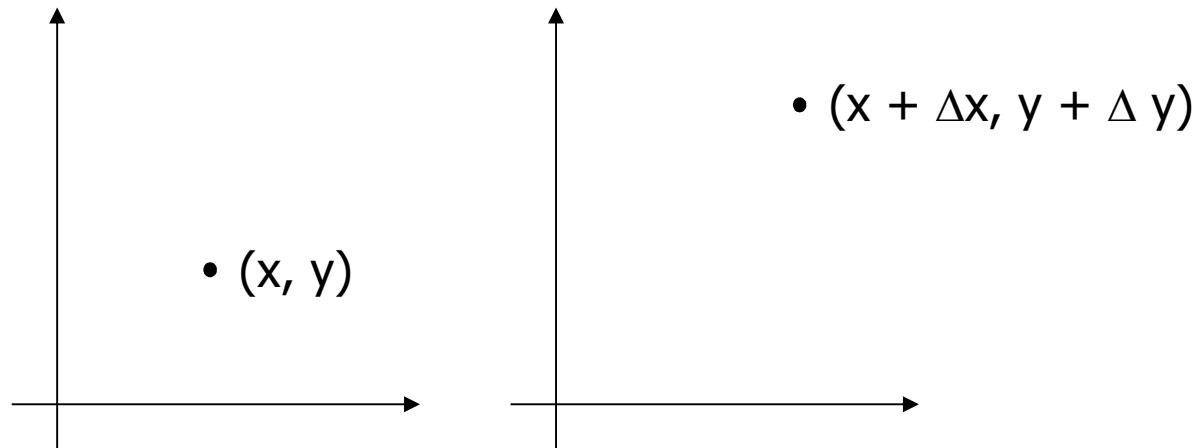
$$\cos(\Phi + \Psi) = \cos \Phi \cos \Psi - \sin \Phi \sin \Psi$$

$$\sin(\Phi + \Psi) = \sin \Phi \cos \Psi + \cos \Phi \sin \Psi$$

$$\begin{aligned}
 x' &= r \cos(\Phi + \Psi) \\
 &= r \cos\Phi \cos\Psi - r \sin\Phi \sin\Psi \\
 &= x \cos\Phi - y \sin\Phi \\
 y' &= r \sin(\Phi + \Psi) \\
 &= r \sin\Phi \cos\Psi + r \cos\Phi \sin\Psi \\
 &= x \sin\Phi + y \cos\Phi
 \end{aligned}$$

$$[x, y, 1] \cdot \begin{bmatrix} \cos\Phi & \sin\Phi & 0 \\ -\sin\Phi & \cos\Phi & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x', y', 1]$$

Traslazione



$$[x, y, 1] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \Delta x & \Delta y & 1 \end{bmatrix} = [x + \Delta x, y + \Delta y, 1]$$

Trasformazione:

$$[x, y, z, 1] \cdot A_1 \cdot A_2 \cdot \dots \cdot A_n$$

Stessa trasformazione applicata a punti diversi \Rightarrow calcolo una volta per tutte del prodotto

$$A_1 \cdot A_2 \cdot \dots \cdot A_n$$

Prodotto di 2 matrici

- Due matrici A $nr_1 \times nc_1$ e B $nr_2 \times nc_2$ sono compatibili se e solo se $nc_1 = nr_2$
- Ipotesi di semplificazione: matrici quadrate di dimensione $n \times n$
- Algoritmo semplice: 3 cicli annidati, complessità per matrici quadrate $T(n) = \Theta(n^3)$

```
void matMult(int **A,int **B,int **C,int nr1,int nc1,int nc2){  
    int i, j, k;  
    for (i=0; i<nr1; i++)  
        for (j=0; j<nc2; j++) {  
            C[i][j] = 0;  
            for (k=0; k<nc1; k++)  
                C[i][j] = C[i][j] + A[i][k]*B[k][j];  
        }  
}
```



02matr_mult

Prodotto in catena di n matrici

Data la sequenza di n matrici compatibili a 2 a 2 $A_1, A_2, A_3, \dots, A_n$ dove A_i ha dimensioni $p_{i-1} \times p_i$ con $i = 1, 2, \dots, n$ calcolare il prodotto

$$A_1 \cdot A_2 \cdot A_3 \cdot \dots \cdot A_n$$

Le dimensioni delle matrici sono memorizzate in un vettore di $n+1$ interi p .

Parentesizzazione

- Definisce l'ordine di applicazione delle operazioni di prodotto di due matrici con costo minimo
- Esempio: $A_1 \cdot A_2 \cdot A_3 \cdot A_4$ 5 parentesizzazioni possibili

$$(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)))$$

$$(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$$

$$((A_1 \cdot A_2) \cdot (A_3 \cdot A_4))$$

$$((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$$

$$(((A_1 \cdot A_2) \cdot A_3) \cdot A_4)$$

Costi

Date A_1 $p_0 \times p_1$ e A_2 $p_1 \times p_2$, il costo di $A_1 \cdot A_2$ è legato al numero di moltiplicazioni scalari $p_0 \times p_1 \times p_2$

Esempio: $A_1 \cdot A_2 \cdot A_3$ dove A_1 10x100, A_2 100x5, A_3 5x50

- Parentesizzazione #1: $(A_1 \cdot A_2) \cdot A_3$
 - costo di $A_1 \cdot A_2$ $10 \times 100 \times 5 = 5000$, risultato A_{12} 10x5
 - costo di $A_{12} \cdot A_3$ $10 \times 5 \times 50 = 2500$
 - costo totale 7500
- Parentesizzazione #2: $A_1 \cdot (A_2 \cdot A_3)$
 - costo di $A_2 \cdot A_3$ $100 \times 5 \times 50 = 25000$, risultato A_{23} 100x50
 - costo di $A_1 \cdot A_{23}$ $10 \times 100 \times 50 = 50000$
 - costo totale 75000

Numero di parentesizzazioni

Per $n \geq 2$ la catena si può spezzare in 2 al punto k , con $1 \leq k \leq n-1$.
Il numero di parentesizzazioni è il prodotto del numero di parentesizzazioni delle 2 catene (la prima lunga k , la seconda lunga $n-k$)

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{1 \leq k \leq n-1} P(k) \cdot P(n-k) & n \geq 2 \end{cases}$$

Si dimostra che $P(n) = C(n-1)$
dove $C(n)$ è detto numero di Catalan e vale

$$C(n) = \frac{1}{(n+1)} \binom{2n}{n} = \Omega(4^n / n^{3/2})$$

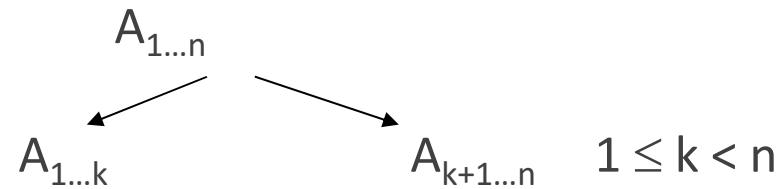
Struttura della soluzione ottima

Notazione:

è una matrice!

$$A_{i \dots j} = A_i \cdot A_{i+1} \cdot \dots \cdot A_j$$

Divisione in sottoproblemi



Costo di $A_{1\dots n}$:

costo di $A_{1\dots k}$ + costo di $A_{k+1\dots n}$ + costo del prodotto $A_{1\dots k} \cdot A_{k+1\dots n}$

Perché sia ottima la soluzione di $A_{1\dots n}$ devono essere ottime le soluzioni di $A_{1\dots k}$ e $A_{k+1\dots n}$.

Dimostrazione per assurdo: se la soluzione di $A_{1\dots k}$ non fosse ottima, ne esisterebbe una migliore, quindi non sarebbe ottima la soluzione di $A_{1\dots n}$. Analogamente per $A_{k+1\dots n}$

Problema con sottostruttura ottima  applicabilità del paradigma della programmazione dinamica

Soluzione ricorsiva

Valore della soluzione ottima:

sottoproblema: determinare il costo minimo della parentesizzazione di $A_{i \dots j}$ con $1 \leq i \leq j \leq n$.

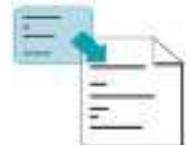
$m[i, j]$: costo minimo per $A_{i \dots j}$

$$m[i, j] = \begin{cases} 0 & \text{se } i = j \\ \min\{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{se } i < j \\ & i \leq k < j \end{cases}$$

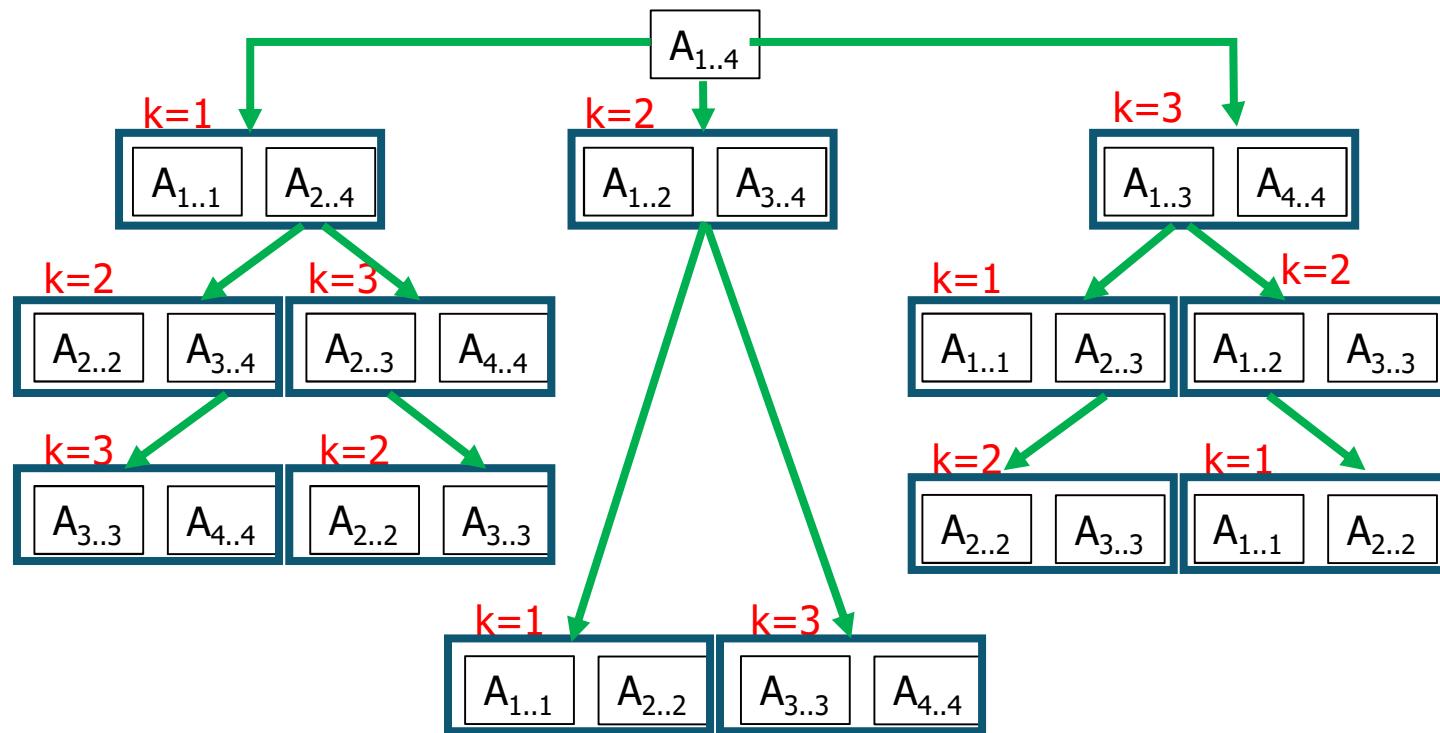
$s[i, j]$ contiene il valore di k che dà una parentesizzazione ottima nella divisione di $A_{i \dots j}$

solo calcolo del costo minimo

```
int minCostR(int *p, int i, int j, int minCost) {  
    int k, cost;  
    if (i == j)  
        return 0;  
    for (k=i; k<j; k++) {  
        cost = minCostR(p, i, k, minCost) +  
               minCostR(p, k+1, j, minCost) +  
               p[i-1]*p[k]*p[j];  
        if (cost < minCost)  
            minCost = cost;  
    }  
    return minCost;  
}  
  
int matrix_chainR(int *p, int n) {  
    return minCostR(p, 1, n, INT_MAX);  
}
```

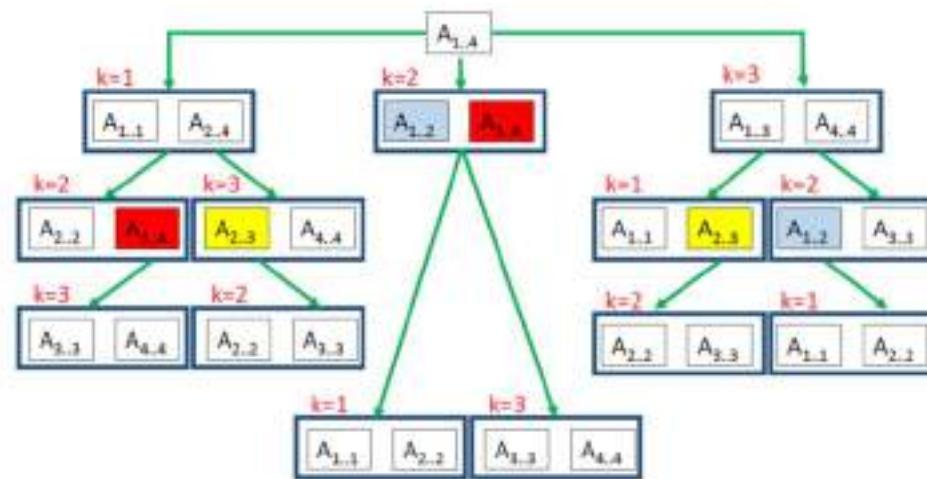


03matr_chain_mult



Limiti della soluzione ricorsiva

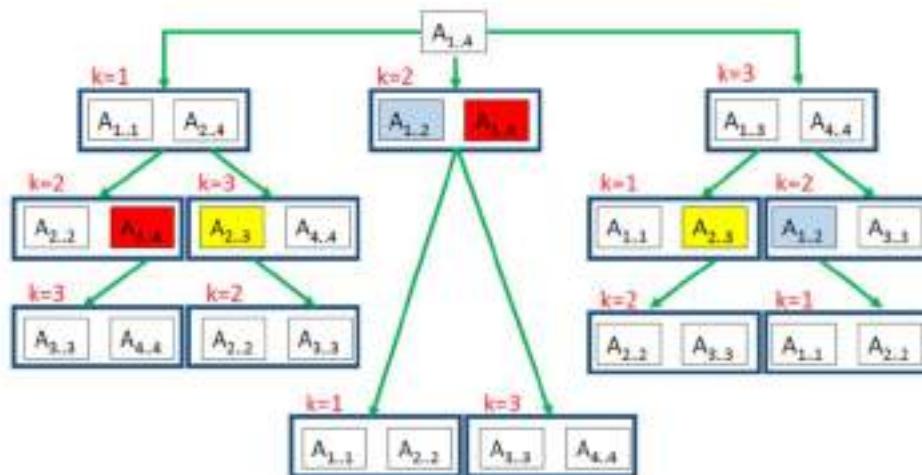
- assunzione di indipendenza dei sottoproblemi



- complessità: $T(n) = O(2^n)$

Numero di sottoproblemi indipendenti: combinazioni ripetute di n elementi presi a 2 a 2

$$C'_{n,2} = \frac{(n+2-1)!}{2!(n-1)!} = \frac{(n+1)!}{2!(n-1)!} = \frac{n(n+1)}{2} = \Theta(n^2)$$



Soluzione ottima: calcolo bottom-up del valore

Strutture dati:

- A_i matrice $p_{i-1} \times p_i$ con $i=1, 2, \dots, n$
- vettore p delle dimensioni
- tabella $m[1\dots n, 1\dots n]$ per memorizzare i costi $m[i, j]$ e identificare il costo minimo
- tabella $s[1\dots n, 1\dots n]$ per tenere traccia del valore ottimo di k per ricostruire la soluzione



non serve per il calcolo del costo minimo, servirà per costruire la soluzione

Passi:

- catene lunghe 1 ($A_{i...j}$ con $i=j$): costi nulli $m[i][j]=0 \forall i=j$
- catene lunghe 2 ($A_{1...2}, A_{2...3} \dots A_{n-1...n}$): nessuna scelta ($k=i$),
costi fissi $m[i][j]=p_{i-1} \times p_i \times p_j$
- catene lunghe 3
 - $A_{1...3}$: scelta per
 - $k=1$ usando $m[1][1], m[2][3]$ e $p_0 \times p_1 \times p_3$
 - $k=2$ usando $m[1][2], m[3][3]$ e $p_0 \times p_2 \times p_3$
 - $A_{2...4}$: scelta per
 - $k=2$ usando $m[2][2], m[3][4]$ e $p_1 \times p_2 \times p_4$
 - $k=3$ usando $m[2][3], m[4][4]$ e $p_1 \times p_3 \times p_4$
 - $A_{n-2...n}$: scelta per
 - $k=n-2$ usando $m[n-2][n-2], m[n-1][n]$ e $p_{n-3} \times p_{n-2} \times p_n$
 - $k=n-1$ usando $m[n-2][n-1], m[n][n]$ e $p_{n-3} \times p_{n-1} \times p_n$
- etc. etc.

calcolo del costo minimo e della soluzione ottima

```
int matrix_chainDP(int *p, int n) {  
    int i, l, j, k, q, **m, **s;  
  
    m = calloc((n+1), sizeof(int *));  
    s = calloc((n+1), sizeof(int *));  
  
    for (i = 0; i <= n; i++) {  
        m[i] = calloc((n+1), sizeof(int));  
        s[i] = calloc((n+1), sizeof(int));  
    }
```

costo 0 in quanto
catene lunghe 1

ciclo su lunghezza crescente delle catene

```

for (l = 2; l <= n; l++)
    for (i = 1; i <= n-l+1; i++) {
        j = i+l-1;
        m[i][j] = INT_MAX;
        for (k = i; k <= j-1; k++) {
            q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
            if (q < m[i][j]) {
                m[i][j] = q;
                s[i][j] = k;
            }
        }
    }
    displaySol(s, 1, n);
    return m[1][n];
}

```

identificazione di i e j

identificazione di k

calcolo costo

scelta

visualizzazione della soluzione ottima

Esempio

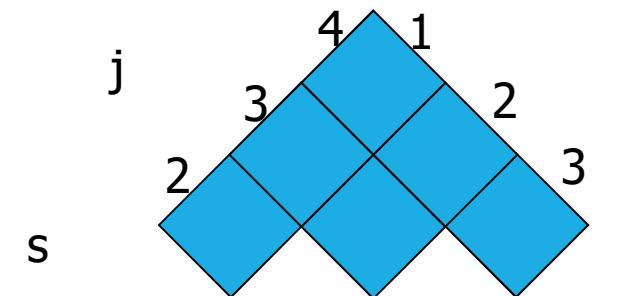
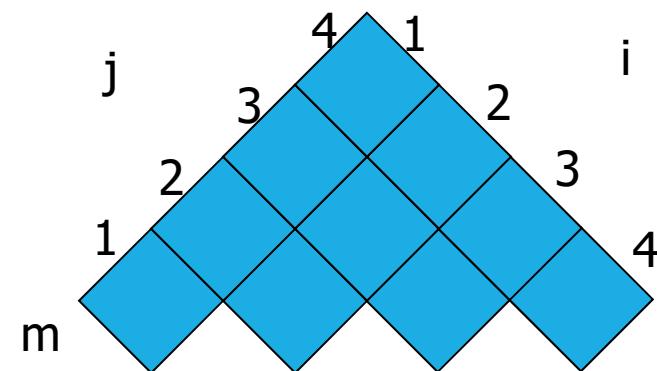
$A_1 \quad 4 \times 4$

$A_2 \quad 4 \times 6$

$A_3 \quad 6 \times 15$

$A_4 \quad 15 \times 10$

$p \quad \boxed{4 \quad 4 \quad 6 \quad 15 \quad 10}$



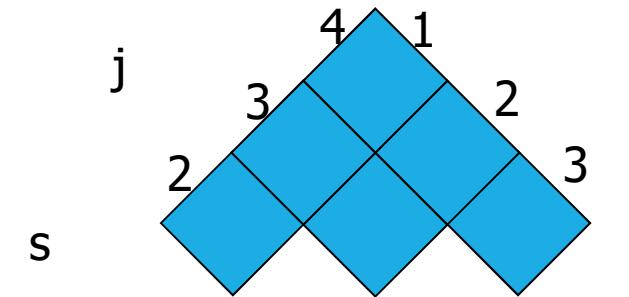
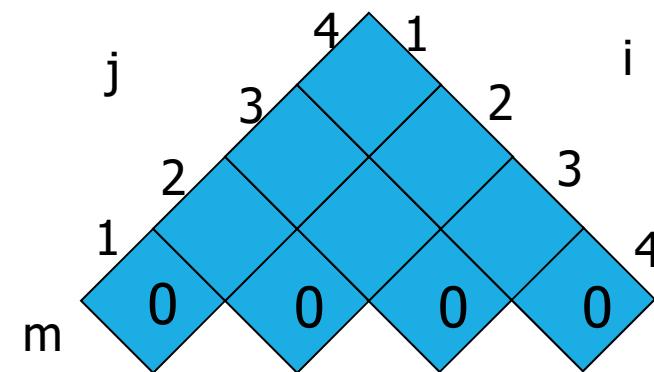
$m[1,1] = 0$

$m[2,2] = 0$

$m[3,3] = 0$

$m[4,4] = 0$

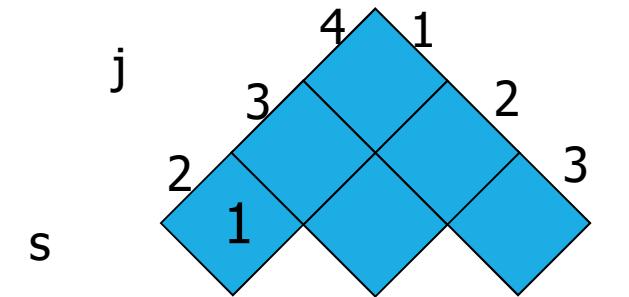
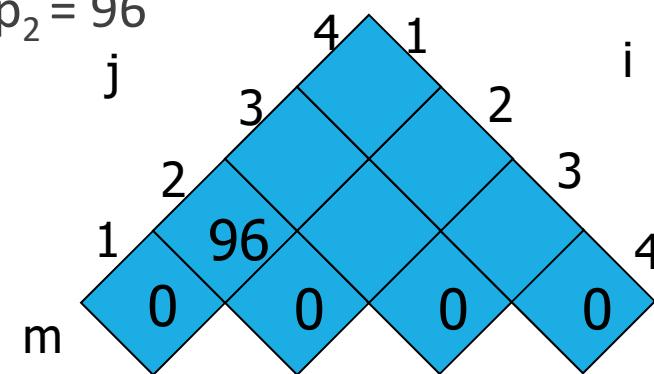
Catene lunghe 1



$$m[1,2] = m[1,1] + m[2,2] + p_0 p_1 p_2 = 96$$

$k=1$

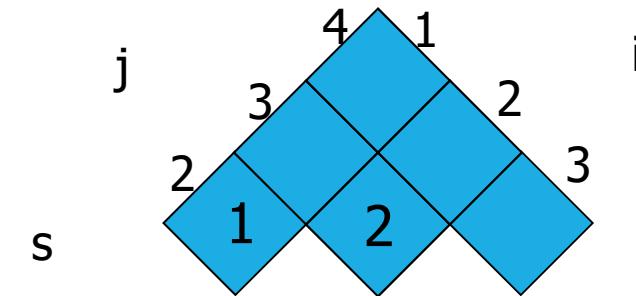
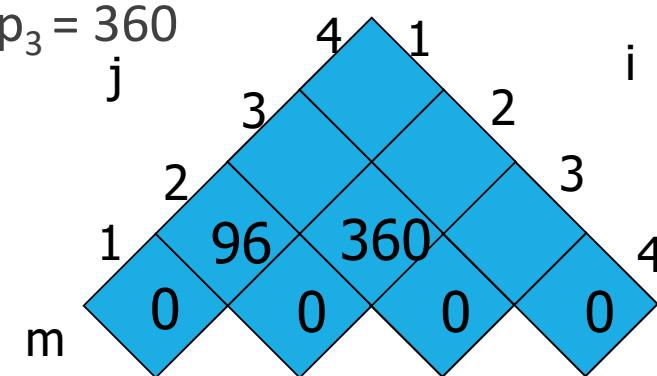
Catene lunghe 2: $A_{1..2}$



$$m[2,3] = m[2,2] + m[3,3] + p_1 p_2 p_3 = 360$$

$k=2$

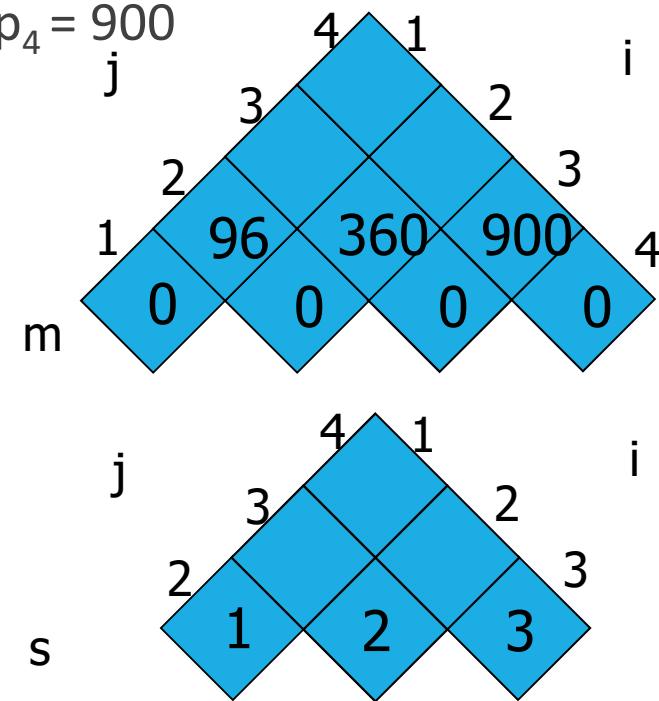
Catene lunghe 2: $A_{2..3}$



$$m[3,4] = m[3,3] + m[4,4] + p_2 p_3 p_4 = 900$$

$k=3$

Catene lunghe 2: $A_{3..4}$



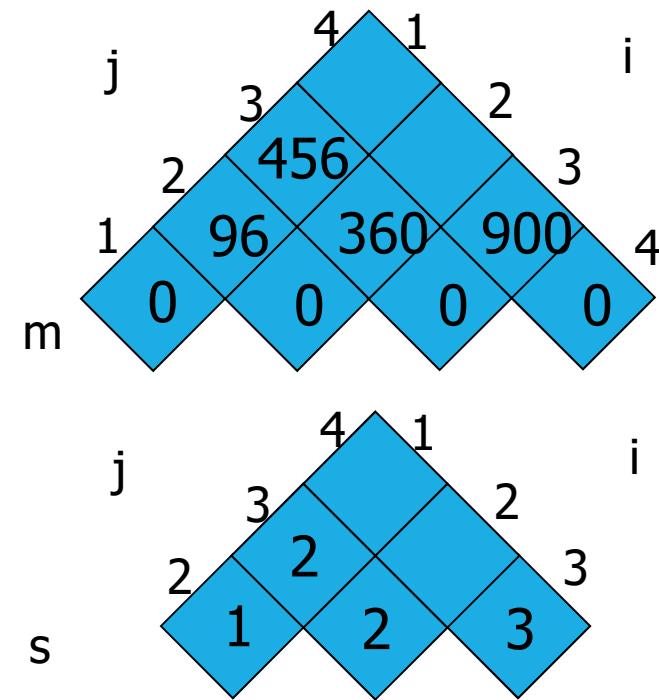
$$m[1,3] = m[1,1] + m[2,3]$$
$$+ p_0 p_1 p_3 = 360 + 240 = 600$$

$k=1$

$$m[1,3] = m[1,2] + m[3,3]$$
$$+ p_0 p_2 p_3 = 96 + 360 = \textcolor{green}{456}$$

$k=2$

Catene lunghe 3: $A_{1..3}$



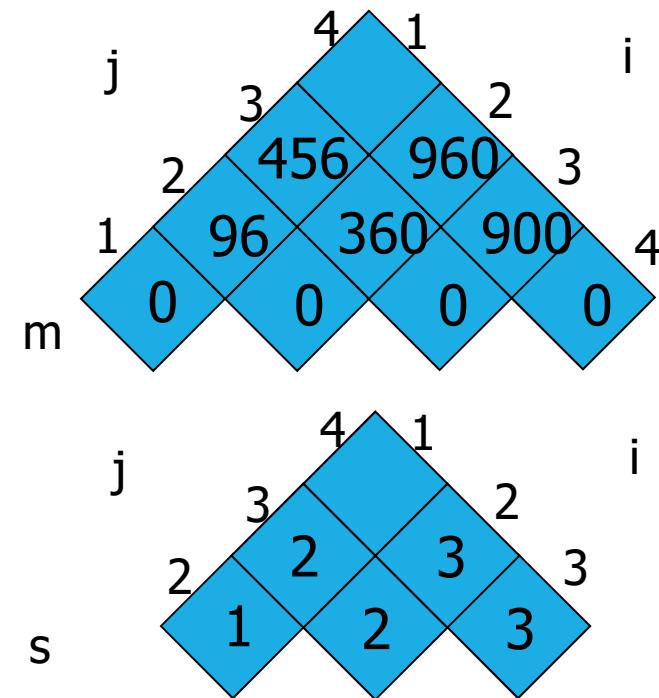
$$m[2,4] = m[2,2] + m[3,4]$$
$$+ p_1 p_2 p_4 = 900 + 240 = 1140$$

$k=2$

$$m[2,4] = m[2,3] + m[4,4]$$
$$+ p_1 p_3 p_4 = 360 + 600 = 960$$

$k=3$

Catene lunghe 3: $A_{2..4}$



Catena lunga 4: $A_{1..4}$

$$m[1,4] = m[1,1] + m[2,4] \\ + p_0 p_1 p_4 = 960 + 160 = 1120$$

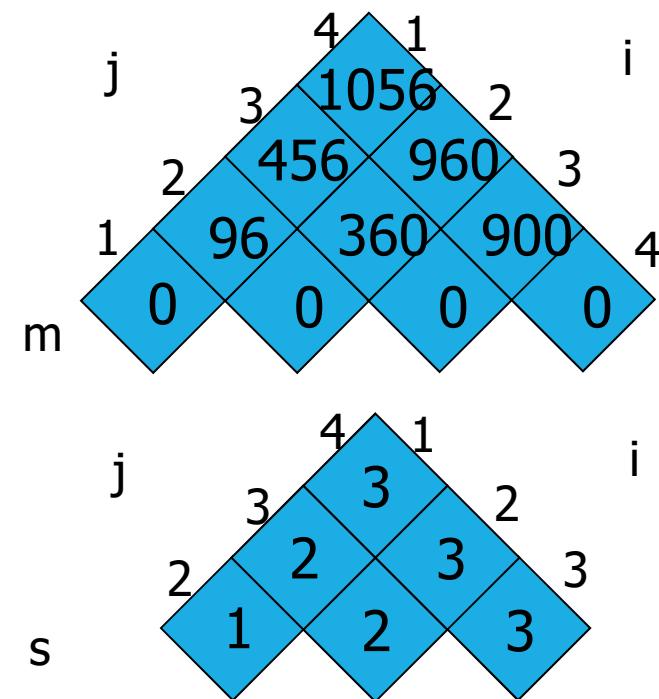
$k=1$

$$m[1,4] = m[1,2] + m[3,4] \\ + p_0 p_2 p_4 = 1236$$

$k=2$

$$m[1,4] = m[1,3] + m[4,4] \\ + p_0 p_3 p_4 = 456 + 600 = \textcolor{green}{1056}$$

$k=3$



Complessità

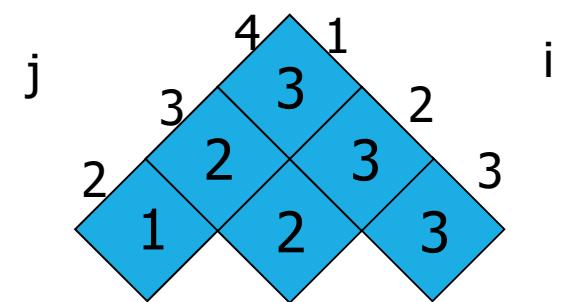
$$T(n) = \Theta(n^3)$$

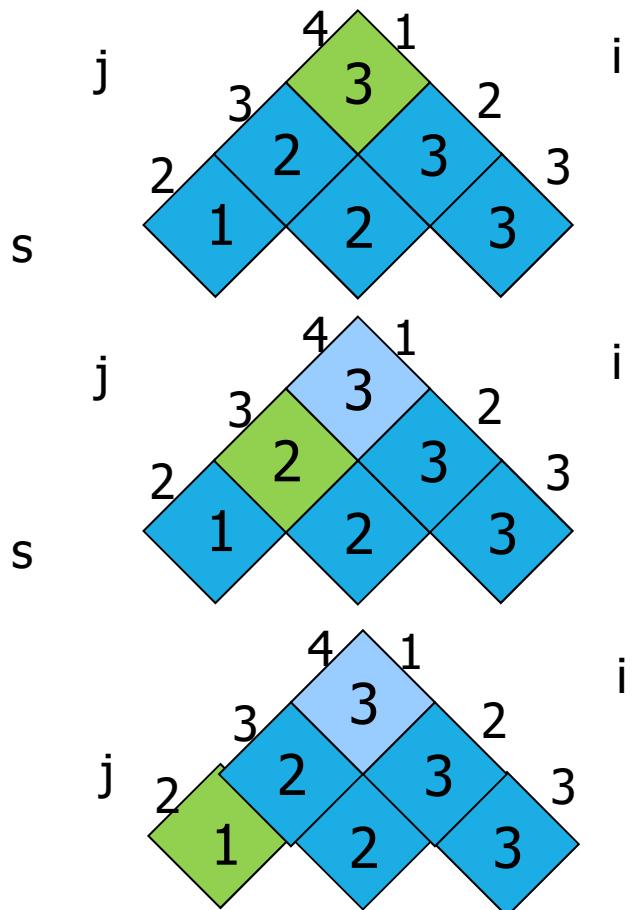
$$S(n) = \Theta(n^2)$$

rispetto al costo esponenziale nel tempo della soluzione ricorsiva

Soluzione ottima: costruzione

```
void displaySol(int **s, int i, int j) {  
    if (j <= i) {  
        printf("A%d", i);  
        return;  
    }  
    printf("(");  
    displaySol(s, i, s[i][j]);  
    printf(") x (");  
    displaySol(s, s[i][j]+1, j);  
    printf(")");  
    return;  
}
```





$$A_{1\dots 4} = (A_{1\dots 3}) \times A_4$$

$$\begin{aligned} A_{1\dots 4} &= (A_{1\dots 3}) \times A_4 \\ &= ((A_{1\dots 2}) \times A_3) \times A_4 \end{aligned}$$

$$\begin{aligned} A_{1\dots 4} &= (A_{1\dots 3}) \times A_4 \\ &= ((A_{1\dots 2}) \times A_3) \times A_4 \\ &= ((A_1) \times (A_2)) \times A_3 \times A_4 \end{aligned}$$

Applicabilità della programmazione dinamica

- Esistenza di una sottostruttura ottima
- Esistenza di molti sottoproblemi in comune:
 - vantaggio rispetto al divide et impera che assume l'indipendenza dei sottoproblemi
 - numero di sottoproblemi polinomiale nella dimensione dei dati in ingresso
 - sottoproblemi di complessità polinomiale
- Approccio bottom-up (parte da tutti i problemi elementari)

Esistenza della sottostruttura ottima

1. Dimostrare che una soluzione del problema consiste nel fare una scelta. Questa scelta genera uno o più sottoproblemi da risolvere
2. Per un dato problema, supporre di conoscere la scelta che porta a una soluzione ottima. Non interessa sapere come è stata determinata tale scelta
3. Fatta la scelta, determinare quali sottoproblemi ne derivano e quale sia il modo migliore per caratterizzare lo spazio di sottoproblemi risultante

4. Dimostrare per assurdo che le soluzioni dei sottoproblemi utilizzate all'interno della soluzione ottima del problema devono essere necessariamente ottime con la tecnica del «taglia & incolla»:
 - supporre che le soluzioni dei sottoproblemi non siano ottime
 - «tagliare» la sottosoluzione non ottima e «incollare» una sottosoluzione ottima
 - verificare che si è generata una contraddizione

Attenzione a non assumere l'esistenza della sottostruttura ottima se non è possibile farlo!

Esempio:

Dato un grafo orientato e non pesato e 2 suoi vertici u e v, trovare il cammino:

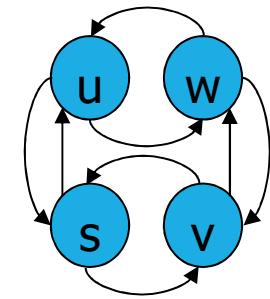
- **minimo** (formato dal minimo numero di archi). È necessariamente semplice: se non lo fosse, si potrebbe eliminare il ciclo e ridurre il numero di archi
- **massimo**: cammino semplice formato dal massimo numero di archi. Se non fosse semplice, percorrendo il ciclo infinite volte il problema perderebbe di significato

Cammino minimo tra u e v nel grafo $G=(V, E)$:

- problema banale: u e v coincidono
- esiste un cammino minimo p : $u \rightarrow_p v$
 - essendo $u \neq v$, esiste un nodo intermedio w (può anche essere u o v) che spezza il cammino p in $p_1 u \rightarrow_{p1} w$ e $p_2 w \rightarrow_{p2} v$
 - se p_1 non fosse ottimo, esisterebbe p'_1 ottimo, che sostituito a p_1 porterebbe a concludere che p non è ottimo: contraddizione
 - analogamente per p_2

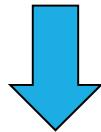
Cammino semplice massimo tra u e v nel grafo $G=(V, E)$:

- (u, w, v) è un cammino massimo semplice tra u e v
- il suo sottocammino (u, w) non è massimo, è massimo il sottocammino (u, s, v, w)
- il suo sottocammino (w, v) non è massimo, è massimo il sottocammino (w, u, s, v)

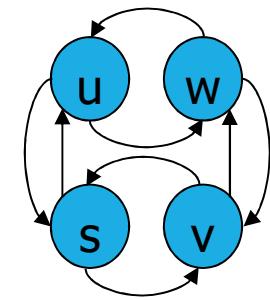


Il problema non presenta una sottostruttura ottima!

Inoltre, combinando i sottocammini semplici e massimi (u, s, v, w) e (w, u, s, v) il cammino che si ottiene non è semplice



La programmazione dinamica non è applicabile.
Il problema dei cammini **massimi** è **NP-completo**.



Divide et impera vs. programmazione dinamica

Divide et impera

- tutti i tipi di problemi
- sottoproblemi indipendenti
- top-down
- ricorsione

Programmazione dinamica

- solo problemi di ottimizzazione
- sottoproblemi in numero limitato
- sottoproblemi condivisi
- bottom-up
- iterazione

Longest Increasing Sequence

Data una sequenza di N interi

$$X = (x_0, x_1, \dots, x_{N-1})$$

si definisce **sottosequenza** di X di lunghezza k ($k \leq N$) un qualsiasi n-upla Y di k elementi di X con indici crescenti i_0, i_1, \dots, i_{k-1} .

Si ricordi:

- **sottosequenza**: indici non necessariamente contigui
- **sottostringa**: indici contigui
- **prefisso** i-esimo di lunghezza i+1 di una sequenza X: $X_i = (x_0, x_1, \dots, x_i)$

- Longest Increasing Sequence:
 - sottosequenza di lunghezza massimale
 - di elementi con valori crescenti
- Problema: determinare una LIS.
- Soluzioni:
 - ricorsione con modello del Calcolo Combinatorio (powerset)
 $(T(n) = O(2^N))$
 - programmazione dinamica se
 - applicabile \Rightarrow sottostruttura ottima
 - conveniente \Rightarrow numero polinomiale di sottoproblemi indipendenti

Applicabilità/convenienza

- Esistenza di una sottostruttura ottima: data una soluzione ottima, se non fosse ottima la soluzione al problema per ogni prefisso (sottoproblema), se ne potrebbe trovare una migliore e di conseguenza la soluzione ottima non sarebbe tale (assurdo)
- Il numero di sottoproblemi indipendenti è polinomiale ($O(N)$) nella dimensione dei dati in ingresso
- La soluzione di ciascun sottoproblema è $O(N)$
- La soluzione con programmazione dinamica ha complessità $O(N^2)$

PS: esistono algoritmi $O(N \log N)$ che esulano da questo corso.

Soluzione ricorsiva

Valore della soluzione ottima:

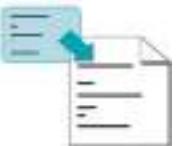
- funzione ricorsiva LISR: dato il prefisso $X_i = (x_0, x_1, \dots, x_i)$
 - se $i=0$, $c[X_i] = 1$
 - $\forall i \ 0 < i < N$ considero tutti i prefissi X_j con $0 \leq j < i$ che soddisfano la condizione $x_j < x_i$
 - l'elemento x_i può essere aggiunto in coda a formare una LIS più lunga di 1, calcolo $1 + c[X_j]$
 - prendo il massimo e lo assegno a $c[X_i]$

$$c[X_i] = \begin{cases} 1 & i=0 \\ \max(1 + c[X_j]) \text{ tale che } 0 \leq j < i \ \& x_j < x_i & i>0 \end{cases}$$

wrapper

```
int LIS(int *val) {  
    return LISR(val, N-1);  
}
```

```
int LISR(int *val, int i) {  
    int j, ris;  
    if (i == 0) → c[X0]=1  
        return 1;  
    ris = 1;  
    for (j=0; j < i; j++)  
        if (val[j] < val[i]) → max(1+c[Xj]) tale che 0≤j<i && xj<xi  
            ris = max(ris, 1 + LISR(val, j));  
    return ris;  
}
```

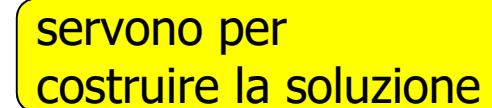


04LIS

Soluzione ottima: calcolo bottom-up del valore

Strutture dati:

- `val` vettore di input di N interi
- L vettore di N interi per memorizzare la lunghezza della LIS per ogni prefisso i -esimo
- P vettore di N interi per memorizzare l'indice dell'elemento precedente nella LIS
- `last` intero per memorizzare l'indice dell'ultimo elemento nella LIS



servono per costruire la soluzione

Passi:

- il valore minimo di una LIS è 1 (sequenza monotona decrescente)
- per lunghezze crescenti del sottovettore considerato (prefisso i da 1 a N-1)
 - individuare la LIS del prefisso i che soddisfa le condizioni
 - registrare in P l'indice del valore precedente
- stampa ricorsiva dei valori mediante percorimento di P

Esempio

val	11	7	13	15	8	14
L	1					
P	-1					

i=0
 $c[X_0] = 1$

val	11	7	13	15	8	14
L	1	1				
P	-1	-1				

i=1
 $c[X_1] = 1$

val	11	7	13	15	8	14
L	1	1	2			
P	-1	-1	0			

i=2
 $c[X_2] = 2$

val	11	7	13	15	8	14
L	1	1	2	3		
P	-1	-1	0	2		

i=3
 $c[X_3] = 3$

val	11	7	13	15	8	14
L	1	1	2	3	2	
P	-1	-1	0	2	1	

i=4
 $c[X_4] = 3$

val	11	7	13	15	8	14
L	1	1	2	3	2	3
P	-1	-1	0	2	1	2

i=5
 $c[X_5] = 3$

```

#define N 7

void LISDP(int *val) {
    int i, j, ris=1, L[N], P[N], last=1;
    L[0] = 1; P[0] = -1;
    for (i=1; i<N; i++) {
        L[i] = 1; P[i] = -1;
        for (j=0; j<i; j++)
            if ((val[j] < val[i]) && (L[i] < 1 + L[j])) {
                L[i] = 1 + L[j]; P[i] = j;
            }
        if (ris < L[i]) {
            ris = L[i]; last = i;
        }
    }
    printf("one of the Longest Increasing Sequences is ");
    LISprint(val, P, last);
    printf("and its length is %d\n", ris);
}

```

calcolo del costo minimo e di una soluzione ottima

visualizzazione di una soluzione ottima

```

void LISprint(int *val, int *P, int i) {
    if (P[i]==-1) {
        printf("%d ", val[i]);
        return;
    }
    LISprint(val, P, P[i]);
    printf("%d ", val[i]);
}

```

last = 3

val	11	7	13	15	8	14
L	1	1	2	3	2	3
P	-1	0	2	1	2	

una LIS = (11, 13, 15)

Longest Common Subsequence

Date 2 sequenze X e Y, Z è una sottosequenza comune se è sottosequenza sia di X che di Y.

Esempio:

X =

A	C	G	G	T	A	C
0	1	2	3	4	5	6

 Y =

C	T	G	A	C	A
0	1	2	3	4	5

Sottosequenza comune:

C	G	A
---	---	---

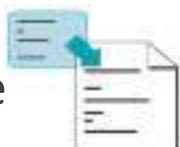
Sottosequenze comuni di lunghezza massima (LCS):

C	G	C	A
---	---	---	---

C	T	A	C
---	---	---	---

La determinazione della LCS trova applicazione nei confronti di DNA.

Essa è trattata nei lucidi di approfondimento disponibili sul Portale della Didattica.



05LCS

Lo zaino (discreto)

Dato un insieme di N oggetti ciascuno dotato di peso w_j e di valore v_j e dato un peso massimo cap , determinare il sottoinsieme S di oggetti tali che:

- $\sum_{j \in S} w_j x_j \leq \text{cap}$
- $\sum_{j \in S} v_j x_j = \text{MAX}$
- $x_j \in \{0,1\}$

Ogni oggetto o è preso ($x_j = 1$) o lasciato ($x_j = 0$).

Ogni oggetto esiste in una sola istanza.

Tipologia 3 nelle slide di programmazione

3

Esempio

N= 4

cap = 10

Nome	A	B	C	D
Valore v _i	10	6	8	9
Peso w _i	8	4	2	3

Soluzione:

insieme {B, C, D} con valore massimo 23

Struttura della soluzione ottima

- Problema $P(N, \text{cap})$
- Sottoproblema $P(i, \text{cap})$: problema per i primi i oggetti
- $S(i, \text{cap})$: soluzione ottima per $P(i, \text{cap})$
- 2 casi:
 - l'oggetto i (ultimo degli oggetti correnti) appartiene alla soluzione ottima
 - l'oggetto i non appartiene alla soluzione ottima

Dimostrazione per assurdo:

- I. l'oggetto i appartiene a $S(i, \text{cap}) \Rightarrow S(i, \text{cap}) - \{i\}$ è una soluzione ottima
se non lo fosse, allora esisterebbe una soluzione S' con valore maggiore e compatibile con la capacità. Se a S' si riaggiungesse l'oggetto i , allora la soluzione risultante, certamente compatibile con la capacità, avrebbe valore maggiore della soluzione di partenza, contraddicendo l'ipotesi di soluzione ottima.
- II. l'oggetto i non appartiene a $S(i, \text{cap}) \Rightarrow S(i, \text{cap})$ è una soluzione ottima per $P(i-1, \text{cap})$
se non lo fosse, allora esisterebbe una soluzione S' con valore maggiore e compatibile con la capacità, contraddicendo l'ipotesi di soluzione ottima.

Il numero di sottoproblemi indipendenti è $\Theta(N \cdot \text{cap})$: la programmazione dinamica è **conveniente**.

Si identificano 2 fasi:

- ottimizzazione: calcolo del massimo valore compatibile con la capacità
- ricerca: identificazione degli elementi

Approccio ricorsivo “divide et impera” al solo problema di ottimizzazione.

Soluzione ricorsiva

Valore della soluzione ottima:

- caso terminale: non ci sono oggetti ($i < 0$) o la capacità disponibile è nulla ($j = 0$)
- caso non terminale:
 - l'oggetto i -esimo non può essere preso perché, aggiungendolo alla soluzione, si eccede la capacità
 - l'oggetto i -esimo può essere preso, ma non è detto che convenga
 - se conviene, l'oggetto viene preso
 - altrimenti l'oggetto non viene preso.

Valutazione delle convenienza: termini di paragone:

- $\text{maxv}[i-1, j]$: massimo valore con capacità disponibile j considerando gli oggetti da 0 a $i-1$, quindi non l'oggetto i
- $v[i] + \text{maxv}[i-1, j-w[i]]$: valore dell'oggetto i sommato al massimo valore ottenuto considerando gli oggetti da 0 a $i-1$ e una capacità disponibile $j-w[i]$ tale da poter contenere l'oggetto i di peso $w[i]$

Se $\text{maxv}[i-1, j] \geq v[i] + \text{maxv}[i-1, j-w[i]]$

- non prendo l'oggetto i e $\text{maxv}[i, j] = \text{maxv}[i-1, j]$
- altrimenti prendo l'oggetto i e $\text{maxv}[i, j] = v[i] + \text{maxv}[i-1, j-w[i]]$.

massimo valore di i oggetti
con capacità disponibile j

$$\maxv[i, j] = \begin{cases} 0 & \text{l'oggetto } i \text{ non viene preso} \\ \maxv[i-1, j] & \text{non ci sono oggetti o capacità disponibile 0} \\ \max\{\maxv[i-1, j], \maxv[i-1, j-w[i]]+v[i]\} & \text{peso dell'oggetto } i \text{ eccede capacità disponibile } j \end{cases}$$

non ci sono oggetti
o capacità disponibile 0

peso dell'oggetto
i eccede capacità
disponibile j

se $i < 0$ o $j = 0$
se $w[i] > j$
se $w[i] \leq j$

l'oggetto i non
viene preso

se è possibile prendere l'oggetto i, prenderlo
se migliora il valore rispetto a non prenderlo

```
int maxValR(Item *items, int i, int j) {  
    if ((i < 0) || (j == 0))  
        return 0;  
    if (items[i].size >j)  
        return maxValR(items, i-1, j);  
    return max(maxValR(items, i-1, j),  
               maxValR(items, i-1, j-items[i].size)+items[i].value);  
}  
  
int KnapmaxVal(Item *items, int N, int cap) {  
    return maxValR(items, N, cap);  
}
```



06knapsack

Soluzione ottima: calcolo bottom-up del valore

N oggetti identificati da indici 1,2,..., N. Oggetto fittizio all'indice 0 di peso e valore 0.

Capacità crescenti da 0 a cap. Zaino fittizio all'indice 0 con capacità 0.

Strutture dati:

- vettore items di N elementi
- tabella $(N+1) \times (cap+1)$ maxVal[0...N, 0...cap] per memorizzare i valori e identificare il massimo
- non serve un'ulteriore struttura dati per la costruzione della soluzione

Esempio

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

$N = 4$
oggetti 1,2,3 4
 $\text{cap} = 10$

		j										
		0	1	2	3	4	5	6	7	8	9	10
0		0	0	0	0	0	0	0	0	0	0	0
1		0	0	0	0	0	0	0	0	0	0	0
i		0	0	0	0	0	0	0	0	0	0	0
2		0	0	0	0	0	0	0	0	0	0	0
3		0	0	0	0	0	0	0	0	0	0	0
4		0	0	0	0	0	0	0	0	0	0	0
		zaino fittizio										
		maxval										

Passi:

- 2 cicli **for** annidati di scansione degli N oggetti (i da 1 a N) e della capacità (j da 1 a cap)
- l'oggetto corrente i si trova in `items[i-1]`
- se il peso dell'oggetto corrente `items[i-1].size` eccede la capacità disponibile j, l'oggetto non viene preso e `maxval[i, j] = maxval[i-1, j]`
- altrimenti si valuta se l'oggetto corrente conviene come nella soluzione ricorsiva.

Valutazione delle convenienza: termini di paragone:

- $\text{maxval}[i-1, j]$: massimo valore con capacità disponibile j considerando gli oggetti da 0 a $i-1$, quindi non l'oggetto i
- $\text{items}[i-1].value + \text{maxv}[i-1, j - \text{items}[i-1].size]$: valore dell'oggetto i sommato al massimo valore ottenuto considerando gli oggetti da 0 a $i-1$ e una capacità disponibile $j - \text{items}[i-1].size$ tale da poter contenere l'oggetto i di peso $\text{items}[i-1].size$

Se $\text{maxval}[i-1, j] \geq \text{maxval}[i-1][j - \text{items}[i-1].size] + \text{items}[i-1].value$

- non prendo l'oggetto i e $\text{maxval}[i, j] = \text{maxval}[i-1, j]$
- altrimenti prendo l'oggetto i e
$$\text{maxval}[i, j] = \text{maxval}[i-1][j - \text{items}[i-1].size] + \text{items}[i-1].value$$

```

int KNPmaxvalDP(Item *items, int N, int cap) {
    int i, j, **maxval;
    // allocazione matrice maxval di dimensioni (N+1)x(cap+1)
    for (i=1; i<=N; i++)
        for (j=1; j <=cap; j++) {
            if (items[i-1].size > j)
                maxval[i][j] = maxval[i-1][j];
            else
                maxval[i][j] = max(maxval[i-1][j],
                                    maxval[i-1][j-items[i-1].size] +
                                    items[i-1].value);
        }
    printf("Maximum booty is: \n");
    displaySol(items, N, cap, maxval);
    printf("\n");
    return maxval[N][cap];
}

```

calcolo del valore massimo e
di una soluzione ottima

visualizzazione di una
soluzione ottima

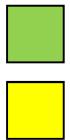
Esempio

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Oggetto 1 (i=1) valore 10, peso 8

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
i	2	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0
	maxval										

Verde: celle con valore già calcolato



Giallo: celle con valore da calcolare



Rosso: celle di confronto



$$\forall j \ 1 \leq j \leq 7$$

$\text{items}[i-1].size > j \Rightarrow$ l'oggetto 1 non viene preso
 $\Rightarrow \text{maxval}[i, j] = \text{maxval}[i-1, j]$

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Verde: celle con valore già calcolato 
 Giallo: celle con valore da calcolare 
 Rosso: celle di confronto 

Oggetto 1  (i=1) valore 10, peso 8

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	10	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0

maxval

j = 8

preso/non preso

items[i-1].size \leq j $8 \leq 8 \Rightarrow$ l'oggetto 1 può essere preso
conveniente/non conveniente

maxval[i-1,j] < maxval[i-1][j-items[i-1].size]+items[i-1].value)

0	0	10
---	---	----

\Rightarrow l'oggetto 1 conviene e viene preso maxval[i,j] = 10

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Verde: celle con valore già calcolato 
 Giallo: celle con valore da calcolare 
 Rosso: celle di confronto 

Oggetto 1  (i=1) valore 10, peso 8

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	10	10	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0

maxval

j = 9

preso/non preso

items[i-1].size \leq j 8 \leq 9 \Rightarrow l'oggetto 1 può essere preso
conveniente/non conveniente

maxval[i-1,j] < maxval[i-1][j-items[i-1].size]+items[i-1].value)

0	0	10
---	---	----

\Rightarrow l'oggetto 1 conviene e viene preso maxval[i,j] = 10

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Verde: celle con valore già calcolato 
 Giallo: celle con valore da calcolare 
 Rosso: celle di confronto 

Oggetto 1  (i=1) valore 10, peso 8

	0	1	2	3	4	5	6	7	8	9	10
j	0	0	0	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0

maxval

j = 10

preso/non preso

items[i-1].size $\leq j$ $8 \leq 10 \Rightarrow$ l'oggetto 1 può essere preso
conveniente/non conveniente

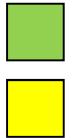
maxval[i-1,j] < maxval[i-1][j-items[i-1].size]+items[i-1].value)

	0	0	10
--	---	---	----

\Rightarrow l'oggetto 1 conviene e viene preso maxval[i,j] = 10

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Verde: celle con valore già calcolato



Giallo: celle con valore da calcolare



Rosso: celle di confronto



Oggetto 2 (i=2) valore 6, peso 4

j

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	10	10	10
i	2	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0

maxval

$$\forall j \ 1 \leq j < 4$$

$\text{items}[i-1].size > j \Rightarrow \text{l'oggetto } 2 \text{ non viene preso}$
 $\Rightarrow \text{maxval}[i, j] = \text{maxval}[i-1, j]$

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Verde: celle con valore già calcolato

Giallo: celle con valore da calcolare

Rosso: celle di confronto

1

1

1

Oggetto 2  (i=2) valore 6, peso 4

j

j = 4

preso/non preso

`items[i-1].size <= j` $\leq 4 \Rightarrow$ l'oggetto 2 può essere preso

conveniente/non conveniente

`maxval[i-1,i] < maxval[i-1][i-items[i-1].size]+items[i-1].value)`

progetto? conviene e viene

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Oggetto 2  (i=2) valore 6, peso 4

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	10	10	10
i	2	0	0	0	0	6	6	6	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0
	maxval										

Verde: celle con valore già calcolato 

Giallo: celle con valore da calcolare 

Rosso: celle di confronto 

Analogamente per j = 5, 6 e 7

preso/non preso: l'oggetto 2 può essere preso

conveniente/non conveniente: l'oggetto 2 conviene, preso

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

Oggetto 2  (i=2) valore 6, peso 4

Verde: celle con valore già calcolato

Giallo: celle con valore da calcolare

Rosso: celle di confronto

j = 8

`items[i-1].size < 8` \Rightarrow l'oggetto 2 può essere preso conveniente/non conveniente

$\maxval[i-1, j] > \maxval[i-1][j - \text{items}[i-1].size] + \text{items}[i-1].value$

10 0 6

⇒ l'oggetto 2 non conviene, quindi non viene preso e
 $\text{maxval}[i,j] = \text{maxval}[i-1,j] = 10$

Analogamente per $j = 9$ e $j = 10$.

Si procede allo stesso modo per $i = 3$ e $i = 4$.

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

$N = 4$
 $\text{cap} = 10$
 j

maxval =	0	0	0	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0	0	10	10	10
	0	0	0	0	6	6	6	6	10	10	10
	0	0	8	8	8	8	14	14	14	14	18
	0	0	8	9	9	17	17	17	17	23	23

configurazione finale

Soluzione ottima: costruzione

Iterazione: partendo da $\text{maxval}[N][\text{cap}]$ si scandiscono tutti gli oggetti:

- se la stima $\text{maxval}[i][j]$ che include l'oggetto corrente i è uguale a quella che non lo include $\text{maxval}[i-1][j]$ significa che l'oggetto corrente i non è stato preso, quindi $\text{sol}[i-1] = 0$
- altrimenti l'oggetto corrente i è stato preso ($\text{sol}[i-1]=1$). L'oggetto preso ha «consumato» una certa capacità, quindi j viene aggiornato a $j - \text{items}[i-1].\text{size}$.

Si ricordi che l'oggetto corrente i si trova in $\text{items}[i-1]$ e il suo flag di presenza/assenza in $\text{sol}[i-1]$.

	0	1	2	3
valore	10	6	8	9
peso	8	4	2	3
oggetto	1	2	3	4

sol 1 2 3 4

0	1	1	1
---	---	---	---

	j									
maxval =	0	0	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0	10	10	10
	0	0	0	0	6	6	6	10	10	10
	0	0	8	8	8	8	14	14	14	18
	0	0	8	9	9	17	17	17	17	23
										23

Soluzione:
insieme {B, C, D} con valore massimo 23

```

void displaySol(Item *items, int N, int cap, int **maxval){
    int i, j, *sol;
    sol = calloc(N, sizeof(int));
    j = cap;
    for (i=N; i>=1; i--)
        if (maxval[i][j] == maxval[i-1][j])
            sol[i-1] = 0; oggetto non preso
        else {
            sol[i-1] = 1; oggetto preso
            j-= items[i-1].size;
        }
    for (i=0; i<N; i++) visualizzazione degli oggetti presi
        if (sol[i])
            ITEMstore(items[i]);
}

```

Complessità

KNAPmaxValDP: $T(n) = \Theta(N \cdot \text{cap})$: conta solo il numero di sottoproblemi, in quanto ognuno ha costo $\Theta(1)$.

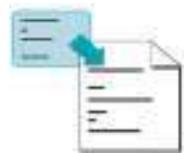
Memoization

- Approccio simile al divide et impera ricorsivo, quindi top-down
- Memorizzazione delle soluzioni ai sottoproblemi
- Lookup: evita di risolvere problemi già trattati
- NB: alcuni autori la denominano Programmazione dinamica top-down

Esempio: numeri di Fibonacci

array knownF

```
unsigned long fib(int n, unsigned long *knownF) {  
    unsigned long t;  
    if (knownF[n] != -1)  
        return knownF[n];  
    if(n == 0 || n == 1) {  
        knownF[n] = n;  
        return n;  
    }  
    t = fib(n-2, knownF) + fib(n-1, knownF);  
    knownF[n] = t;  
    return t;  
}
```



07fibonacciM

Esempio: sequenza di Hofstadter

$$H(0)=0$$

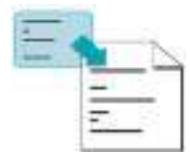
$$H(n)=n-H(H(H(n-1))) \quad \text{per } n > 0$$

```
int H(int n) {  
    if (n == 0)  
        return 0;  
    return n - H(H(H(n-1)));  
}
```

ricorsivo

```
typedef struct hm_ {int val;int calc;} hm;  
int HM(int n, hm *hmem) {  
    if (n == 0) {  
        array hmem di struct  
        hmem[n].val = 0; hmem[n].calc = 1;  
        return hmem[n].val;  
    }  
    if (hmem[n].calc == 1) return hmem[n].val;  
    hmem[n].val=n-HM(HM(HM(n-1,hmem),hmem),hmem);  
    hmem[n].calc = 1;  
    return hmem[n].val;  
}
```

memoization

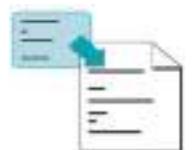


08hofstadterM

Esempio: parentesizzazione ottima

```
int matrix_chainM(int *p, int n) { matrice m
    int i, j, **m;
    m = malloc((n+1)*sizeof(int *));
    for (i = 0; i <= n; i++)
        m[i] = malloc((n+1)*sizeof(int));
    for (i = 0; i <= n; i++)
        for (j = 0; j <= n; j++)
            m[i][j] = INT_MAX;
    return lookup(p, 1, n, m);
}
```

Complessità $T(n) = O(n^3)$
 $S(n) = \Theta(n^2)$



03matrix_chain_mult

```
int lookup(int *p, int i, int j, int **m) {
    int k, q;
    if (m[i][j] < INT_MAX)
        return m[i][j];
    if (i==j)
        m[i][j] = 0;
    else
        for (k= i; k <j; k++) {
            q = lookup(p, i, k, m) +
                lookup(p, k+1, j, m) + p[i-1]*p[k]*p[j];
            if (q < m[i][j])
                m[i][j] = q;
        }
    return m[i][j];
}
```

Esempio: il problema dello zaino

Dato un insieme di N oggetti ciascuno dotato di peso w_j e di valore v_j e dato un peso massimo cap , determinare il sottoinsieme S di oggetti tali che:

- $\sum_{j \in S} w_j x_j \leq \text{cap}$
- $\sum_{j \in S} v_j x_j = \text{MAX}$
- $x_j \in \{0,1\}$

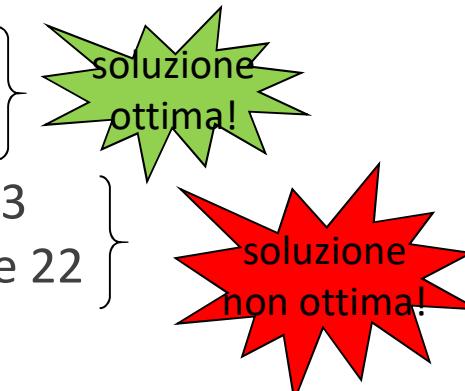
Ogni oggetto o è preso ($x_j = 1$) o lasciato ($x_j = 0$). **Ogni oggetto esiste in un numero arbitrario di istanze.**

Esempio

$N = 5 \ cap = 17$

		A	B	C	D	E
Valore	v_i	4	5	10	11	13
Peso	p_i	3	4	7	8	9

Possibili soluzioni:

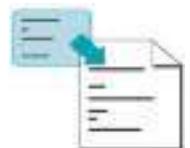
- D, E: peso 17, valore 24
 - A, C, C: peso 17, valore 24
 - A, A, B, C: peso 17, valore 23
 - A, A, A, B, B: peso 17, valore 22
- 

Soluzione ricorsiva: calcolo del valore

- principio di moltiplicazione: le scelte sono gli oggetti (i)
- l'oggetto *i* può essere scelto se il suo peso è compatibile con la capacità disponibile, cioè se la capacità disponibile space dopo aver preso l'oggetto è ≥ 0
$$\text{space} = \text{cap-items}[i].size \geq 0;$$
- se scelto, si ricalcola la capacità disponibile
$$\text{space} = \text{cap-items}[i].size;$$
- ricorsione: determinazione della soluzione ottima per zaino di capacità disponibile space
- se si migliora il risultato corrente, lo si aggiorna
$$\text{if}((t=\text{knapR}(\text{space})+\text{items}[i].value) > \text{max})$$
$$\text{max} = t;$$

```
int KNAPmaxvalR(Item *items, int N, int cap) {  
    int i, space, max, t;  
    for (i = 0, max = 0; i < N; i++)  
        if ((space = cap-items[i].size) >= 0)  
            if ((t=KNAPmaxvalR(items,N,space)+items[i].value)>max)  
                max = t;  
    return max;  
}
```

ricorsivo



09knapsackM

```
int KNPmaxvalM(Item *items,int N,int cap,int *maxKnown){  
    int i, space, max, t;  
    if (maxKnown[cap] != -1)  
        return maxKnown[cap];  
    for (i=0, max=0; i < N; i++)  
        if ((space = cap-items[i].size) >= 0)  
            if ((t=KNPmaxvalM(items,N,space,maxKnown)+  
                 items[i].value)>max)  
                max = t;  
    maxKnown[cap] = max;  
    return max;  
}
```

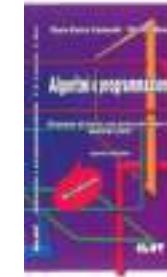
memoization

Riferimenti

- Programmazione dinamica:
 - Cormen 16
 - Montresor 13
 - Crescenzi 2.7
- Problema del ladro e dello zaino:
 - Sedgewick 5.3

Esercizi di teoria

- 8. Programmazione dinamica
 - 8.1 Prodotto in catena di matrici
 - 8.2 Longest Common Subsequence



Approfondimenti

- Longest Common Subsequence

Longest Common Subsequence

Paolo Camurati



Confronto di DNA

Struttura del DNA: sequenza di basi (adenina A, citosina C, guanina G, timina T)

Confronto di 2 DNA: determinazione del grado di somiglianza:

- una sequenza è una sottosequenza dell'altra
- trasformazione di una sequenza nell'altra con numero minimo di operazioni
- date 2 sequenze, identificazione di una o tutte le sottosequenze più lunghe comuni.

Sottosequenza

Date 2 sequenze di simboli

$X = (x_0, x_1, \dots, x_{m-1})$ e $Y = (y_0, y_1, \dots, y_{n-1})$ con $n \leq m$

Y si dice **sottosequenza** di X se esiste una sequenza crescente i_0, i_1, \dots, i_{n-1} di indici di X tali che $x_{i_j} = y_j \quad \forall j = 0, 1, \dots, n-1$.

- **sottosequenza**: indici non necessariamente contigui
- **sottostringa**: indici contigui
- **prefisso** i -esimo di una sequenza X :

$X_i = (x_0, x_1, \dots, x_i)$

Esempio

$S_1 = \boxed{A \ C \ G \ C \ T \ A \ C}$ e $S_2 = \boxed{C \ G \ T \ C}$

S_2 è una sottosequenza di S_1 con indici 1, 2, 4, 6

S_2 non è una sottostringa di S_1 .

```
int subseq(char *s1, char *s2) {
    while (*s1 != '\0') {
        if (*s2 == *s1) {
            s2++;
            if (*s2 == '\0')
                return 1;
        }
        s1++;
    }
    return 0;
}
```

Longest Common Subsequence

Date 2 sequenze X e Y, Z è una sottosequenza comune se è sottosequenza sia di X che di Y.

Esempio:

X =

A	C	G	C	T	A	C
0	1	2	3	4	5	6

 Y =

C	T	G	A	C	A
0	1	2	3	4	5

Sottosequenza comune:

C	G	A
---	---	---

Sottosequenze comuni di lunghezza massima (LCS):

C	G	C	A
---	---	---	---

C	T	A	C
---	---	---	---

Esempio

X = ACCG**GTCGAGTGC**CGGAAGCCGGCCGAA

Y = **GTCGTTCGGAATGCCGTTGCTCTGTAAA**

LCS:

GTCGTCGGAAGCCGGCCGAA

Struttura della soluzione ottima

Una LCS di 2 sequenze X e Y contiene al suo interno una LCS dei prefissi delle 2 sequenze.



La Programmazione Dinamica è applicabile.

Teorema

Date 2 sequenze non vuote $X=(x_0, x_1, \dots, x_{m-1})$ e $Y=(y_0, y_1, \dots, y_{n-1})$ e una qualsiasi loro LCS $Z=(z_0, z_1, \dots, z_{k-1})$:

1. se $x_{m-1}=y_{n-1}$ allora $z_{k-1}=x_{m-1}=y_{n-1}$ e Z_{k-2} è una LCS di X_{m-2} e Y_{n-2}
2. se $x_{m-1} \neq y_{n-1}$ e $z_{k-1} \neq x_{m-1}$, allora Z è LCS di X_{m-2} e Y
3. se $x_{m-1} \neq y_{n-1}$ e $z_{k-1} \neq y_{n-1}$, allora Z è LCS di X e Y_{n-2}

Dimostrazione

1. se $x_{m-1} = y_{n-1}$ allora
 - I. $z_{k-1} = x_{m-1}$
 - II. Z_{k-2} è una LCS di X_{m-2} e Y_{n-2}

I. $z_{k-1} = x_{m-1}$

se z_{k-1} non fosse x_{m-1} , accodando x_{m-1} a Z otterremmo una sottosequenza di X e Y di lunghezza $|Z|+1 = k+1$, il che contraddice il fatto che Z è una LCS

$X = \begin{array}{ccccccccc} A & B & C & B & D & A & B \\ 0 & 1 & 2 & 3 & 4 & 5 & m-1 \end{array} \quad \begin{array}{ccccccccc} B & D & C & A & B \\ 0 & 1 & 2 & 3 & n-1 \end{array}$

se $z_{k-1} \neq x_{m-1}$ $Z = \boxed{B} \boxed{C} \boxed{A} \boxed{\neq B}$ allora $\exists Z = \boxed{B} \boxed{C} \boxed{A} \boxed{\neq B} \boxed{B}$

$$|Z| = k$$

$$|Z| = k+1$$

contraddice l'ipotesi

$k-1 \quad k$

- II. il prefisso Z_{k-2} è:
 - I. ovviamente una sottosequenza comune dei prefissi X_{m-2} e Y_{n-2}
 - II. a lunghezza massima: per assurdo, se non lo fosse, allora esisterebbe una sottosequenza W comune a X_{m-2} e Y_{n-2} tale che $|W| > |Z_{k-2}| = k - 1$. Accodando x_{m-1} a W si otterrebbe una sottosequenza comune di X e Y di lunghezza $|W| + 1 > k - 1 + 1 = k$, contraddicendo l'ipotesi.

$X = \boxed{A} \boxed{B} \boxed{C} \boxed{B} \boxed{D} \boxed{A} \boxed{B}$ $Y = \boxed{B} \boxed{D} \boxed{C} \boxed{A} \boxed{B}$

0 1 2 3 4 $m-2$ $m-1$

0 1 2 $n-2$ $n-1$

se $Z_{k-2} \boxed{B} \boxed{C} \boxed{A}$ non fosse massima allora $\exists W = \boxed{\quad \quad \quad}$
 $|W| > |Z_{k-2}| = k-1$

accodando \boxed{B} a W si otterrebbe una lunghezza $|W| + 1 = k$

$|W| + 1 = k$ $\boxed{\quad \quad \quad} \boxed{B}$

contraddice l'ipotesi

2. se $x_{m-1} \neq y_{n-1}$ e $z_{k-1} \neq x_{m-1}$, allora Z è:

 - I. ovviamente una sottosequenza comune del prefisso X_{m-2} e di Y
 - II. di lunghezza massima. Per assurdo, se non lo fosse, allora esisterebbe una sottosequenza W comune a X_{m-2} e Y tale che $|W| > |Z| = k$. Il fatto che $z_{k-1} \neq x_{m-1}$ implica che W sia anche una sottosequenza comune di $X=X_{m-1}$ e Y, ma essendo $|W| > k$ si contraddice l'ipotesi.

X =  Y = 

se Z [C G C A] non fosse massima
0 1 2 k-1

allora $\exists W =$
per X_{m-2} e $Y \quad |W| > |Z| = k$

contraddice l'ipotesi

3. dimostrazione analoga al caso precedente

Se la sottostruttura del problema è ottima



la programmazione dinamica è **applicabile**.

Il numero di sottoproblemi indipendenti è $\Theta(nm)$:

la programmazione dinamica è **conveniente**.

Si identificano 2 fasi:

- ottimizzazione: calcolo della lunghezza della LCS
- ricerca: identificazione della LCS Z o di tutte le LCS.

Approccio ricorsivo “divide et impera” al solo problema di ottimizzazione.

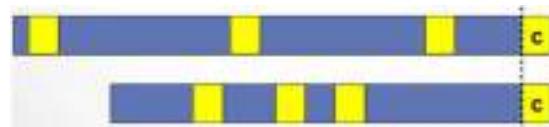
Soluzione ricorsiva

Valore della soluzione ottima:

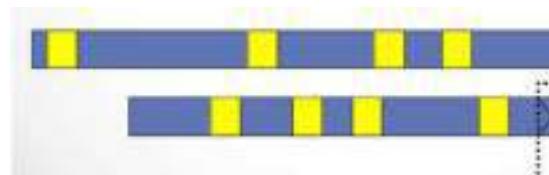
- caso terminale: primo elemento o di X o di Y, ritorna 0
- caso non terminale:
 - l'elemento in coda a X e Y è lo stesso, ritorna 1 + la lunghezza della LCS del prefisso di X e Y accorciato di 1
 - l'elemento in coda a X e Y non è lo stesso, ritorna il massimo tra
 - la lunghezza della LCS del prefisso di X accorciato di 1 e Y (invariata)
 - la lunghezza della LCS di X (invariata) e del prefisso di Y accorciato di 1

Soluzione:

se l'ultimo carattere **c** in ciascuna sequenza è uguale, esiste una LCS più lunga che lo contiene ottenibile da quella in cui l'ultimo carattere è rimosso.



se l'ultimo carattere differisce, la LCS è quella ottenuta rimuovendo l'ultimo carattere in una **sola** delle due sequenze



$$c[i, j] = \begin{cases} 0 & \text{se } i=0 \text{ e } j=0 \\ c[i-1, j-1] + 1 & \text{se } x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & \text{se } x_i \neq y_j \end{cases}$$

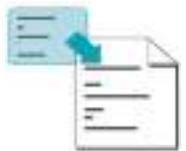
Lunghezza della LCS
dei prefissi X_i e Y_j

```

int lengthR(char *X, char *Y, int i, int j) {
    if ((i == 0) || (j == 0))
        return 0;
    if (X[i] == Y[j])
        return 1 + lengthR(X, Y, i-1, j-1);
    else
        return max(lengthR(X, Y, i-1, j), lengthR(X, Y, i, j-1));
}

int LCSlength(char *X, char *Y) {
    return lengthR(X, Y, strlen(X), strlen(Y));
}

```

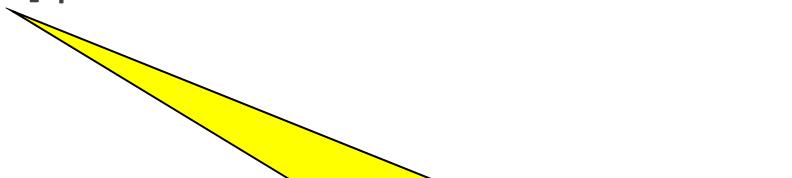


LCS

Soluzione ottima: calcolo bottom-up del valore

Strutture dati:

- tabella $c[0...m, 0...n]$ per memorizzare le lunghezze $c[i, j]$ dei prefissi X_i e Y_j
- tabella $b[0...m, 0...n]$ per la costruzione dalla LCS



non serve per il calcolo del costo minimo, servirà per costruire la soluzione

Passi:

- 2 cicli **for** annidati di scansione delle stringhe X e Y
- 3 casi:
 - ultimi caratteri $X[i-1]$ e $Y[j-1]$ uguali: la lunghezza di una LCS per i prefissi X_i e Y_j è quella dei prefissi X_{i-1} e Y_{j-1} incrementata di 1
 - ultimi caratteri $X[i-1]$ e $Y[j-1]$ diversi: la lunghezza di una LCS per i prefissi X_i e Y_j è la maggiore tra le lunghezze delle LCS per i prefissi X_{i-1} e Y_j e X_i e Y_{j-1}



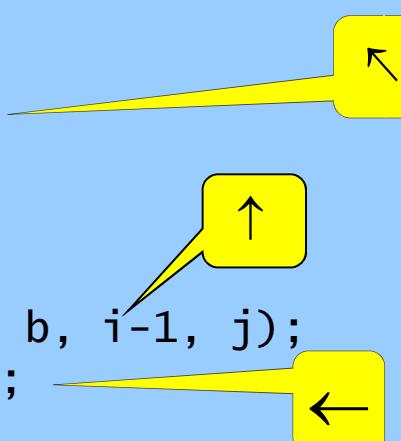
calcolo del costo minimo e di una soluzione ottima

```
int LCSlengthDP(char *X, char *Y) {  
    int i, j, m, n, **c, **b;  
    m = strlen(X); n = strlen(Y);  
    // allocazione matrici c e b di dimensioni (m+1)x(n+1)  
    for (i=1; i<=m; i++)  
        for (j=1; j <=n; j++)  
            if (X[i-1]==Y[j-1]) { c[i][j] = c[i-1][j-1]+1; b[i][j] = 1; }  
            else {  
                if (c[i-1][j]>=c[i][j-1]) {c[i][j]=c[i-1][j]; b[i][j]=2;}  
                else {c[i][j] = c[i][j-1]; b[i][j] = 3;}  
            }  
    printf("LCS is: ");  
    displaySol(X, Y, b);  
    printf("\n");  
    return c[m][n];  
}
```

visualizzazione di una
soluzione ottima

Soluzione ottima: costruzione

```
void displaySolR(char *X, char *Y, int **B, int i, int j) {  
    if ((i==0) || (j==0)) {return;}  
    if (B[i][j]==1) {  
        displaySolR(X, Y, B, i-1, j-1);  
        printf("%c", X[i-1]);  
        return;  
    }  
    if (B[i][j]==2) displaySolR(X, Y, B, i-1, j);  
    else displaySolR(X, Y, B, i, j-1);  
}  
void displaySol(char *X, char *Y, int **B) {  
    int m, n;  
    m = strlen(X); n = strlen(Y);  
    displaySolR(X, Y, B, m, n);  
}
```



Esempio

$X = \boxed{A|B|C|B|D|A|B}$ $Y = \boxed{B|D|C|A|B|A}$

i

j

$C = \begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$

$b = \begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \uparrow & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$

$$X = \boxed{A|B|C|B|D|A|B} \quad Y = \boxed{B|D|C|A|B|A}$$

i

j

$$C = \begin{array}{cccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \color{red}{0} & 0 & \color{yellow}{0} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$
$$b = \begin{array}{cccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \uparrow & \color{yellow}{\uparrow} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$X = \boxed{A|B|C|B|D|A|B} \quad Y = \boxed{B|D|C|A|B|A}$$

i

j

$$c = \begin{array}{ccccccc} 0 & 0 & 0 & 0 & \text{green} & 0 & 0 \\ 0 & 0 & \text{red} & 0 & \text{yellow} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$
$$b = \begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \uparrow & \uparrow & \uparrow & \text{yellow} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$X = \boxed{A \ B \ C \ B \ D \ A \ B} \quad Y = \boxed{B \ D \ C \ A \ B \ A}$$

i

$$Y = \boxed{B \mid D \mid C \mid A \mid B \mid A}$$

i

$$X = \boxed{A \ B \ C \ B \ D \ A \ B} \quad Y = \boxed{B \ D \ C \ A \ B \ A}$$

i

$$Y = \boxed{B \mid D \mid C \mid A \mid B \mid A}$$

$$X = \boxed{A|B|C|B|D|A|B} \quad Y = \boxed{B|D|C|A|B|A}$$

i

j

etc.

$$X = \boxed{A \ B \ C \ B \ D \ A \ B} \quad Y = \boxed{B \ D \ C \ A \ B \ A}$$

i

B D C A B A

j

$$C = \begin{array}{|c|c|c|c|c|c|c|} \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ \hline 0 & 1 & 1 & 1 & 1 & 1 & 2 & 2 \\ \hline 0 & 1 & 1 & 2 & 2 & 2 & 2 & 2 \\ \hline 0 & 1 & 1 & 2 & 2 & 3 & 3 & 3 \\ \hline 0 & 1 & 2 & 2 & 2 & 3 & 3 & 3 \\ \hline 0 & 1 & 2 & 2 & 3 & 3 & 4 & 4 \\ \hline 0 & 1 & 2 & 2 & 3 & 4 & 4 & 4 \\ \hline \end{array}$$

configurazione finale

$$X = \boxed{A B C B D A B} \quad Y = \boxed{B D C A B A}$$

i

$$Y = \boxed{B \ D \ C \ A \ B \ A}$$

1

$$C = \begin{array}{|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ \hline 0 & 1 & 1 & 1 & 1 & 1 & 2 \\ \hline 0 & 1 & 1 & 2 & 2 & 2 & 2 \\ \hline 0 & 1 & 1 & 2 & 2 & 3 & 3 \\ \hline 0 & 1 & 2 & 2 & 2 & 3 & 3 \\ \hline 0 & 1 & 2 & 2 & 3 & 3 & 4 \\ \hline 0 & 1 & 2 & 2 & 3 & 4 & 4 \\ \hline \end{array}$$

$$Z = \boxed{B \quad C \quad B \quad A}$$

Complessità

LCSlengthDP: $T(n) = \Theta(mn)$

displaySol: $T(n) = O(m+n)$

$S(n) = \Theta(n^2)$

rispetto al costo esponenziale nel tempo della soluzione ricorsiva.
È possibile evitare la matrice b, in quanto il valore di $c[i,j]$ dipende solo da $c[i-1,j-1]$, $c[i-1, j]$ e $c[i, j-1]$. Non cambia la $S(n)$.

```

void displaySolR(char *X, char *Y, int **C, int i, int j) {
    if ((i==0) || (j==0)) { return; }
    if (X[i-1]==Y[j-1]) {
        displaySolR(X, Y, C, i-1, j-1);
        printf("%c", X[i-1]);
        return;
    }
    if (C[i-1][j] >= C[i][j-1]) displaySolR(X, Y, C, i-1, j);
    else displaySolR(X, Y, C, i, j-1);
}

void displaySol(char *X, char *Y, int **C) {
    int m, n;
    m = strlen(X); n = strlen(Y);
    displaySolR(X, Y, C, m, n);
}

```

Il paradigma greedy

Paolo Camurati



Il paradigma greedy

Per i problemi di ottimizzazione il paradigma greedy è un'alternativa:

- all'approccio divide et impera
 - alla programmazione dinamica
- in generale:
- di minor complessità, quindi più rapido
 - non sempre in grado di ritornare sempre una soluzione ottima.

Principi:

- a ogni passo: per *tentare* di trovare una soluzione globalmente ottima si scelgono soluzioni *localmente ottime*
- le scelte fatte ai singoli passi non vengono successivamente riconsiderate (*no backtrack*)
- scelte localmente ottime sulla base di una funzione di **appetibilità** (costo).

- Vantaggi
- algoritmo molto semplice
- tempo di elaborazione molto ridotto
- Svantaggi
 - soluzione non necessariamente ottima, in quanto non è detto che lo spazio delle possibilità sia esplorato in maniera esaustiva.

Algoritmo

Appetibilità note in partenza e non modificate:

- partenza: soluzione vuota
- si ordinano le scelte per appetibilità decrescenti
- si eseguono le scelte in ordine decrescente, aggiungendo, ove possibile, il risultato alla soluzione.

Appetibilità modificabili:

- come prima con modifica delle appetibilità e coda a priorità.

Selezione di attività (1)

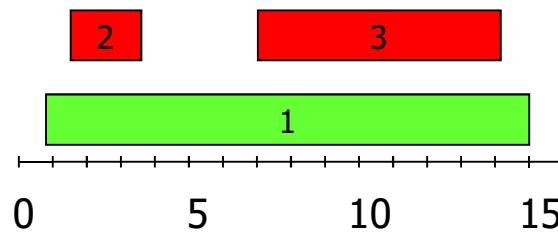
intervallo aperto a destra

- Input: insieme di n attività caratterizzate da tempo di inizio e tempo di fine $[s, f]$
- Output: insieme con il **massimo numero** di attività compatibili
- Compatibilità: $[s_i, f_i]$ e $[s_j, f_j]$ non si sovrappongono, cioè $s_i \geq f_j$ oppure $s_j \geq f_i$
- Approccio greedy: ordinamento delle attività in base a una funzione di appetibilità.

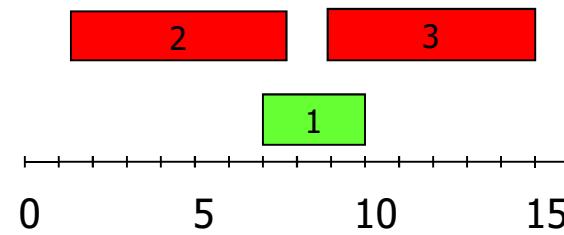


Funzioni di appetibilità

- Funzioni che non portano sempre a soluzioni ottime:
 - ordinamento per tempo di inizio crescente: un'attività che inizia presto ma dura a lungo impedisce di selezionarne altre



- Funzioni che non portano sempre a soluzioni ottime:
 - ordinamento per durata crescente: un'attività breve che interseca 2 lunghe impedisce di selezionarle

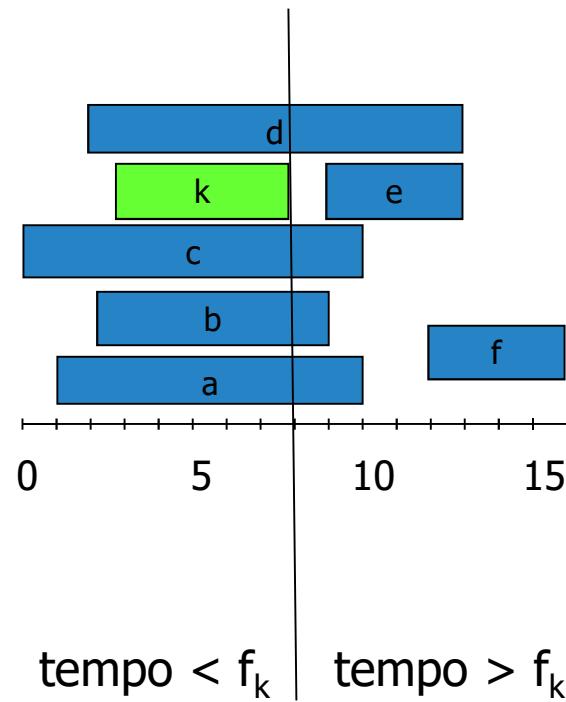


- Funzione che porta sempre a soluzioni ottime:
 - ordinamento per tempo di fine crescente:
 - supponiamo che k sia l'attività che finisce per prima:
 - scegliere k non pregiudica nulla circa le attività che iniziano dopo che è finita
 - se ci sono attività che iniziano prima di k , visto che finiscono dopo k , esse si intersecano e quindi al più se ne può scegliere una, quindi si sceglie proprio k

Si può scegliere una sola tra a, b, c, d e k perché si intesecano certamente tra di loro.

Le attività a, b, c, d potrebbero pregiudicare la scelta di e ed f, k certamente no.

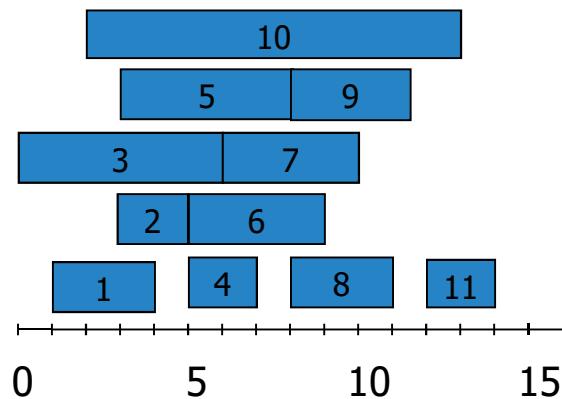
Si sceglie k.



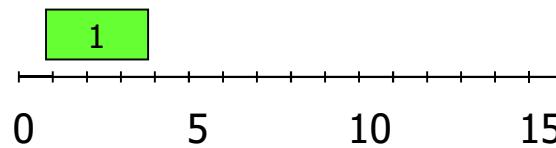
Esempio



i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

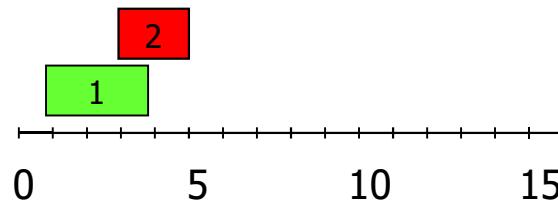


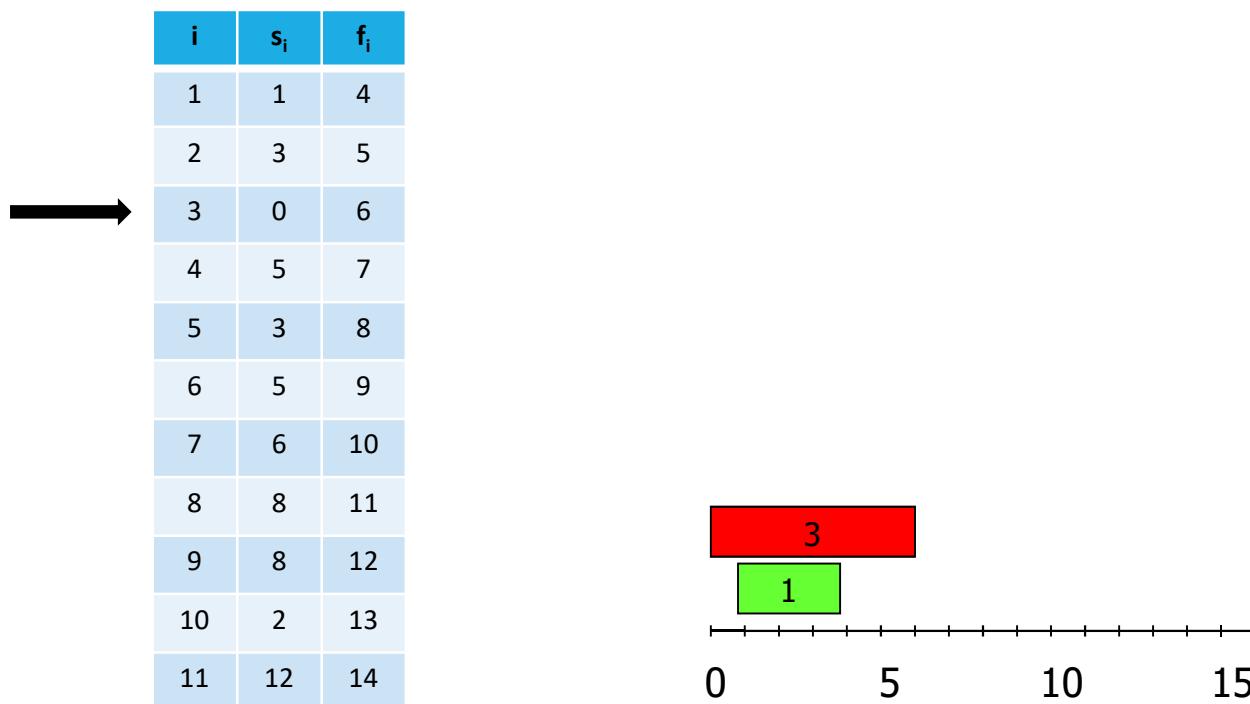
già ordinate per tempo f_i

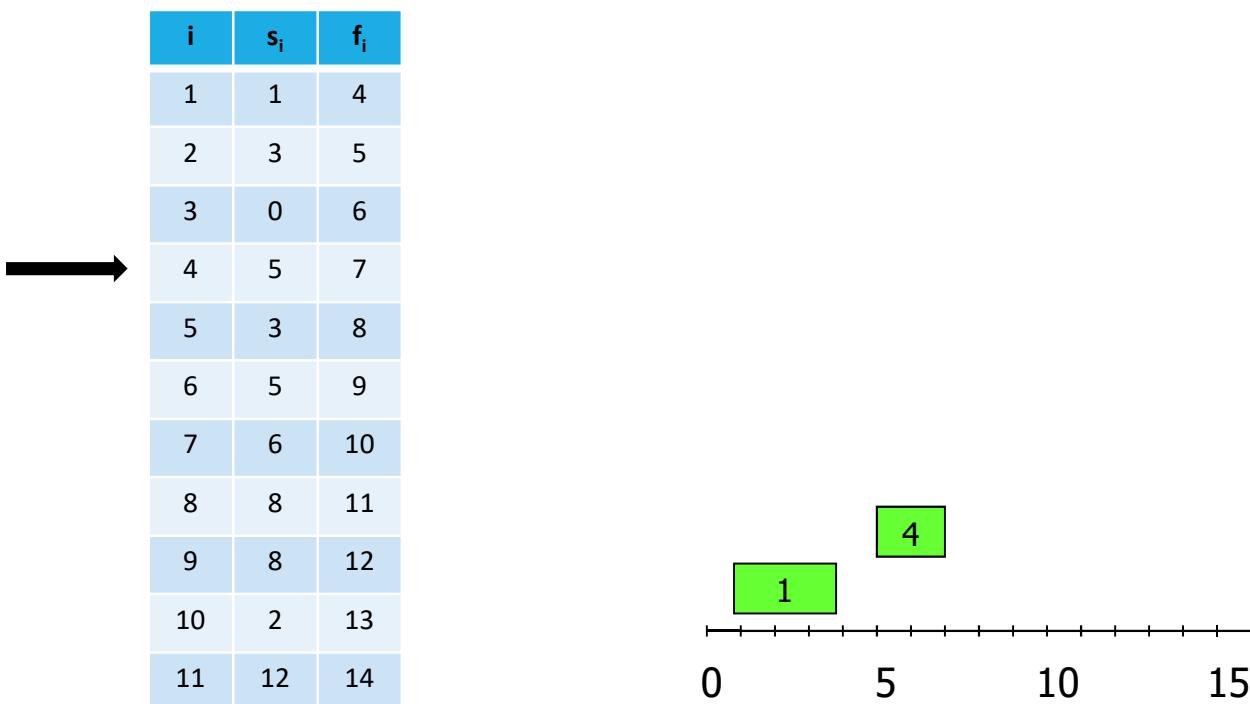




i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

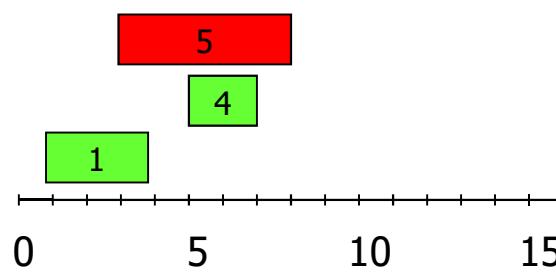


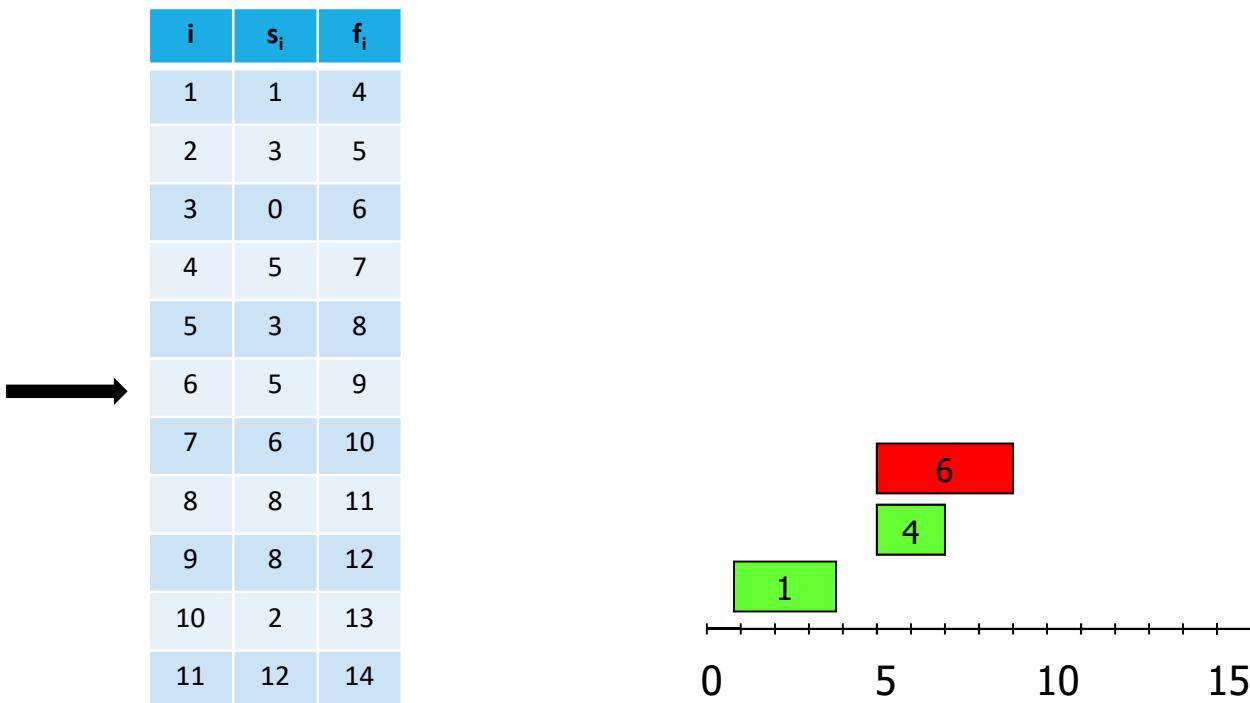






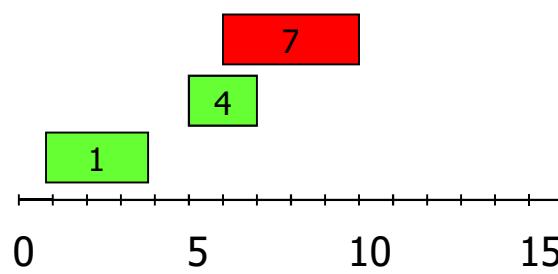
i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

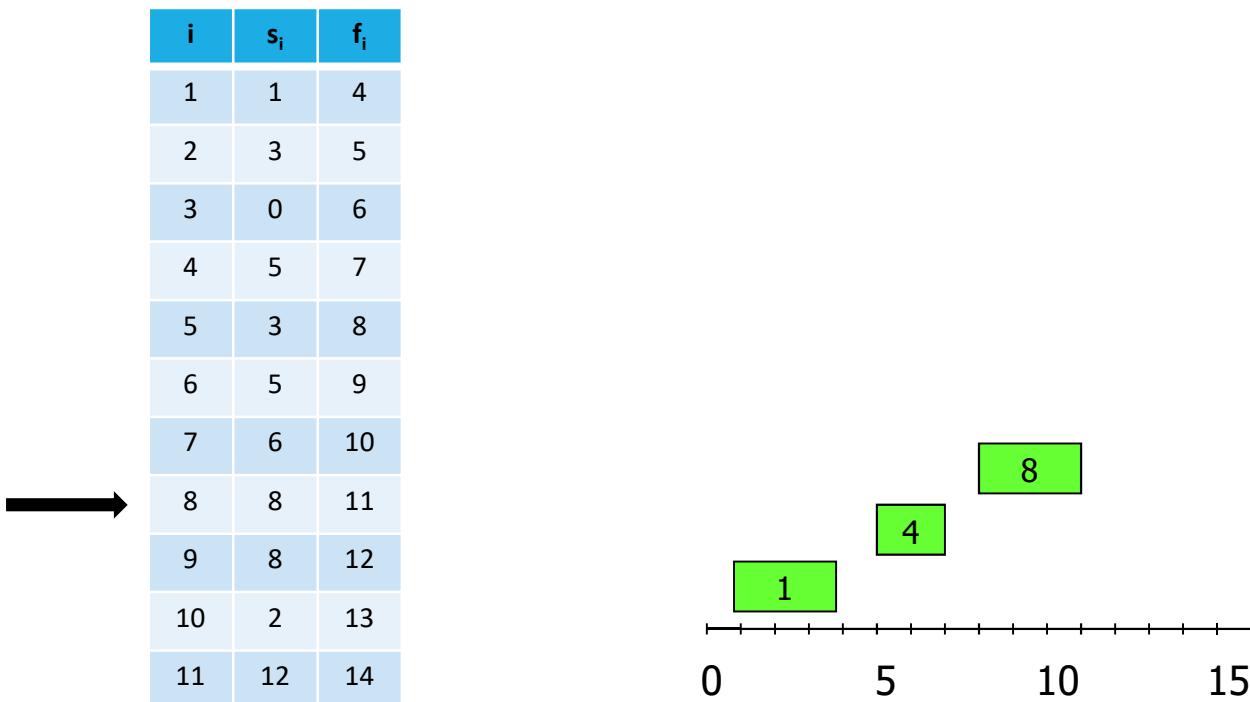






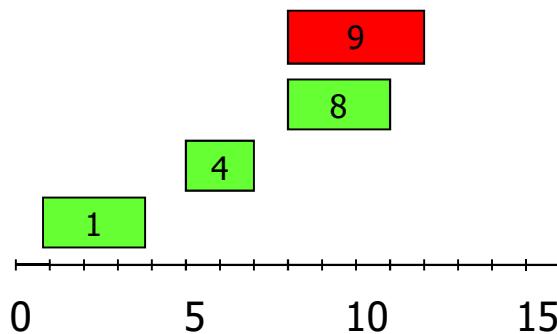
i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

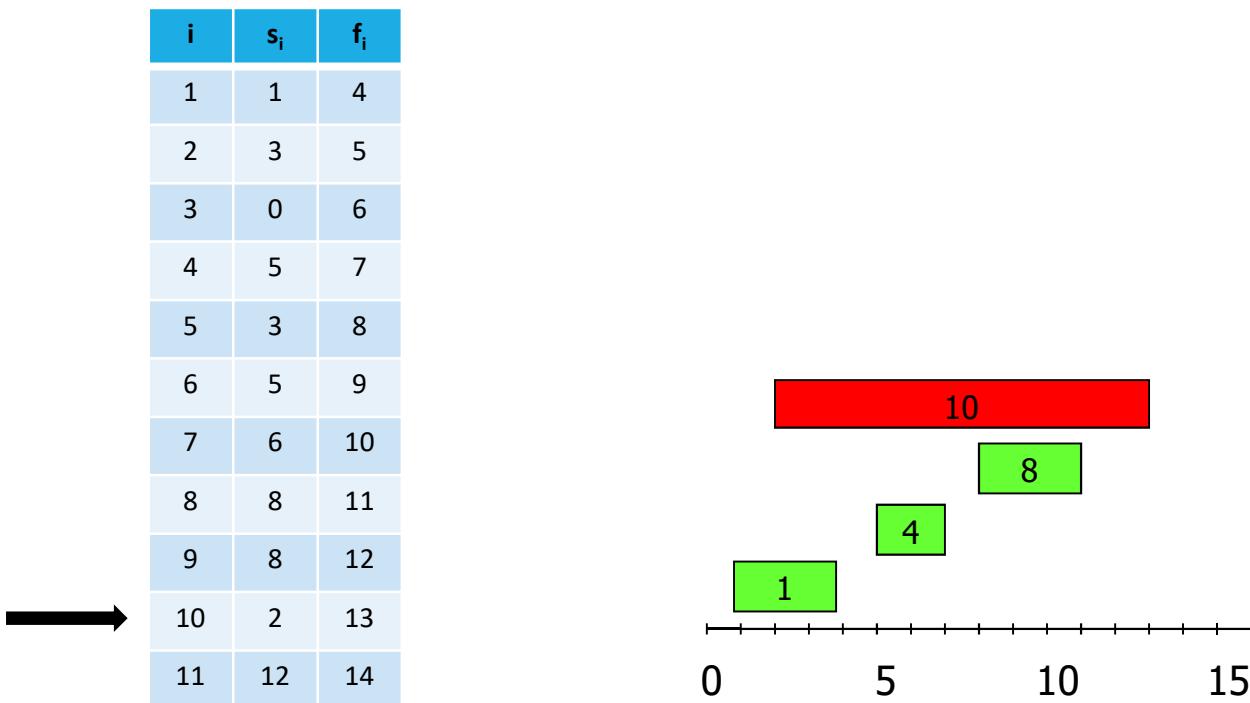




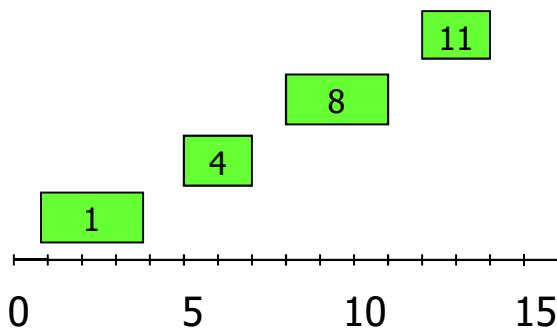


i	s _i	f _i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14





i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



Quasi ADT Item

Item | name | start | stop | selected

Item.h

```
typedef struct {
    char name[MAXC];
    int start;
    int stop;
    int selected;
} Item;
typedef int Key;
int ITEMlt(Item A, Item B);
int ITEMgt(Item A, Item B);
Item ITEMscan();
void ITEMstore(Item A);
Key KEYgetStart(Item A);
Key KEYgetStop(Item A);
Key KEYgetSel(Item A);
void KEYsetSel(Item *pA);
```

3

Tipologia 3
nelle slide di
programmazione



01activityselection

Item.c

```
Item ITEMscan() {
    Item A;
    printf("name, start, stop: ");
    scanf("%s %d %d", A.name, &A.start, &A.stop);
    return A;
}

void ITEMstore(Item A) {
    printf("name= %s \t start= %d \t stop = %d \n",
           A.name, A.start, A.stop);
}

int ITEMlt(Item A, Item B) {
    return (A.stop < B.stop);
}

int ITEMgt(Item A, Item B) {
    return (A.stop > B.stop);
}
```

```
Key KEYgetStop(Item A) {
    return A.stop;
}

Key KEYgetStart(Item A) {
    return A.start;
}

Key KEYgetSel(Item A) {
    return A.selected;
}

void KEYsetSel(Item *pA){
    pA->selected = 1;
}
```

client.c

```
void select(Item *act, int n) {  
    int i, stop;  
  
    KEYsetSel(&act[0]);  
    stop = KEYgetStop(act[0]);  
  
    for (i=1; i<n; i++)  
        if (KEYgetStart(act[i]) >= stop) {  
            KEYsetSel(&act[i]);  
            stop = KEYgetStop(act[i]);  
        }  
}
```

```
int main() {
    int n, i;
    Item *act;
    printf("No. of activities: "); scanf("%d", &n);
    act = calloc(n,sizeof(Item));
    printf("Input activities: \n");
    for (i=0; i<n; i++) act[i] = ITEMscan();

    MergeSort(act, n);

    select(act, n);

    printf("Selected activities: \n");
    for (i=0; i<n; i++)
        if (KEYgetSel(act[i])==1)
            ITEMstore(act[i]);
    return 0;
}
```

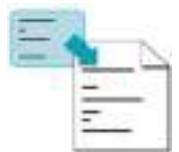
Selezione di attività (2)

- Input: insieme di n attività caratterizzate da tempo di inizio e tempo di fine $[s, f]$ e durata $f-s$
- Output: insieme con **somma delle durate massima**
- L'approccio greedy con ordinamento delle attività per tempo di fine crescente non dà sempre una soluzione ottima

Il cambiamonete

- Input: monetazione, resto da erogare
- Output: resto con numero minimo di monete
- Appetibilità: valore della moneta
- Approccio greedy: a ogni passo moneta di maggior valore inferiore al resto residuo.

```
for (i=0; i < numden; i++) {  
    coins[i] = amount / den[i];  
    amount = amount - (amount/den[i])*den[i];  
    printf("n. of %d cent coins = %d\n",den[i],coins[i]);  
}
```



02changemachine

Esempio

Monetazione:

25, 10, 5, 1

Resto:

67

Passo	Resto residuo
0	17
1	10
2	7
3	2

Moneta scelta

2 x 25
1 x 10
1 x 5
2 x 1



Esempio

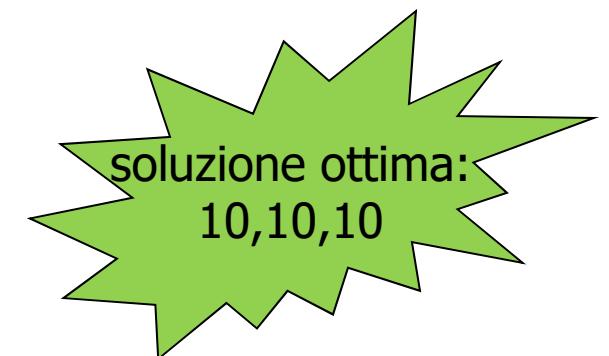
Monetazione:

25, 10, 1

Resto:

30

Passo	Resto residuo	Moneta scelta
0	5	
1	0	1x25 5x1



Il problema dello zaino (discreto)

Dato un insieme di N oggetti ciascuno dotato di peso w_j e di valore v_j e dato un peso massimo cap , determinare il sottoinsieme S di oggetti tali che:

- $\sum_{j \in S} w_j x_j \leq \text{cap}$
- $\sum_{j \in S} v_j x_j = \text{MAX}$
- $x_j \in \{0,1\}$

Ogni oggetto o è preso ($x_j=1$) o lasciato ($x_j=0$).

Appetibilità: valore specifico v_j / w_j decrescente.

Approccio greedy: a ogni passo aggiungo l'oggetto a massimo valore specifico compatibile con il peso disponibile.

Esempio

cap = 50

	i=1	i=2	i=3
Valore v_i	60	100	120
Peso w_i	10	20	30
Val. spec. v_i/w_i	6	5	4

Passo	Residuo	Oggetto	Valore
0	50	1	60
1	40	2	100
2	20		

Oggetti: 1 e 2
Valore 160:



Oggetti: 2 e 3
Valore 220:



Il problema dello zaino (continuo)

Dato un insieme di N oggetti ciascuno dotato di peso w_j e di valore v_j e dato un peso massimo cap , determinare il sottoinsieme S di oggetti tali che:

- $\sum_{j \in S} w_j x_j \leq P$
- $\sum_{j \in S} v_j x_j = MAX$
- $0 \leq x_j \leq 1$

Ogni oggetto può essere preso per una frazione x_j .

Appetibilità: valore specifico v_j / w_j decrescente.

Approccio greedy: ad ogni passo aggiungo la frazione di oggetto a massimo valore specifico compatibile con il peso disponibile.

Esempio

cap = 50

	i=0	i=1	i=2
Valore v_i	60	100	120
Peso w_i	10	20	30
Val. spec. v_i/w_i	6	5	4

Passo	Residuo	Oggetto	Valore	Frazione
0	50	1	60	1.00
1	40	2	100	1.00
2	20	3	120	0.66

Oggetti: 1, 2 e 2/3 di 3

Valore 240:



Quasi ADT Item

item [n(ame)|w(eight)|v(alue)|r(atio)|f(ract)]

3

Item.h

```
typedef struct {char n[MAX]; float w; float v;
                float r; float f;} Item;

typedef float Key;

int     ITEMeq(Item A, Item B);
int     ITEMgt(Item A, Item B);
Item   ITEMscan();
void   ITEMstore(Item A);
Key    KEYgetW(Item A);
Key    KEYgetV(Item A);
Key    KEYgetF(Item A);
void   KEYsetF(Item *pA, float f);
```

Tipologia 3
nelle slide di
programmazione



03fractionalknapsack

```

. . .
Item ITEMscan() {
    printf("name, weight, value: ");
    scanf("%s %f %f", A.name, &(A.w), &(A.v));
    A.r = A.v/A.w;
    return A;
}
void ITEMstore(Item A) {
    printf("name= %s \t weight= %.2f \t value = %.2f \t
           fract= %.2f \n", A.name, A.w, A.v, A.f);
}
int ITEMeq(Item A, Item B) {return (A.r == B.r);}
int ITEMgt(Item A, Item B) {return (A.r > B.r);}
Key KEYgetW(Item A) {return (A.w);}
Key KEYgetV(Item A) {return (A.v);}
Key KEYgetF(Item A) {return (A.f);}
void KEYsetF(Item *pA, float f) {pA->f = f;}

```

main.c

```
...
float knapsack(int n, Item *objects, float cap) {
    float stolen = 0.0, res = cap;
    int i;

    for (i=0; i<n && (KEYgetW(objects[i]) <= res); i++) {
        KEYsetF(&objects[i], 1.0);
        stolen = stolen + KEYgetV(objects[i]);
        res = res - objects[i].weight;
    }

    KEYsetF(&objects[i], res/KEYgetW(objects[i]));
    stolen = stolen + KEYgetF(objects[i])*KEYgetV(objects[i]);

    return stolen;
}
```

```
int main() {
    int n, i; float cap, stolen=0.0; Item *objects;
    printf("No. of objects: "); scanf("%d", &n);
    objects = calloc(n, sizeof(Item));
    printf("Input objects: \n");
    for (i=0; i<n; i++)
        objects[i] = ITEMscan();
    printf("Capacity of knapsack: "); scanf("%f", &cap);
    MergeSort(objects, n);
    stolen = knapsack(n, objects, cap);
    printf("Results: \n");
    for(i = 0; i < n; i++)
        ITEMstore(objects[i]);
    printf("Total amount stolen: %.2f \n", stolen);
    return 0;
}
```

Codici di Huffman (1950)

- Codice: stringa di bit associata a un simbolo $s \in S$
 - a lunghezza fissa
 - a lunghezza variabile
- Codifica: da simbolo a codice
- Decodifica: da codice a simbolo.



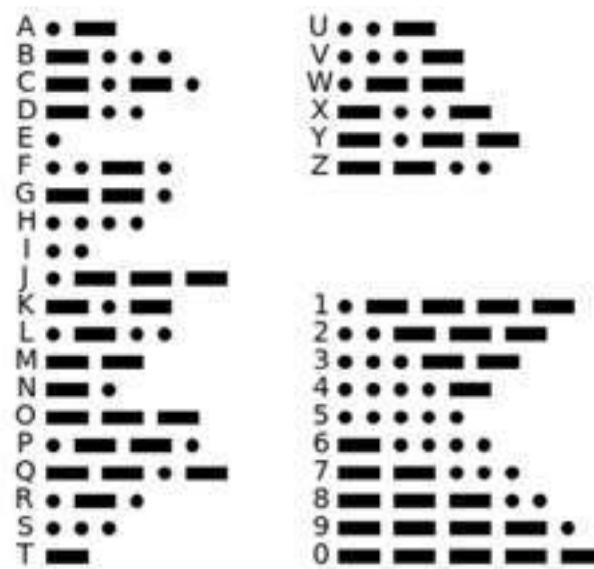
Codici a lunghezza fissa:

- numero di bit $n = \lceil \log_2 (\text{card}(S)) \rceil$
- vantaggio: facilità di decodifica
- uso: simboli isofrequenti

Codici a lunghezza variabile:

- svantaggio: difficoltà di decodifica
- vantaggio: risparmio di spazio di memoria
- uso: simboli con frequenze diverse.

Esempio: alfabeto Morse con durata specificata di lineette e punti e pause tra simboli (punti e lineette) e tra parole.



Esempio

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

file con 100.000 caratteri

codice fisso: $3 \times 100.000 = 300.000$ bit

codice variabile:

$(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1.000 = 224.000$ bit

Codici prefissi

Codice (libero da) prefisso:
nessuna parola di codice valida è un prefisso di un'altra parola di codice.

Codifica: giustapposizione di stringhe

Decodifica: percorrimento di albero binario.

PS: il codice Morse non è libero da prefisso, ma ci sono le durate e le pause.

Esempio

$$a = 0$$

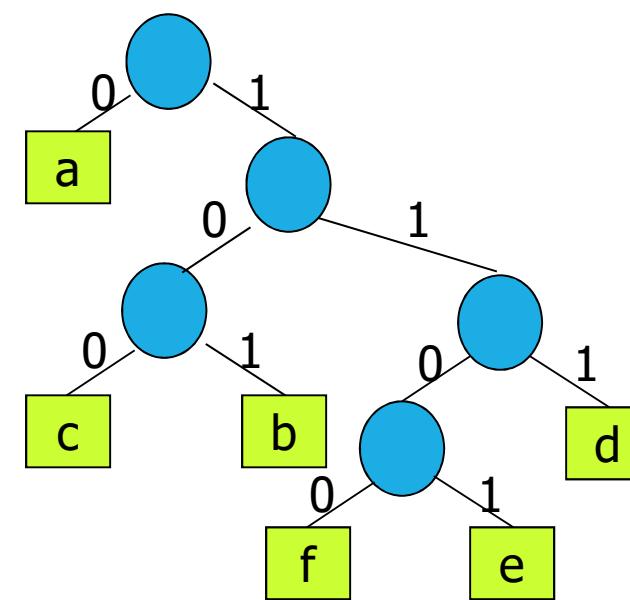
$$b = 101$$

$$c = 100$$

$$d = 111$$

$$e = 1101$$

$$f = 1100$$



Codifica:

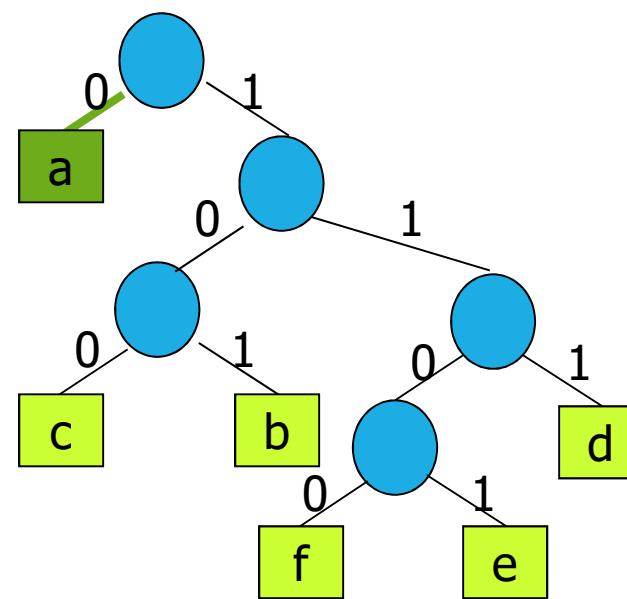
a b f a a c

0101110000100

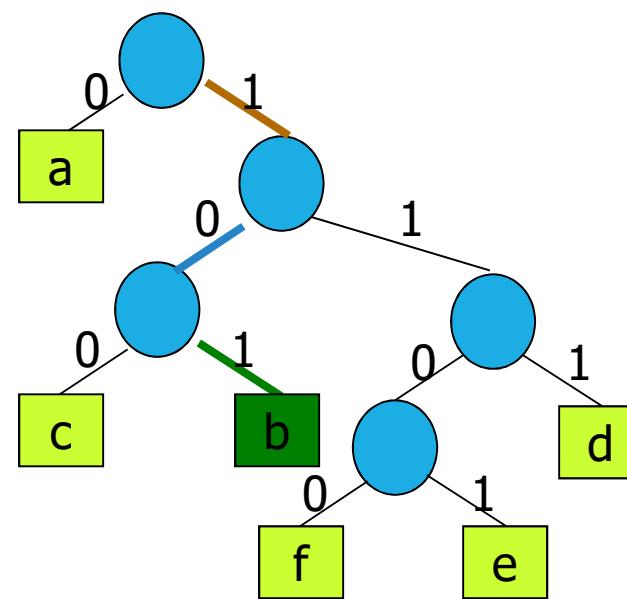
Decodifica:

0101110000100

a

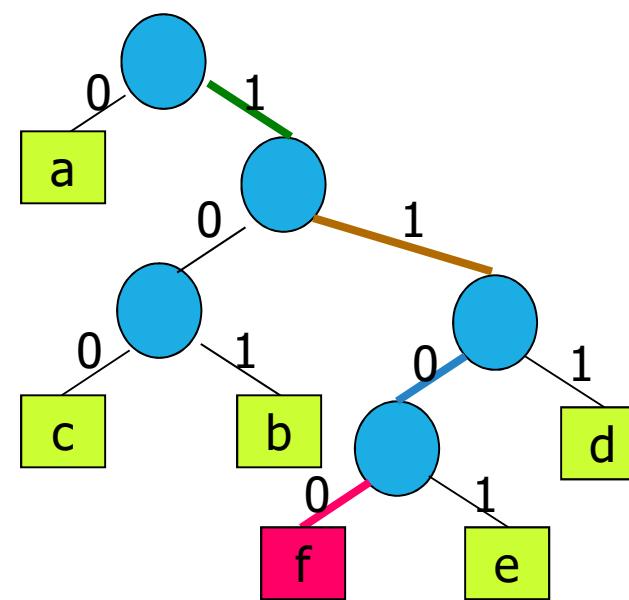


Decodifica:
101110000100
ab

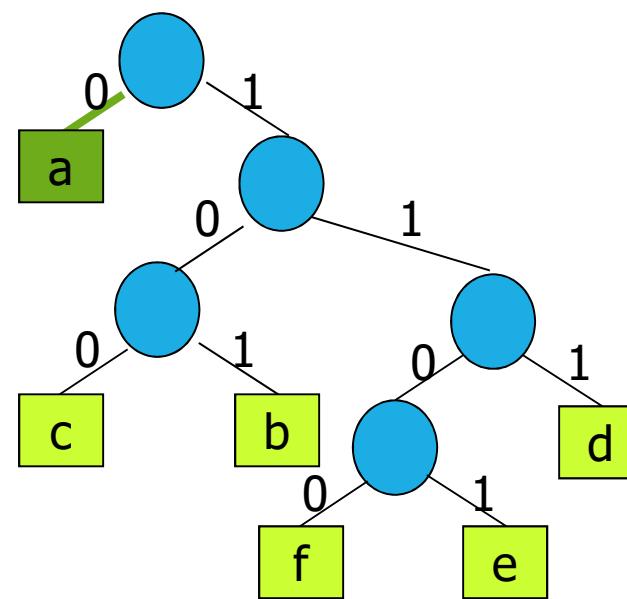


Decodifica:

110000100
abf



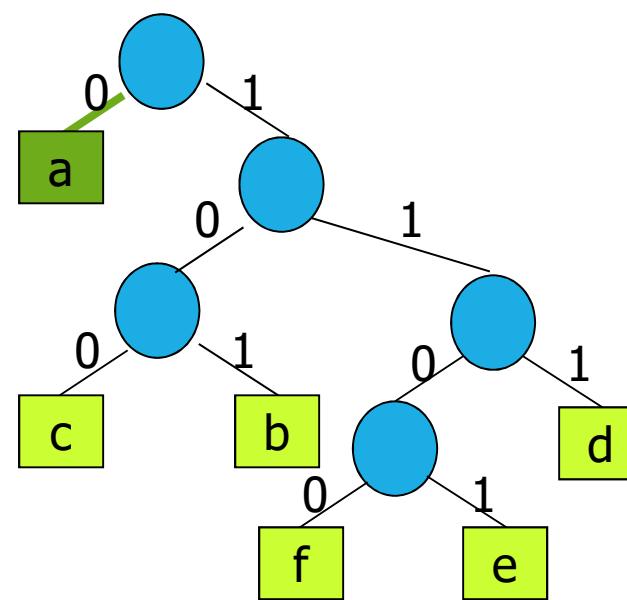
Decodifica:
00100
abfa



Decodifica:

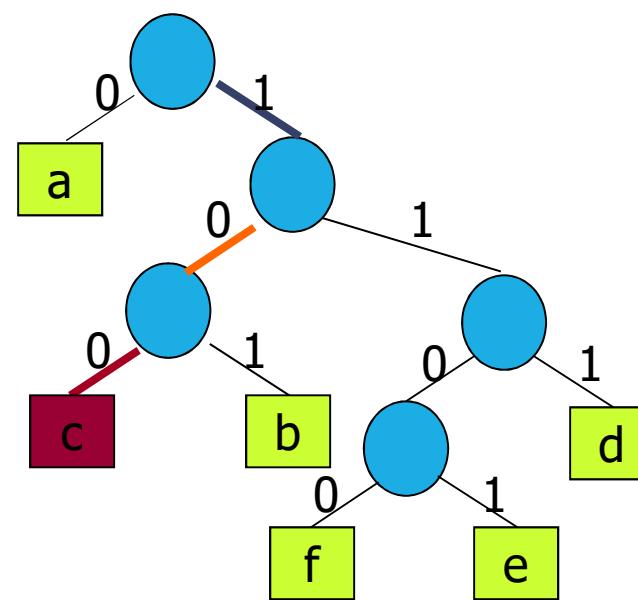
0100

abfaa



Decodifica:

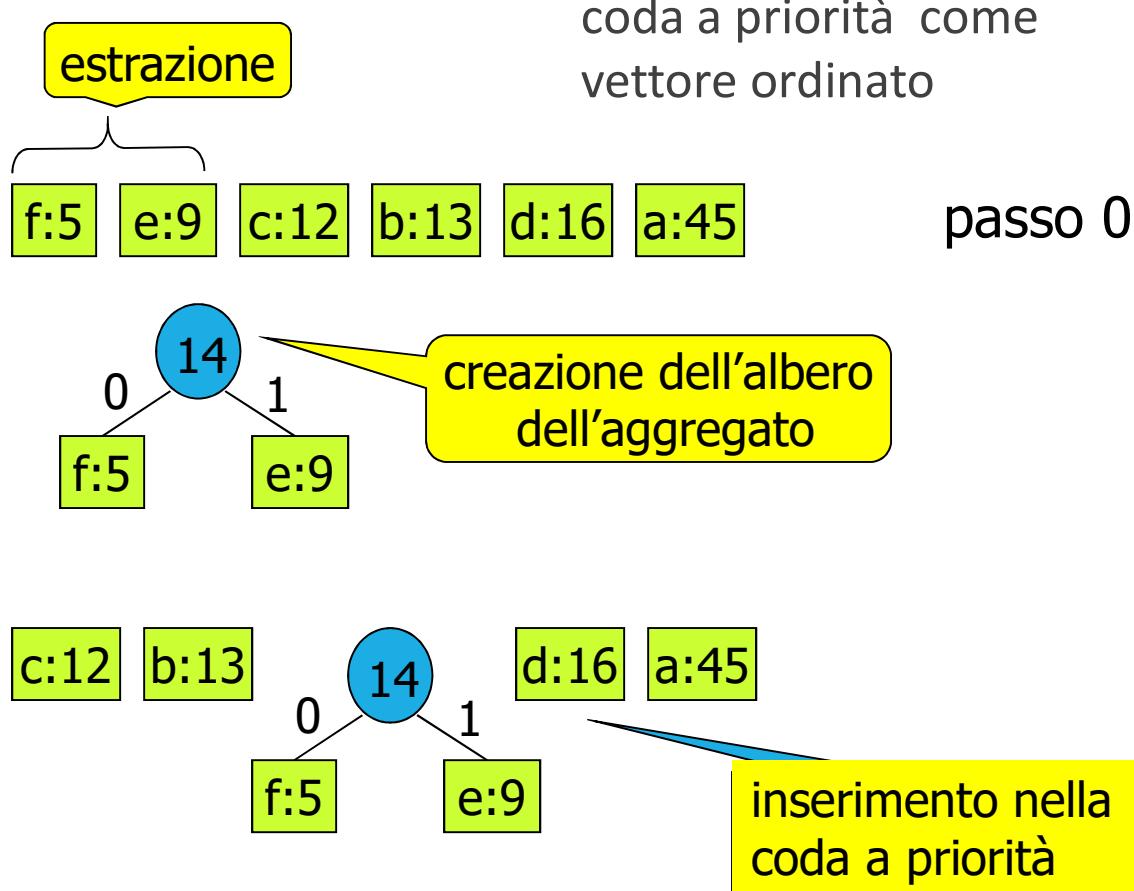
100
abfaac

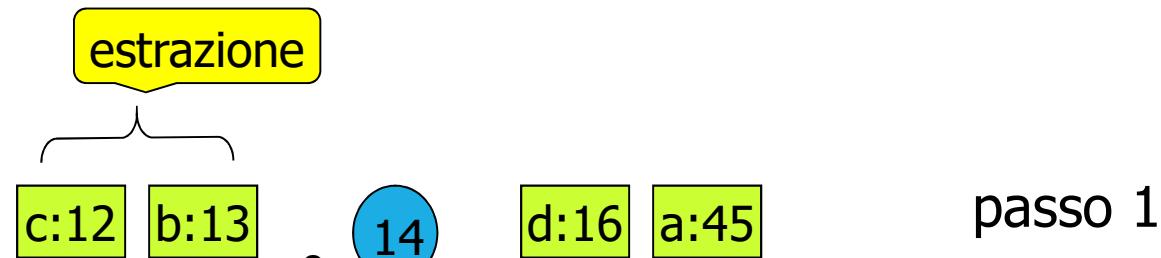


Costruzione dell'albero binario

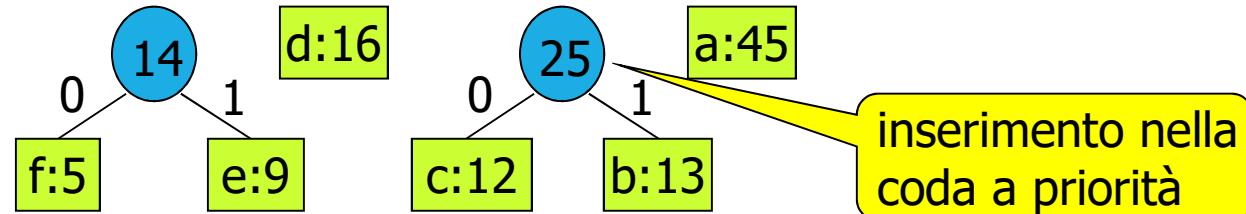
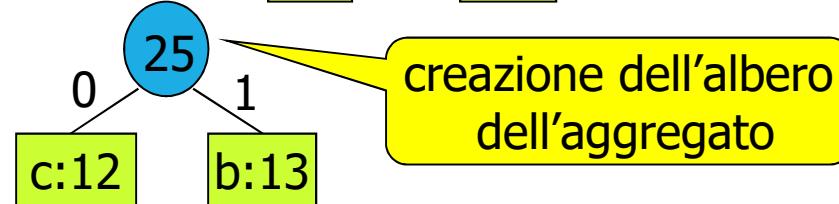
Struttura dati: coda a priorità (frequenze crescenti, proprietà greedy)

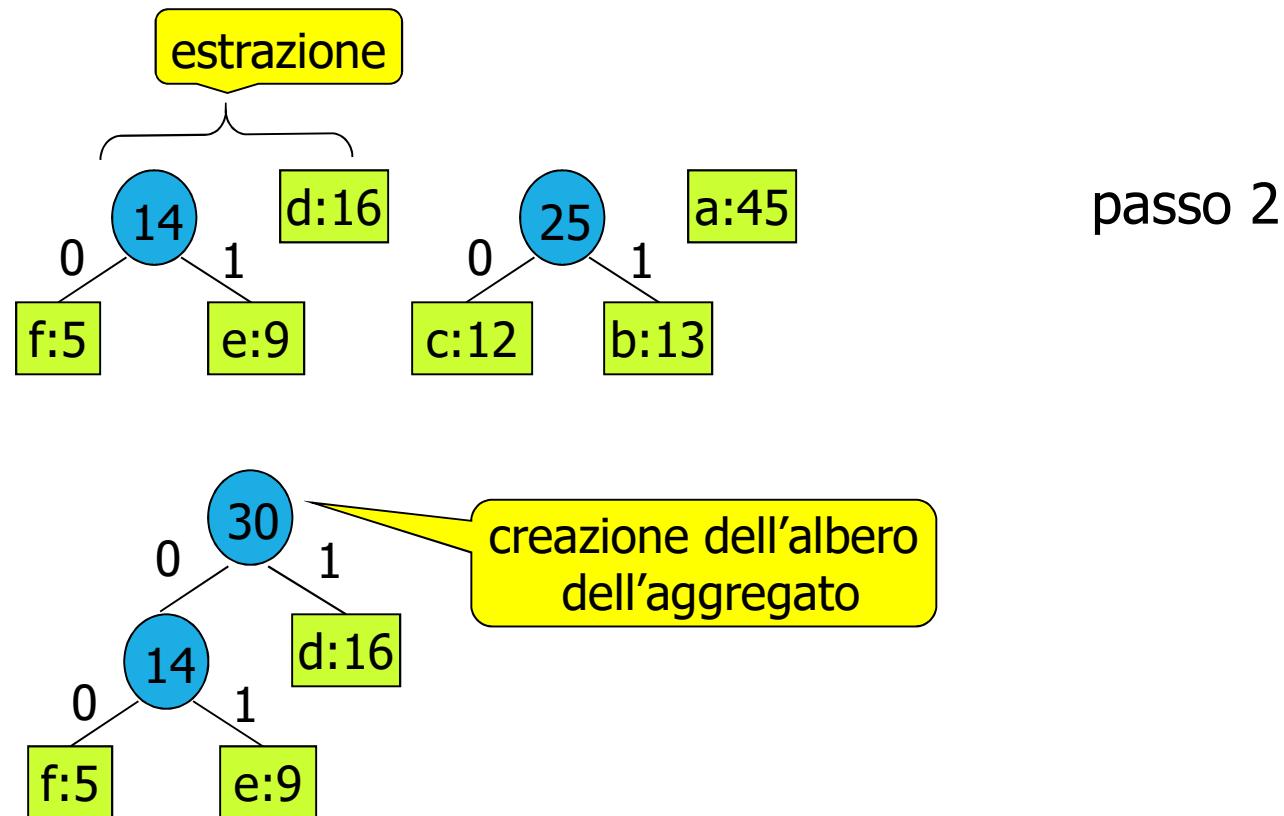
- Inizialmente: simbolo = foglia
- Passo i-esimo:
 - estrazione dei 2 simboli (o aggregati) a minor frequenza
 - costruzione dell'albero binario (aggregato di simboli): nodo = simbolo o aggregato, frequenza = somma delle frequenze
 - inserimento nella coda a priorità
- Terminazione: coda vuota.



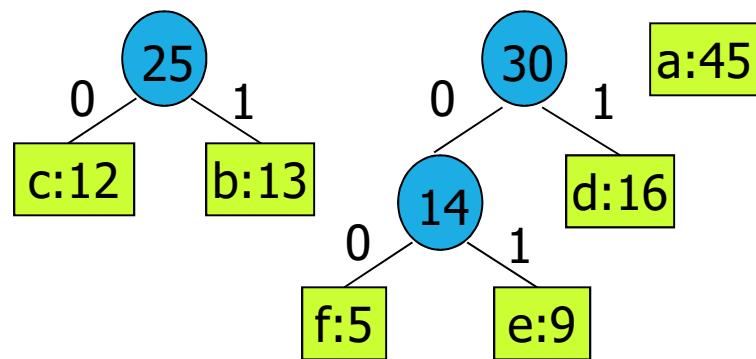


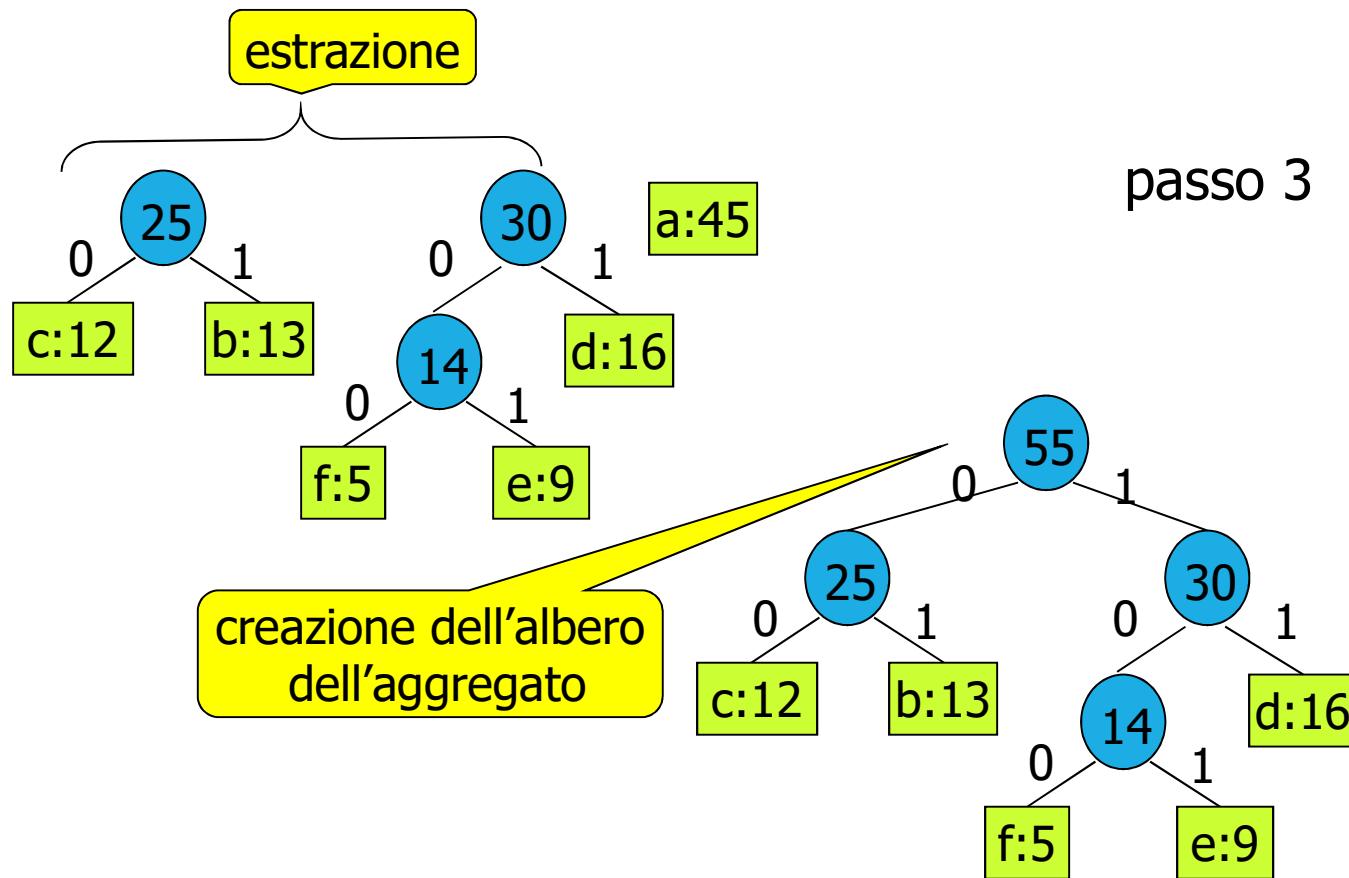
passo 1



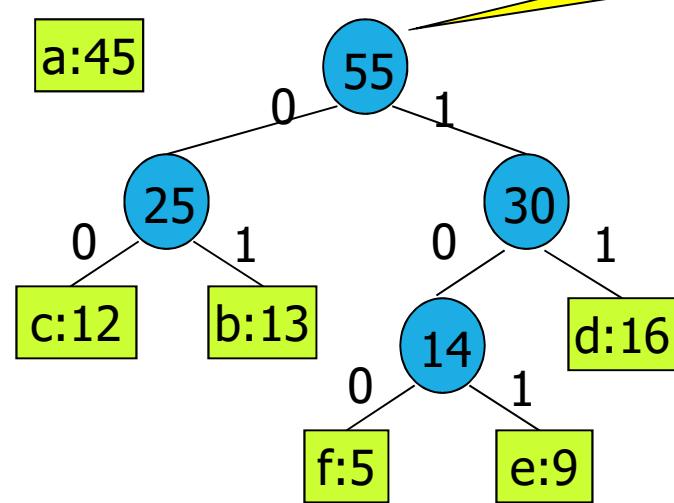


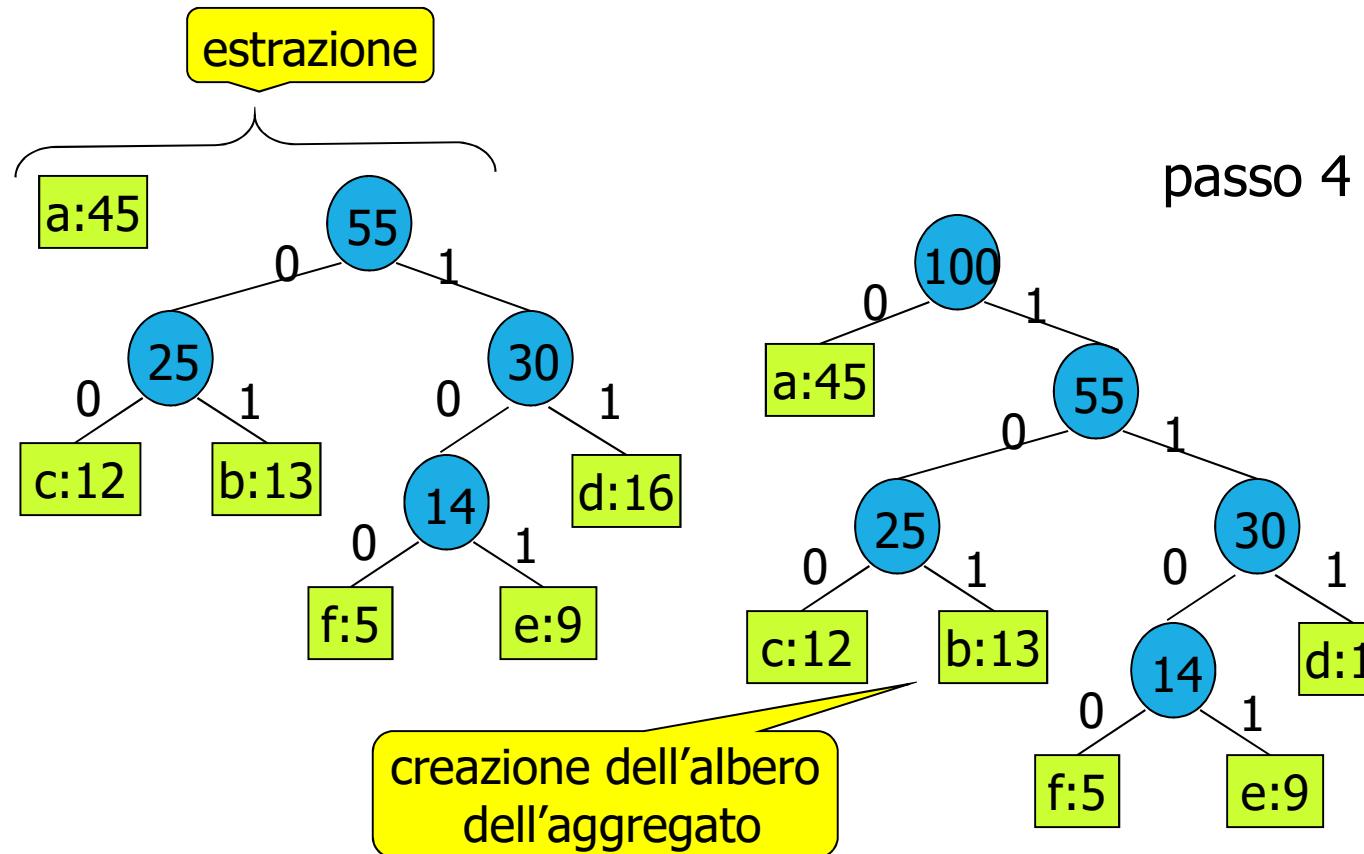
inserimento nella
coda a priorità





inserimento nella
coda a priorità



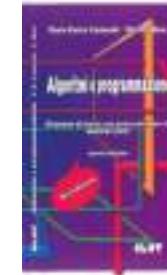


Riferimenti

- Selezione di attività:
 - Cormen 17.1
- Problema del ladro e dello zaino:
 - Cormen 17.2
- Codici di Huffman
 - Cormen 17.3

Esercizi di teoria

- 9. Paradigma greedy
 - 9.1 Activity selection
 - 9.2 Codici di Huffman





POLITECNICO
DI TORINO

Dipartimento
di Automatica e Informatica

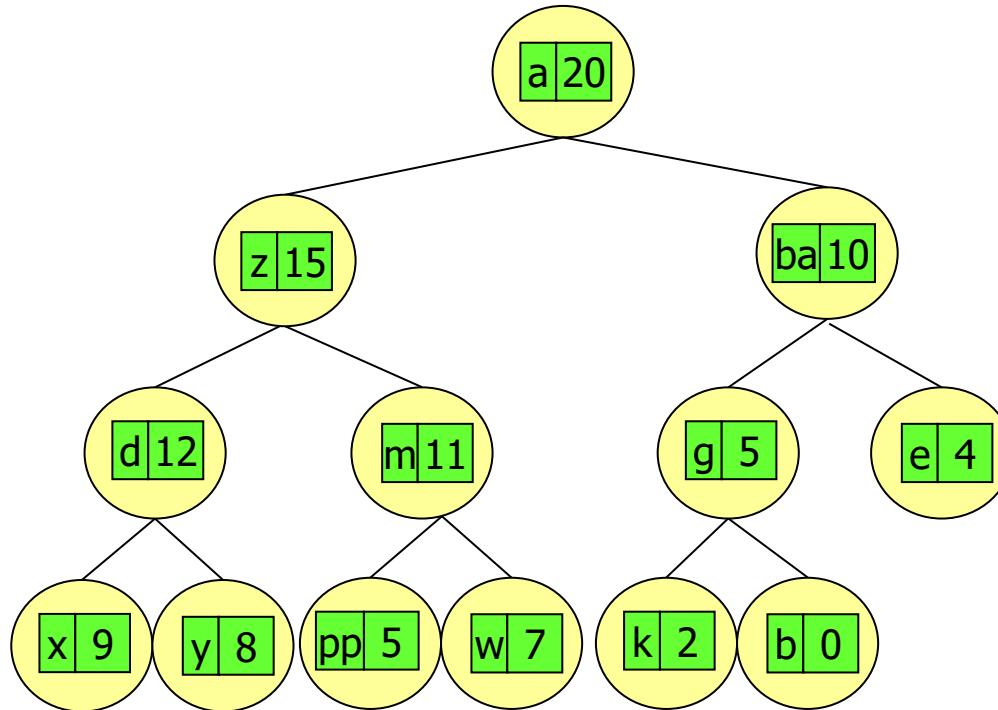
Code a Priorità e Heap

Gianpiero Cabodi e Paolo Camurati

ADT Heap

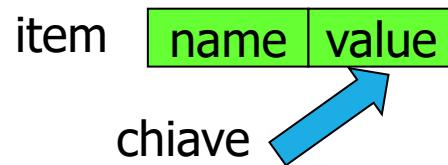
- Definizione: albero binario con
 - **proprietà strutturale:** quasi completo (tutti i livelli completi, tranne eventualmente l'ultimo, riempito da SX a DX) \Rightarrow quasi bilanciato
 - **proprietà funzionale:**
$$\forall i \neq r \text{ } \text{key}(\text{parent}(i)) \geq \text{key}(i)$$
 - conseguenza: chiave max nella radice
- Implementazione: mediante vettore.

Esempio



Item

- Quasi ADT Item
- Dati:
 - Nome (stringa), valore (intero)
 - Chiave = valore
 - Tipologia 3



ADT di classe Heap

Heap.h

```
typedef struct heap *Heap;

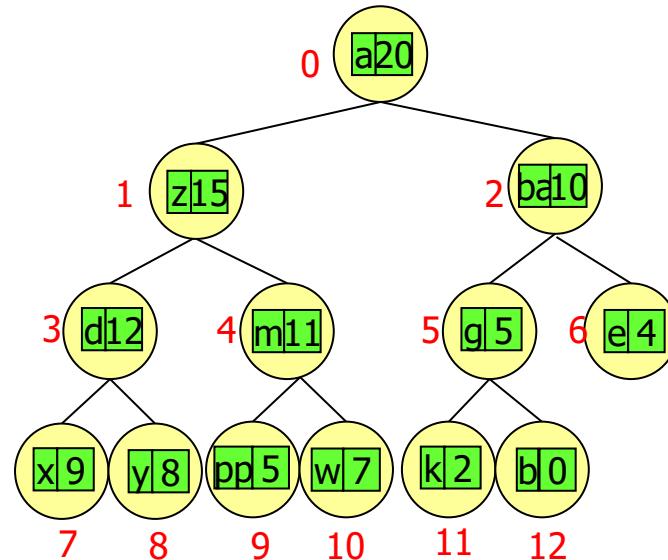
Heap    HEAPinit(int maxN);
Void    HEAPfree(Heap h);
void    HEAPfill(Heap h, Item val);
void    HEAPsort(Heap h);
void    HEAPdisplay(Heap h);
```

Implementazione

- Struttura dati: vettore di Item $h \rightarrow A[0..maxN-1]$
- $h \rightarrow \text{heapsize}$: numero di elementi in heap $h \rightarrow A$
- radice in $h \rightarrow A[0]$
- dato $h \rightarrow A[i]$:
 - il figlio SX è $h \rightarrow A[\text{LEFT}(i)]$ dove $\text{LEFT}(i) = 2i+1$
 - il figlio DX è $h \rightarrow A[\text{RIGHT}(i)]$ dove
 $\text{RIGHT}(i) = 2i+2$
 - il padre è $h \rightarrow A[\text{PARENT}(i)]$ dove
 $\text{PARENT}(i) = (i-1)/2$

Esempio

maxN = 15



h->A

a	z	ba	d	m	g	e	x	y	pp	w	k	b		
20	15	10	12	11	5	4	9	8	5	7	2	0		

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

h->heapsize 13

Heap.c

```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"
#include "Heap.h"

struct heap { Item *A; int heapsize; };

int LEFT(int i) { return (i*2 + 1); }
int RIGHT(int i) { return (i*2 + 2); }
int PARENT(int i) { return ((i-1)/2); }

Heap HEAPinit(int maxN) {
    Heap h;
    h = malloc(sizeof(*h));
    h->A = malloc(maxN*sizeof(Item));
    h->heapsize = 0;
    return h;
}
```

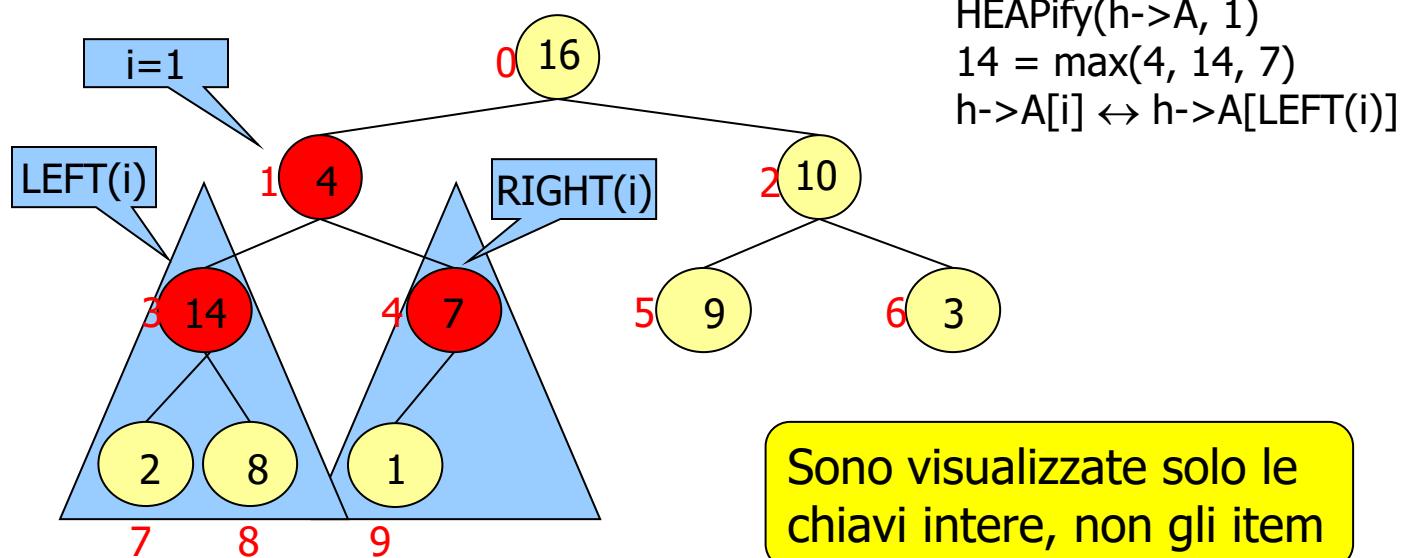
usata per inserire valori,
non necessariamente
il risultato sarà uno heap

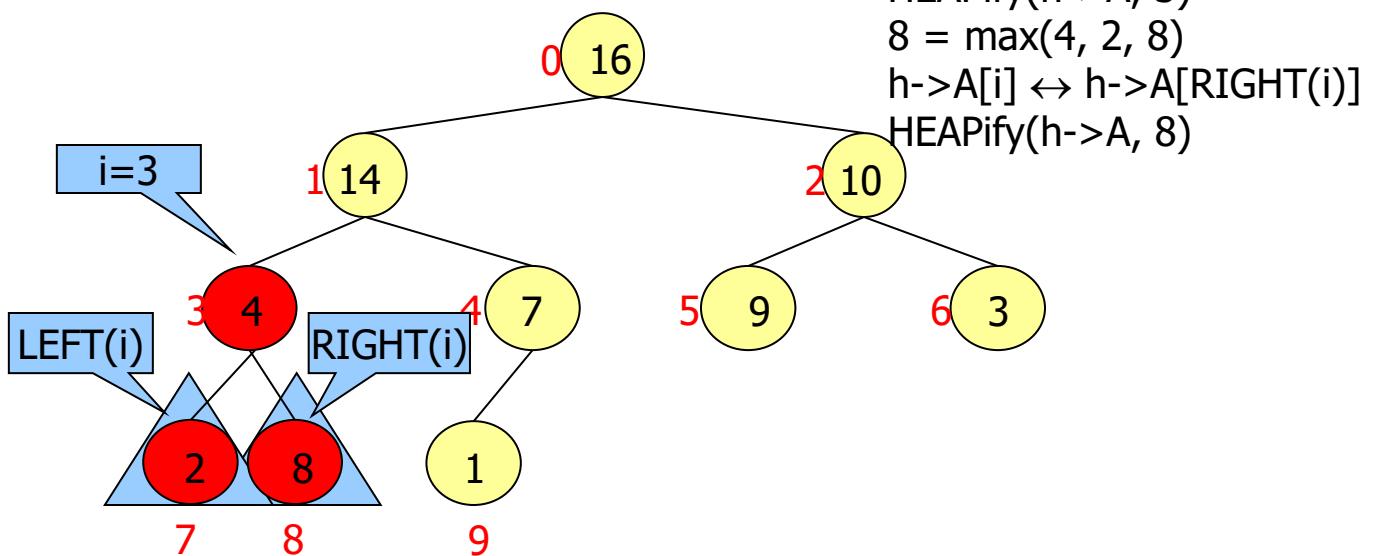
```
void HEAPfree(Heap h) {  
    free(h->A);  
    free(h);  
}  
  
void HEAPfill(Heap h, Item item) {  
    int i;  
    i = h->heapsize++;  
    h->A[i] = item;  
    return;  
}  
  
void HEAPdisplay(Heap h) {  
    int i;  
    for (i = 0; i < h->heapsize; i++)  
        ITEMstore(h->A[i]);  
}
```

Funzione HEAPify

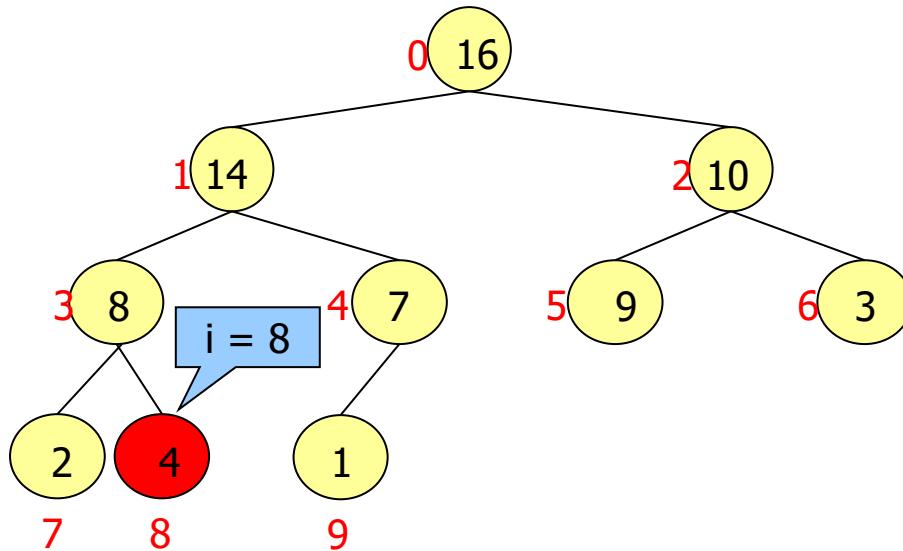
- Trasforma in heap i, LEFT(i), RIGHT(i), dove LEFT(i) e RIGHT(i) sono già heap
- assegna ad A[i] il max tra A[i], A[LEFT(i)] e A[RIGHT(i)]
- se c'è stato scambio $A[i] \leftrightarrow A[LEFT(i)]$, applica ricorsivamente HEAPify su sottoalbero con radice LEFT(i)
- analogamente per scambio $A[i] \leftrightarrow A[RIGHT(i)]$.
- Complessità: $T(n) = O(\lg n)$.

Esempio





HEAPify($h \rightarrow A$, 8)
foglia
terminazione.



```
void HEAPify(Heap h, int i) {
    int l, r, largest;
    l = LEFT(i);
    r = RIGHT(i);
    if ((l < h->heapsize) &&
        KEYcmp(KEYget(h->A[l]), KEYget(h->A[i])) == 1)
        largest = l;
    else
        largest = i;
    if ((r < h->heapsize) &&
        KEYcmp(KEYget(h->A[r]), KEYget(h->A[largest])) == 1)
        largest = r;
    if (largest != i) {
        Swap(h, i, largest);
        HEAPify(h, largest);
    }
}
```

Funzione HEAPbuild

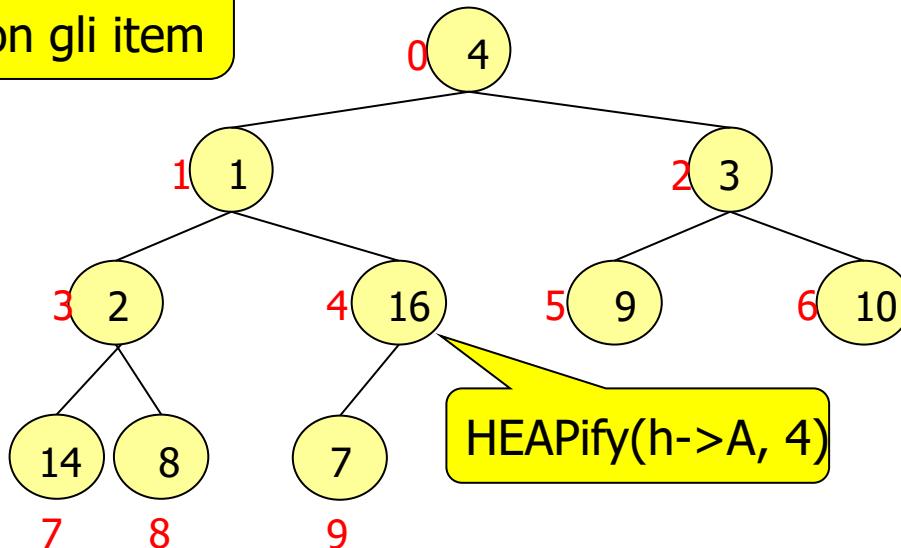
- Trasforma un albero binario memorizzato in vettore A in uno heap:
 - le foglie sono heap
 - applica HEAPify a partire dal padre dell'ultima foglia o coppia di foglie fino alla radice.

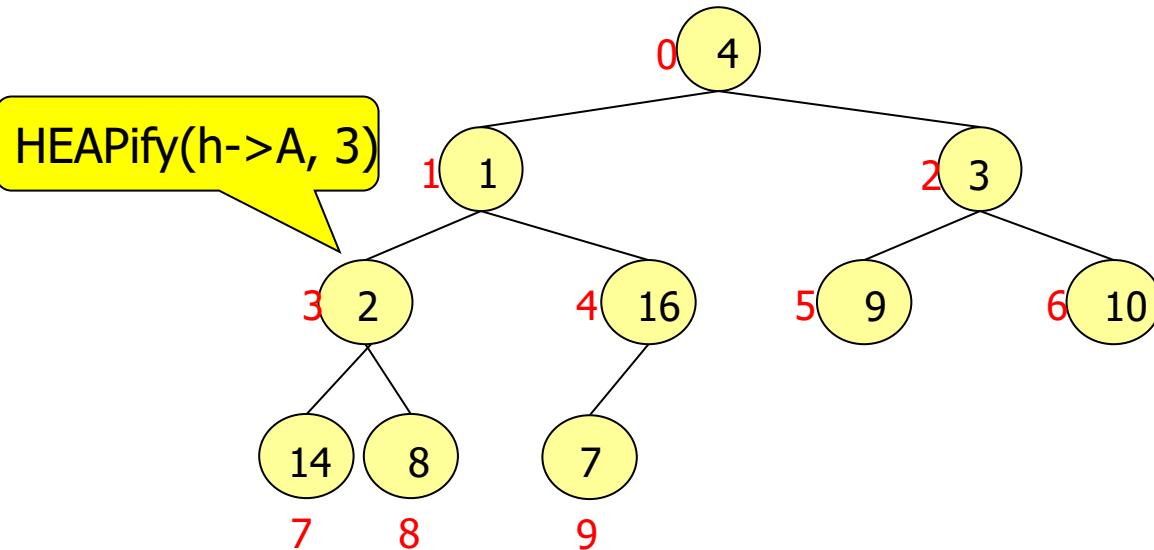
```
void HEAPbuild (Heap h) {  
    int i;  
    for (i=PARENT(h->heapsize-1); i >= 0; i--)  
        HEAPify(h, i);  
}
```

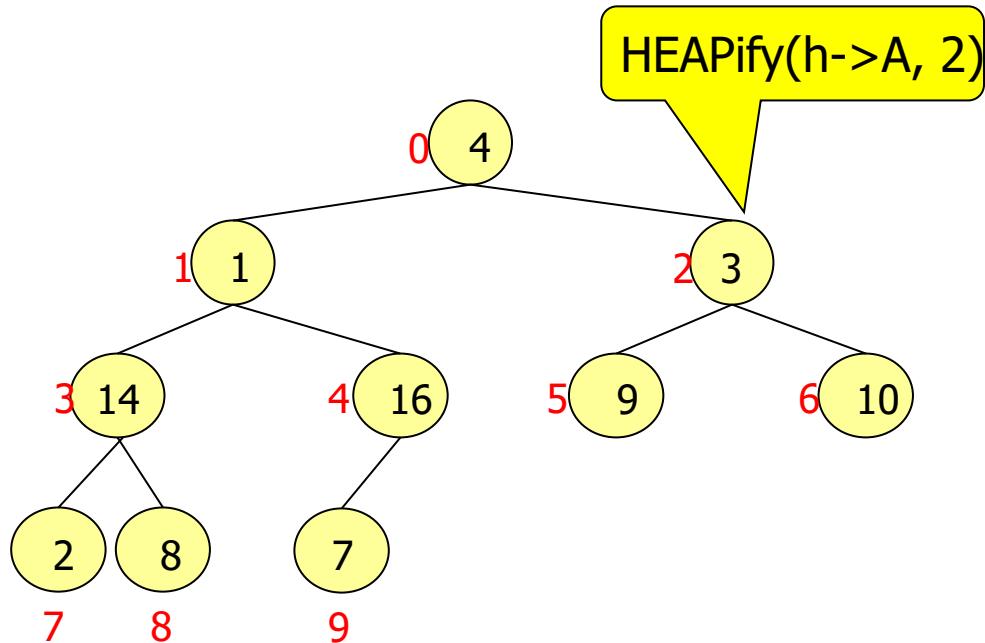
Esempio

HEAPbuild($h \rightarrow A$)

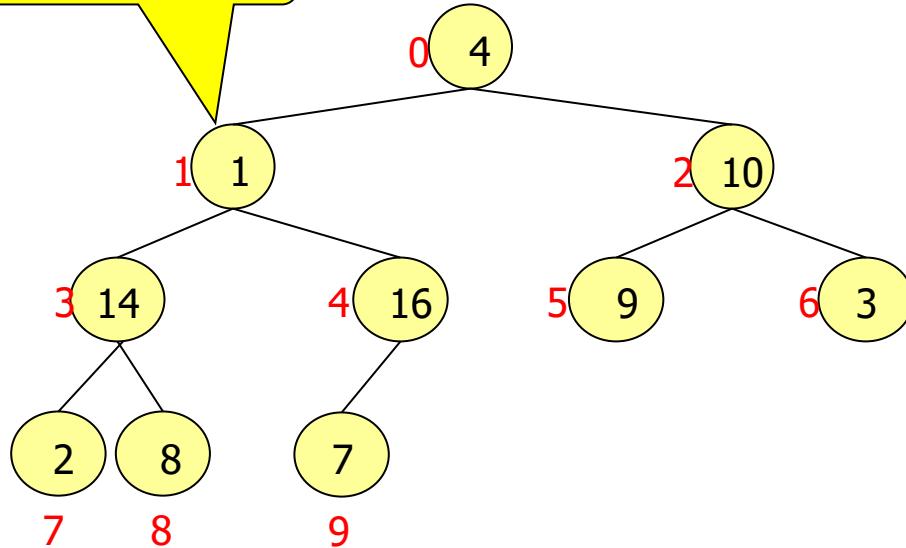
Sono visualizzate solo le chiavi intere, non gli item



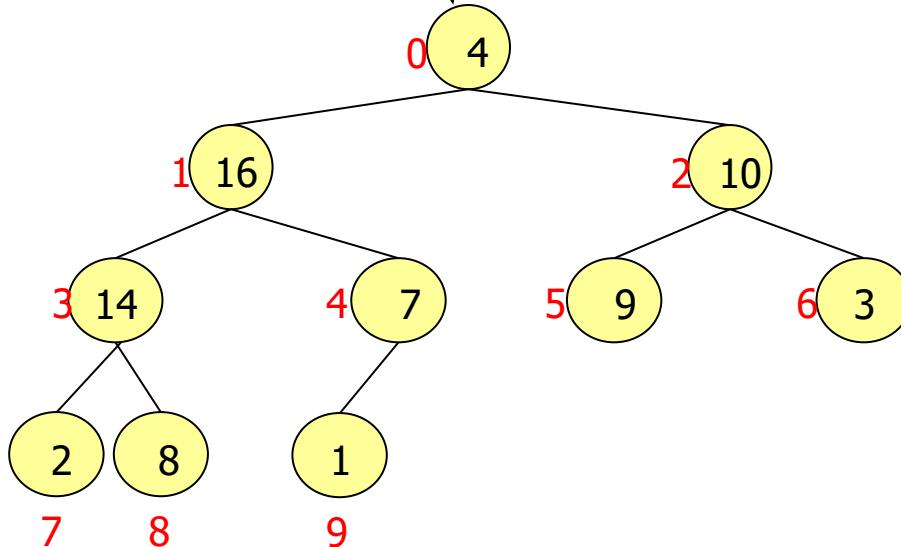


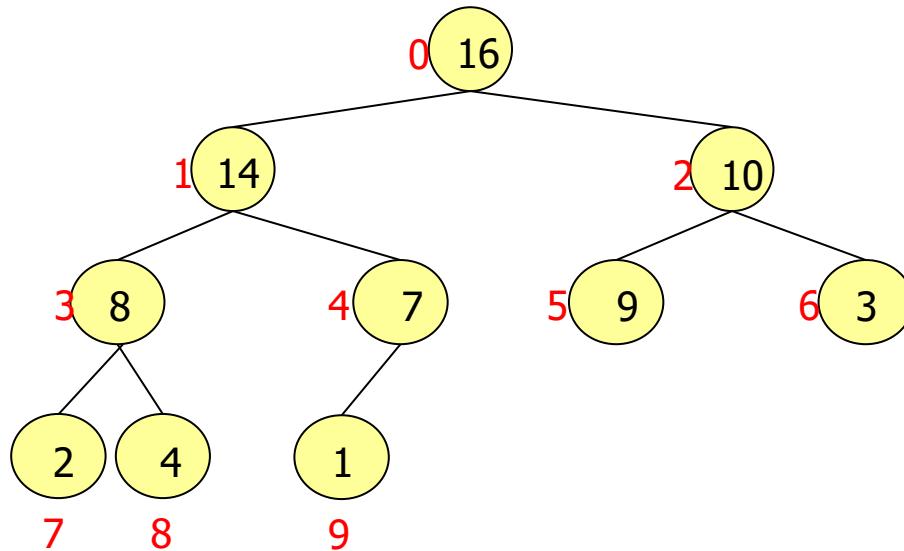


HEAPIfy($h \rightarrow A$, 1)



HEAPIfy($h \rightarrow A$, 0)





- Analisi di complessità:

- intuitiva ed imprecisa: n passi ciascuno di costo $\log n$, quindi $T(n) = O(n \lg n)$
- precisa: $T(n) = O(n)$.

Risoluzione per sviluppo (unfolding).

$$T(n) = 2T(n/2) + \log_2(n)$$

$$T(n/2) = 2T(n/4) + \log_2(n/2)$$

$$T(n/4) = 2T(n/8) + \log_2(n/4)$$

2 sottoalberi

Heapify

Sostituendo in $T(n)$:

$$T(n) = \sum_{i=0}^{\log_2 n} 2^i \log_2(n/2^i)$$

$$\sum_{i=0}^k i x^i = x (1 - (k+1)x^k + kx^{k+1}) / (1-x)^2$$

$$= \log_2 n \sum_{i=0}^{\log_2 n} 2^i - \sum_{i=0}^{\log_2 n} i 2^i$$

$$= \log_2 n (2n-1) - 2(1 - (\log_2 n + 1)n + 2n \log_2 n)$$

$$= 2n - \log_2 n - 2$$

$$= O(n)$$

Funzione HEAPsort

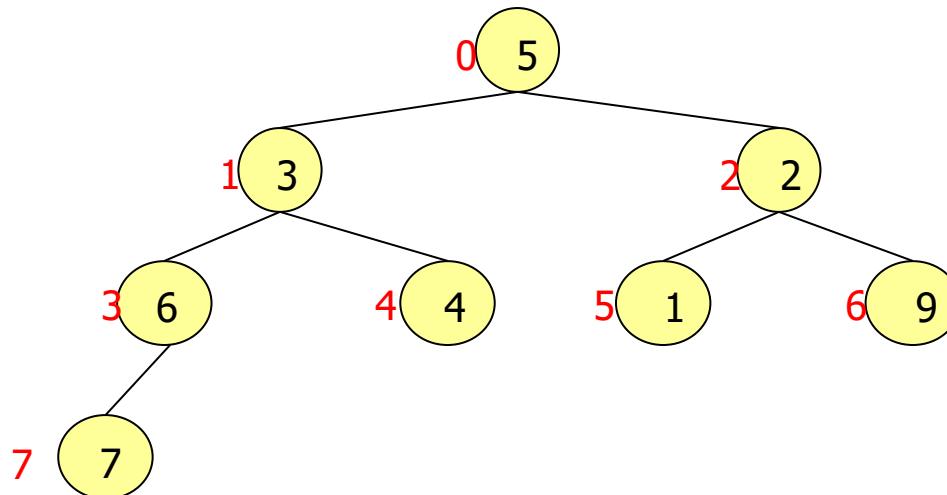
- Trasforma il vettore in uno heap mediante HEAPbuild
- Scambia il primo e ultimo elemento
- Riduci la dimensione dello heap di 1
- Ripristina la proprietà di heap
- Ripeti fino a esaurimento dello heap.
- Caratteristiche:
 - complessità: $T(n) = O(n \lg n)$.
 - in loco
 - non stabile

Esempio

Sono visualizzate solo le chiavi intere, non gli item

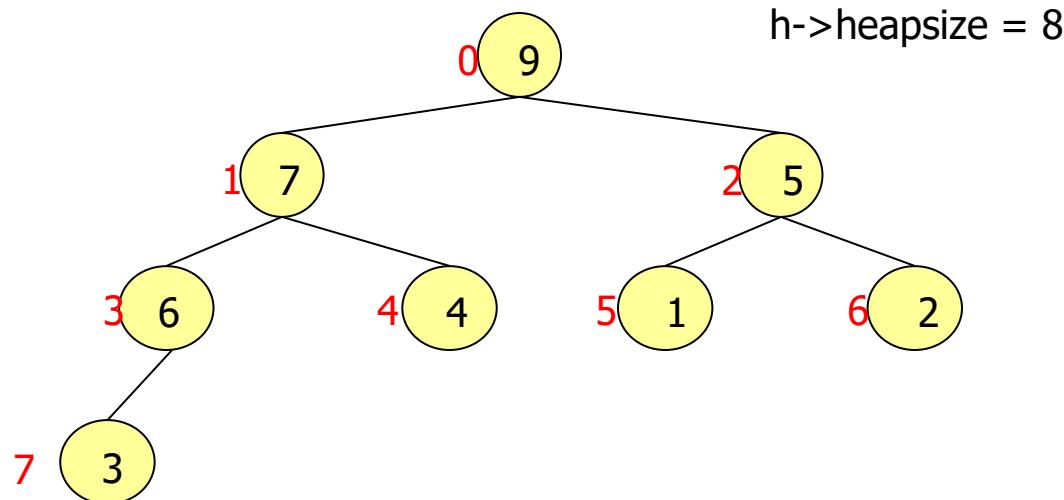
Configurazione iniziale

$h \rightarrow A$ [5 | 3 | 2 | 6 | 4 | 1 | 9 | 7]



Applico HEAPbuild

h->A [9 | 7 | 5 | 6 | 4 | 1 | 2 | 3]

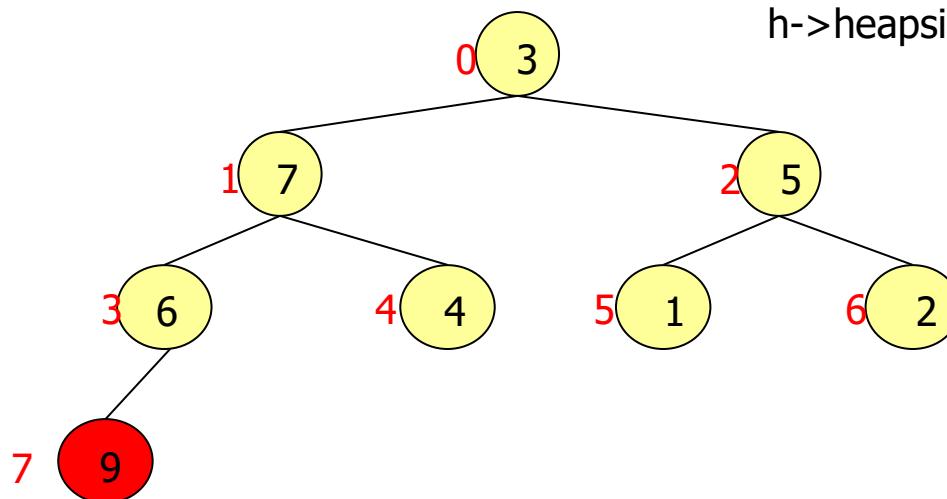


$h \rightarrow A[0] \leftrightarrow h \rightarrow A[h \rightarrow \text{heapsize} - 1]$

$h \rightarrow \text{heapsize}--$

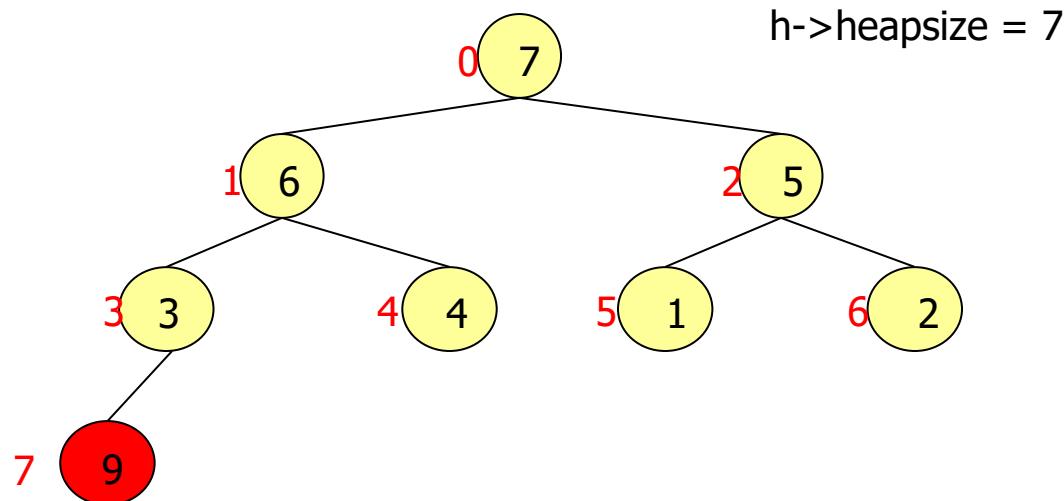


$h \rightarrow \text{heapsize} = 7$



HEAPify($h \rightarrow A$, 0)

$h \rightarrow A$ [7 | 6 | 5 | 3 | 4 | 1 | 2 | 9]

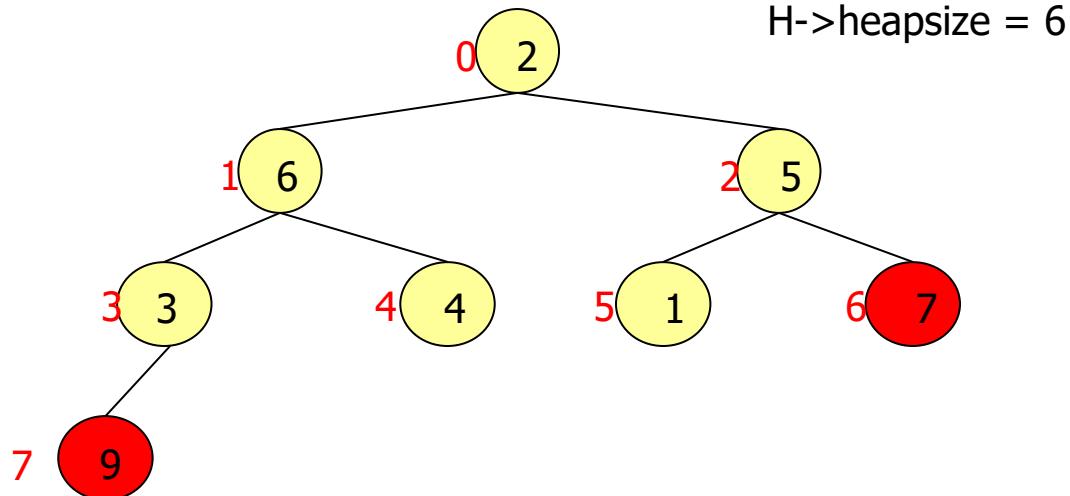


$h \rightarrow A[0] \leftrightarrow h \rightarrow A[h \rightarrow \text{heapsize} - 1]$

$h \rightarrow \text{heapsize}--$

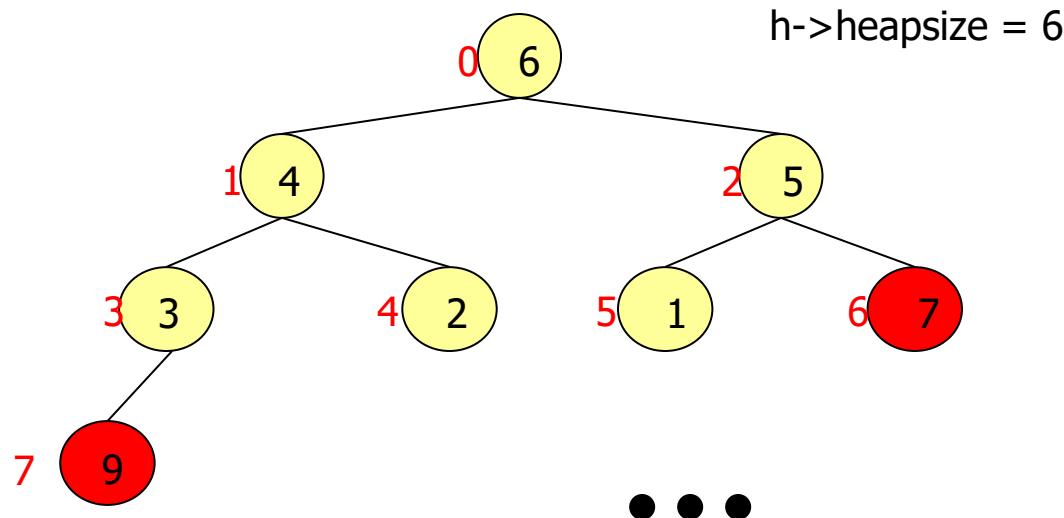


$H \rightarrow \text{heapsize} = 6$



HEAPify($h \rightarrow A$, 0)

$h \rightarrow A$ [6 | 4 | 5 | 3 | 2 | 1 | 7 | 9]



```
void HEAPSORT(Heap h) {  
    int i, j;  
    HEAPbuild(h);  
    j = h->heapsize;  
    for (i = h->heapsize-1; i > 0; i--) {  
        Swap (h,0,i);  
        h->heapsize--;  
        HEAPify(h,0);  
    }  
    h->heapsize = j;  
}
```

Coda a priorità

Definizione:

- struttura dati PQ per mantenere un set di elementi di tipo Item, ciascuno dei quali include un campo priorità
- operazioni principali: inserzione, estrazione del massimo, lettura del massimo, cambio di priorità.

ADT di classe Coda a Priorità

PQ.h

```
typedef struct pqueue *PQ;

PQ    PQinit(int maxN);
void  PQfree(PQ pq);
int   PQempty(PQ pq);
void  PQinsert(PQ pq, Item val);
Item  PQextractMax(PQ pq);
Item  PQshowMax(PQ pq);
void  PQdisplay(PQ pq);
int   PQsize(PQ pq);
void  PQchange(PQ pq, Item val);
```

discussione a parte

Implementazione della struttura dati:

- vettore/lista non ordinato
 - vettore/lista ordinato
 - heap di dati/indici.
- } non considerati qui,
cfr Tipi di Dato Astratto

Complessità

	PQinsert	PQshowMax	PQextractMax
Vettore non ordinato	1	N	N
Lista non ordinata	1	N	N
Vettore ordinato	N	1	1
Lista ordinata	N	1	1
Heap di item/indici	$\log N$	1	$\log N$

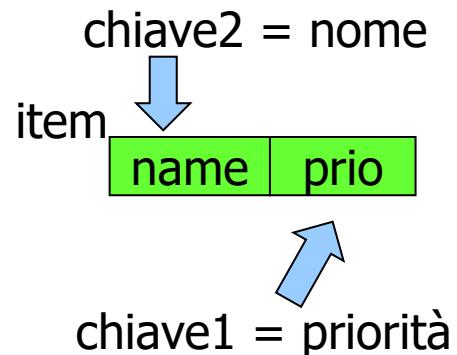
max in coda

max in testa

Cosa contiene l'ADT Coda a priorità?

I soluzione: la **coda a priorità contiene dati** (lo heap che realizza la coda a priorità contiene i dati), l'ADT è una struct con:

1. la coda a priorità: vettore (heap) pq->A di dati di tipo Item (quasi ADT, tipologia 3)
2. heapsize: intero.



Utente

item $\xrightarrow{\text{PQinsert}(pq, \text{item})}$

item $\xleftarrow{\text{PQshowMax}(pq)}$

item $\xleftarrow{\text{PQextractMax}(pq)}$

ADT I cat. coda a priorità di dati

$pq \rightarrow A$

a	q	zz	qa	cd	s	w	c		
81	70	20	48	5	9	19	15		

0 1 2 3 4 5 6 7 8 9

$pq \rightarrow \text{heapsize}$ 8

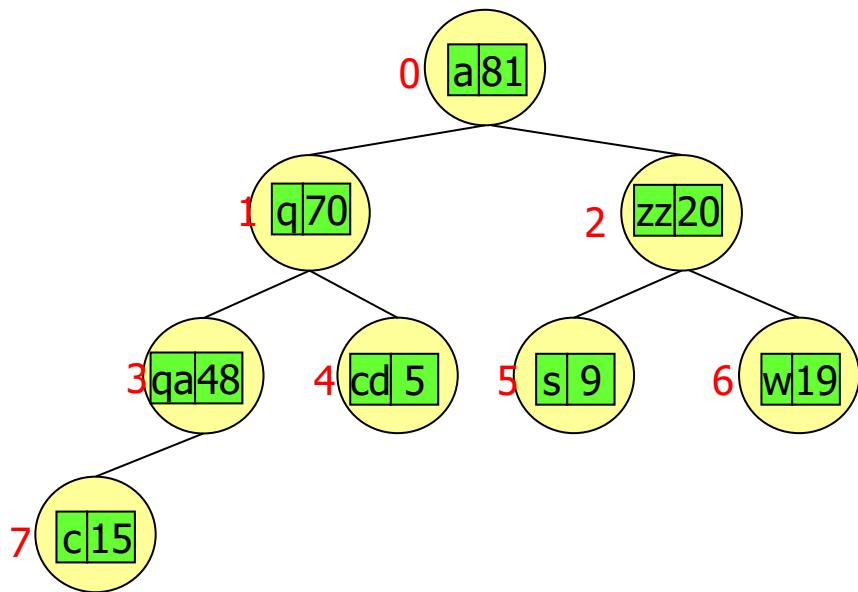
`pq->A`

a	q	zz	qa	cd	s	w	c		
81	70	20	48	5	9	19	15		

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

`0`

`pq->heapsize` 8



ADT di I cat. Coda a priorità

PQ.c

```
#include <stdlib.h>
#include "Item.h"
#include "PQ.h"

struct pqueue { Item *A; int heapsize; };

static int LEFT(int i) { return (i*2 + 1); }
static int RIGHT(int i) { return (i*2 + 2); }
static int PARENT(int i) { return ((i-1)/2); }

PQ PQinit(int maxN){
    PQ pq = malloc(sizeof(*pq));
    pq->A = (Item *)malloc(maxN*sizeof(Item));
    pq->heapsize = 0;
    return pq;
}
```

```
void PQfree(PQ pq){
    free(pq->A);
    free(pq);
}

int PQempty(PQ pq) { return pq->heapsize == 0; }

int PQsize(PQ pq) { return pq->heapsize; }

Item PQshowMax(PQ pq) { return pq->A[0]; }

void PQdisplay(PQ pq) {
    int i;
    for (i = 0; i < pq->heapsize; i++)
        ITEMstore(pq->A[i]);
}
```

Funzione PQinsert

- Aggiunge una foglia all'albero (cresce per livelli da SX a DX, rispettando la proprietà strutturale)
- Risale dal nodo corrente (inizialmente la foglia appena creata) fino al più alla radice. Confronta la chiave del dato contenuto nel padre con la chiave del dato da inserire, facendo scendere il dato del padre nel figlio se la chiave da inserire è maggiore, altrimenti inserisce il dato nel nodo corrente.

Complessità: $T(n) = O(\lg n)$.

```
void PQinsert (PQ pq, Item val) {
    int i;
    i = pq->heapsize++;
    while((i>=1) &&
          (KEYcmp(KEYget(pq->A[PARENT(i)]),KEYget(val))== -1)){
        pq->A[i] = pq->A[PARENT(i)];
        i = PARENT(i);
    }
    pq->A[i] = val;
    return;
}
```

Esempio

Inserzione di r 75

0	1	2	3	4	5	6	7	8	9
a	q	zz	qa	cd	s	w	c		
81	70	20	48	5	9	19	15		

pq->heapsize 8

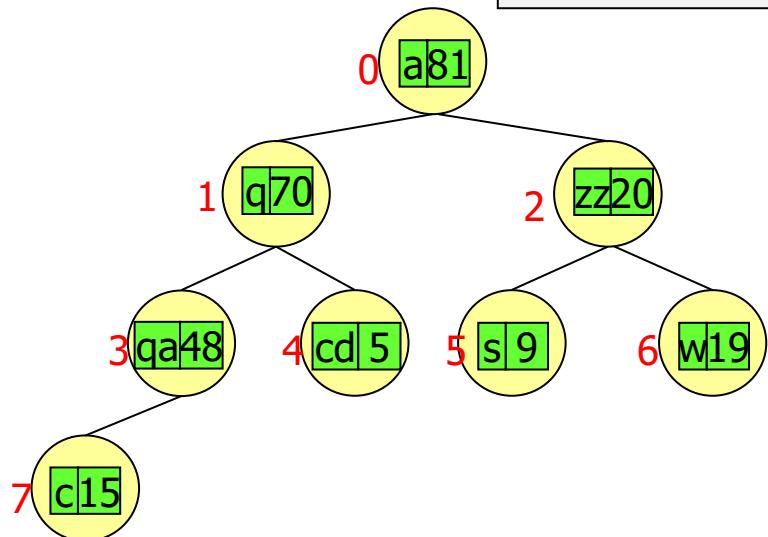


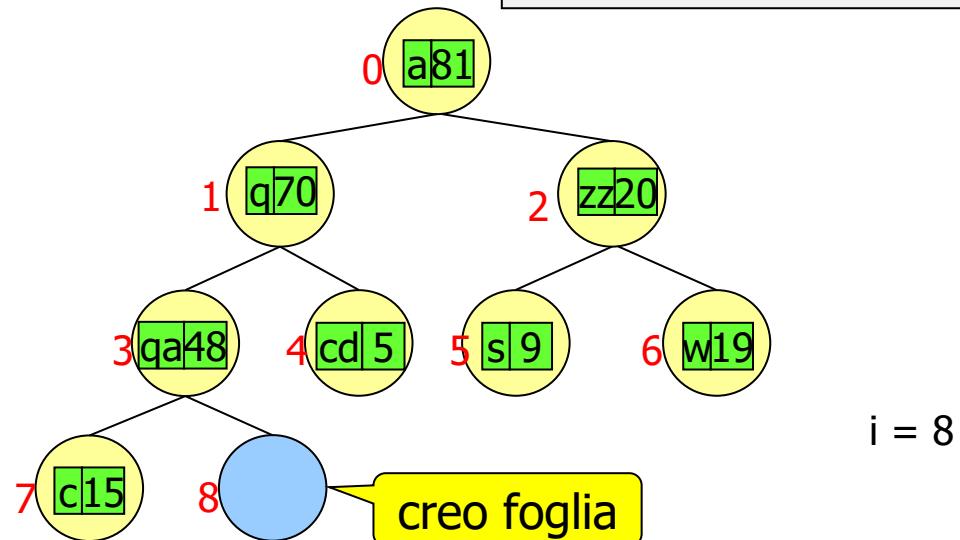
Diagram illustrating a priority queue (pq) structure and its heap representation.

Priority Queue State:

a	q	zz	qa	cd	s	w	c	
81	70	20	48	5	9	19	15	

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

Heap Size: pq->heapsize = 9



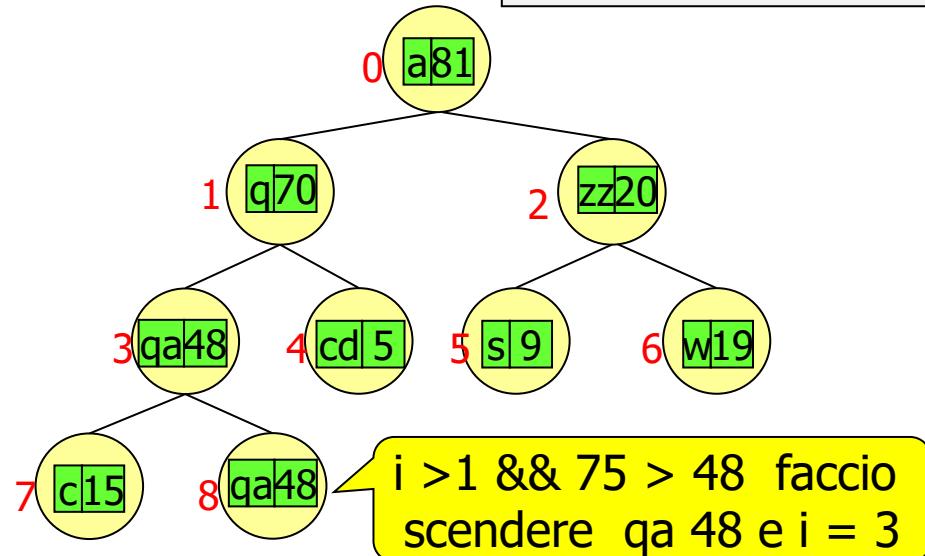
i = 8

pq->A

a	q	zz	qa	cd	s	w	c	qa	
81	70	20	48	5	9	19	15	48	

0 1 2 3 4 5 6 7 8 9

pq->heapsize 9

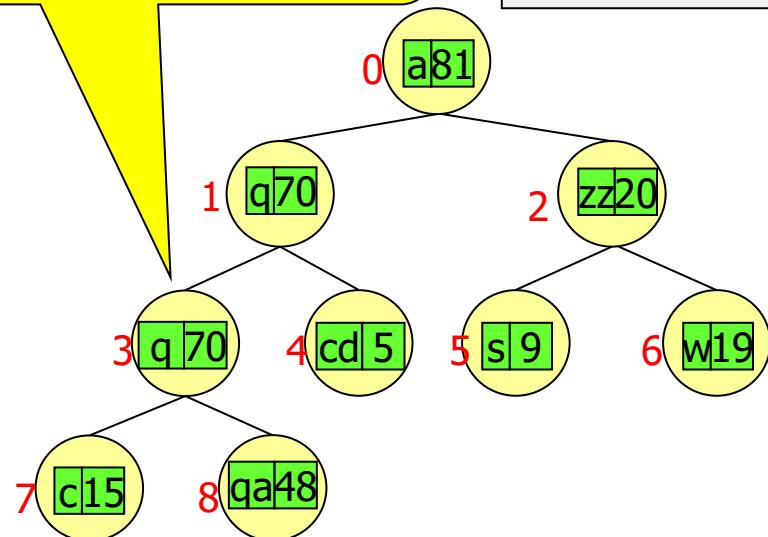


pq->A	a	q	zz	q	cd	s	w	c	qa	
	81	70	20	70	5	9	19	15	48	

0 1 2 3 4 5 6 7 8 9

pq->heapsize 9

i > 1 && 75 > 70
faccio scendere
q 70 e i = 1



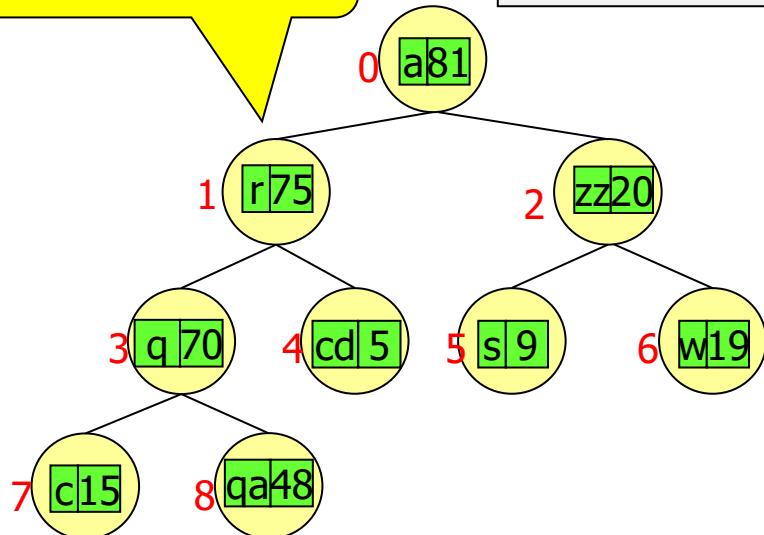
`pq->A`

a	r	zz	q	cd	s	w	c	qa	
81	75	20	70	5	9	19	15	48	

0 1 2 3 4 5 6 7 8 9

`pq->heapsize` 9

$i > 1 \&& 75 < 81$
inserisco r 75



Funzione PQextractMax

- Modifica lo heap, estraendone il valore massimo, che è contenuto nella radice:
 - scambia la radice con l'ultima delle foglie (quella più a destra nell'ultimo livello)
 - riduce di 1 della dimensione dello heap
 - ripristina le proprietà dello heap mediante applicazione di HEAPify.

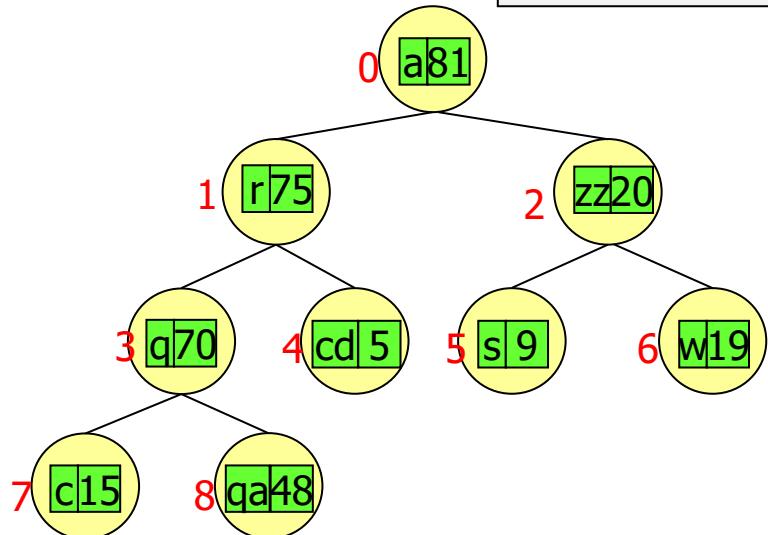
Complessità: $T(n) = O(\lg n)$.

```
Item PQextractMax(PQ pq) {  
    Item val;  
    Swap (pq, 0,pq->heapsize-1);  
    val = pq->A[pq->heapsize-1];  
    pq->heapsize--;  
    HEAPify(pq, 0);  
    return val;  
}
```

Esempio

0	1	2	3	4	5	6	7	8	9
a	r	zz	q	cd	s	w	c	qa	
81	75	20	70	5	9	19	15	48	

pq->heapsize 9

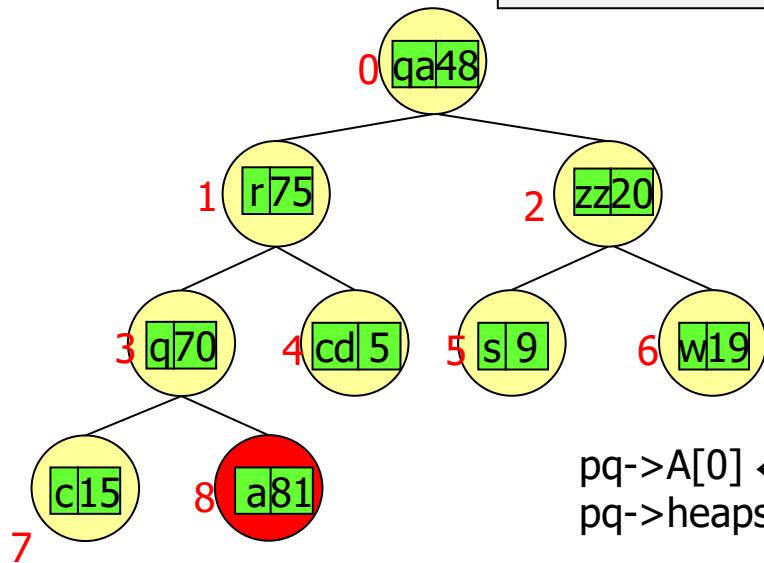


`pq->A`

qa	r	zz	q	cd	s	w	c	a	
48	75	20	70	5	9	19	15	81	

0 1 2 3 4 5 6 7 8 9

`pq->heapsize` 8



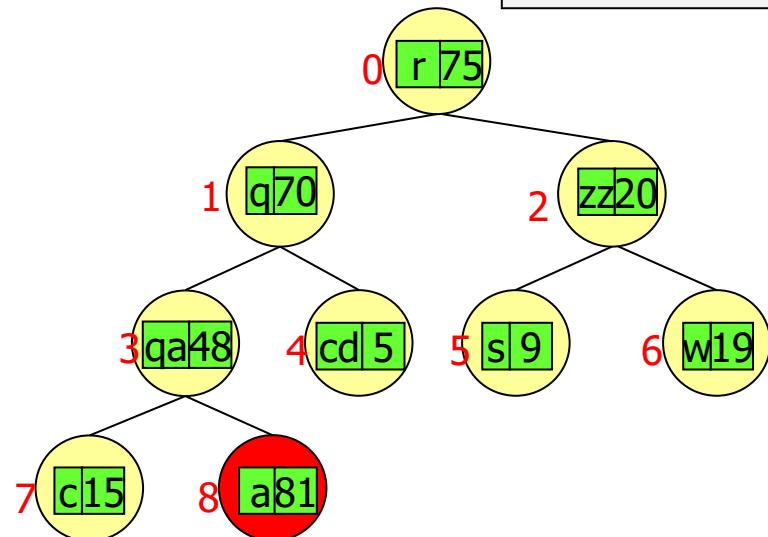
`pq->A[0] ↔ pq->A[pq->heapsize-1]`
`pq->heapsize--`

HEAPIfy(pq->A, 0)

0	1	2	3	4	5	6	7	8	9
r 75	q 70	zz 20	qa 48	cd 5	s 9	w 19	c 15	a 81	

0 1 2 3 4 5 6 7 8 9

pq->heapsize 8



Funzione PQchange

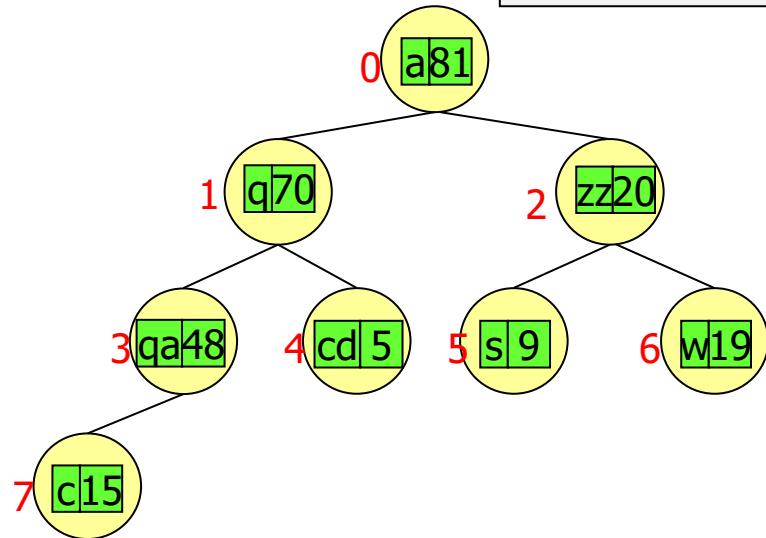
- Modifica la priorità di un elemento, la cui **posizione** (indice nello heap) viene calcolata con una scansione di costo lineare
- O risale dalla posizione data fino al più alla radice confrontando la chiave del padre con la chiave modificata, facendo scendere la chiave del padre nel figlio se la chiave modificata è maggiore, altrimenti la inserisce nel nodo corrente
- O applica HEAPify a partire dalla posizione data.

Complessità: $T(n) = O(n) + O(\lg n) = O(n)$.

Esempio

$pq \rightarrow A$	a	q	zz	qa	cd	s	w	c		
	81	70	20	48	5	9	19	15		
	0	1	2	3	4	5	6	7	8	9

$pq \rightarrow \text{heapsize}$ 8



Cambio la priorità di w da 19
a 90. L'elemento si trova
all'indice 6.

pq->A

a	q	zz	qa	cd	s	zz	c		
81	70	20	48	5	9	20	15		

0 1 2 3 4 5 6 7 8 9

pq->heapsize 8

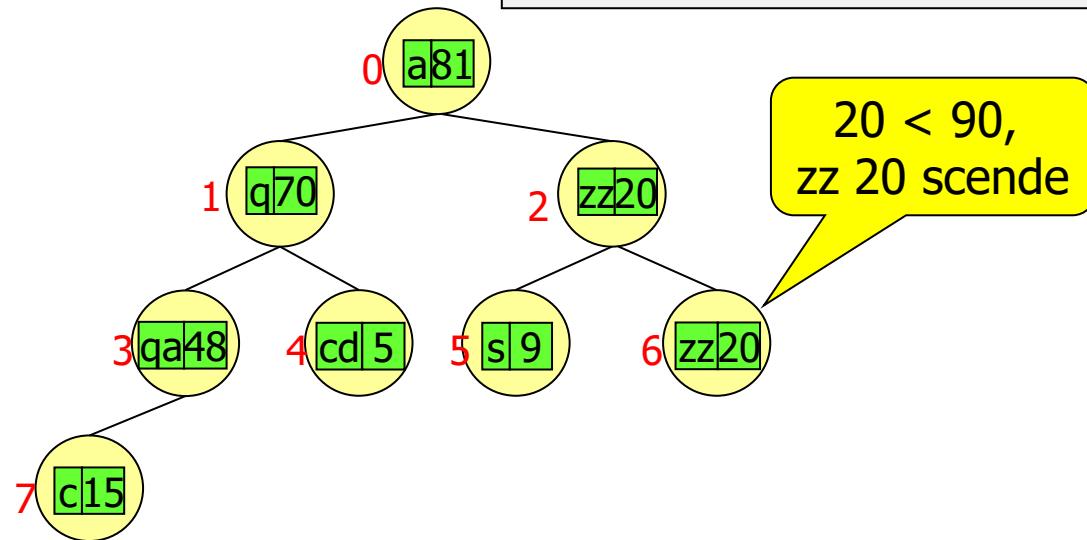


Diagram illustrating a priority queue (pq) structure and its heap representation.

Priority Queue State:

- Root Node:** 0 a81
- Left Child (Node 1):** q70
- Right Child (Node 2):** a81
- Left-Left Child (Node 3):** qa48
- Left-Right Child (Node 4):** cd5
- Right-Left Child (Node 5):** s9
- Right-Right Child (Node 6):** zz20
- Bottom-Left Child (Node 7):** c15

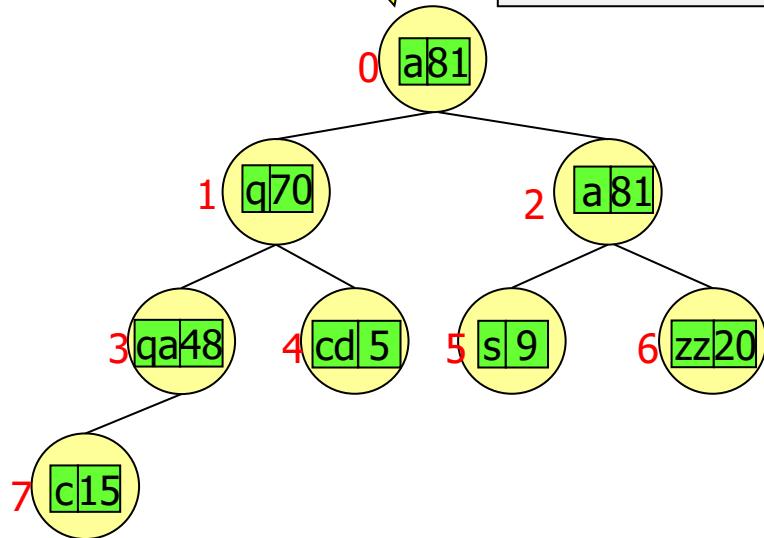
Heap Representation:

a	q	a	qa	cd	s	zz	c		
81	70	81	48	5	9	20	15		

Metadata:

- pq->A:** A pointer to the array containing the heap elements.
- pq->heapsize:** A value indicating the current size of the heap, which is 8.

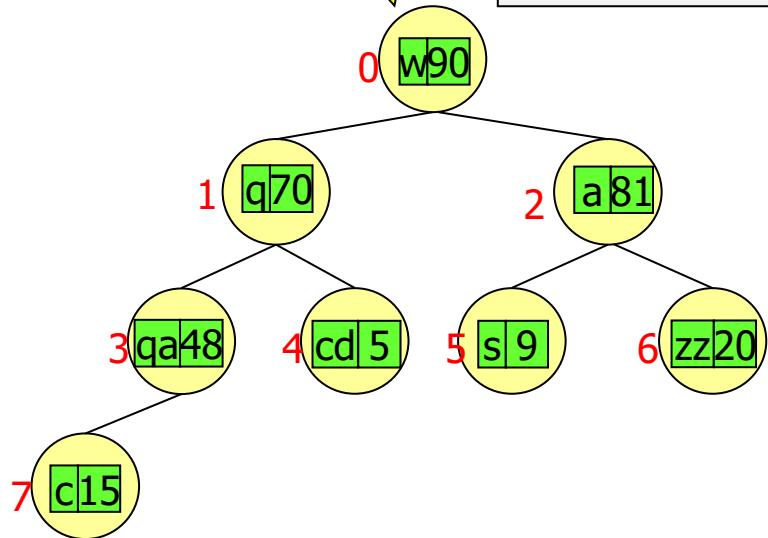
Note: A yellow speech bubble contains the text: $81 < 90$, a 81 scende.



radice:
inserisco w 90

pq->A	w	q	a	qa	cd	s	zz	c		
	90	70	81	48	5	9	20	15		
	0	1	2	3	4	5	6	7	8	9

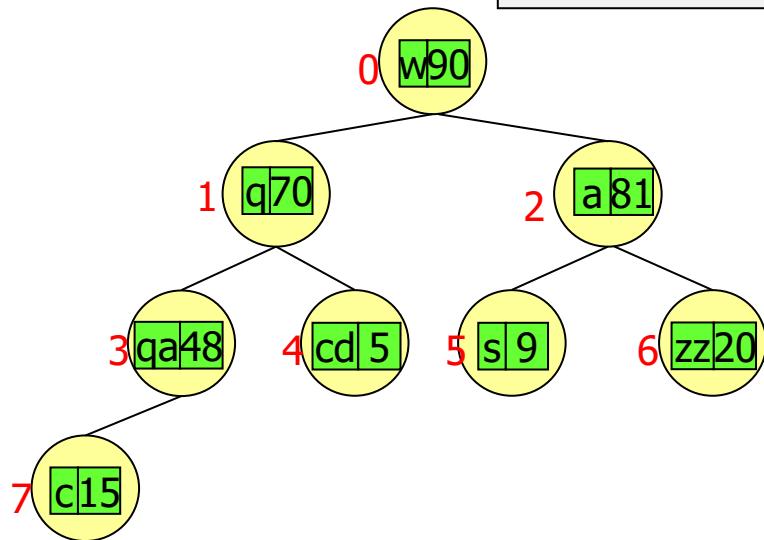
pq->heapsize 8



Esempio

0	1	2	3	4	5	6	7	8	9
w	q	a	qa	cd	s	zz	c		
90	70	81	48	5	9	20	15		

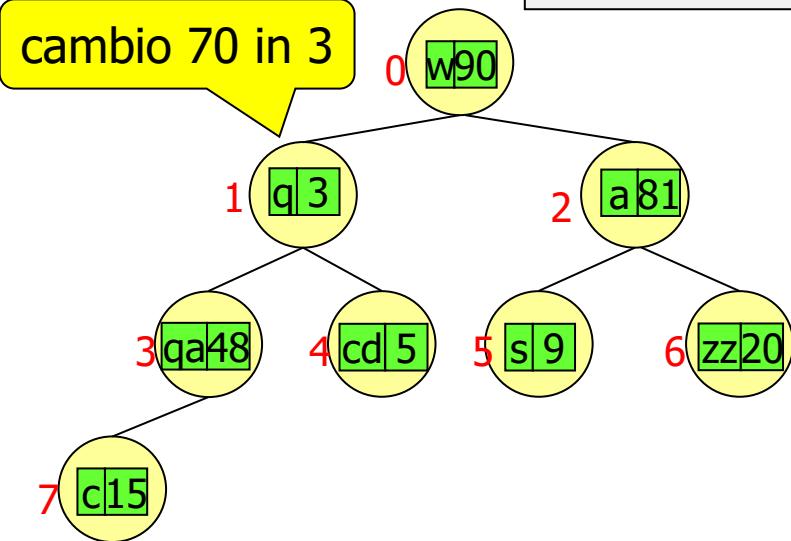
pq->heapsize 8



Cambio la priorità di q da 70
a 3. L'elemento si trova
all'indice 1.

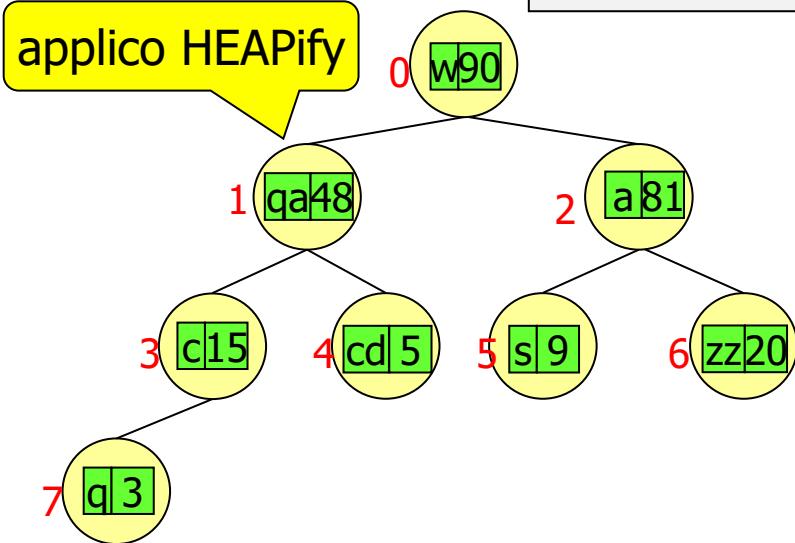
<code>pq->A</code>	w	q	a	qa	cd	s	zz	c		
	90	3	81	48	5	9	20	15		
	0	1	2	3	4	5	6	7	8	9

`pq->heapsize` 8



<code>pq->A</code>	w	qa	a	c	cd	s	zz	q		
	90	48	81	15	5	9	20	3		
	0	1	2	3	4	5	6	7	8	9

`pq->heapsize` 8



```
void PQchange (PQ pq, Item val) {
    int i, found = 0, pos;
    for (i = 0; i < pq->heapsize && found == 0; i++)
        if (NAMEcmp(NAMEget(&(pq->A[i])), NAMEget(&val))==0) {
            found = 1;
            pos = i;
        }

    if (found==1) {
        while(pos>=1 &&
              PRIOget(pq->A[PARENT(pos)])<PRIOget(val)){
            pq->A[pos] = pq->A[PARENT(pos)];
            pos = PARENT(pos);
        }
        pq->A[pos] = val;
        HEAPify(pq, pos);
    }
    else
        printf("key not found!\n");
    return;
}
```

È possibile migliorare PQchange O(n)?

Occorre fare in modo che NON si debba **cercare** l'item nella coda

- Soluzione: L'Item ricorda/sa «dove» si trova nella coda (gestito dalle operazioni su PQ)

Possibili implementazioni:

A. In coda si inseriscono solo «riferimenti» ad Item (es. puntatori)

- l'Item ha un campo pos (posizione in coda prioritaria)
- Il modulo Item fornisce le operazioni ITEMsetPos ITEMgetPos che permettono di ottenere la posizione di un item con costo O(1)

B. L'Item è un indice (oppure ha come campo valore un indice in un vettore): in pratica è un riferimento a «dove» sono collocate le informazioni. La priorità può essere parte dell'item oppure può essere «affiancata» all'item/indice

- La coda sfrutta internamente la corrispondenza dato-indice, associa a ogni item una casella unica di un vettore
- E' possibile ottenere la posizione di un Item in coda con tempo O(1)

C. Non è possibile gestire un riferimento ad Item con puntatore o indice, si usa la chiave come riferimento (deve essere univoca, senza possibili duplicati)

- il modulo PQ usa la chiave dell'item per gestire una corrispondenza chiave-posizione mediante una tabella di simboli efficiente (es. Hash O(1), BST bilanciato O(lg(n)))

È possibile migliorare PQchange O(n)?

Occorre fare in modo che NON si debba **cercare** l'item nella coda

- Soluzione: L'Item ricorda/sa «dove» si trova nella coda (gestito dalle operazioni su PQ)

Possibili implementazioni:

- A. In coda si inseriscono solo «riferimenti» ad Item (es. puntatori)

- l'Item ha un campo pos (posizione in coda prioritaria)
- Il modulo Item fornisce le operazioni ITEMsetPos ITEMgetPos che permettono di ottenere la posizione di un item con costo O(1)

- B. L'Item è un indice (oppure ha come campo valore un indice in un vettore): in pratica è un riferimento a «dove» sono collocate le informazioni. La priorità può essere parte dell'item oppure può essere «affiancata» all'item/indice

- La coda sfrutta internamente la corrispondenza dato-indice, associa a ogni item una casella unica di un vettore
- E' possibile ottenere la posizione di un Item in coda con tempo O(1)

- C. Non è possibile gestire un riferimento ad Item con puntatore o indice, si usa la chiave come riferimento (deve essere univoca, senza possibili duplicati)

- il modulo PQ usa la chiave dell'item per gestire una corrispondenza chiave-posizione mediante una tabella di simboli efficiente (es. Hash O(1), BST bilanciato O(lg(n)))

Coda prioritaria di indici

Non si inseriscono in coda gli item ma coppie (indice, priorità), quindi si adotta la versione di «chiave affiancata al dato» (la priorità è un parametro aggiuntivo) invece che «chiave parte del dato»

- Il vettore $pq \rightarrow qp$ (posizione in coda) serve per implementare Pqchange efficiente, identificando la posizione dell'elemento nello heap con costo $O(1)$ (l'elemento è un indice) senza bisogno di una scansione lineare.

PQ.h

```
typedef struct pqueue *PQ;

PQ      PQinit(int maxN);
void    PQfree(PQ pq);
int     PQempty(PQ pq);
int     PQsize(PQ pq);
void    PQinsert(PQ pq, int index, int prio);
int     PQshowMax(PQ pq);
int     PQextractMax(PQ pq);
void    PQdisplay(PQ pq);
void    PQchange(PQ pq, int index, int prio);
```

PQ.h

```
typedef struct pqueue *PQ;
```

```
PQ      PQinit(int maxN);  
void    PQfree(PQ pq);  
int     PQempty(PQ pq);  
int     PQsize(PQ pq);  
void    PQinsert(PQ pq, int index, int prio);  
int     PQshowMax(PQ pq);  
int     PQextractMax(PQ pq);  
void    PQdisplay(PQ pq);  
void    PQchange(PQ pq, int index, int prio);
```

Scompare il tipo Item
Si gestiscono indici e priorità

PQ.c

```
...
typedef struct {int index; int prio} heapItem;
struct pqueue {heapItem *A; int heapsize; int *qp};

PQ PQinit(int maxN) {
    int i;
    PQ pq = malloc(sizeof(*pq));
    pq->A = malloc(maxN*sizeof(heapItem));
    pq->qp = malloc(maxN*sizeof(int));
    for (i=0; i < maxN; i++){
        pq->A[i].index = -1; pq->qp[i] = -1;
    }
    pq->heapsize = 0;
    return pq;
}
```

PQ.c

```
...
typedef struct {int index; int prio} heapItem;
struct pqueue {heapItem *A; int heapsize; int *qp};

PQ PQinit(int maxN) {
    int i;
    PQ pq = malloc(sizeof(*pq));
    pq->A = malloc(maxN*sizeof(heapItem));
    pq->qp = malloc(maxN*sizeof(int));
    for (i=0; i < maxN; i++){
        pq->A[i].index = -1; pq->qp[i] = -1;
    }
    pq->heapsize = 0;
    return pq;
}
```

Item interno al modulo PQ
per gestire la coppia
(indice,priorità)

PQ.c

```
...
typedef struct {int index; int prio} heapItem;
struct pqueue {heapItem *A; int heapsize; int *qp};

PQ PQinit(int maxN) {
    int i;
    PQ pq = malloc(sizeof(*pq));
    pq->A = malloc(maxN*sizeof(heapItem));
    pq->qp = malloc(maxN*sizeof(int));
    for (i=0; i < maxN; i++){
        pq->A[i].index = -1; pq->qp[i] = -1;
    }
    pq->heapsize = 0;
    return pq;
}
```

Si assume che maxN sia,
oltre che il massimo numero
di dati in coda,
il limite superiore agli indici

```
void PQfree(PQ pq) {  
    free(pq->qp);  
    free(pq->A);  
    free(pq);  
}  
  
int PQempty(PQ pq){  
    return pq->heapsize == 0;  
}  
  
int PQsize(PQ pq) {  
    return pq->heapsize;  
}
```

```
void PQinsert (PQ pq, int index, int prio){  
    int i, j;  
    i=pq->heapsize++;  
    while((i>=1) &&  
          (pq->A[PARENT(i)].prio)<prio)){  
        pq->A[i] = pq->A[PARENT(i)];  
        pq->qp[pq->A[i].index] = i;  
        i = PARENT(i);  
    }  
    pq->A[i].index = index;  
    pq->A[i].prio = prio;  
    pq->qp[index] = i;  
}
```



aggiorno pq->qp

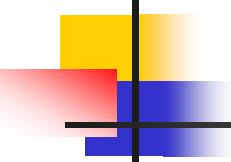
```
static void Swap(PQ pq, int pos1, int pos2){  
    heapItem temp;  
    int index1, index2;  
    temp = pq->A[pos1];  
    pq->A[pos1] = pq->A[pos2];  
    pq->A[pos2] = temp;  
    // update correspondence index-pos  
    index1 = pq->A[pos1].index;  
    index2 = pq->A[pos2].index;  
    pq->qp[index1] = pos1;  
    pq->qp[index2] = pos2;  
}
```

```
static void HEAPify(PQ pq, int i) {
    int l, r, largest;
    l = LEFT(i);
    r = RIGHT(i);
    if (l < pq->heapsize && (pq->A[l].prio > pq->A[i].prio))
        largest = l;
    else
        largest = i;
    if (r < pq->heapsize && (pq->A[r].prio > pq->A[largest].prio))
        largest = r;
    if (largest != i) {
        Swap(pq, i, largest);
        HEAPify(pq, largest);
    }
}
```

```
int PQextractMax(PQ pq) {
    int res;
    int j=0;

    Swap (pq, 0, pq->heapsize-1);
    res = pq->A[pq->heapsize-1].index;
    pq->qp[res]=-1;
    pq->heapsize--;
    pq->A[pq->heapsize].index=-1; // redundant
    HEAPify(pq, 0);
    return res;
}
```

```
void PQchange (PQ pq, int index, int prio) {  
    int pos = pq->qp[index];  
    heapItem temp = pq->A[pos];  
    temp.prio = prio; // new prio  
  
    while ((pos>=1) && (pq->A[PARENT(pos)].index < prio) {  
        pq->A[pos] = pq->A[PARENT(pos)];  
        pq->qp[pq->A[pos].index] = pos;  
        pos = PARENT(pos);  
    }  
    pq->A[pos] = temp;  
    pq->qp[index] = pos;  
  
    HEAPify(pq, pos);  
}
```

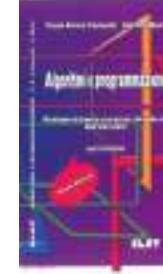


Riferimenti

- Heap:
 - Cormen 7.2, 7.3
 - Sedgewick 9.2, 9.3
- Heapsort:
 - Cormen 7.4
 - Sedgewick 9.4
- Code a priorità:
 - Cormen 7.5
 - Sedgewick 9.1, 9.6

Esercizi di teoria

- 7. Code a priorità e heap
 - 7.1 Heap
 - 7.2 Heap Sort
 - 7.3 Code a priorità



Gli Alberi Binari di Ricerca (Binary Search Trees, BST)

Paolo Camurati e Gianpiero Cabodi

Alberi Binari

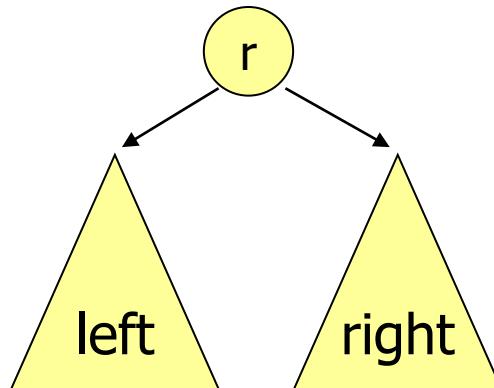
DEFINIZIONI

Alberi binari

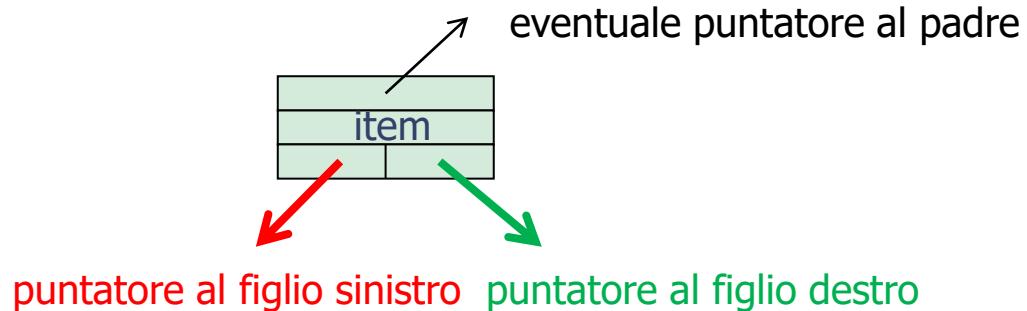
Definizione ricorsiva:

Un albero binario T è:

- un insieme vuoto di nodi vuoto
- una terna formata da una radice, un sottoalbero sinistro e un sottoalbero destro.



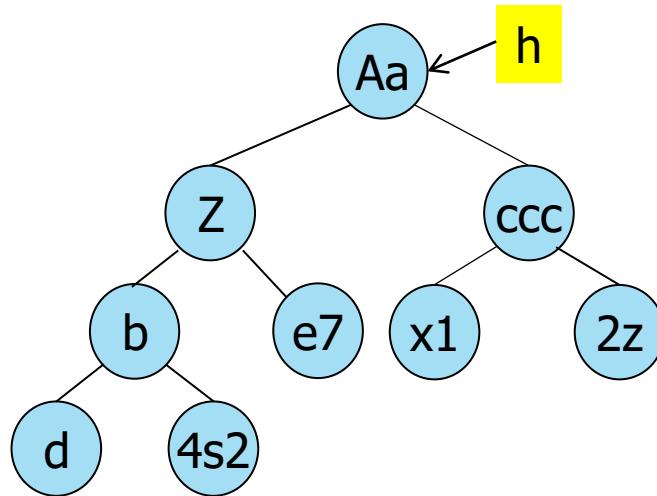
Nodo:



```
typedef struct node* link;
struct node {
    Item item;
    link left;
    link right;
};
```

Accesso

All'albero si accede tramite il puntatore h alla radice:



Calcolo di parametri

numero di nodi

```
int count(link root) {  
    if (root == NULL)  
        return 0;  
    return count(root->left) + count(root->right) + 1;  
}
```

```
int height(link root) {  
    int u, v;  
    if (root == NULL)  
        return -1;  
    u = height(root->left); v = height(root->right);  
    if (u>v)  
        return u+1;  
    return v+1;  
}
```

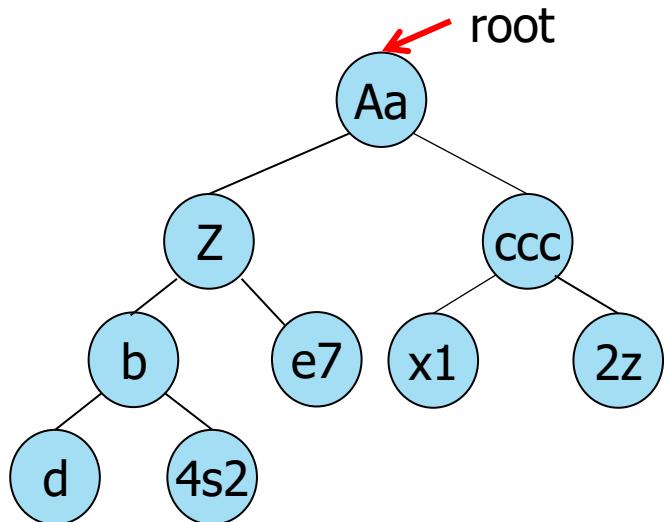
altezza

Visite

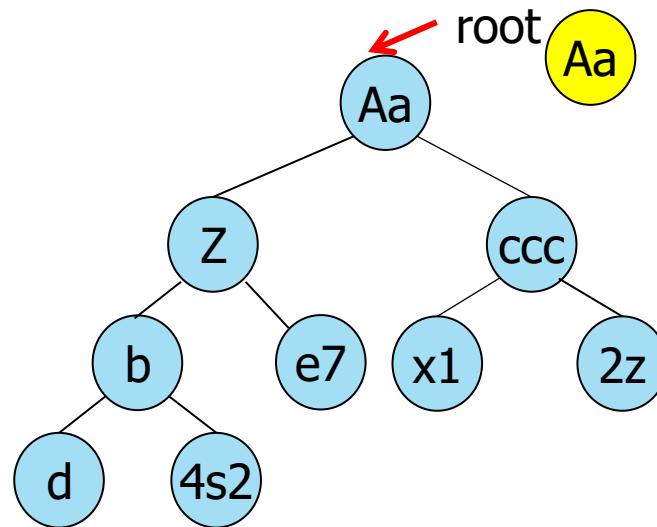
Attraversamento o visita: elenco dei nodi secondo una strategia:

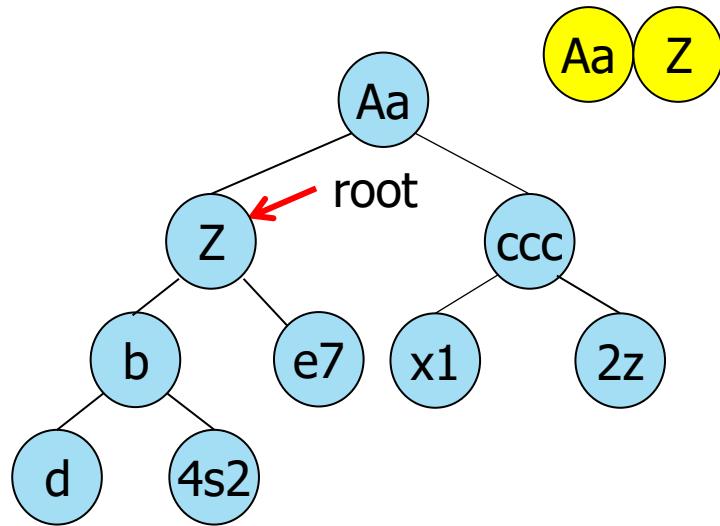
- **pre-ordine**: root, Left(root), Right(root)
- **in-ordine**: Left(root), root, Right(root)
- **post-ordine**: Left(root), Right(root), root

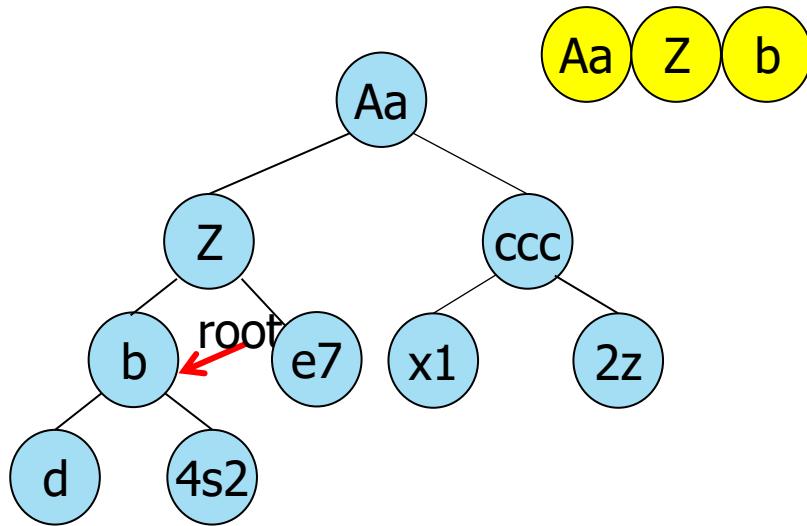
Pre-ordine

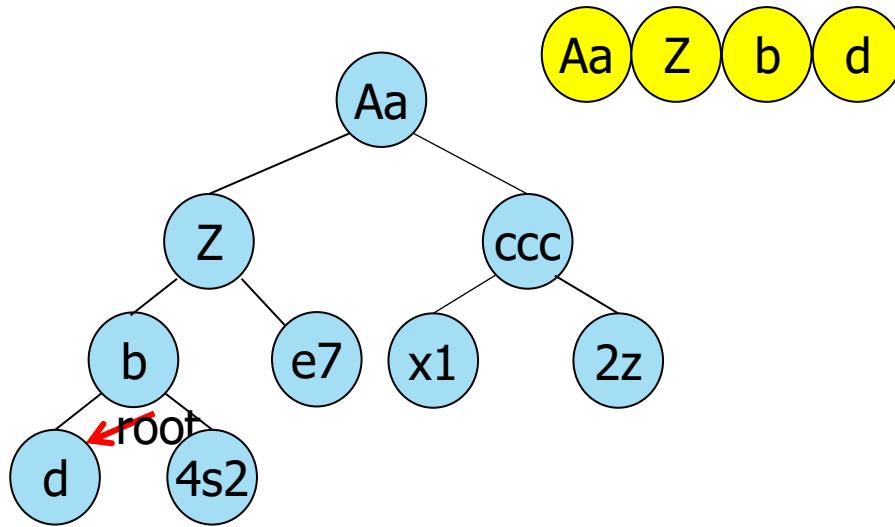


```
void preOrder(link root){  
    if (root == NULL)  
        return;  
    printf("%s ",root->name);  
    preOrder(root->left);  
    preOrder(root->right);  
}
```

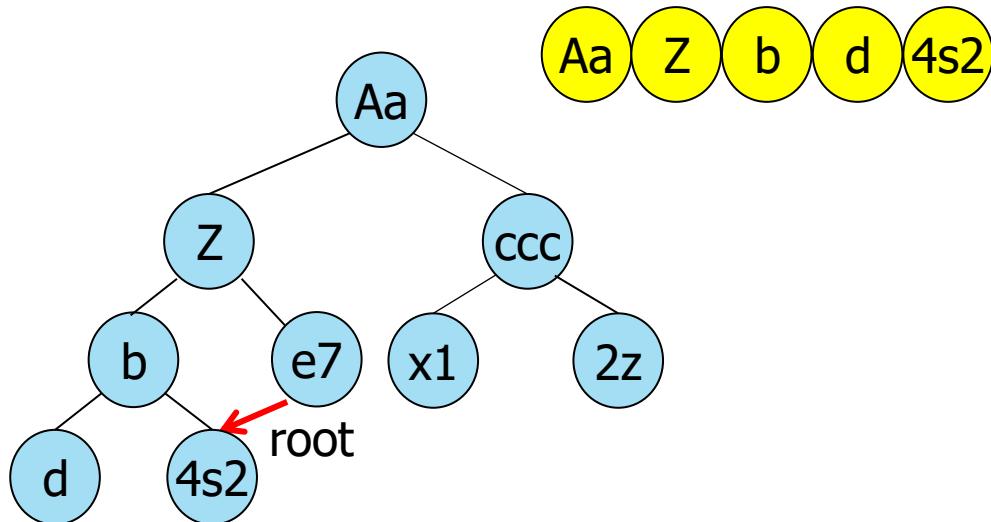


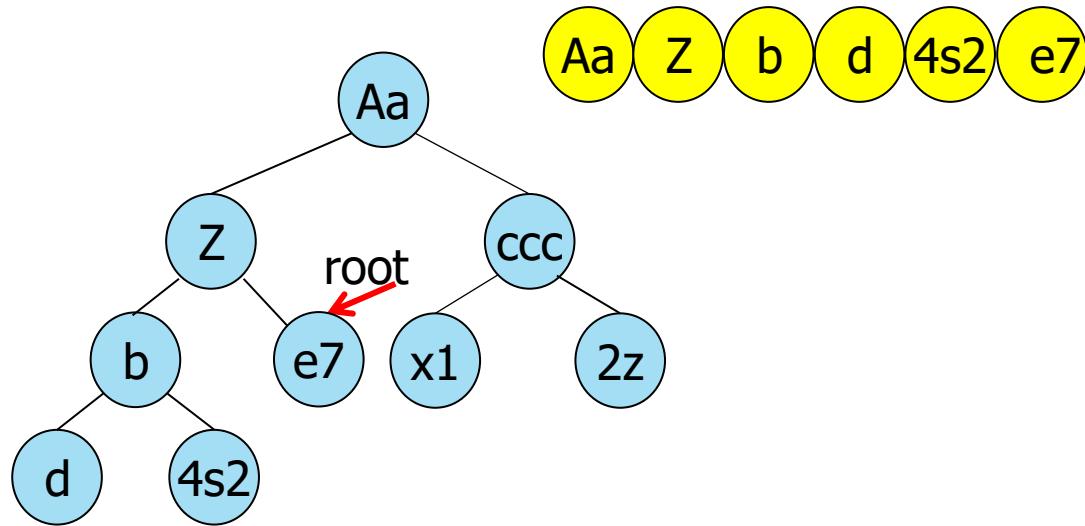


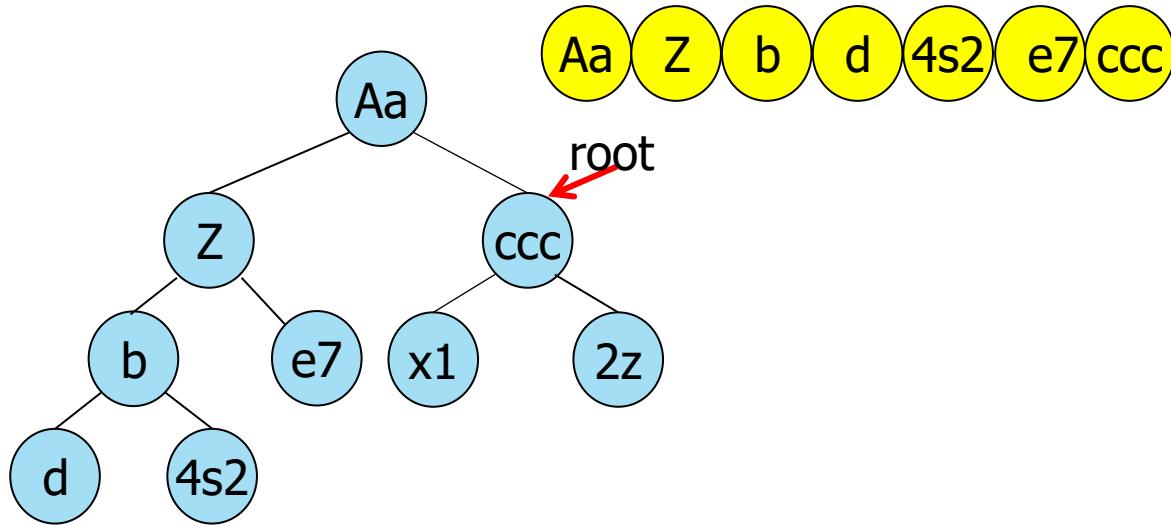


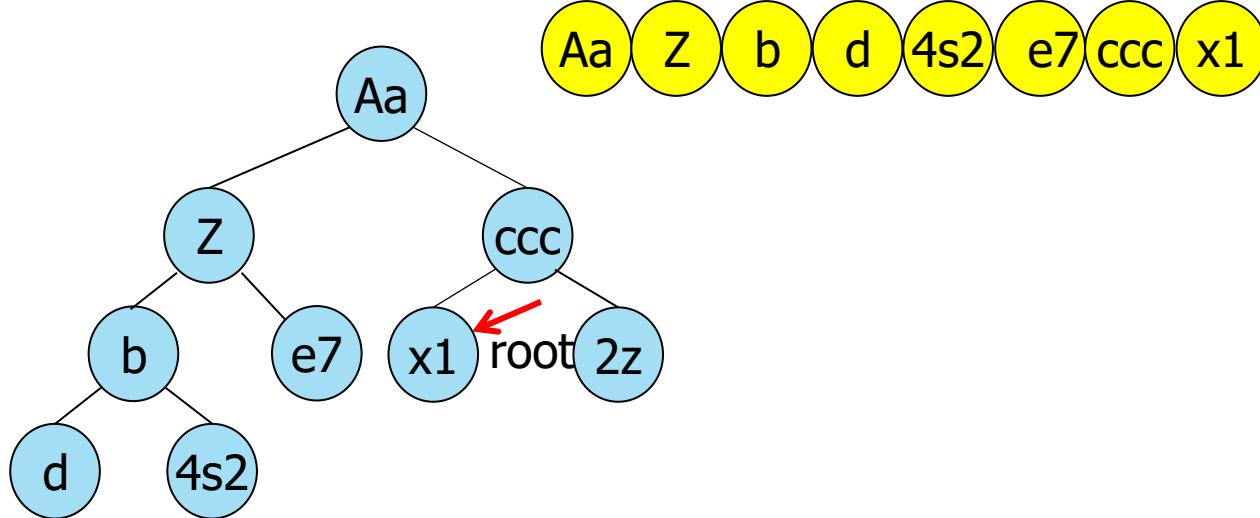


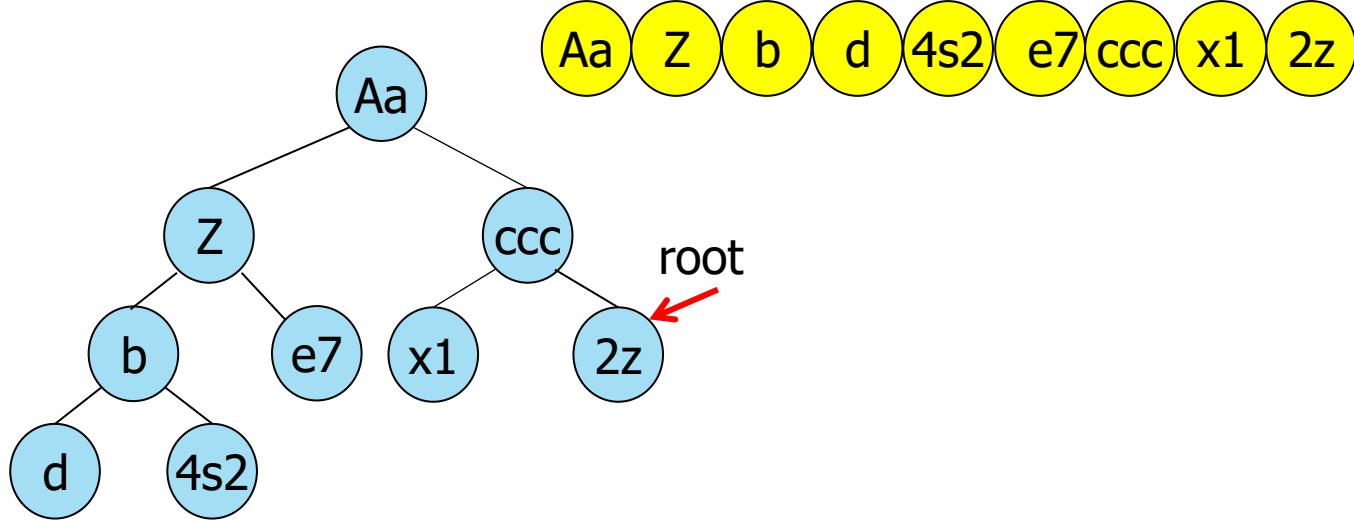
Aa Z b d



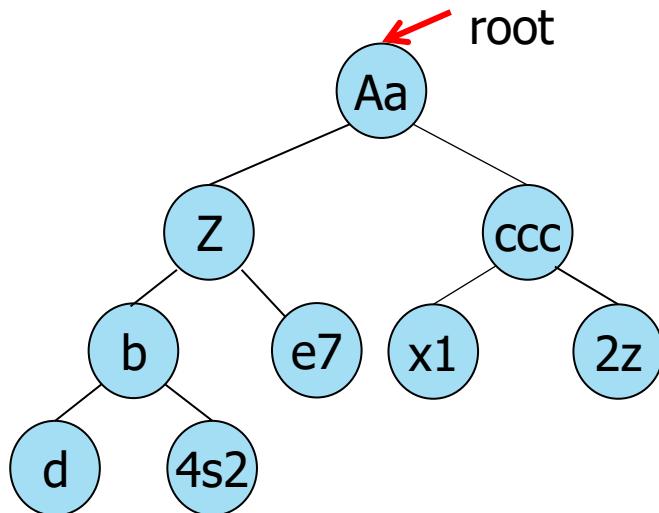




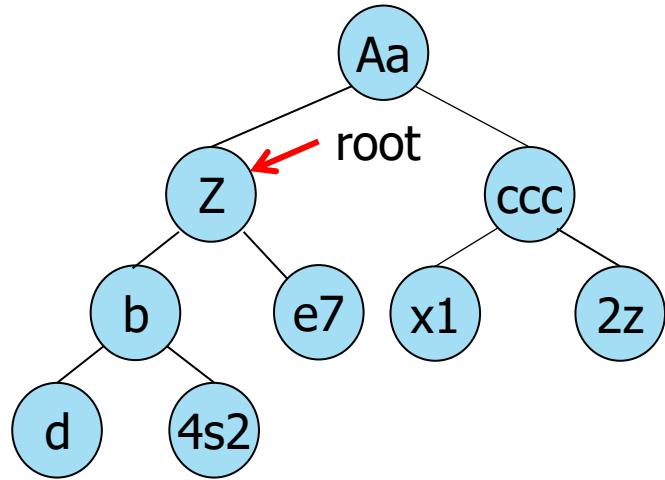


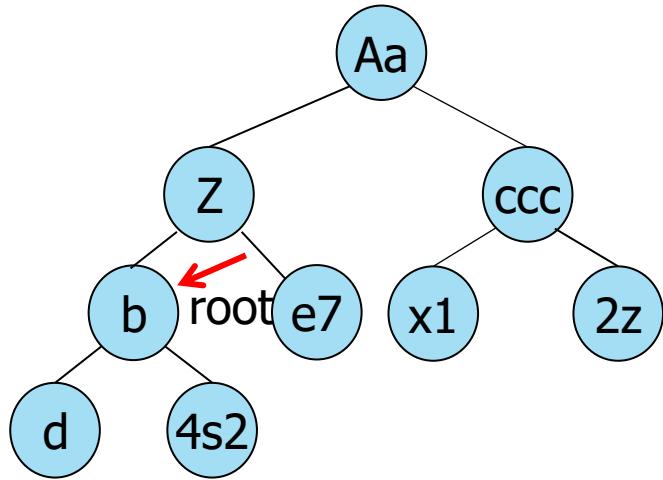


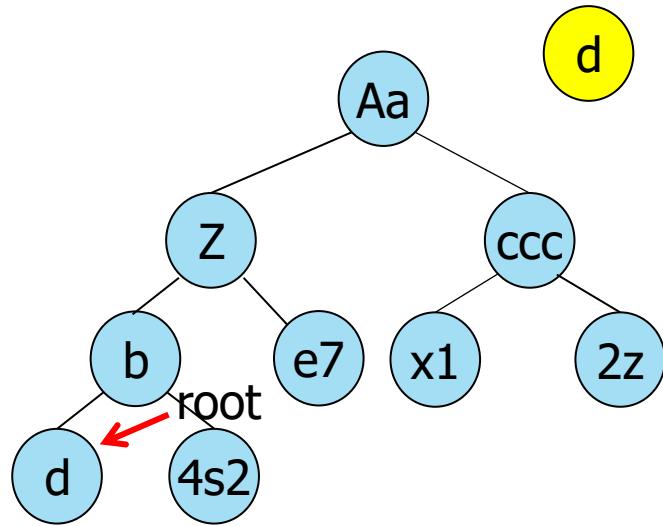
In-ordine

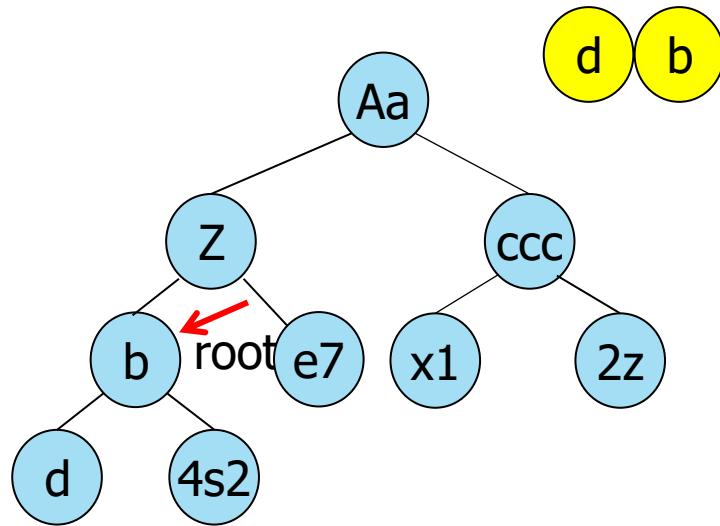


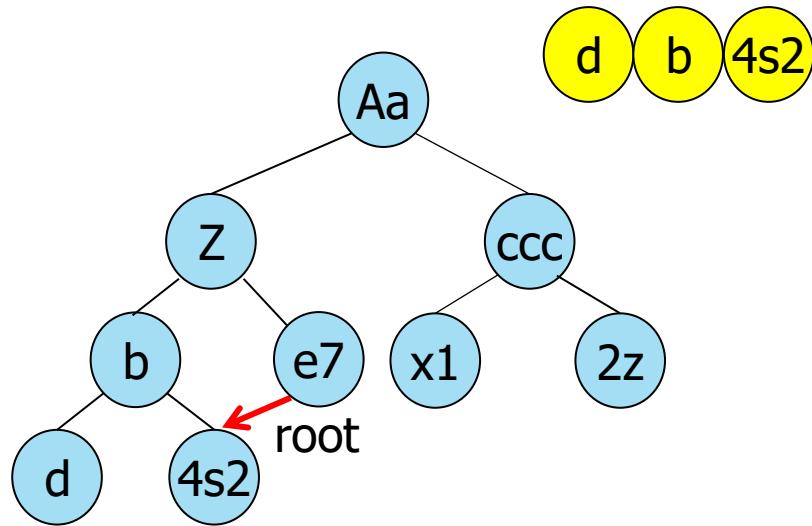
```
void inorder(link root){  
    if (root == NULL)  
        return;  
    inorder(root->left);  
    printf("%s ",root->name);  
    inorder(root->right);  
}
```



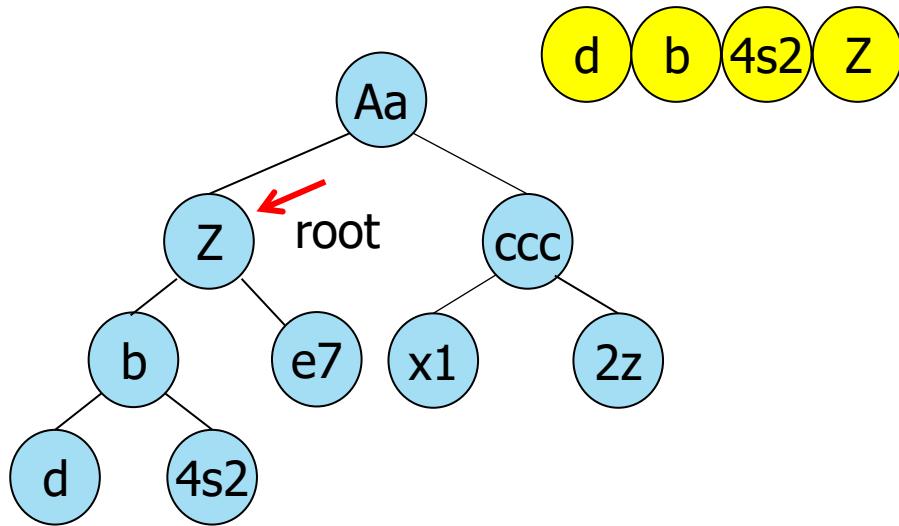




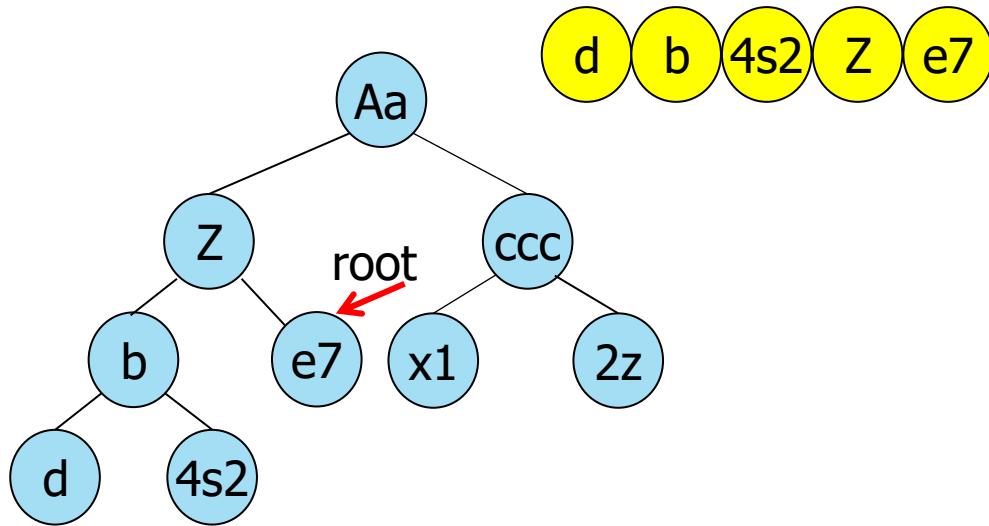


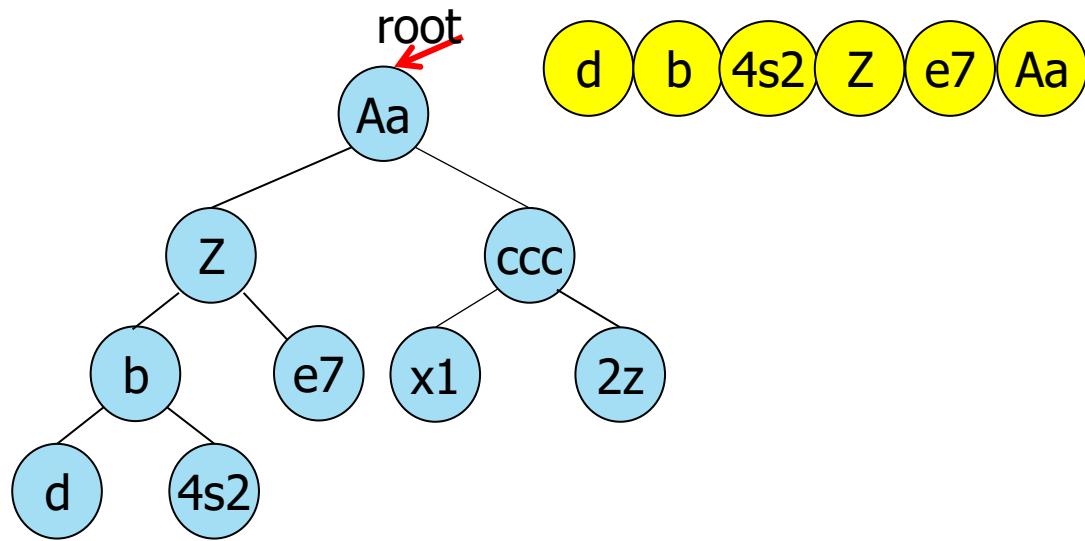


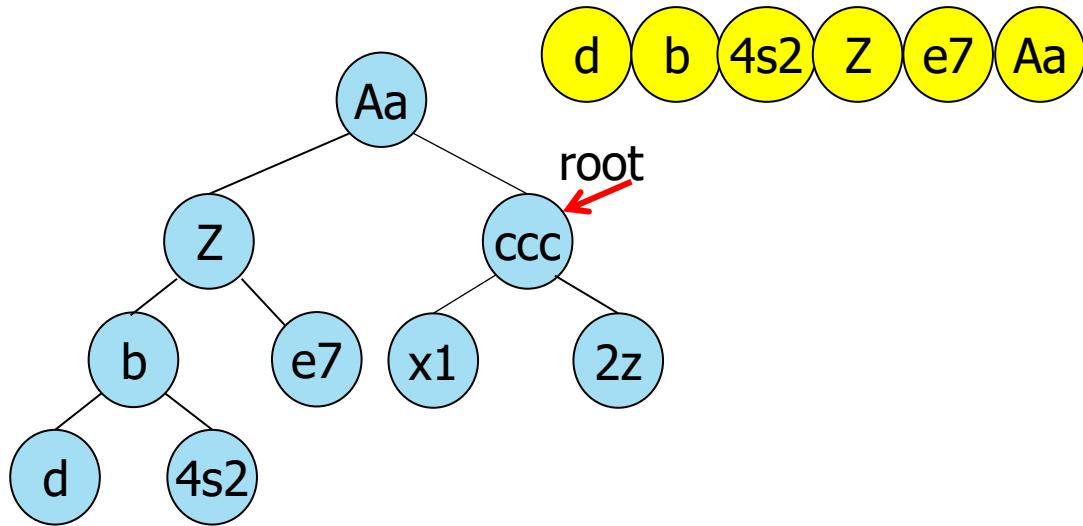
d b 4s2

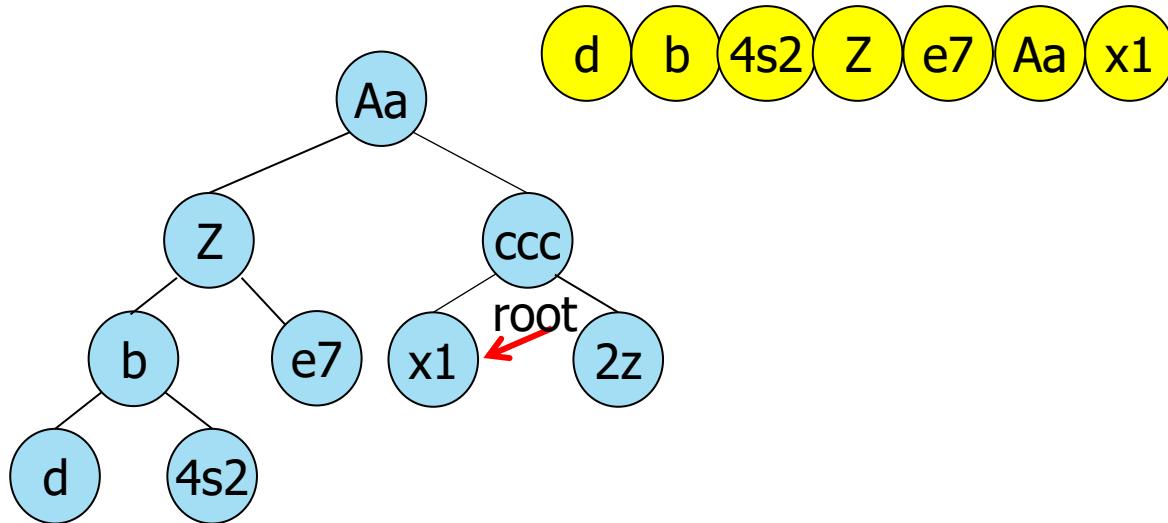


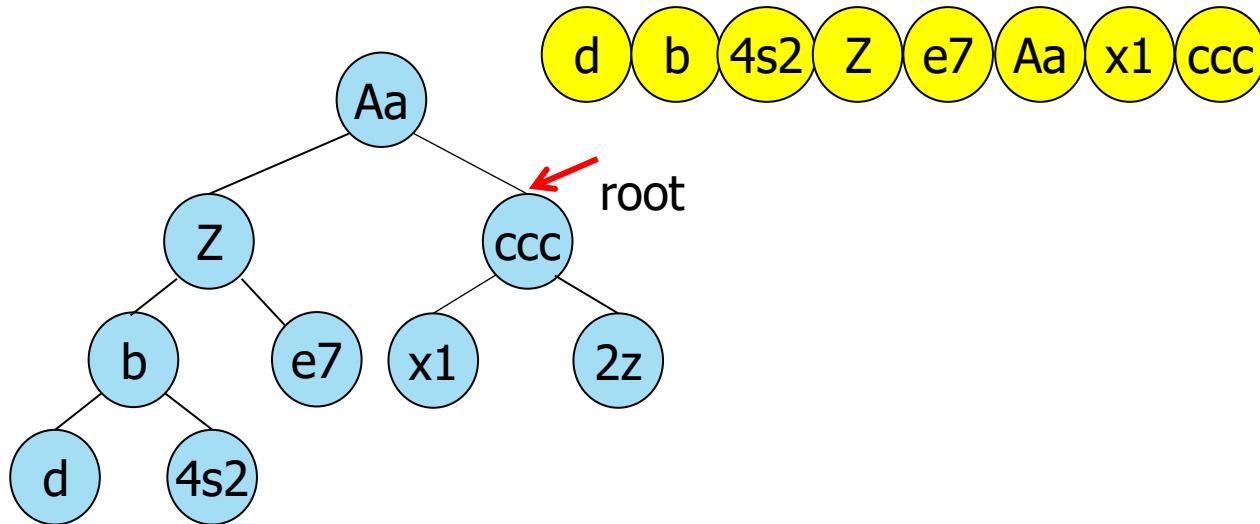
d b 4s2 z

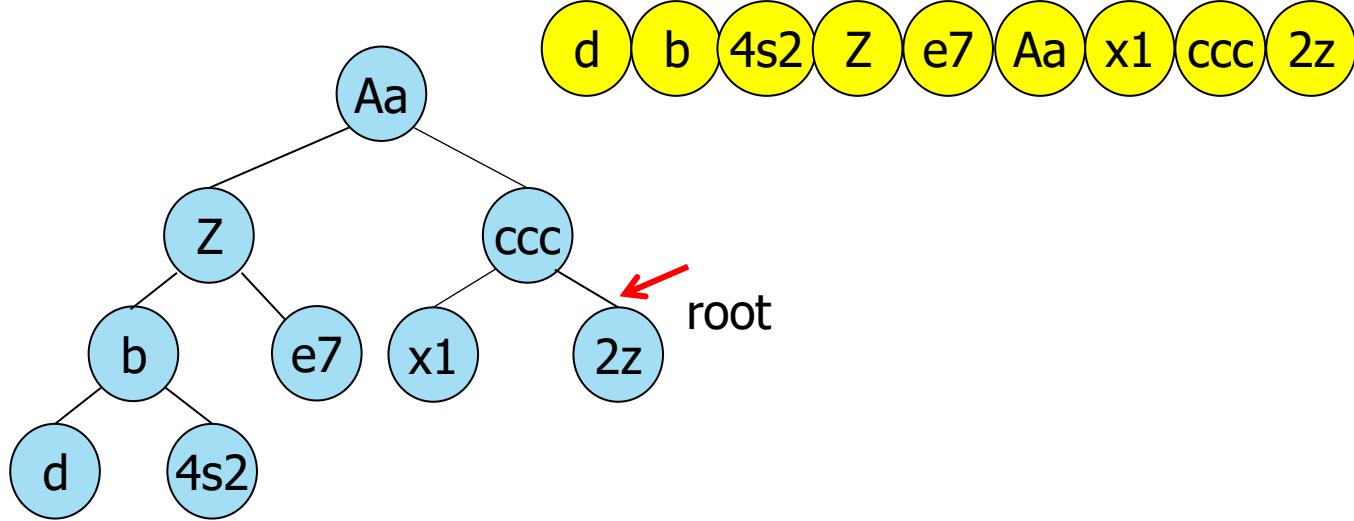




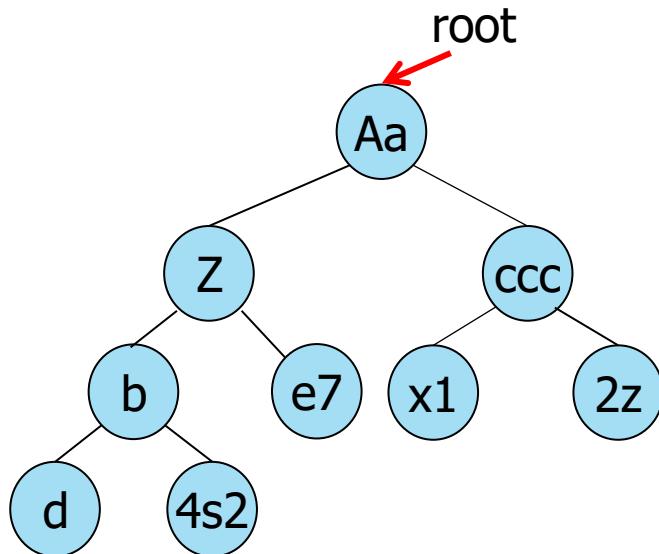




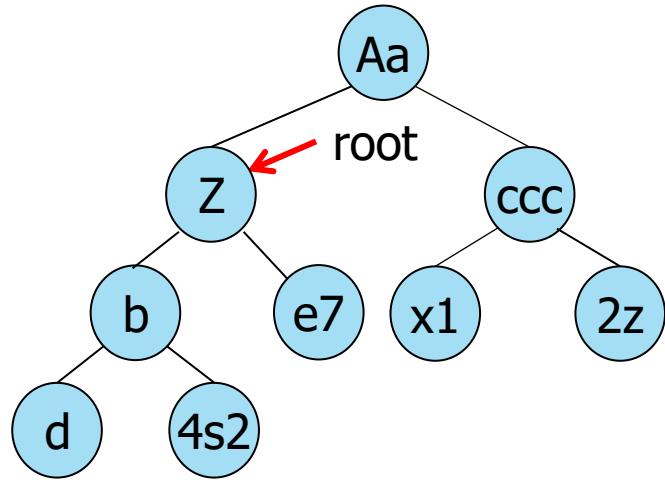


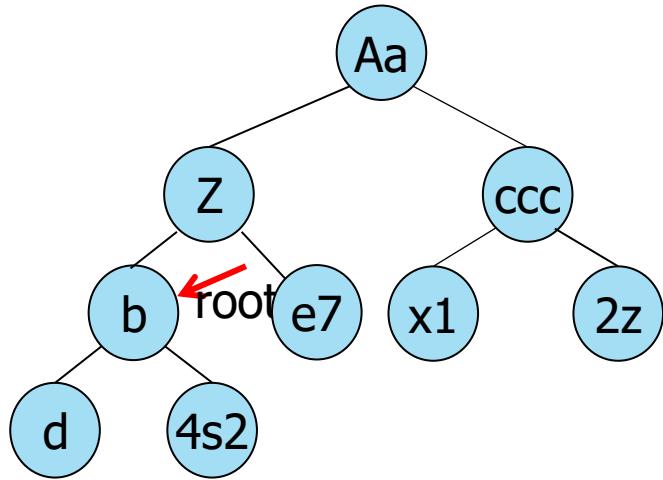


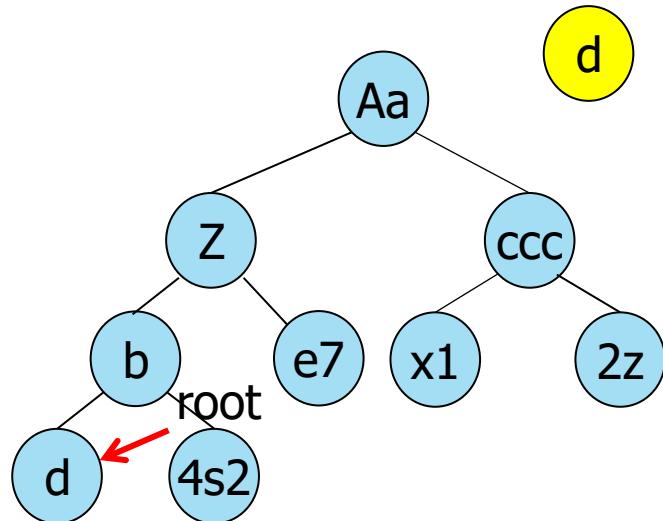
Post-ordine

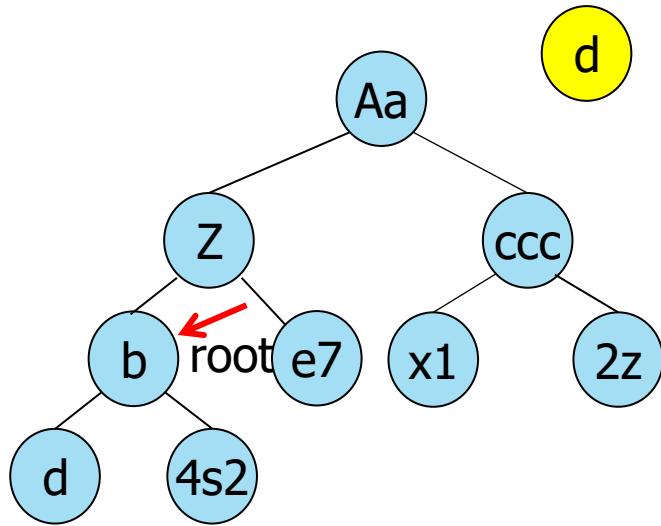


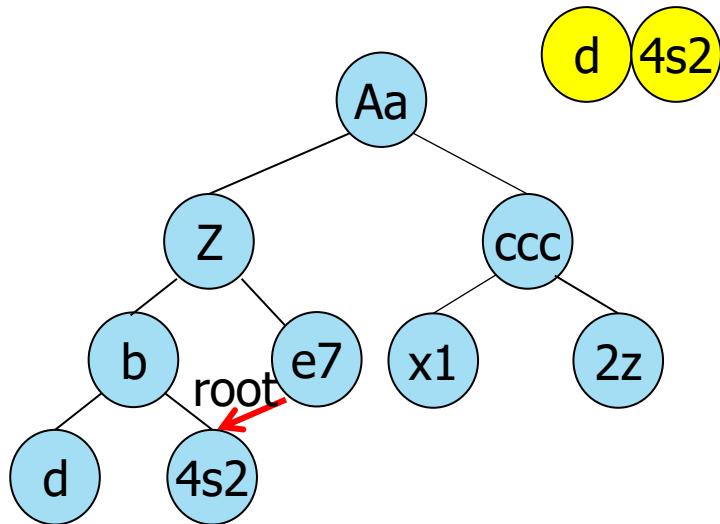
```
void postorder(link root){  
    if (root == NULL)  
        return;  
    postOrder(root->left);  
    postOrder(root->right);  
    printf("%s ", root->name);  
}
```

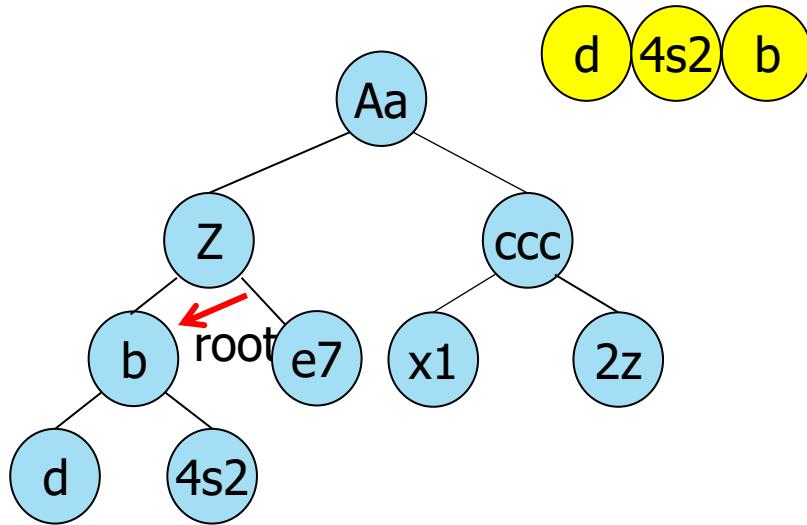


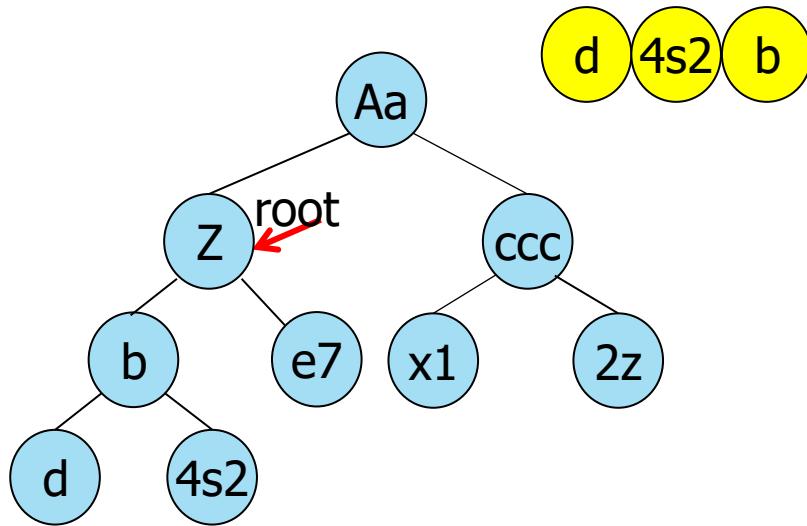


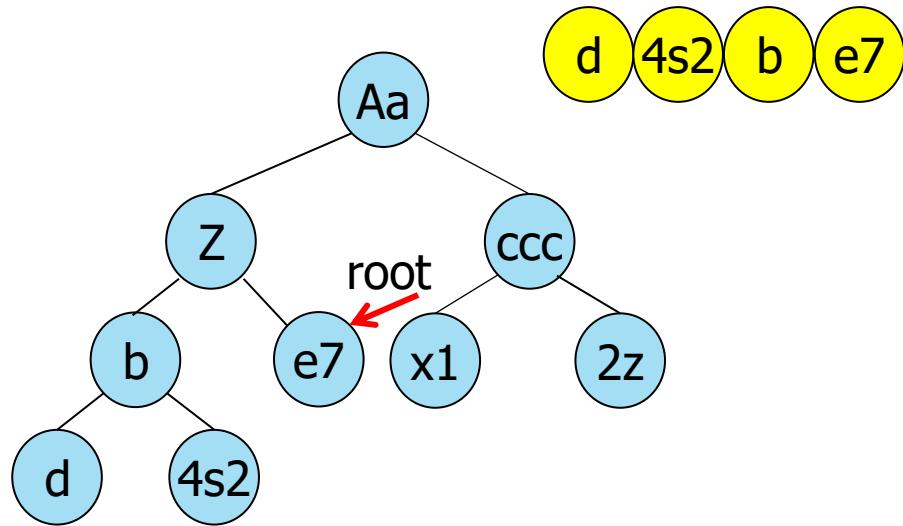


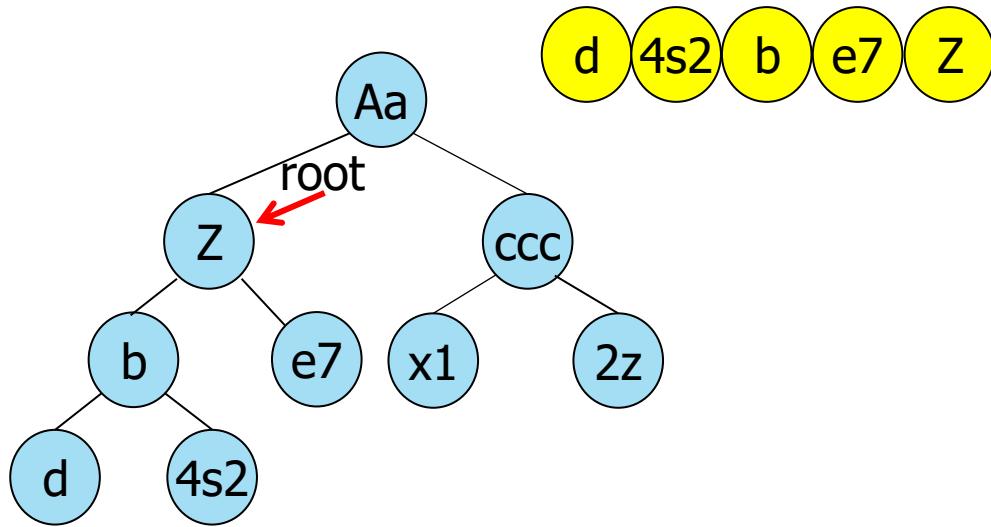


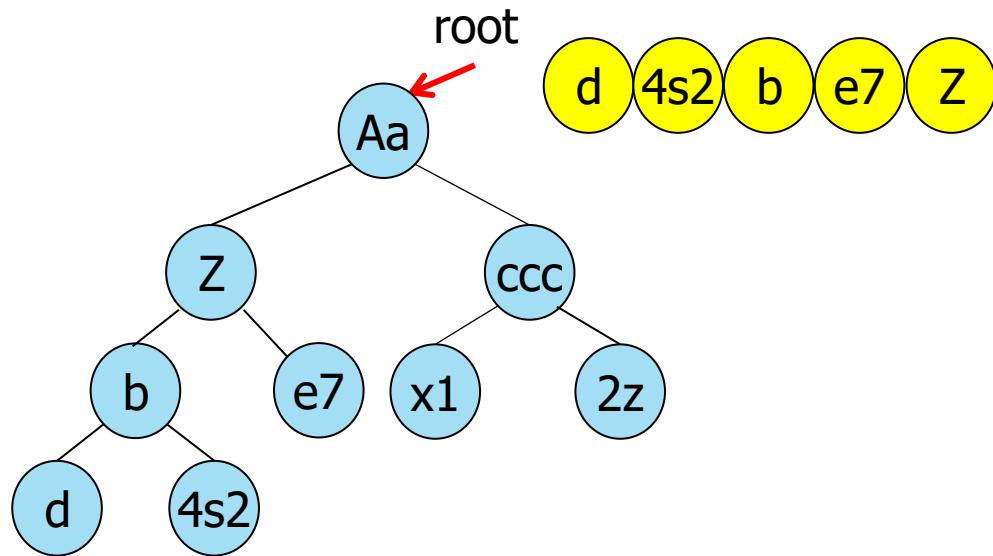


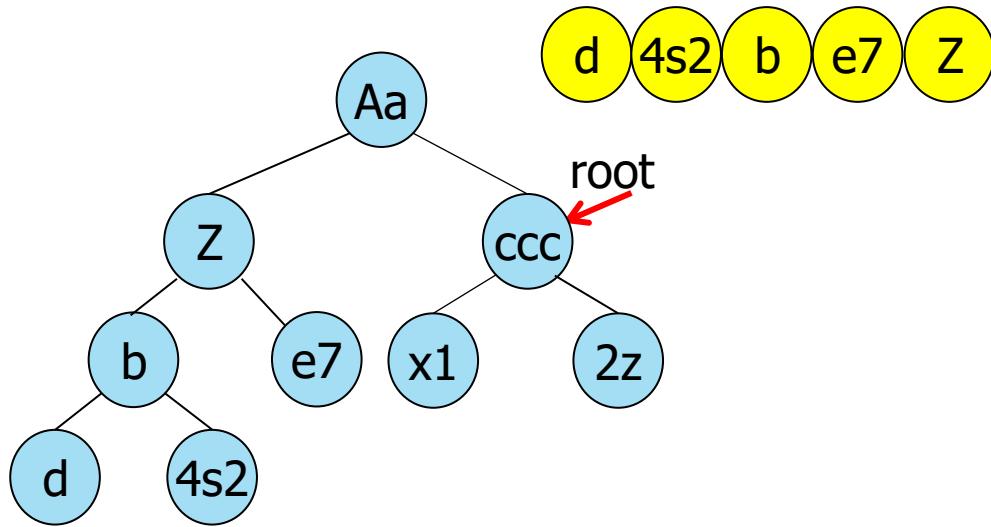






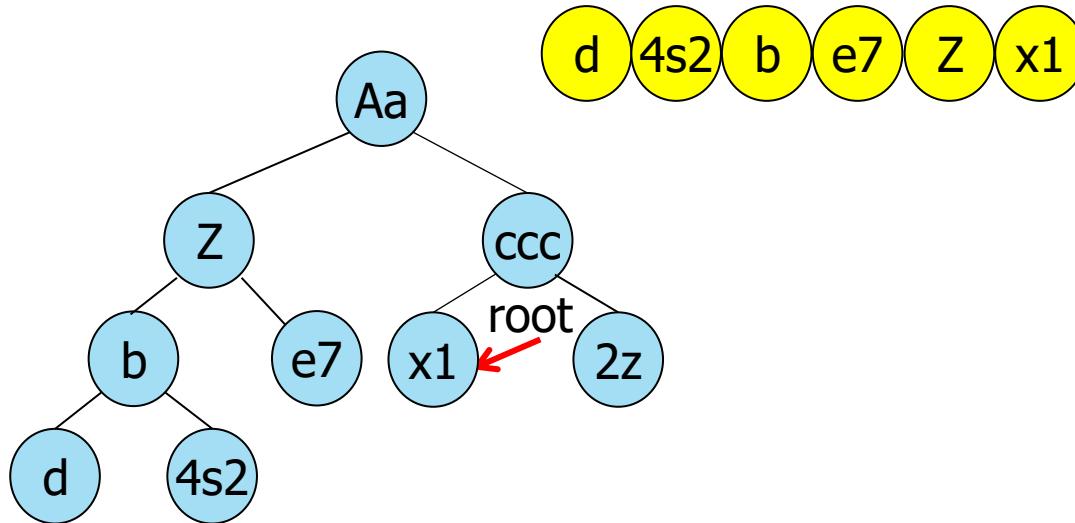


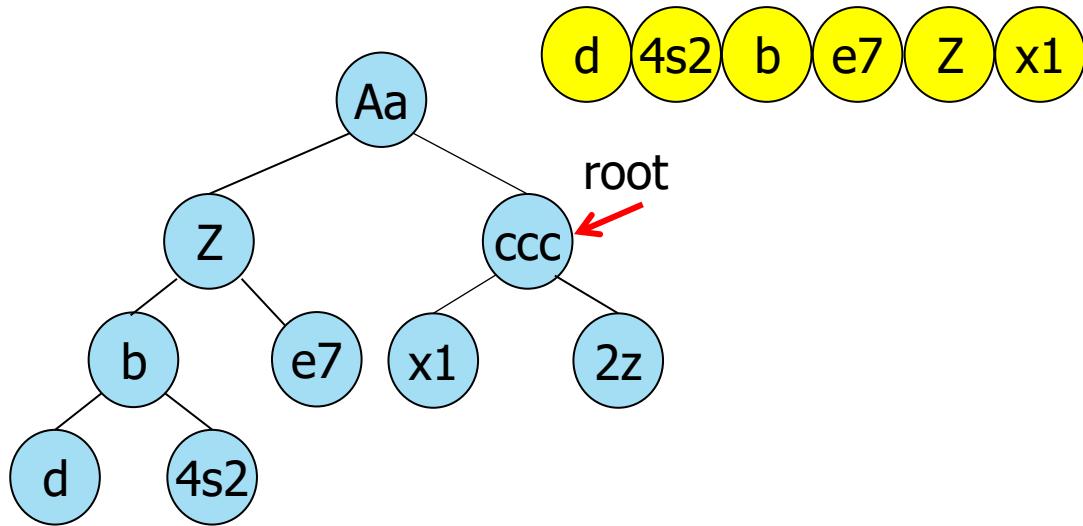


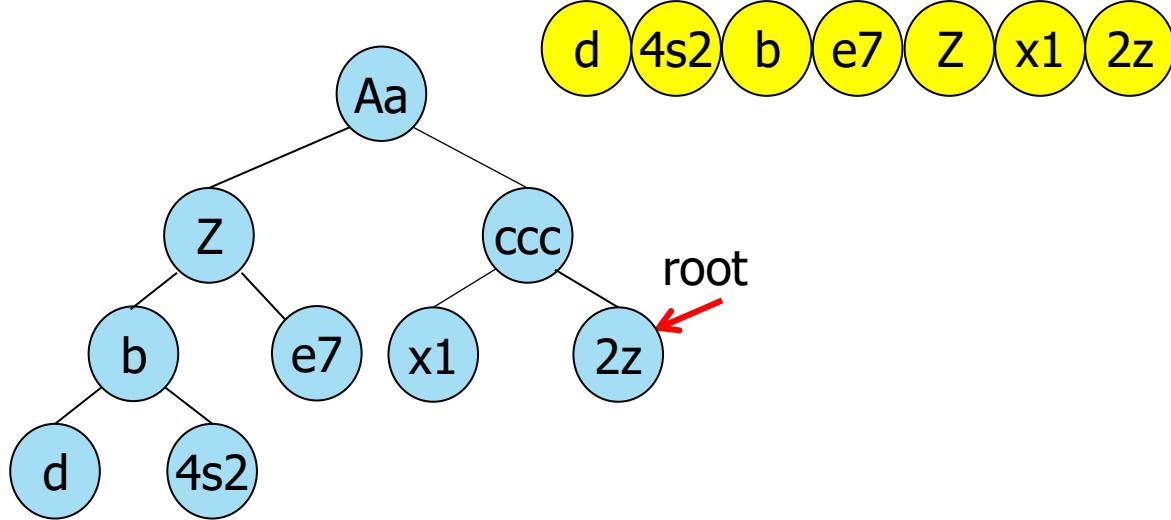


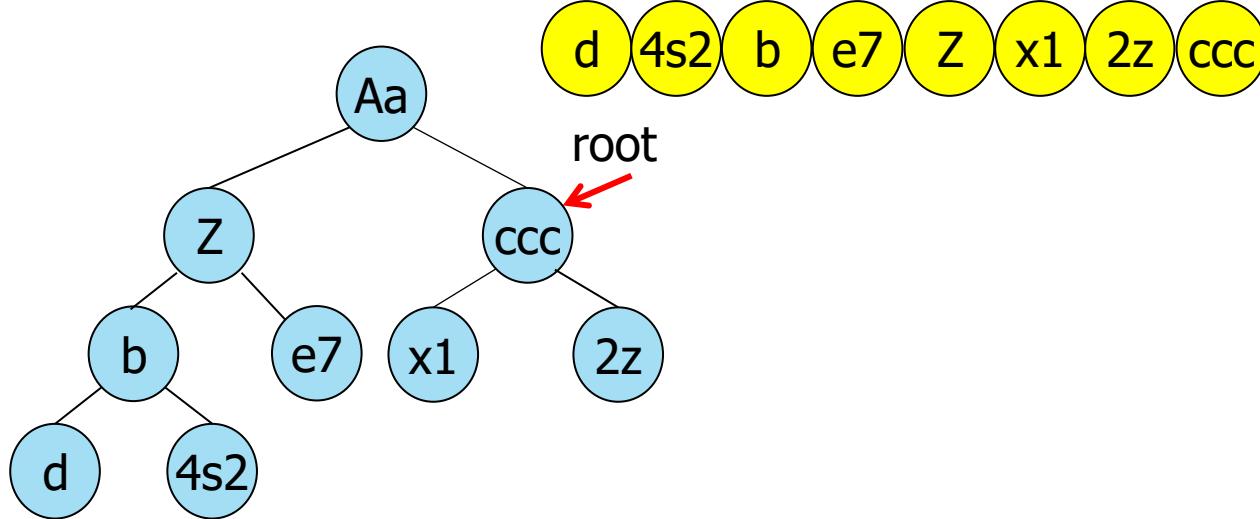
d 4s2 b e7 z

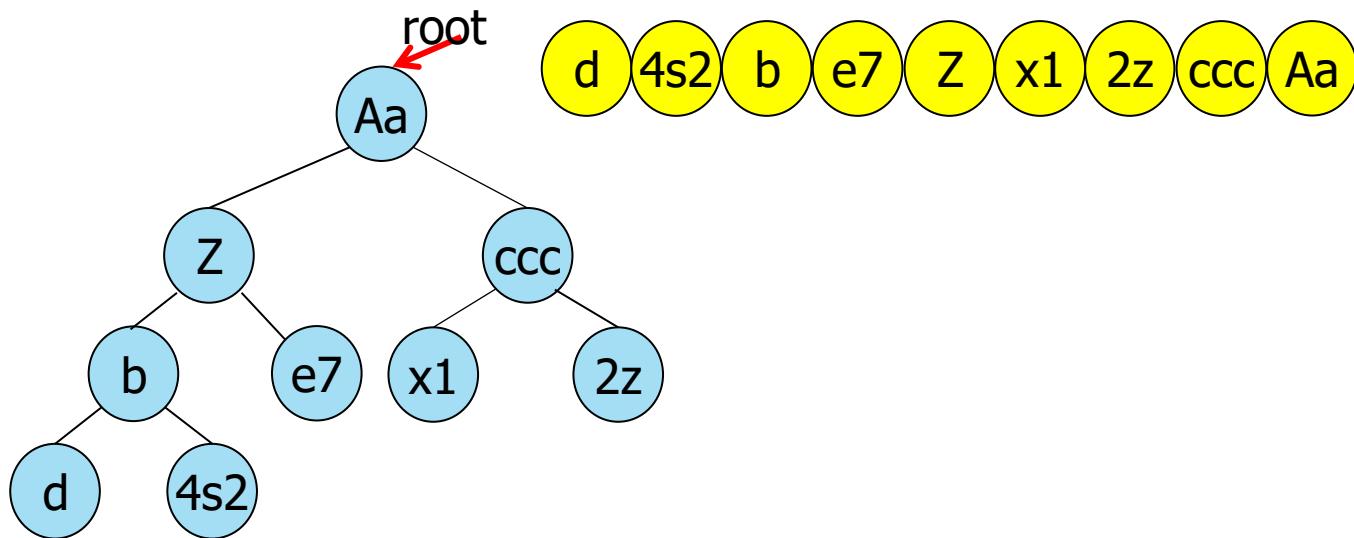
root











Analisi di complessità

Ipotesi 1: albero completo:

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 2$, $b = 2$ (sottoproblemi di dimensione $n-1$, approssimati conservativamente a n)

divide and conquer
 $a = 2$ $b = 2$

Equazione alla ricorrenze:

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

$$T(1) = 1$$

$$T(n) = O(n)$$

decrease and conquer
 $a = 1$ $k_i = 1$

Ipotesi 2: albero totalmente sbilanciato (degenerato in una lista) :

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 1$, $k_i = 1$

Equazione alla ricorrenze:

$$T(n) = 1 + T(n-1) \quad n > 1$$

$$T(1) = 1$$

$$T(n) = O(n)$$

Alberi Binari

ESEMPI DI USO

Alberi binari ed espressioni

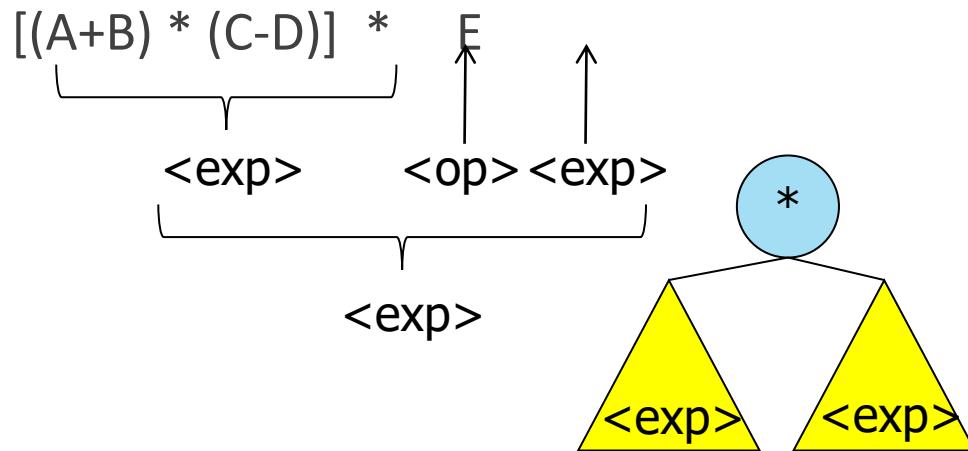
Data un'espressione algebrica in forma infissa (con parentesi per cambiare le precedenze tra operatori), ricostruirne l'albero binario in base alla grammatica semplificata:

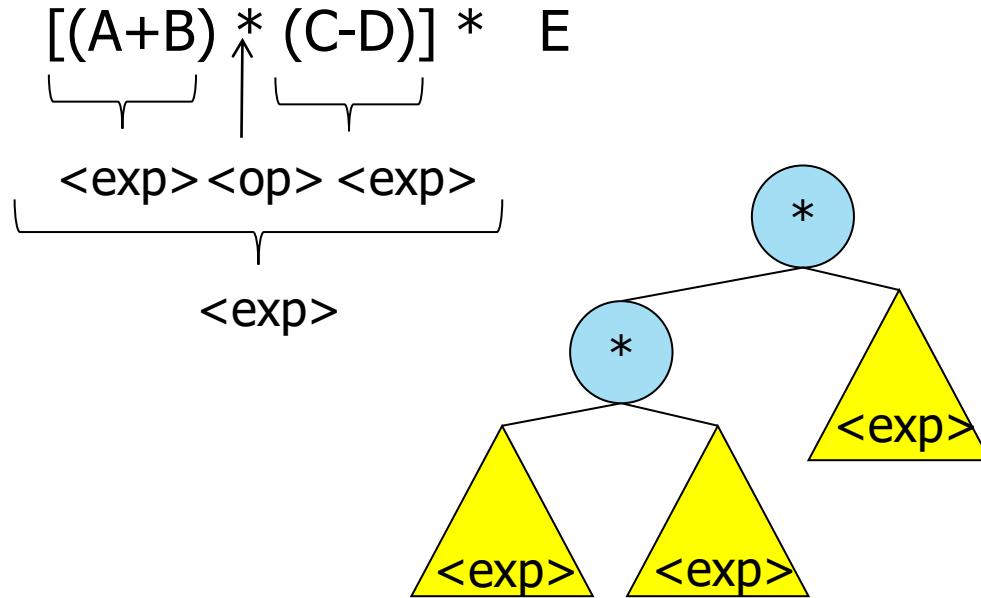
- $\langle \text{exp} \rangle = \langle \text{operand} \rangle \mid \langle \text{exp} \rangle \langle \text{op} \rangle \langle \text{exp} \rangle$
- $\langle \text{operand} \rangle = A \dots Z$
- $\langle \text{op} \rangle = + \mid * \mid - \mid /$

ricorsione

condizione di
terminazione

Dal parsing dell'espressione si ottiene

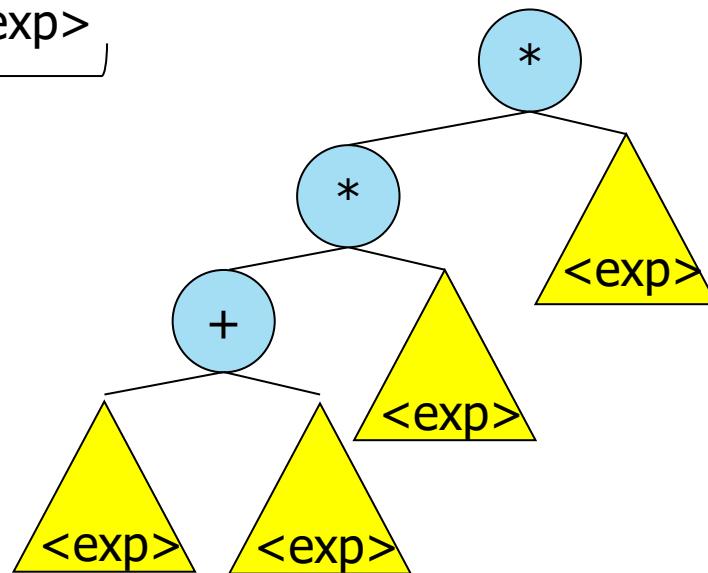


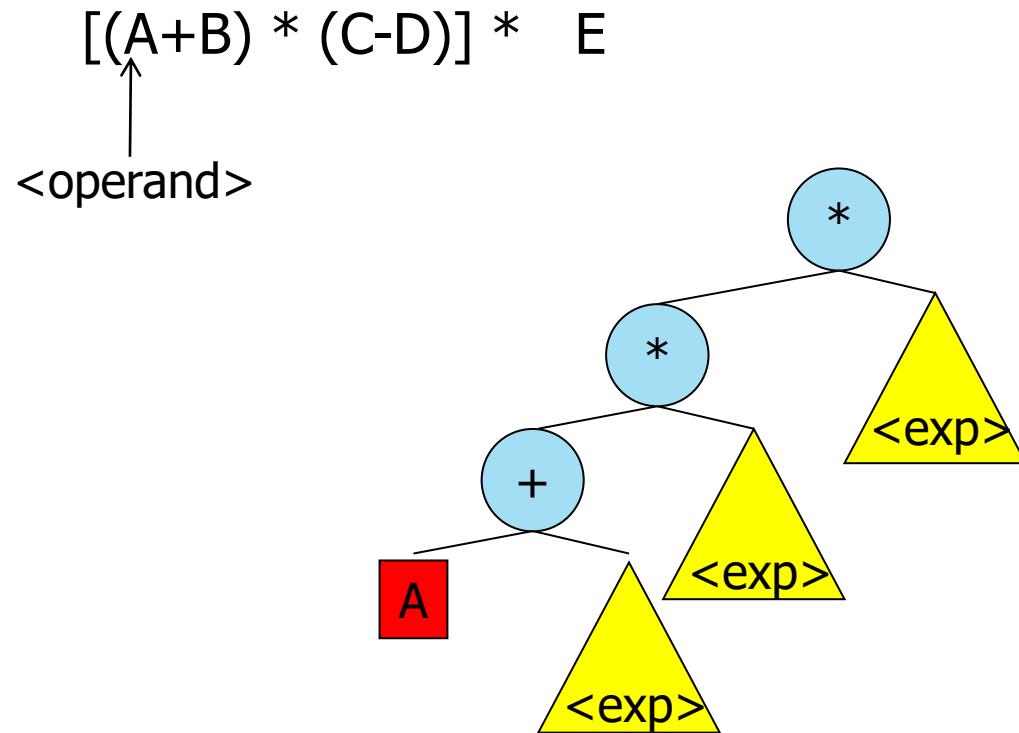


$$[(A+B) * (C-D)] * E$$

$\underbrace{<\text{exp}> <\text{op}> <\text{exp}>}_{<\text{exp}>}$

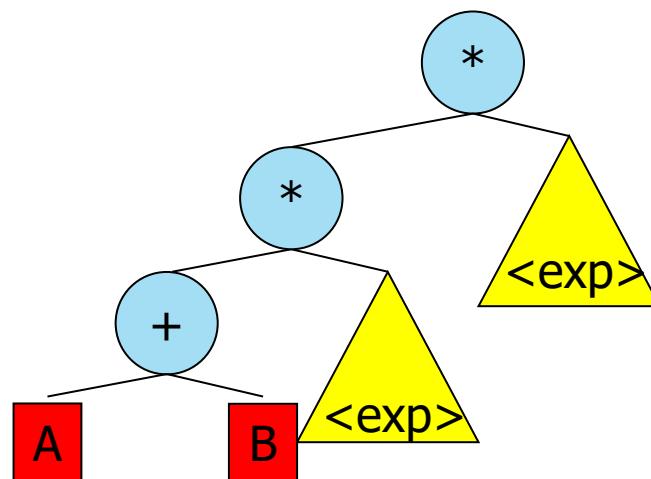
$<\text{exp}>$

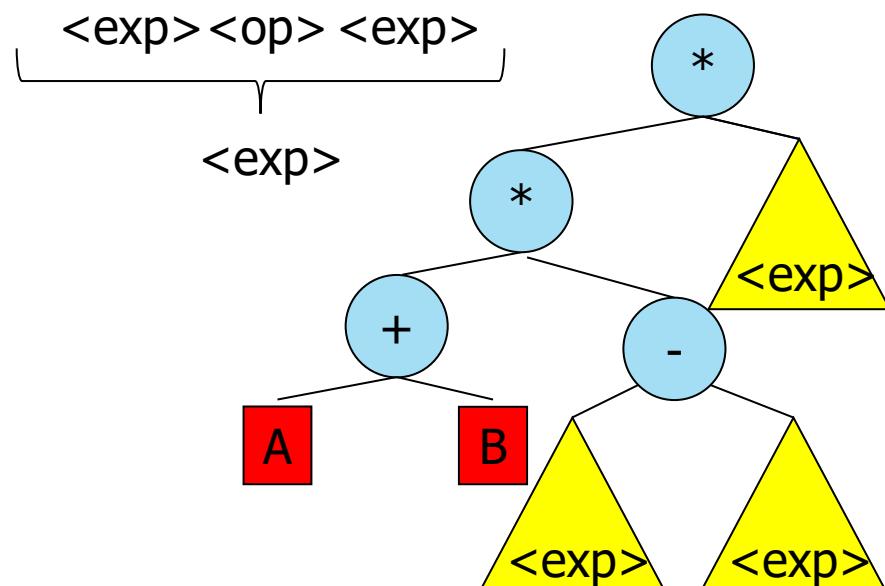


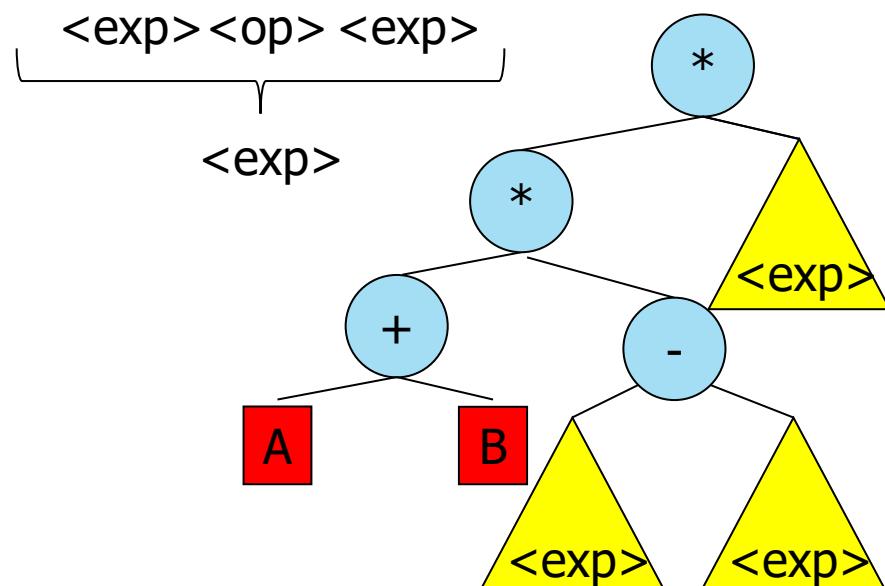


$$[(A+B) * (C-D)] * E$$

↑
<operand>

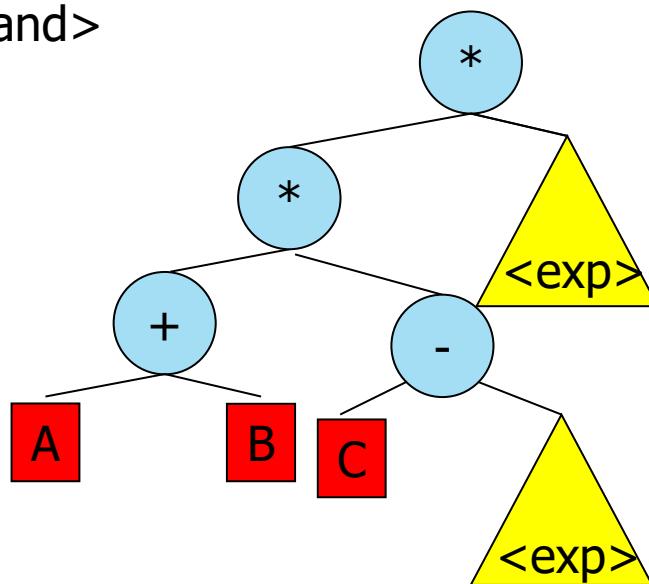


$$[(A+B) * (C-D)] * E$$


$$[(A+B) * (C-D)] * E$$


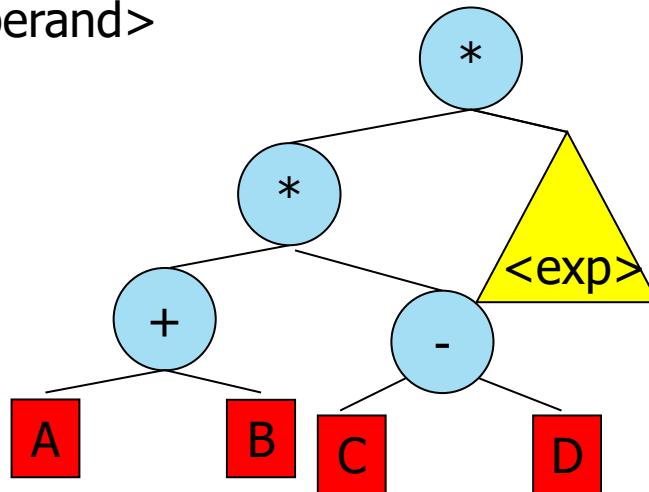
$$[(A+B) * (C-D)] * E$$

<operand>



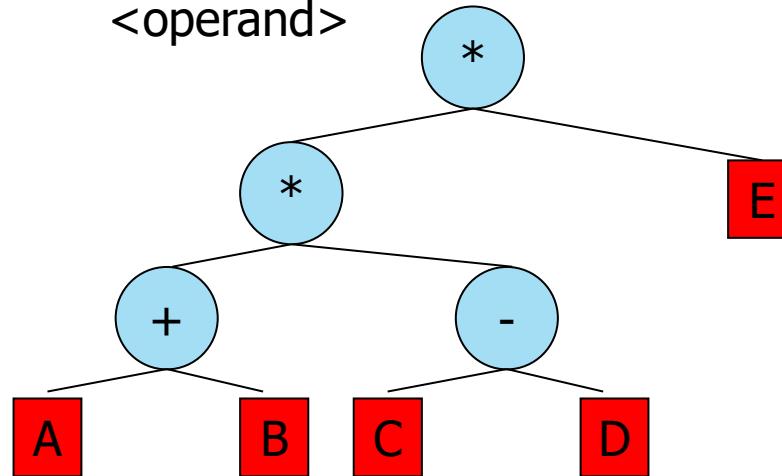
$$[(A+B) * (C-D)] * E$$

↑
<operand>



$$[(A+B) * (C-D)] * E$$

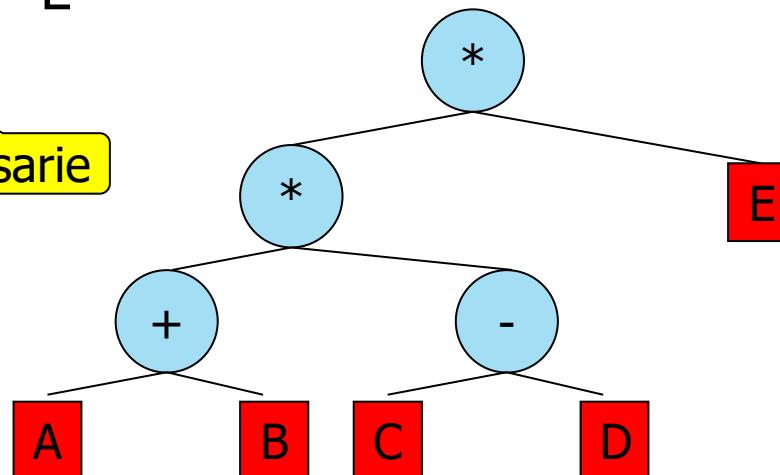
↑
<operand>



L'attraversamento in post-ordine dell'albero dà la forma postfissa (Notazione Polacca Inversa o Reverse Polish Notation) dell'espressione

A B + C D - * E *

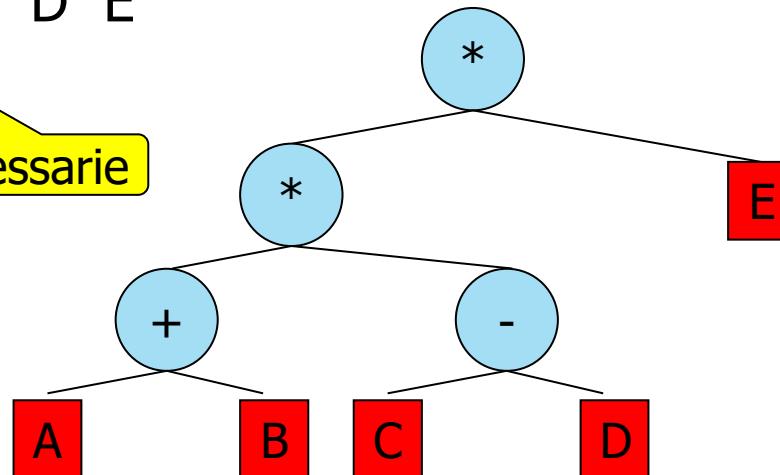
parentesi non più necessarie



L'attraversamento in pre-ordine dell'albero dà la forma prefissa (Notazione Polacca), poco usata in pratica, dell'espressione

* * + A B - C D E

parentesi non più necessarie



Valutazione di espressione in forma prefissa

Grammatica per espressioni in forma prefissa (Notazione Polacca) semplificate (solo operatori + e *, operandi interi positivi)

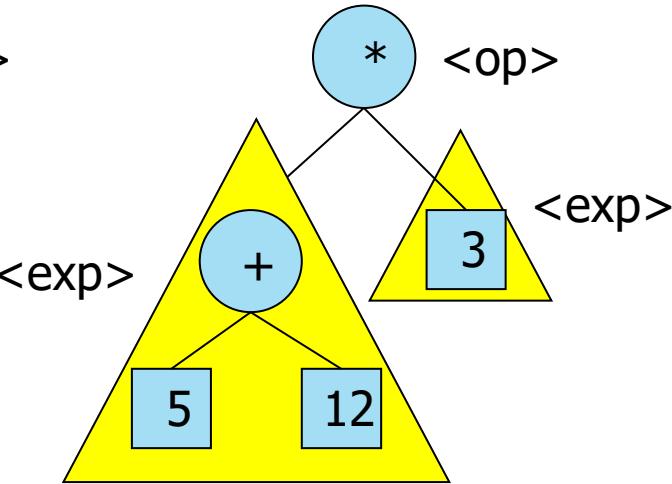
- $\langle \text{exp} \rangle = \langle \text{operand} \rangle \mid \langle \text{op} \rangle \langle \text{exp} \rangle \langle \text{exp} \rangle$
- $\langle \text{operand} \rangle = \text{digit} \mid \text{digit catenate operand}$
- $\langle \text{digit} \rangle = 0..9$
- $\langle \text{op} \rangle = + \mid *$

Esempio:

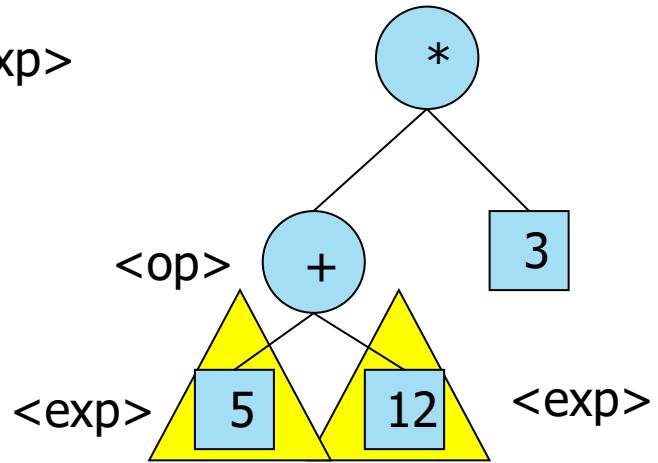
- in forma infissa $(5 + 12) * 3$
- in forma prefissa $* + 5 12 3$

$* + 5 \ 12 \ 3$

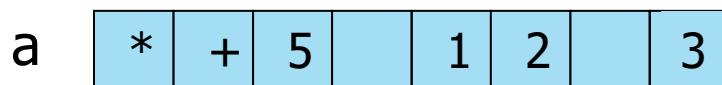
$\begin{array}{c} * \\ | \\ \text{} \end{array}$ $\begin{array}{c} + \\ | \\ \text{} \end{array}$ $\begin{array}{c} 5 \\ | \\ \text{} \end{array}$



+ 5 12 <exp>
<op> <exp>

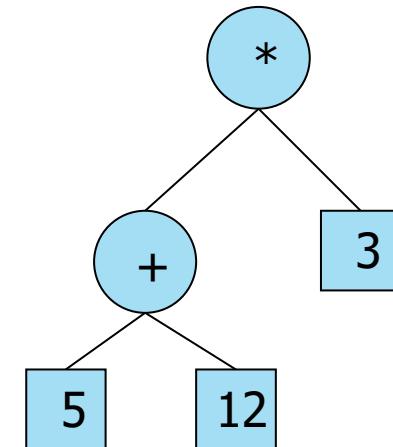


Espressione prefissa memorizzata in vettore a di caratteri (spazi per separare):



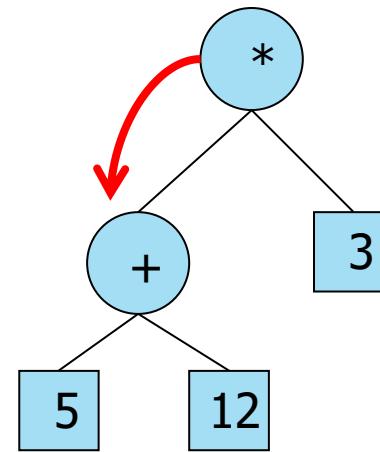
Valutazione: scansione

- se $a[i]$ è un operatore op, ritorna $\text{eval}(\text{op})$
- terminazione: se $a[i]$ è una cifra, calcola il valore dell'intero formato da cifre fino allo spazio, ritorna l'intero

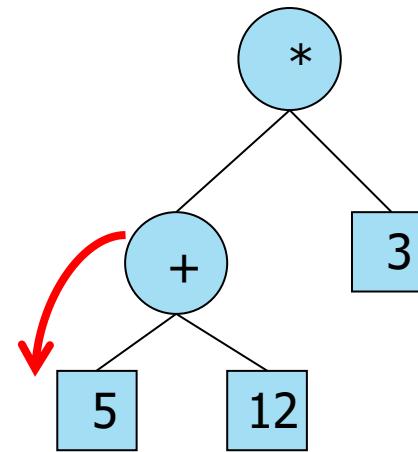
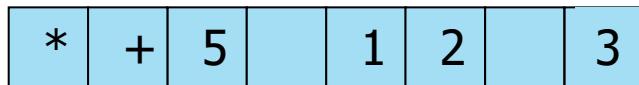


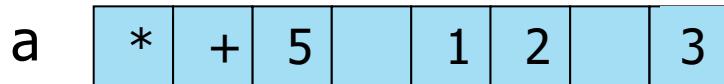
a

*	+	5		1	2		3
---	---	---	--	---	---	--	---

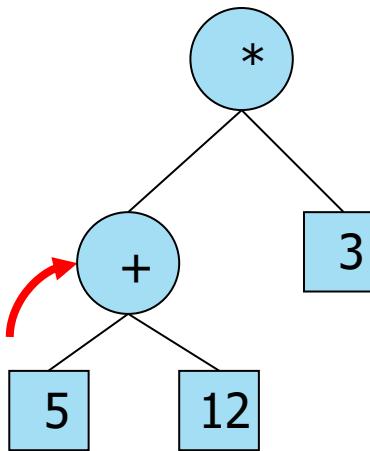


a

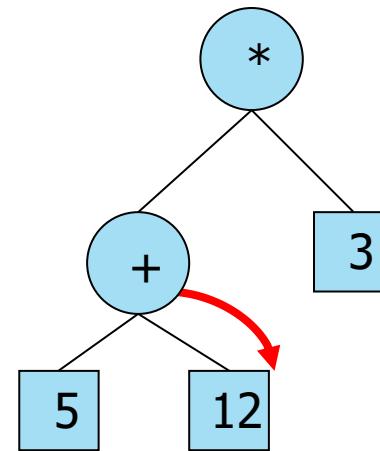
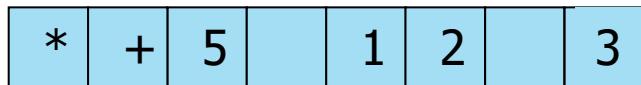




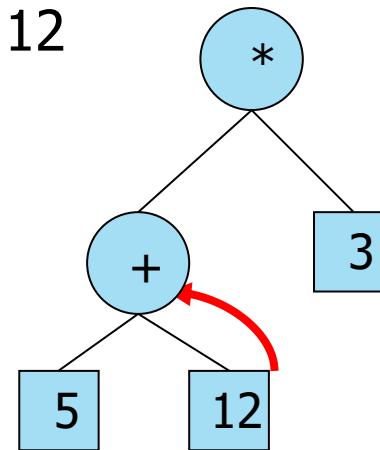
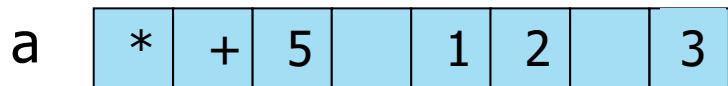
return 5



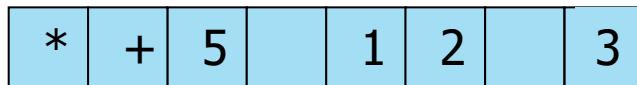
a



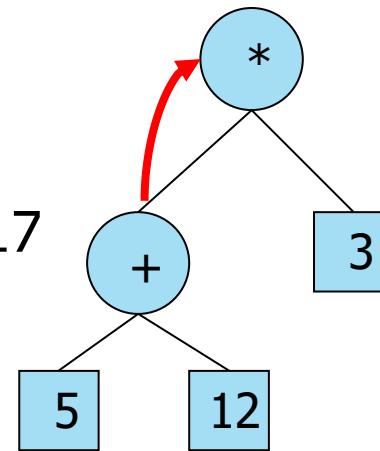
return 12



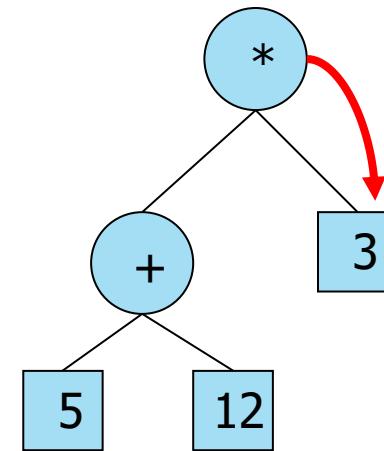
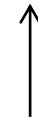
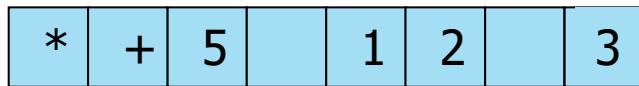
a



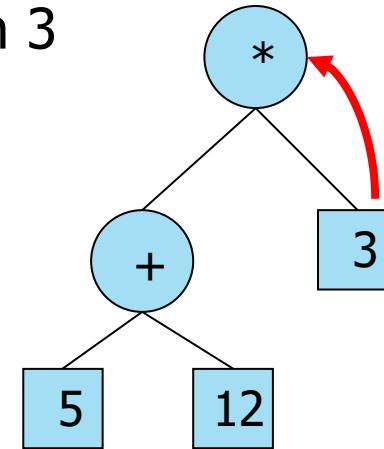
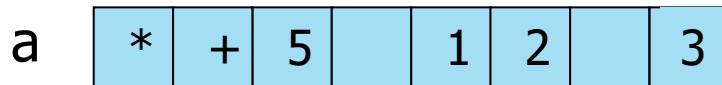
return 17



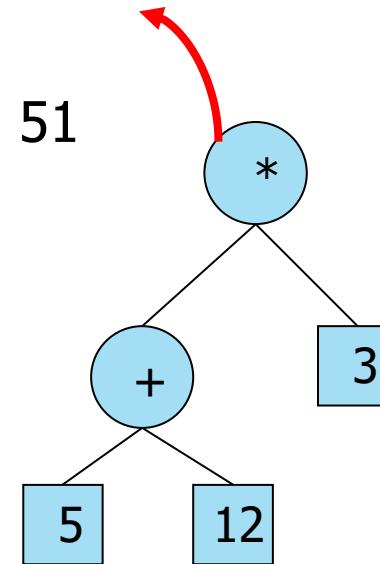
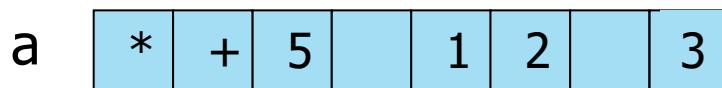
a



return 3



return 51



Nel main si dichiarano variabili globali:

```
int eval() {  
    int x = 0;  
    while (a[i] == ' ')  
        i++;  
    if (a[i] == '+') {  
        i++;  
        return eval() + eval();  
    }  
    if (a[i] == '*') {  
        i++;  
        return eval() * eval();  
    }  
    while ((a[i] >= '0') && (a[i] <= '9'))  
        x = 10 * x + (a[i++]-'0');  
    return x;  
}
```

```
static char *a;  
static int i;
```

Alberi Binari di Ricerca

BST: DEFINIZIONE E IMPLEMENTAZIONE COME ADT

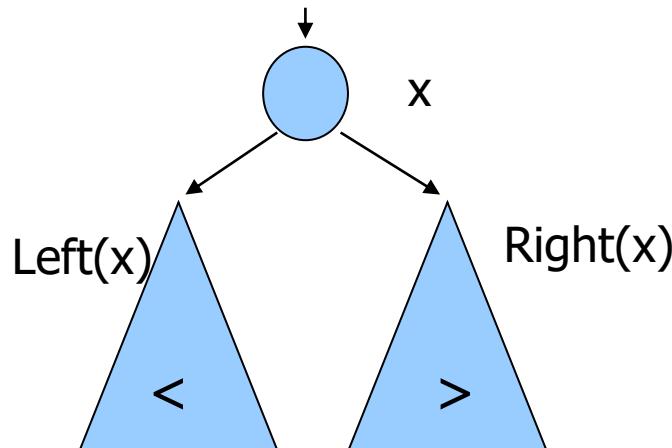
Alberi binari di ricerca (BST)

ADT albero binario con proprietà:

\forall nodo x vale che:

- \forall nodo $y \in \text{Left}(x)$, $\text{key}[y] < \text{key}[x]$
- \forall nodo $y \in \text{Right}(x)$, $\text{key}[y] > \text{key}[x]$

chiavi distinte!



ADT di classe BST

BST.h

```
typedef struct binarysearchtree *BST;

BST    BSTinit() ;
void   BSTfree(BST bst);
int    BSTcount(BST bst);
int    BSTempty(BST bst);
Item   BSTsearch(BST bst, Key k);
void   BSTinsert_leafI(BST bst, Item x);
void   BSTinsert_leafR(BST bst, Item x);
void   BSTinsert_root(BST bst, Item x);
Item   BSTmin(BST bst);
Item   BSTmax(BST bst);
void   BSTvisit(BST bst, int strategy);
```

BST.c

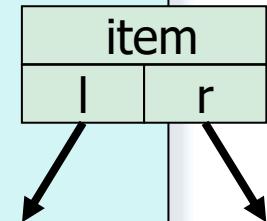
```
#include <stdlib.h>
#include "Item.h"
#include "BST.h"

typedef struct BSTnode* link;
struct BSTnode { Item item; link l; link r; };
struct binarysearchtree { link root; link z; };

static link NEW(Item item, link l, link r) {
    link x = malloc(sizeof *x);
    x->item = item; x->l = l; x->r = r;
    return x;
}

BST BSTinit( ) {
    BST bst = malloc(sizeof *bst) ;
    bst->root= ( bst->z = NEW(ITEMsetNull(), NULL, NULL));
    return bst;
}
```

BSTnode



nodo sentinella

```
void BSTfree(BST bst) {
    if (bst == NULL)
        return;
    treeFree(bst->root, bst->z);
    free(bst->z);
    free(bst);
}

static void treeFree(link h, link z) {
    if (h == z)
        return;
    treeFree(h->l, z);
    treeFree(h->r, z);
    free(h);
}
```

```
static int countR(link h, link z) {
    if (h == z)
        return 0;
    return countR(h->l, z) + countR(h->r, z) +1;
}

int BSTcount(BST bst) {
    return countR(bst->root, bst->z);
}

int BSTempty(BST bst) {
    if (BSTcount(bst) == 0)
        return 1;
    return 0;
}
```

BSTsearch

Ricerca ricorsiva di un nodo che contiene un item con una chiave data:

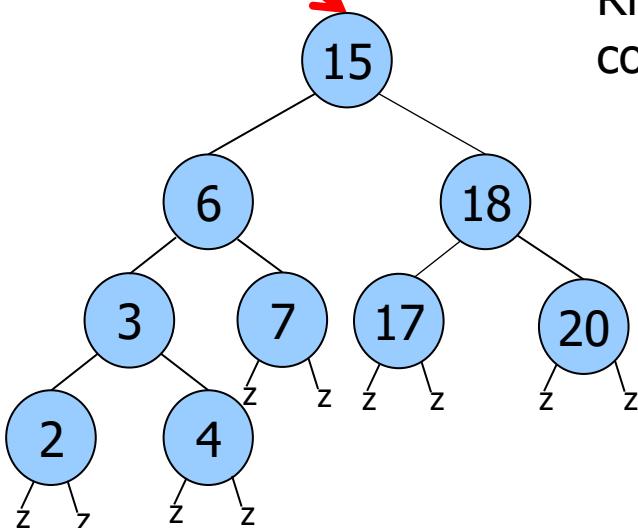
- percorrimento dell'albero dalla radice
- terminazione: la chiave dell'item cercato è uguale alla chiave del nodo corrente (search hit) oppure si è giunti ad un albero vuoto (search miss)
- ricorsione: dal nodo corrente
 - su sottoalbero sinistro se la chiave dell'item cercato è < della chiave del nodo corrente
 - su sottoalbero destro altrimenti

```
Item searchR(link h, Key k, link z) {
    int cmp;
    if (h == z)
        return ITEMsetNull();
    cmp = KEYcmp(k, KEYget(h->item));
    if (cmp == 0)
        return h->item;
    if (cmp == -1)
        return searchR(h->l, k, z);
    return searchR(h->r, k, z);
}
```

```
Item BSTsearch(BST bst, Key k) {
    return searchR(bst->root, k, bst->z);
}
```

Esempio

$h = bst->root$

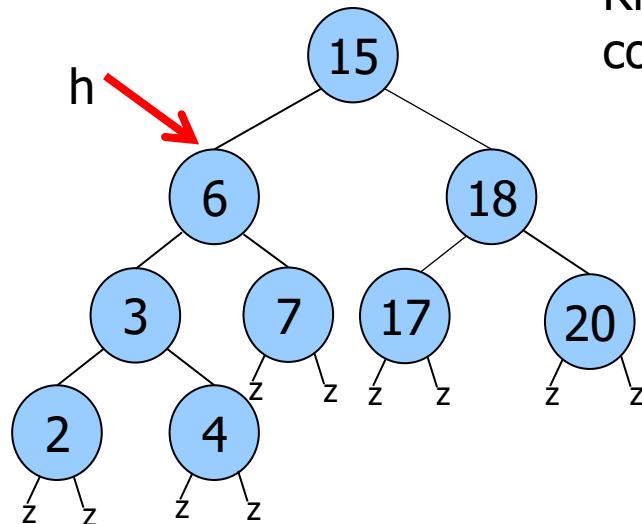


Ricerca dell'item
con chiave 7

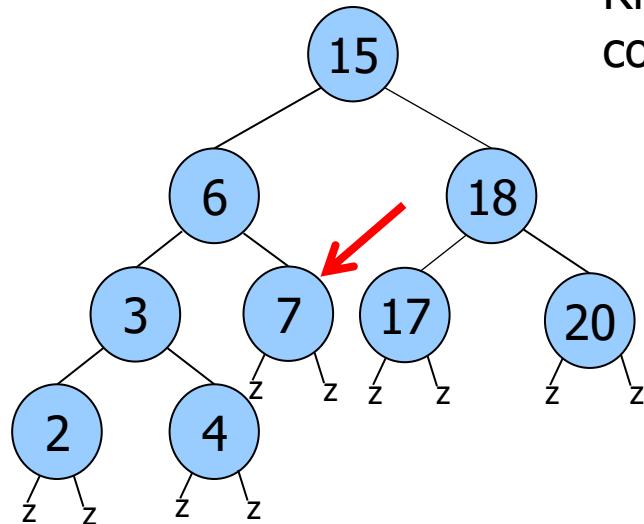
Sono visualizzate solo le
chiavi intere, non gli item

z: nodo sentinella

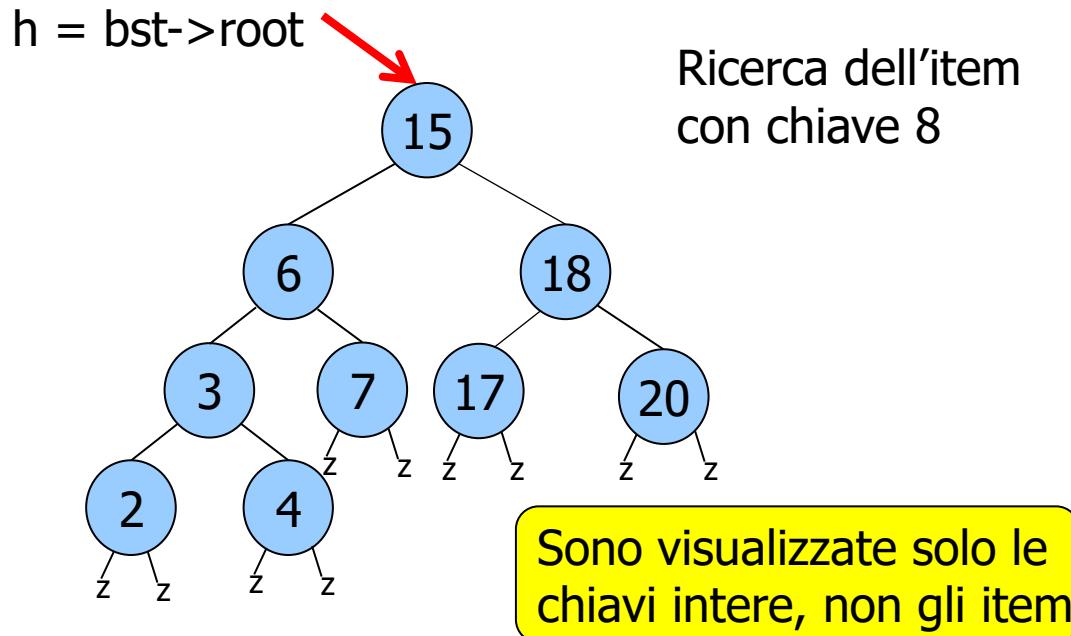
Ricerca dell'item
con chiave 7



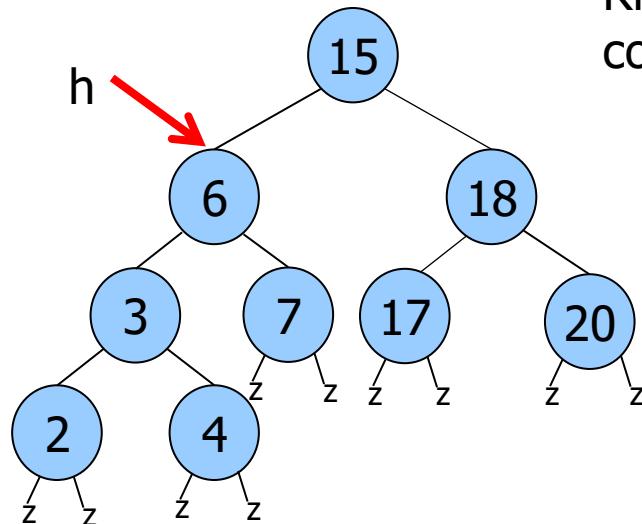
Ricerca dell'item
con chiave 7
Hit!



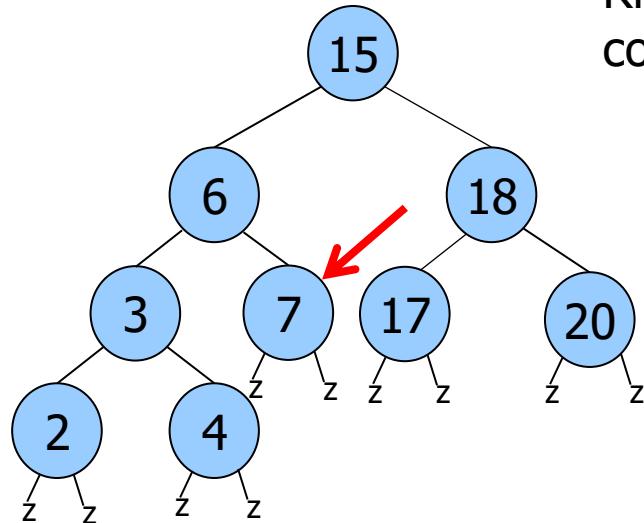
Esempio



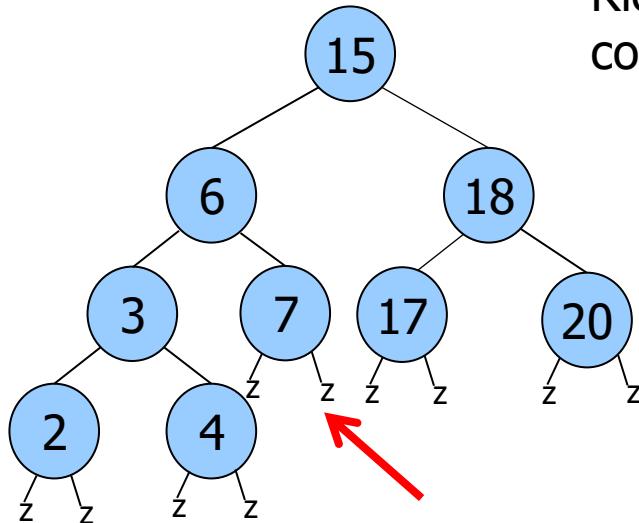
Ricerca dell'item
con chiave 8



Ricerca dell'item
con chiave 8



Ricerca dell'item
con chiave 8
Miss!



BSTmin

```
Item minR(link h, link z) {
    if (h == z)
        return ITEMsetNull();
    if (h->l == z)
        return (h->item);
    return minR(h->l, z);
}

Item BSTmin(BST bst) {
    return minR(bst->root, bst->z);
}
```

Seguire il puntatore al sottoalbero sinistro finché si arriva al nodo sentinella z

BSTmax

```
Item maxR(link h, link z) {
    if (h == z)
        return ITEMsetNull();
    if (h->r == z)
        return (h->item);
    return maxR(h->r, z);
}

Item BSTmax(BST bst) {
    return maxR(bst->root, bst->z);
}
```

Seguire il puntatore al sottoalbero destro finché si arriva al nodo sentinella z

BSTinsert (in foglia)

Inserire in un albero binario di ricerca un nodo che contiene un item \Rightarrow mantenimento della proprietà:

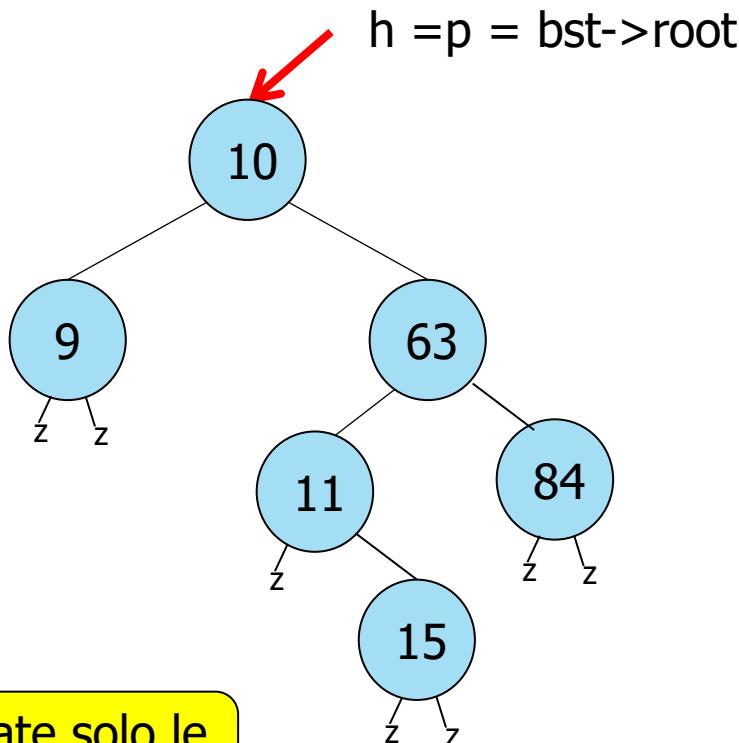
- se il BST è vuoto, creazione del nuovo albero
- inserimento **ricorsivo** nel sottoalbero sinistro o destro a seconda del confronto tra la chiave dell'item e quella del nodo corrente
- inserimento **iterativo**: prima si ricerca la posizione, poi si appende il nuovo nodo.

```
static link insertR(link h, Item x, link z) {
    if (h == z)
        return NEW(x, z, z);
    if (KEYcmp(KEYget(x), KEYget(h->item)) == -1)
        h->l = insertR(h->l, x, z);
    else
        h->r = insertR(h->r, x, z);
    return h;
}

void BSTinsert_leafR(BST bst, Item x) {
    bst->root = insertR(bst->root, x, bst->z);
}
```

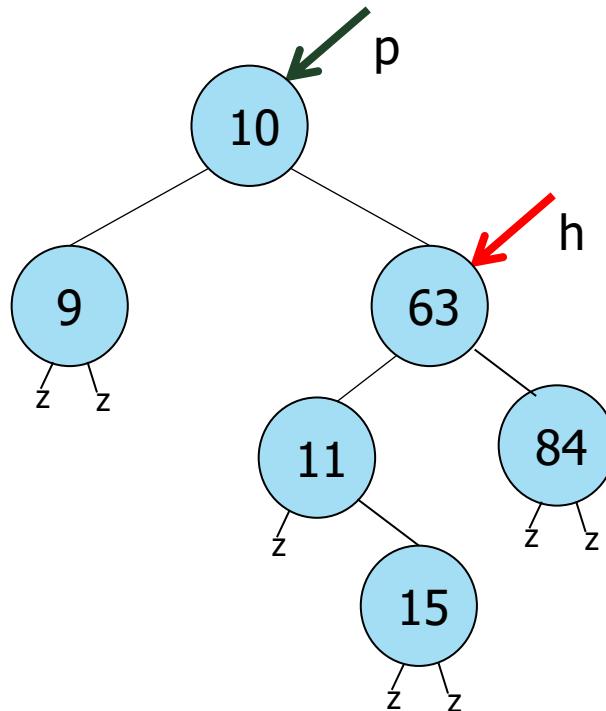
```
void BSTinsert_leafI(BST bst, Item x) {  
    link p = bst->root, h = p;  
    if (bst->root == bst->z) {  
        bst->root = NEW(x, bst->z, bst->z);  
        return;  
    }  
    while (h != bst->z) {  
        p = h;  
        h=(KEYcmp(KEYget(x),KEYget(h->item))==-1) ? h->l : h->r;  
    }  
    h = NEW(x, bst->z, bst->z);  
    if (KEYcmp(KEYget(x), KEYget(p->item))==-1)  
        p->l = h;  
    else  
        p->r = h;  
}
```

Esempio

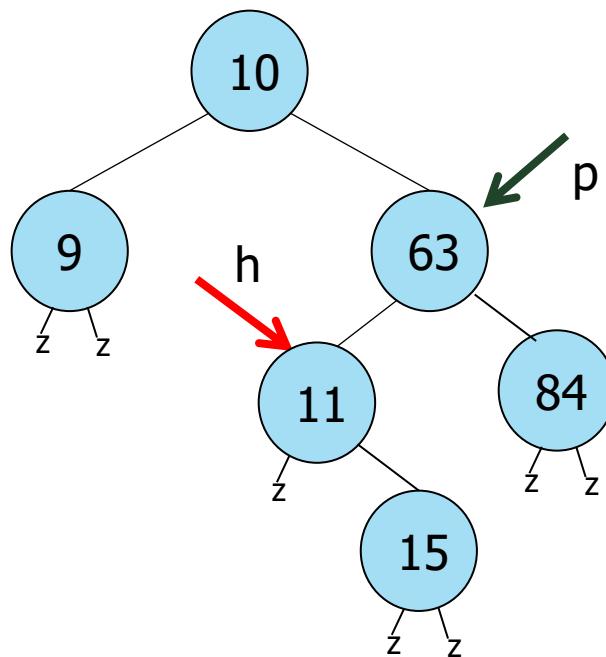


inserimento dell'item con
chiave 62

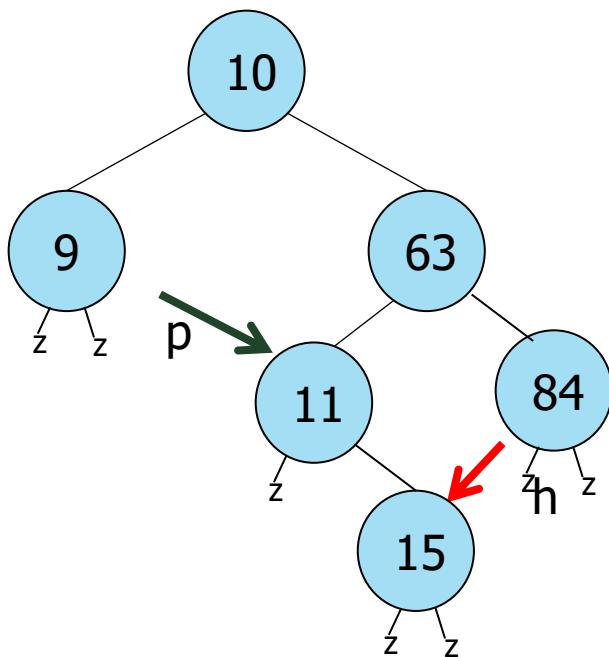
Sono visualizzate solo le
chiavi intere, non gli item



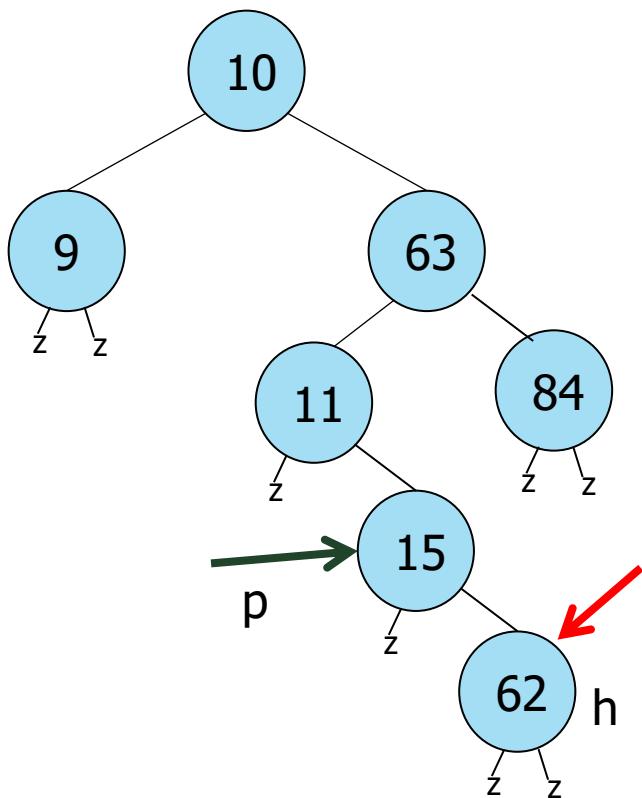
inserimento dell'item con
chiave 62



inserimento dell'item con
chiave 62



inserimento dell'item con
chiave 62



inserimento dell'item con
chiave 62

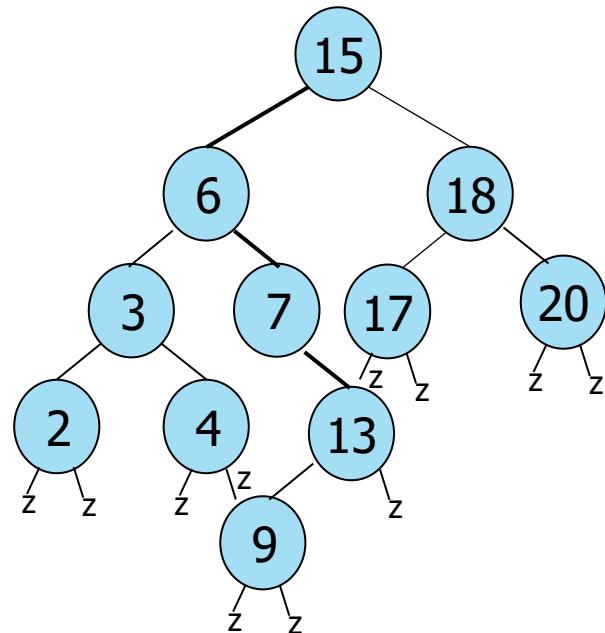
Complessità

Le operazioni hanno complessità $T(n) = O(h)$:

- albero con n nodi completamente bilanciato
 - altezza $h = \alpha(\log_2 n)$
- albero con n nodi completamente sbilanciato ha
 - altezza $h = \alpha(n)$
- $O(\log n) \leq T(n) \leq O(n)$

BSTvisit

Attraversamento in-ordine: **ordinamento crescente delle chiavi.**



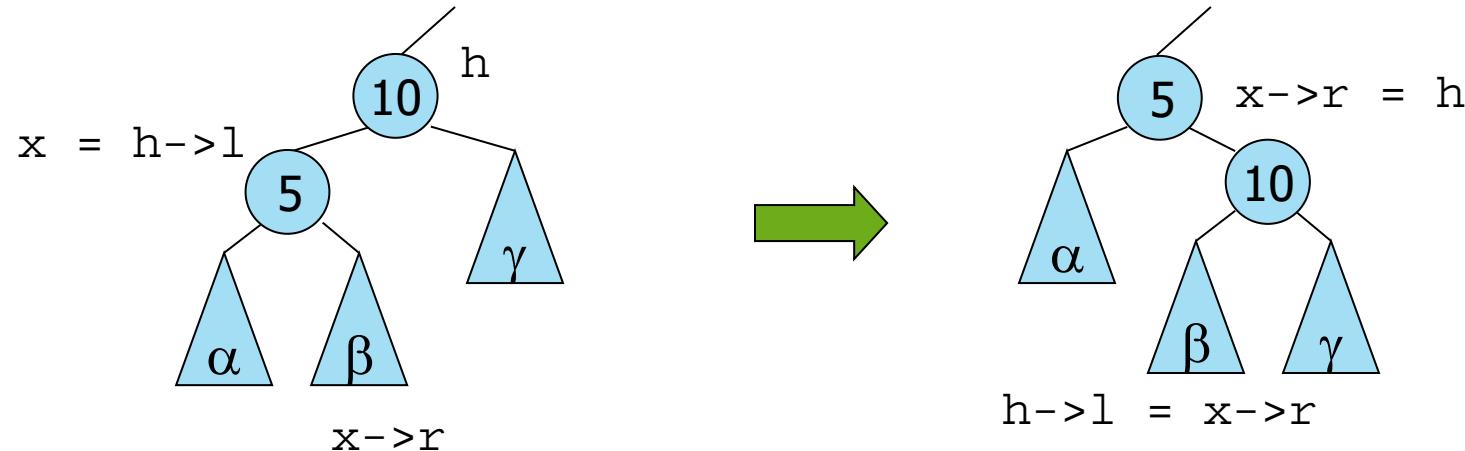
2 3 4 6 7 9 13 15 17 18 20



La chiave **mediana** (inferiore) di un insieme di n elementi è l'elemento che si trova in posizione $\lfloor(n + 1)/2\rfloor$ nella sequenza ordinata degli elementi dell'insieme

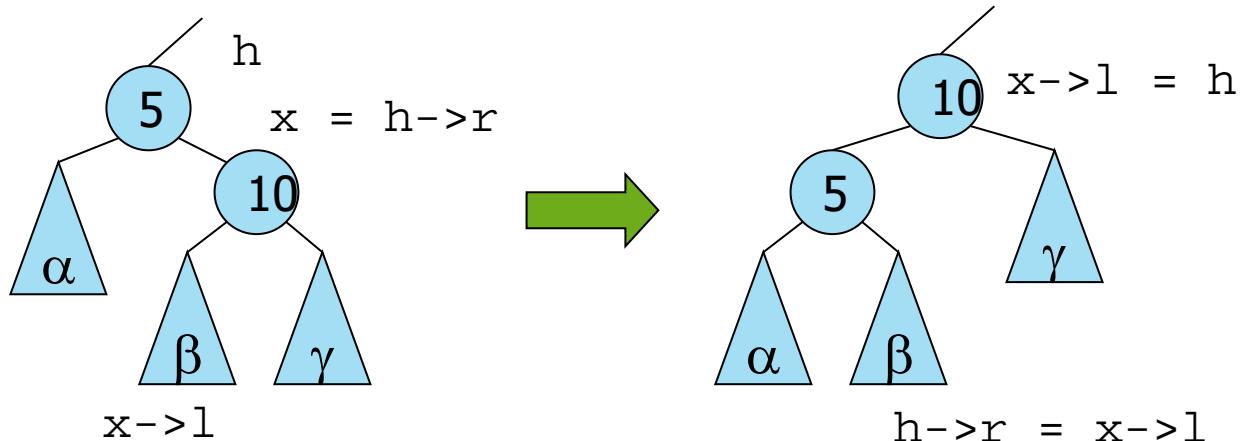
```
static void treePrintR(link h, link z, int strategy) {
    if (h == z)
        return;
    if (strategy == PREORDER)
        ITEMstore(h->item);
    treePrintR(h->l, z, strategy);
    if (strategy == INORDER)
        ITEMstore(h->item);
    treePrintR(h->r, z, strategy);
    if (strategy == POSTORDER)
        ITEMstore(h->item);
}
void BSTvisit(BST bst, int strategy) {
    if (BSTempty(bst))
        return;
    treePrintR(bst->root, bst->z, strategy);
}
```

Rotazione a destra di BST



```
link rotR(link h) {  
    link x = h->l;  
    h->l = x->r;  
    x->r = h;  
    return x;  
}
```

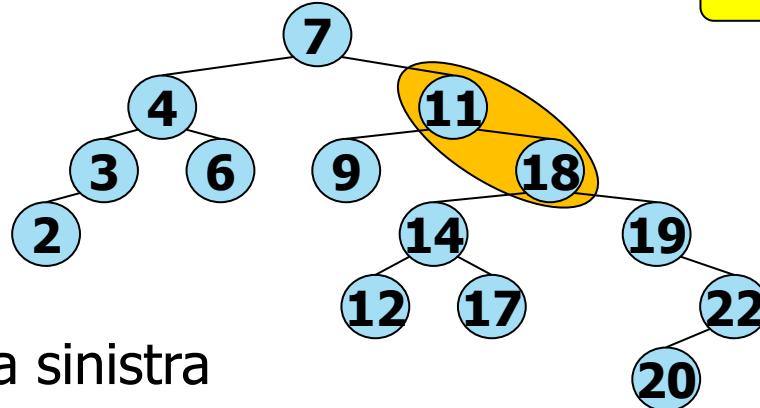
Rotazione a sinistra di BST



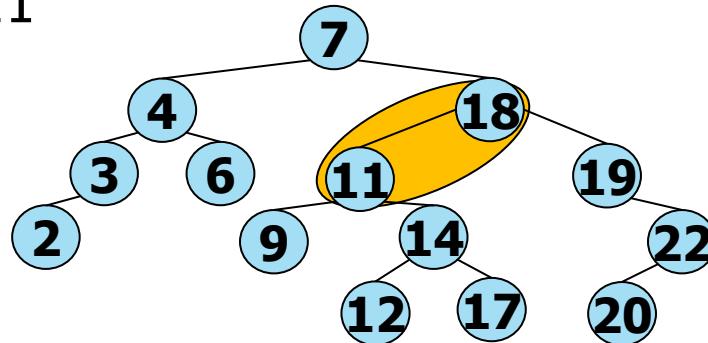
```
link rotL(link h) {  
    link x = h->r;  
    h->r = x->l;  
    x->l = h;  
    return x;  
}
```

Esempio

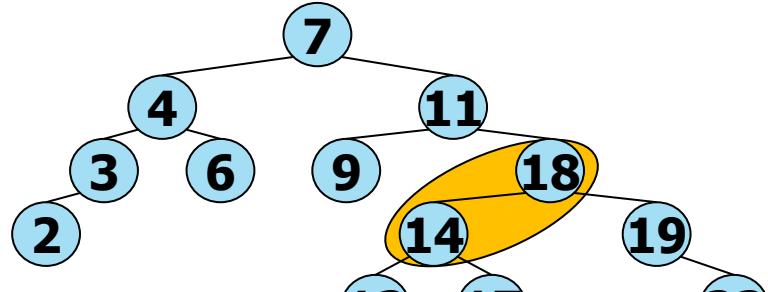
Nodi sentinella z non riportati



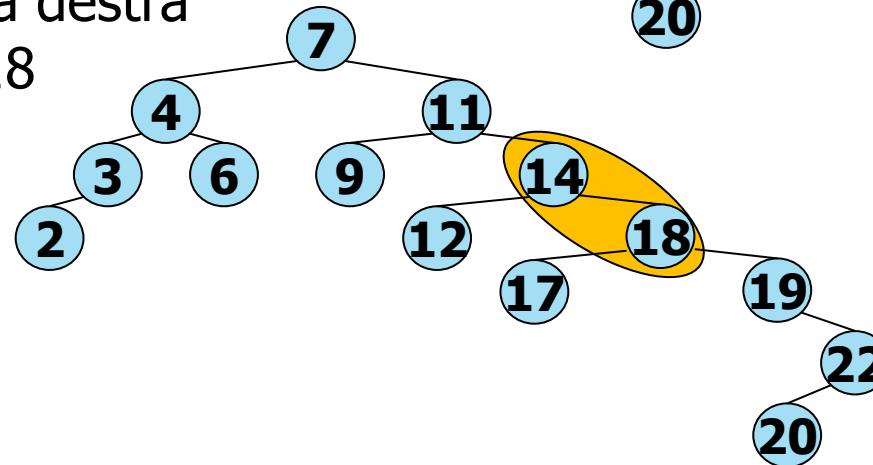
Rotazione a sinistra
attorno a 11



Esempio



Rotazione a destra
attorno a 18

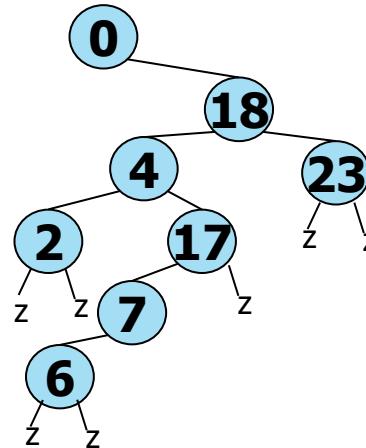
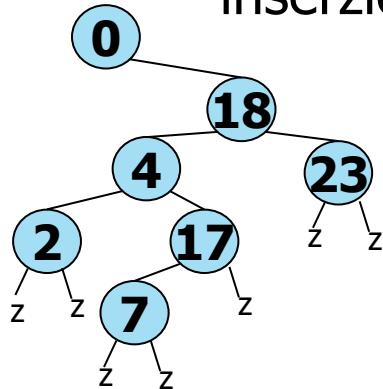


BSTinsert (in radice)

- Inserimento dalle foglie a scelta e non obbligatorio
- Nodi più recenti nella parte alta del BST
- Inserimento ricorsivo alla radice:
 - inserimento nel sottoalbero appropriato
 - rotazione per farlo diventare radice dell'albero principale.

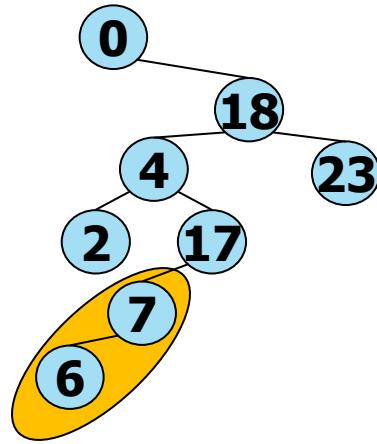
Esempio

inserzione di 6

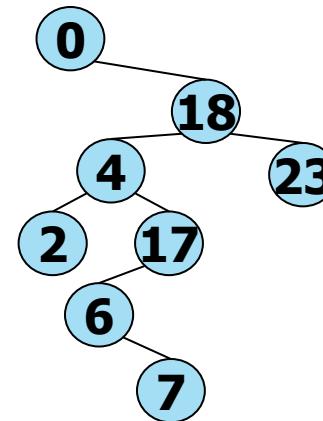


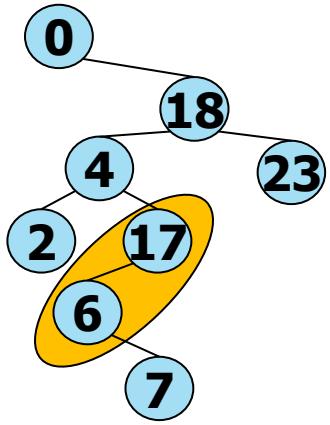
inserzione di 6 alla radice
del sottoalbero opportuno

Nodi sentinella z non più riportati

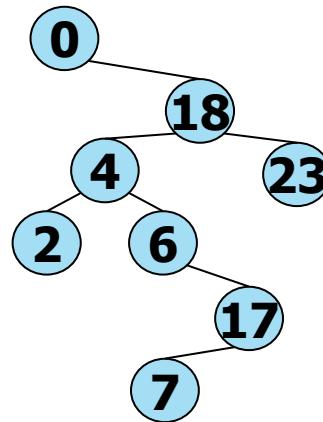


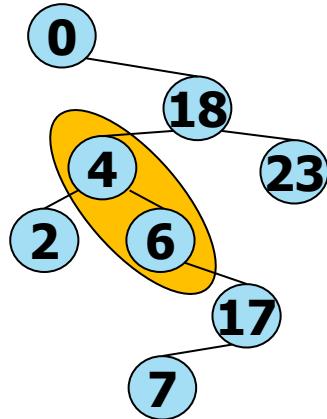
rotazione a DX



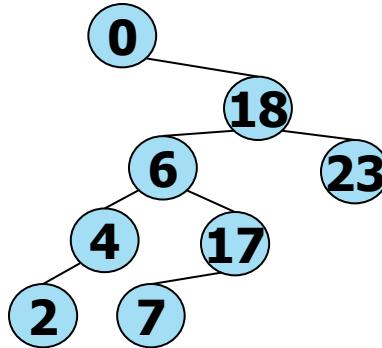


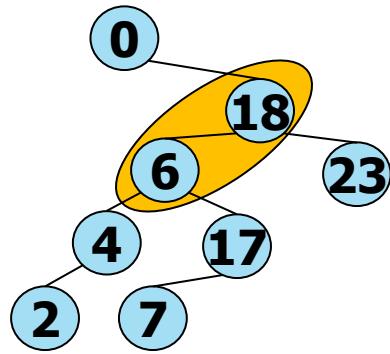
rotazione a DX



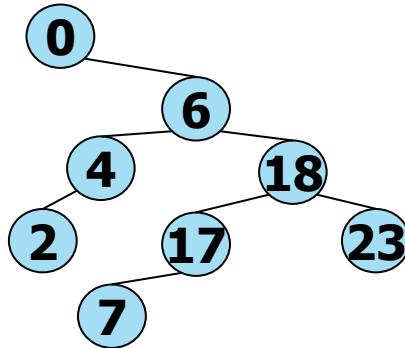


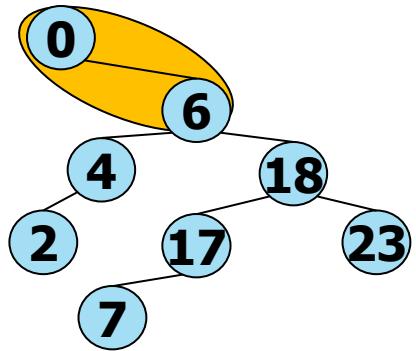
rotazione a SX



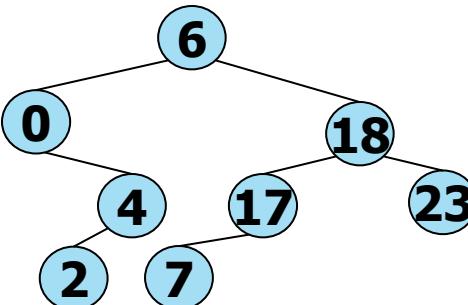


rotazione a DX





rotazione a SX



```
static link insertT(link h, Item x, link z) {
    if (h == z)
        return NEW(x, z, z);
    if (KEYcmp(KEYget(x), KEYget(h->item)) == -1) {
        h->l = insertT(h->l, x, z);
        h = rotR(h);
    }
    else {
        h->r = insertT(h->r, x, z);
        h = rotL(h);
    }
    return h;
}

void BSTinsert_root(BST bst, Item x) {
    bst->root = insertT(bst->root, x, bst->z);
}
```

Estensioni dei BST elementari

PUNTATORE AL PADRE, NUMERO DI NODI NEL SOTTO-ALBERO

Estensione dei BST elementari

Al nodo elementare si possono aggiungere informazioni che permettono lo sviluppo semplice di nuove funzioni:

- puntatore al padre
- numero di nodi dell'albero radicato nel nodo corrente.

Queste informazioni devono ovviamente essere gestite (quando necessario) da tutte le funzioni già viste.

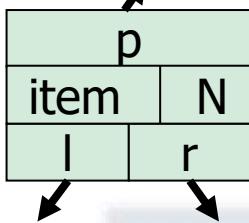
ADT di classe BST

nuove funzioni
funzioni modificate

BST.h

```
typedef struct binarysearchtree *BST;

BST    BSTinit();
void   BSTfree(BST bst); int BSTcount(BST bst);
int    BSTempty(BST bst);
Item   BSTmin(BST bst); Item BSTmax(BST bst);
void   BSTinsert_leafI(BST bst, Item x);
void   BSTinsert_leafR(BST bst, Item x);
void   BSTinsert_root(BST bst, Item x);
Item   BSTsearch(BST bst, Key k);
void   BSTdelete(BST bst, Key k);
Item   BSTselect(BST bst, int r); Order-Statistic BST
void   BSTvisit(BST bst, int strategy);
Item   BSTsucc(BST bst, Key k);
Item   BSTpred(BST bst, Key k);
void   BSTbalance(BST bst);
```



BSTnode



BST.c

```

#include <stdlib.h>
#include "Item.h"      puntatore al padre
#include "BST.h"       dimensione sottoalbero

typedef struct BSTnode* link;
struct BSTnode {Item item; link p; link l; link r; int N;};
struct binarysearchtree { link root; link z; };

static link NEW(Item item, link p, link l, link r, int N){
    link x = malloc(sizeof *x);
    x->item = item;
    x->p = p; x->l = l; x->r = r; x->N = N;
    return x;
}

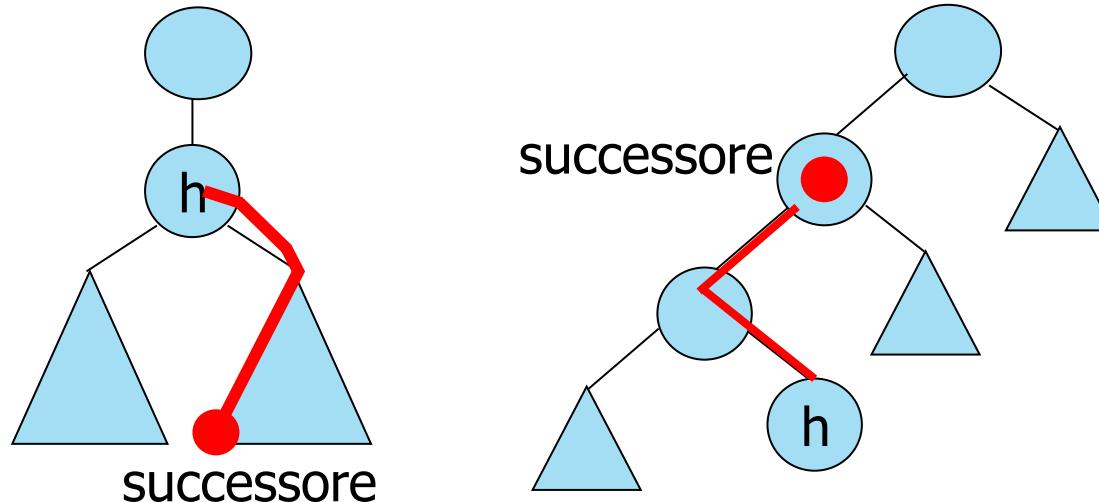
```

```
BST BSTinit( ) {  
    BST bst = malloc(sizeof *bst) ;  
    bst->root=(bst->z=NEW(ITEMsetNull(), NULL, NULL, NULL, 0));  
    return bst;  
}
```

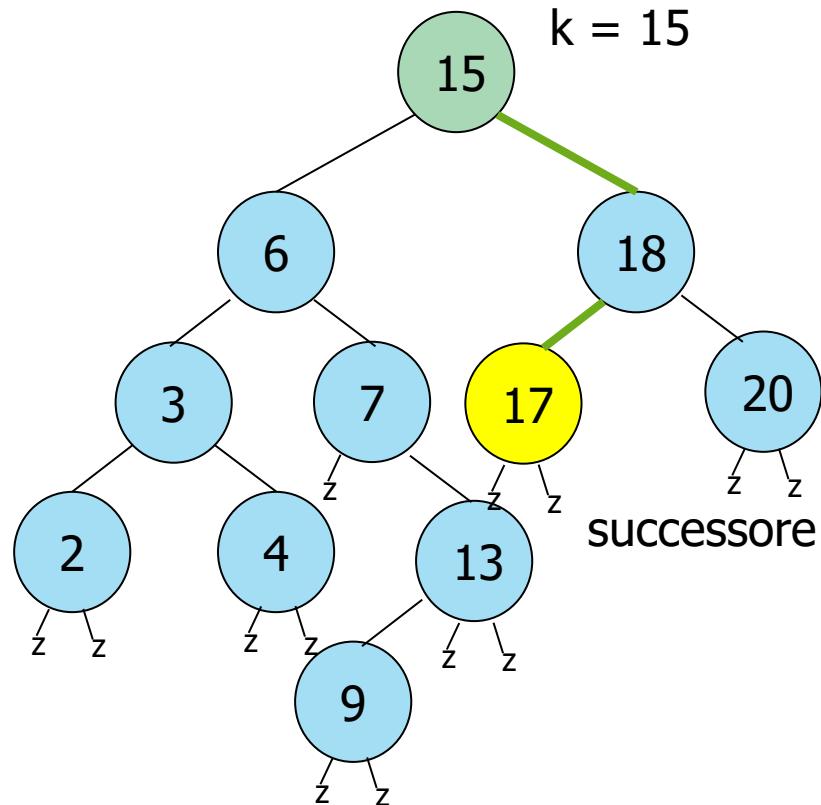
Successore

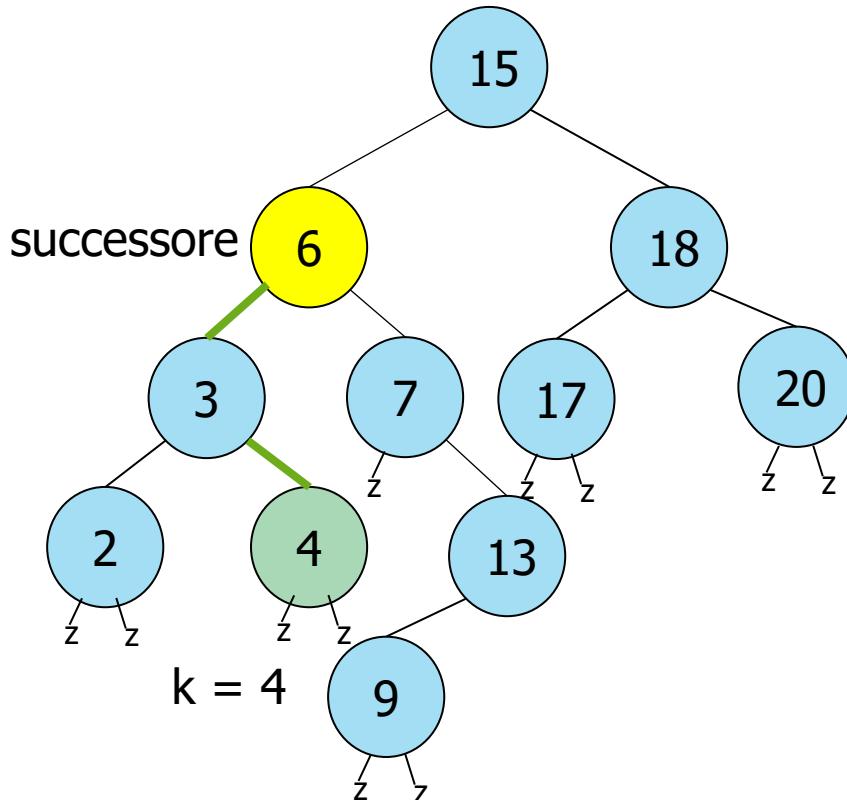
Successore di un item: se esiste, nodo h con un item con la più piccola chiave $>$ della chiave di item, altrimenti item vuoto. Due casi:

- $\exists \text{Right}(h)$: $\text{succ}(\text{key}(h)) = \min(\text{Right}(h))$
- $\nexists \text{Right}(h)$: $\text{succ}(\text{key}(h)) = \text{primo antenato di } h \text{ il cui figlio sinistro è anche un antenato di } h.$



Esempio





```
Item searchSucc(link h, key k, link z) {
    link p;
    if (h == z) return ITEMsetNull();
    if (KEYcmp(k, KEYget(h->item))==0) {
        if (h->r != z) return minR(h->r, z);
        else {
            p = h->p;
            while (p != z && h == p->r) {
                h = p; p = p->p;
            }
            return p->item;
        }
    }
    if (KEYcmp(k, KEYget(h->item))==-1)
        return searchSucc(h->l, k, z);
    return searchSucc(h->r, k, z);
}
Item BSTsucc(BST bst, Key k) {
    return searchSucc(bst->root, k, bst->z);
}
```

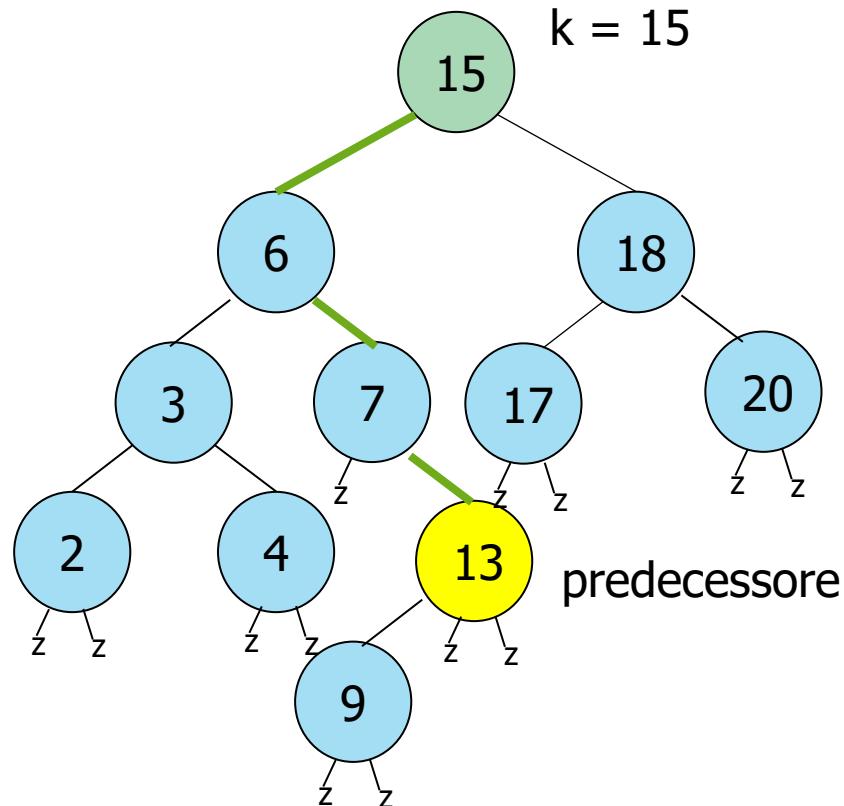
Predecessore

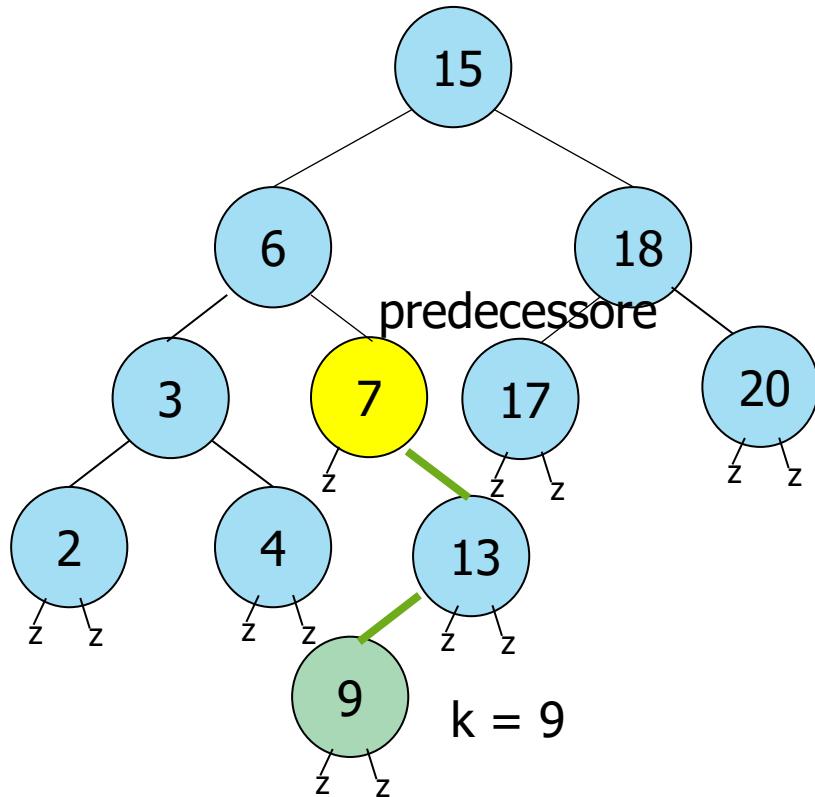
Predecessore di un item: nodo h con item con la più grande chiave $<$ della chiave di item.

Due casi:

- $\exists \text{Left}(h)$: $\text{pred}(\text{key}(h)) = \max(\text{Left}(h))$
- $\nexists \text{Left}(h)$: $\text{pred}(\text{key}(h)) = \text{primo antenato di } h \text{ il cui figlio destro è anche un antenato di } h$.

Esempio





```

Item searchPred(link h, Key k, link z) {
    link p;
    if (h == z) return ITEMsetNull();
    if (KEYcmp(k, KEYget(h->item))==0) {
        if (h->l != z) return maxR(h->l, z);
        else {
            p = h->p;
            while (p != z && h == p->l) {h = p; p = p->p;}
            return p->item;
        }
    }
    if (KEYcmp(k, KEYget(h->item))==-1)
        return searchPred(h->l, k, z);
    return searchPred(h->r, k, z);
}
Item BSTpred(BST bst, Key k) {
    return searchPred(bst->root, k, bst->z);
}

```

BSTinsert (in foglia)

RICORSIVO

```
link insertR(link h, Item x, link z) {  
    if (h == z)  
        return NEW(x, z, z, z, 1);  
    if (KEYcmp(KEYget(x), KEYget(h->item)) == -1) {  
        h->l = insertR(h->l, x, z); h->l->p = h;  
    }  
    else {  
        h->r = insertR(h->r, x, z); h->r->p = h;  
    }  
    (h->N)++;  
    return h;  
}  
void BSTinsert_leafR(BST bst, Item x) {  
    bst->root = insertR(bst->root, x, bst->z);  
}
```

BSTinsert (in foglia)

ITERATIVO

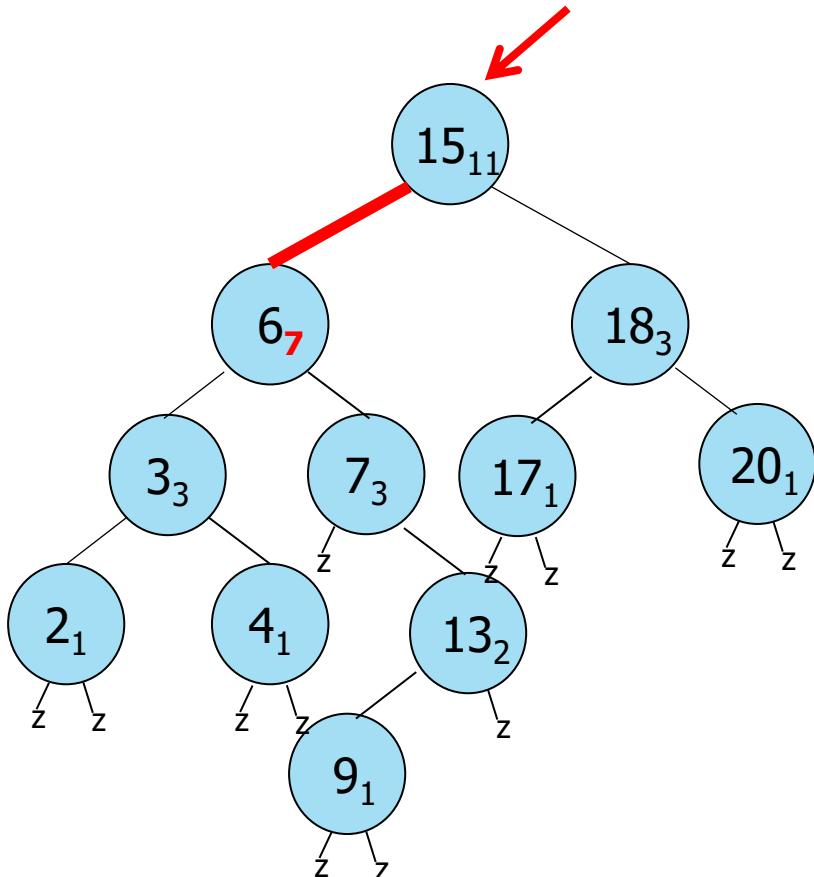
```
void BSTinsert_leafI(BST bst, Item x) {
    link p = bst->root, h = p;
    if (bst->root == bst->z) {
        bst->root = NEW(x, bst->z, bst->z, bst->z, 1);
        return;
    }
    while (h != bst->z) {
        p = h; h->N++;
        h=(KEYcmp(KEYget(x), KEYget(h->item))==-1) ? h->l : h->r;
    }
    h = NEW(x, p, bst->z, bst->z, 1);
    if (KEYcmp(KEYget(x), KEYget(p->item))==-1)
        p->l = h;
    else
        p->r = h;
}
```

BSTselect

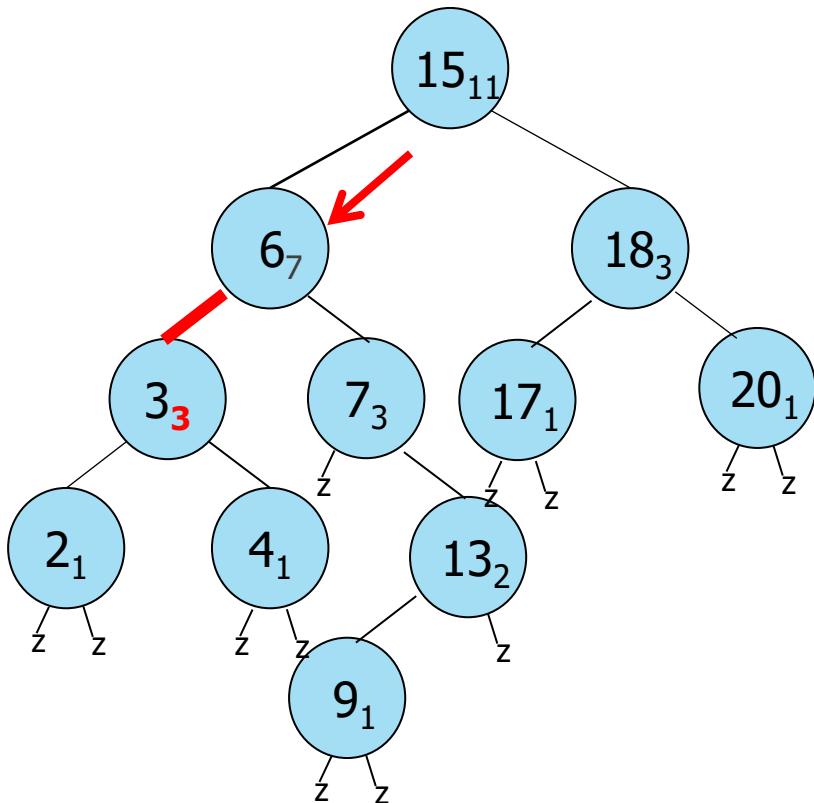
Selezione dell'item con la r-esima chiave più piccola (rango r = chiave in posizione r nell'ordinamento, ad esempio se r=0 item con chiave minima): t è il numero di nodi del sottoalbero sinistro:

- $t = r$: ritorno la radice del sottoalbero
- $t > r$: ricorsione nel sottoalbero sinistro alla ricerca della k-esima chiave più piccola
- $t < r$: ricorsione nel sottoalbero destro alla ricerca della $(r-t-1)$ -esima chiave più piccola

Esempio



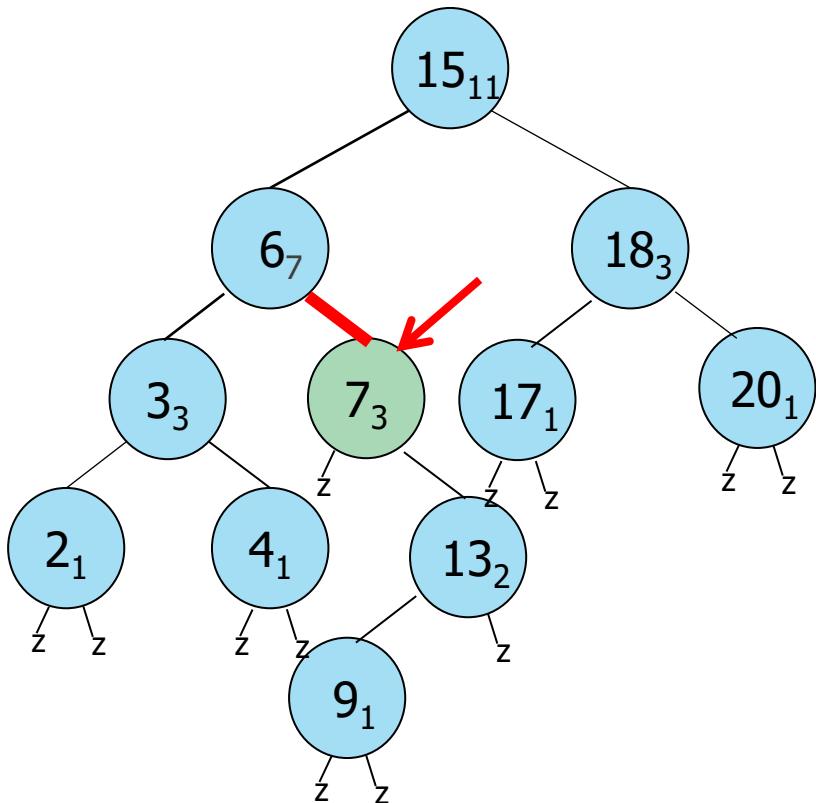
rango = r = 4
quinta più
piccola chiave
t=7
7>4 scendo a SX



$$r = 4$$

$$t=3$$

3<4 scendo a DX
cerco $r=4-3-1=0$

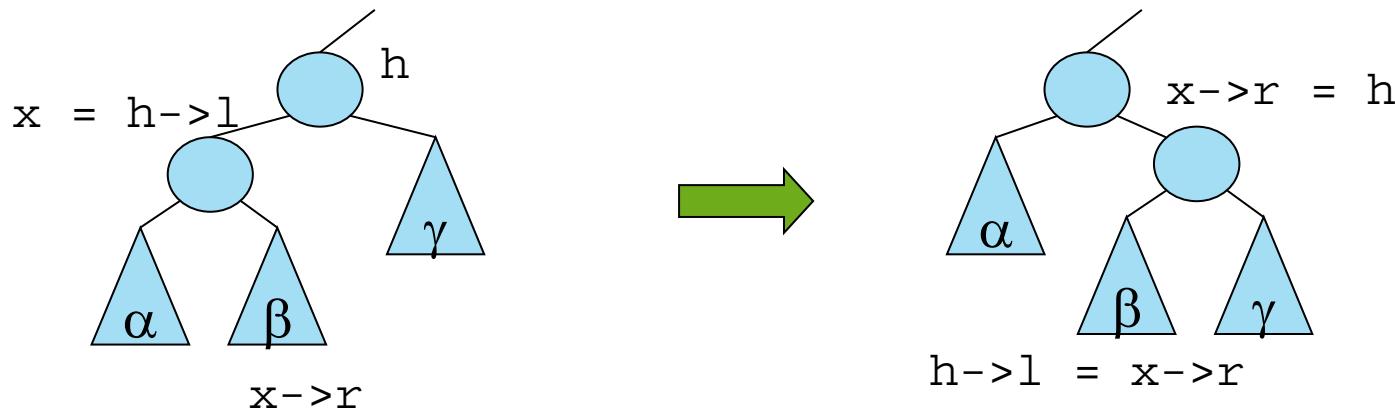


$r = 0$
 $t=0$
return 7

```
Item selectR(link h, int r, link z) {
    int t;
    if (h == z)
        return ITEMsetNull();
    t = h->l->N;
    if (t > r)
        return selectR(h->l, r, z);
    if (t < r)
        return selectR(h->r, r-t-1, z);
    return h->item;
}
```

```
Item BSTselect(BST bst, int r) {
    return selectR(bst->root, r, bst->z);
}
```

Rotazione a destra di BST

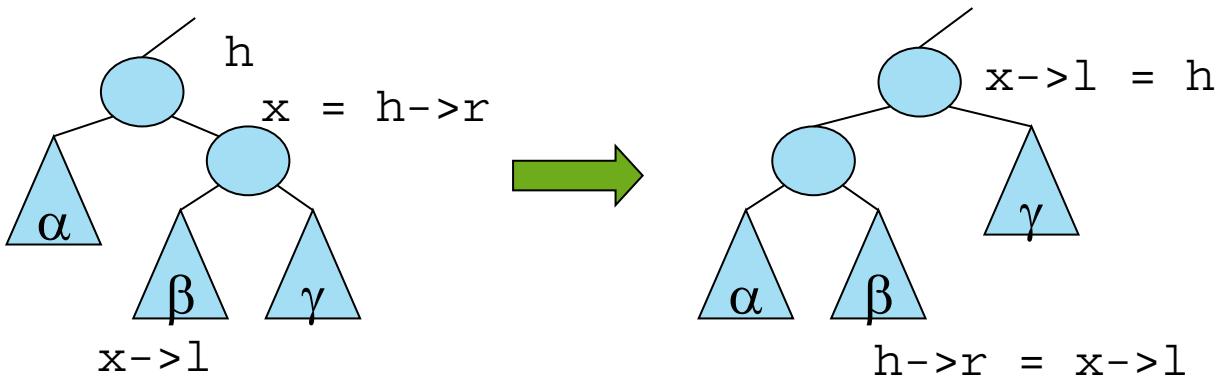


aggiornamento
dimensione
sottoalberi

```
link rotR(link h) {  
    link x = h->l;  
    h->l = x->r;  
x->r->p = h;  
    x->r = h;  
x->p = h->p;  
h->p = x;  
    x->N = h->N;  
    h->N = 1;  
    h->N += (h->l) ? h->l->N : 0;  
    h->N += (h->r) ? h->r->N : 0;  
    return x;  
}
```

aggiornamento
puntatore
al padre

Rotazione a sinistra di BST



aggiornamento
dimensione
sottoalberi

```
link rotL(link h) {  
    link x = h->r;  
    h->r = x->l;  
    x->l->p = h;  
    x->l = h;  
    x->p = h->p;  
    h->p = x;  
    x->N = h->N;  
    h->N = 1;  
    h->N += (h->l) ? h->l->N : 0;  
    h->N += (h->r) ? h->r->N : 0;  
    return x;  
}
```

aggiornamento
puntatore
al padre

BSTinsert (in radice)

```
link insert(link h, Item x, link z) {  
    if ( h == z)  
        return NEW(x, z, z, z, 1);  
    if (KEYcmp(KEYget(x), KEYget(h->item))==-1) {  
        h->l = insertT(h->l, x, z); h = rotR(h); h->N++;  
    }  
    else {  
        h->r = insertT(h->r, x, z); h = rotL(h); h->N++;  
    }  
    return h;  
}  
  
void BSTinsert_root(BST bst, Item x) {  
    bst->root = insertT(bst->root, x, bst->z);  
}
```

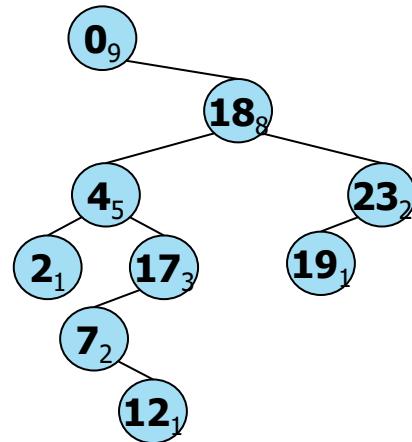
BSTpartition

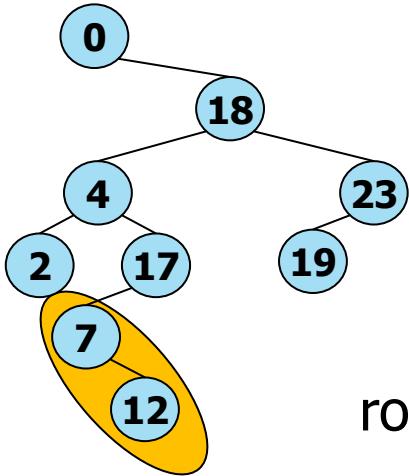
- Riorganizza l'albero avendo l'item con la k-esima chiave più piccola nella radice:
 - porre il nodo come radice di un sottoalbero:
 - $t > k$: ricorsione nel sottoalbero sinistro, partizionamento rispetto alla k-esima chiave più piccola, al termine rotazione a destra
 - $t < k$: ricorsione nel sottoalbero destro, partizionamento rispetto alla $(k-t-1)$ -esima chiave più piccola , al termine rotazione a sinistra
 - Sovente il partizionamento si fa attorno alla chiave mediana

Esempio

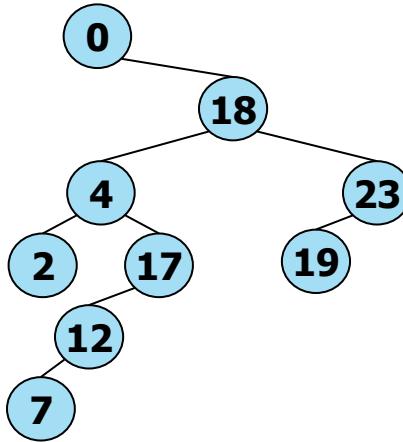
Nodi sentinella z non riportati

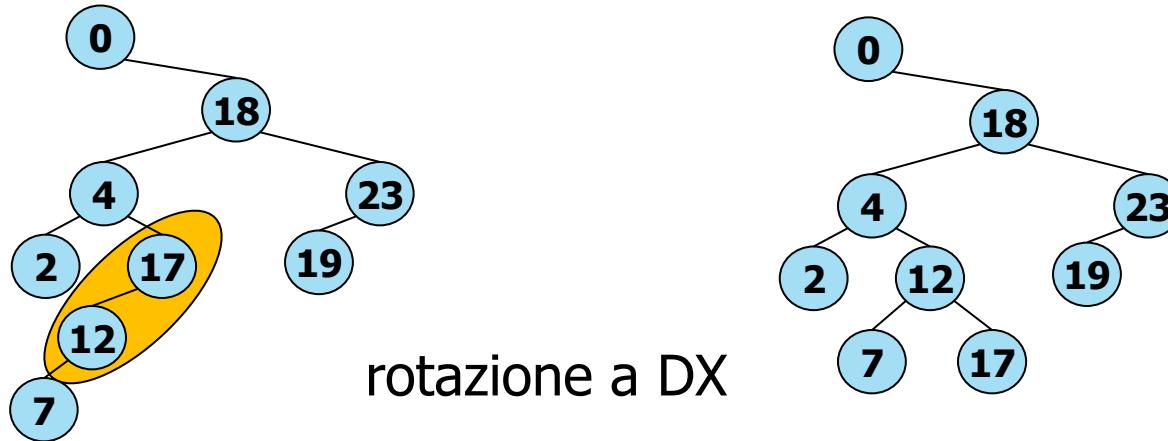
Partizionamento rispetto alla
5a chiave più piccola (12, k=4)

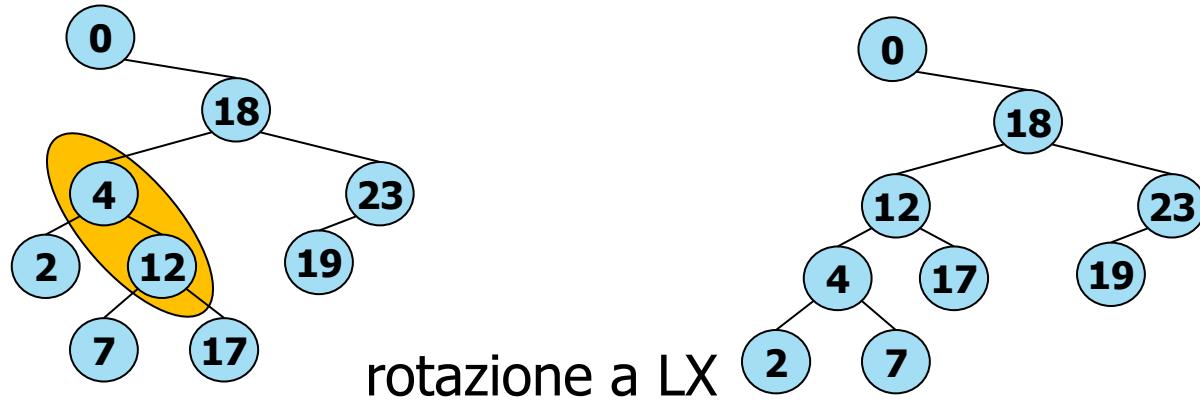


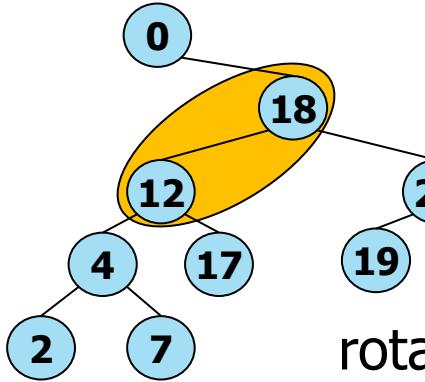


rotazione a SX

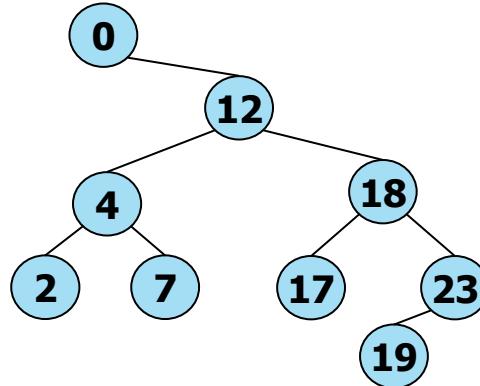


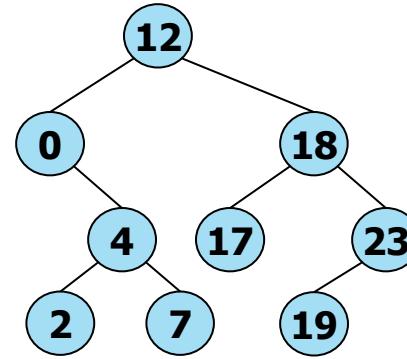
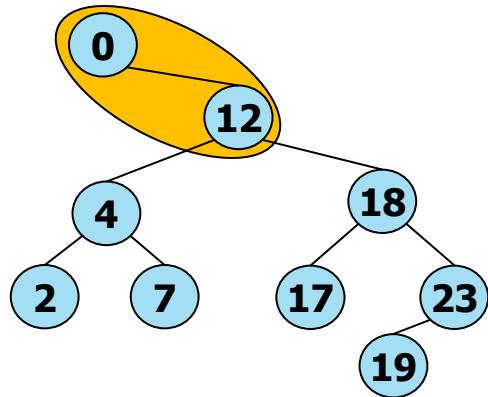






rotazione a DX





rotazione a SX

```
link partR(link h, int r) {
    int t = h->l->N;
    if (t > r) {
        h->l = partR(h->l, r);
        h = rotR(h);
    }
    if (t < r) {
        h->r = partR(h->r, r-t-1);
        h = rotL(h);
    }
    return h;
}
```

BSTdelete

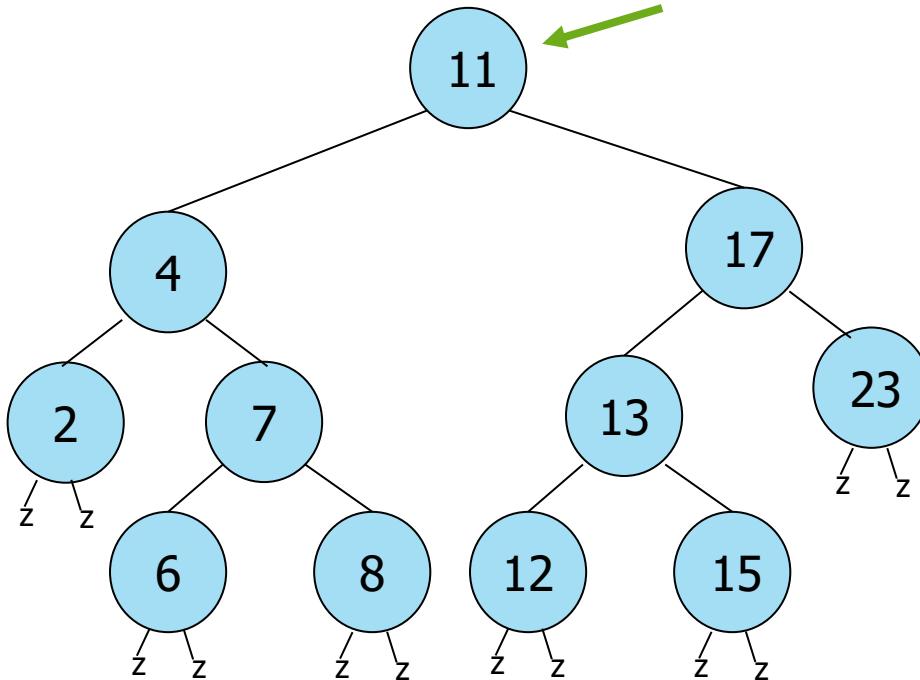
Per cancellare da un albero binario di ricerca un nodo con item con chiave k bisogna mantenere:

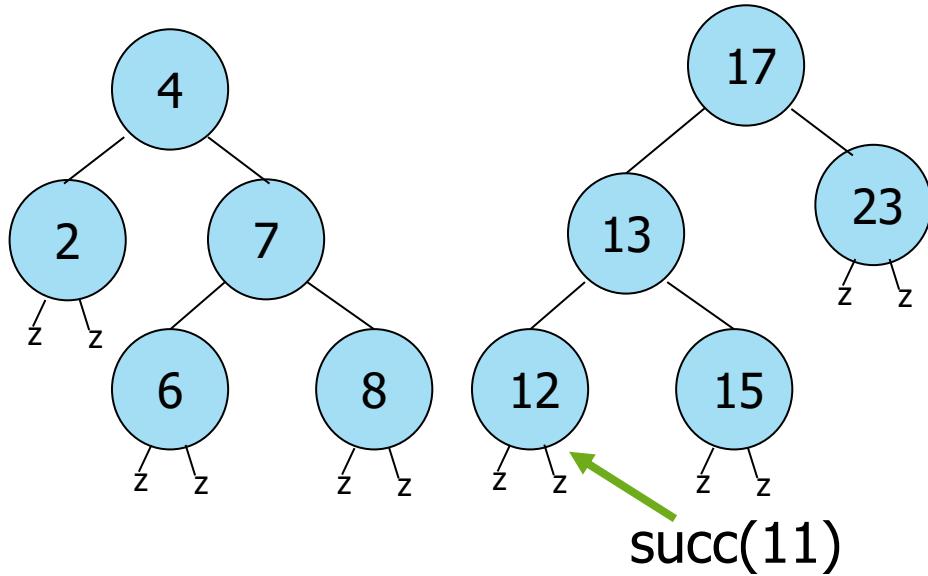
- la proprietà dei BST
- la struttura ad albero binario

Passi:

- controllare se il nodo con l'item da cancellare è in uno dei sottoalberi. Se sì, cancellazione ricorsiva nel sottoalbero
- se è la radice, eliminarlo e ricombinare i 2 sottoalberi. La nuova radice è il successore o il predecessore dell'item cancellato.

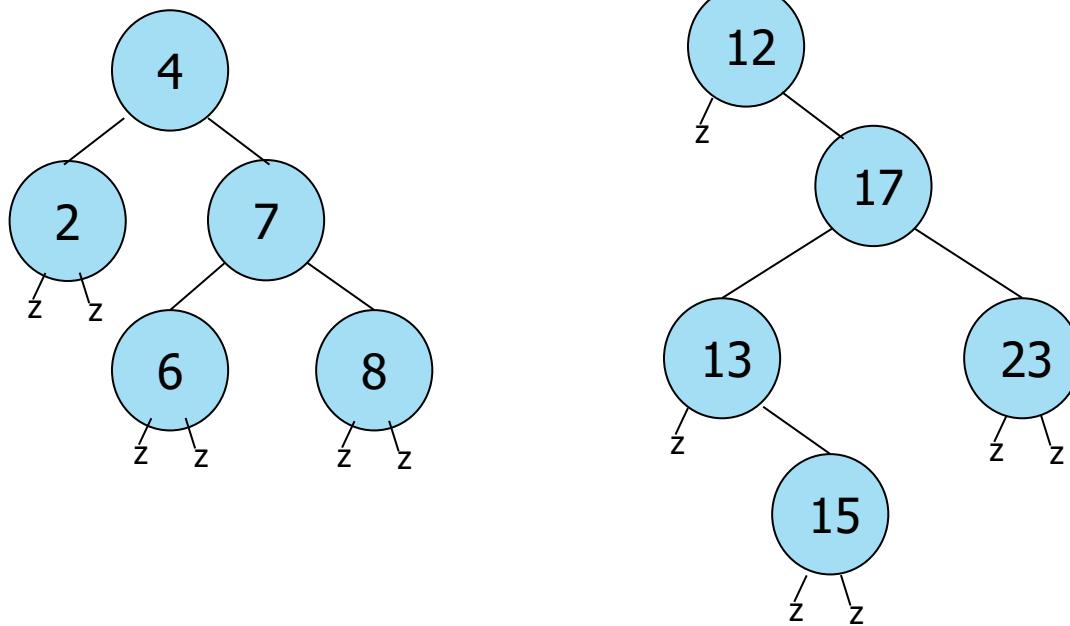
Cancellazione di una radice



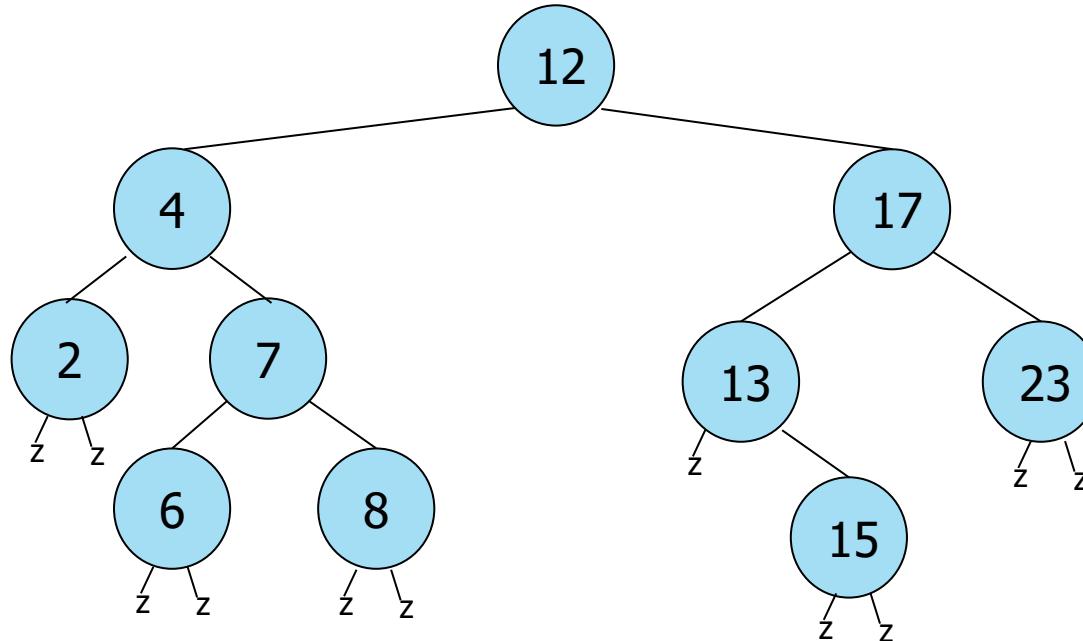


succ(11)

partition rispetto a 12



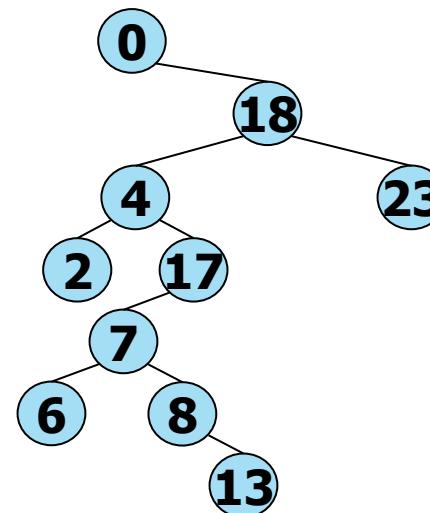
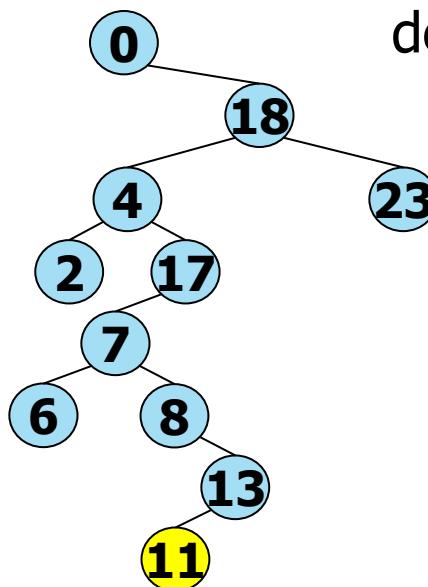
ricostruzione dell'albero con radice 12



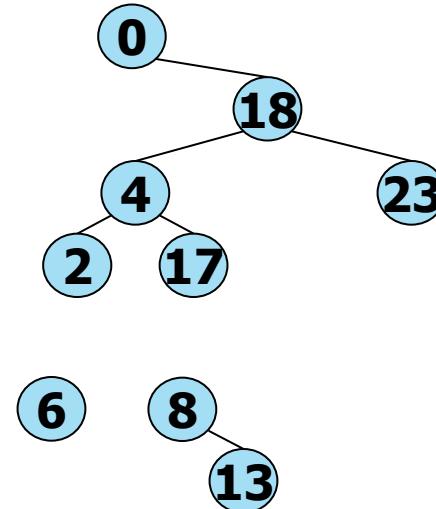
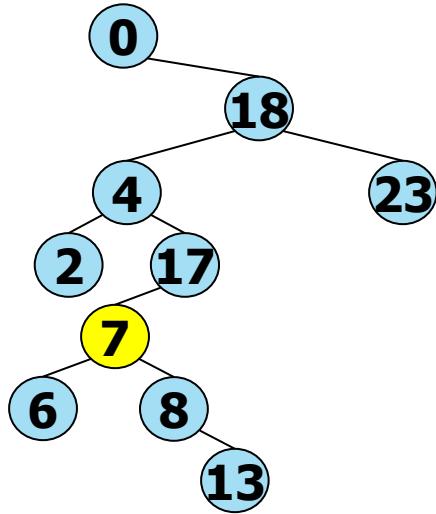
Esempio

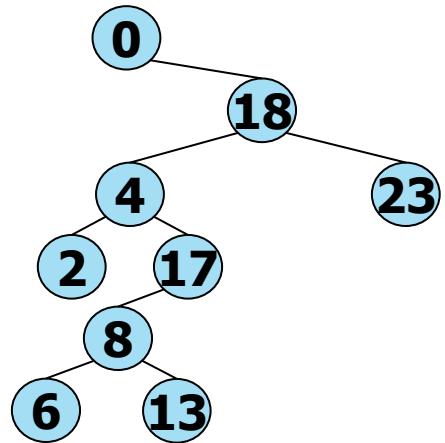
Nodi sentinella z non riportati

cancellazione in sequenza di 11, 7, 4

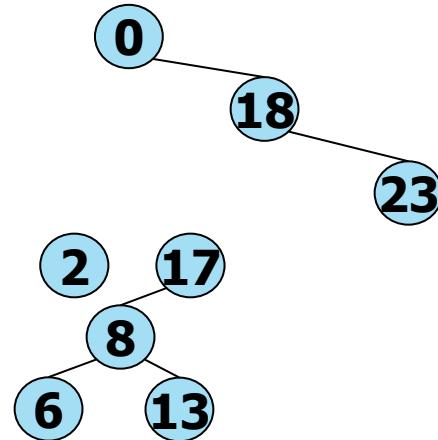
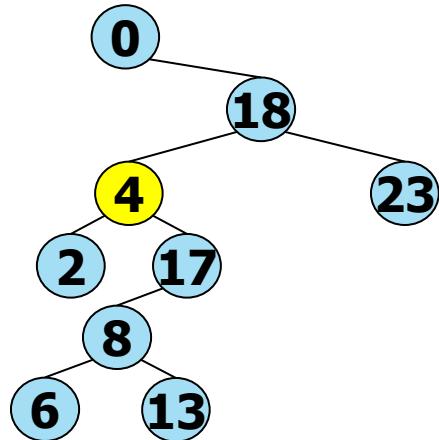


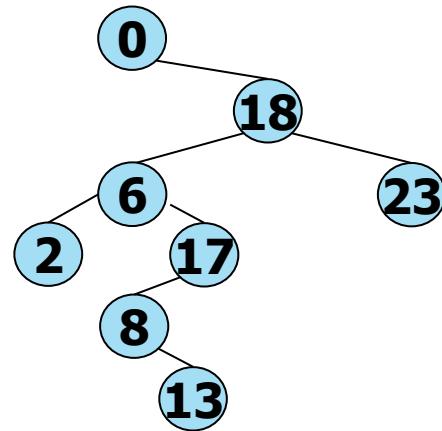
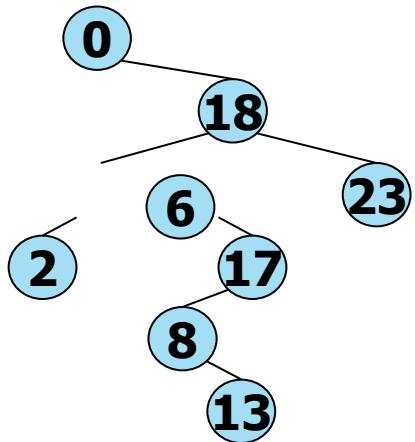
delete 7





delete 4





```
link joinLR(link a, link b, link z) {  
    if (b == z)  
        return a;  
    b = partR(b, 0);  
    b->l = a;  
    a->p = b;   
    b->N = a->N + b->r->N +1;  
    return b;  
}
```

aggiornamento
puntatore
al padre

aggiornamento
dimensione
sottoalberi

```
link deleteR(link h, key k, link z) {  
    link y, p;  
    if (h == z) return z;  
    if (KEYcmp(k, KEYget(h->item)) == -1)  
        h->l = deleteR(h->l, k, z);  
    if (KEYcmp(k, KEYget(h->item)) == 1)  
        h->r = deleteR(h->r, k, z);  
    (h->N)--;  
    if (KEYcmp(k, KEYget(h->item)) == 0) {  
        y = h;  p = h->p;  
        h = joinLR(h->l, h->r, z);  h->p = p;  
        free(y);  
    }  
    return h;  
}  
void BSTdelete(BST bst, key k) {  
    bst->root = deleteR(bst->root, k, bst->z);  
}
```

aggiornamento
numero di nodi

aggiornamento
puntatore
al padre

Bilanciamento

- A priori: vincoli che garantiscono un albero perfettamente bilanciato (B-tree) o con sbilanciamento limitato (alberi 2-3-4, RB-tree)
- Ribilanciamento a richiesta:
 - partizionamento ricorsivo attorno alla chiave mediana inferiore (algoritmo semplice)
 - algoritmo di Day, Stout e Warren, di complessità $O(n)$ costruisce un albero quasi completo (tutti i livelli completi, tranne l'ultimo riempito da sinistra a destra, cfr heap)

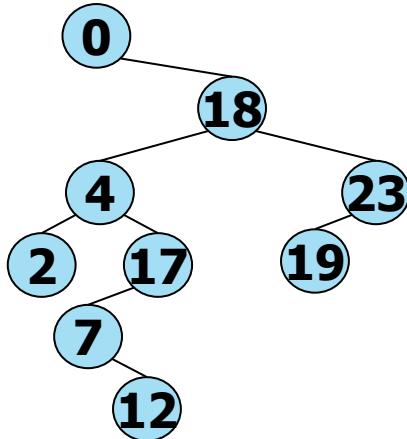
Bisogna valutare il rapporto tra costo e frequenza di ribilanciamento.

```
static link balanceR(link h, link z) {
    int r;
    if (h == z)
        return z;
    r = (h->N+1)/2-1;
    h = partR(h, r);
    h->l = balanceR(h->l, z);
    h->r = balanceR(h->r, z);
    return h;
}

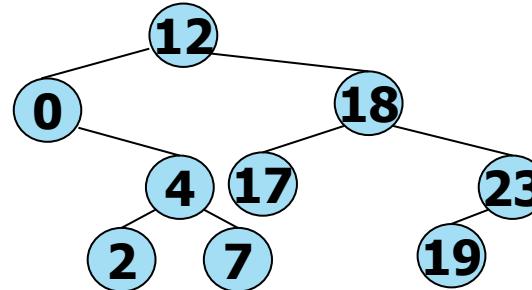
void BSTbalance(BST bst) {
    bst->root = balanceR(bst->root, bst->z);
}
```

Esempio

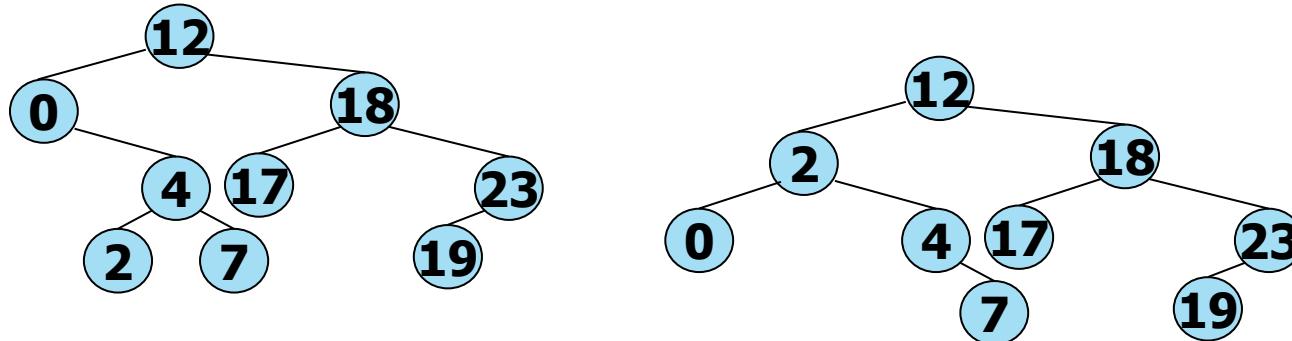
Nodi sentinella z non riportati



La mediana inferiore è 12. partR
con $r=4$ dà:

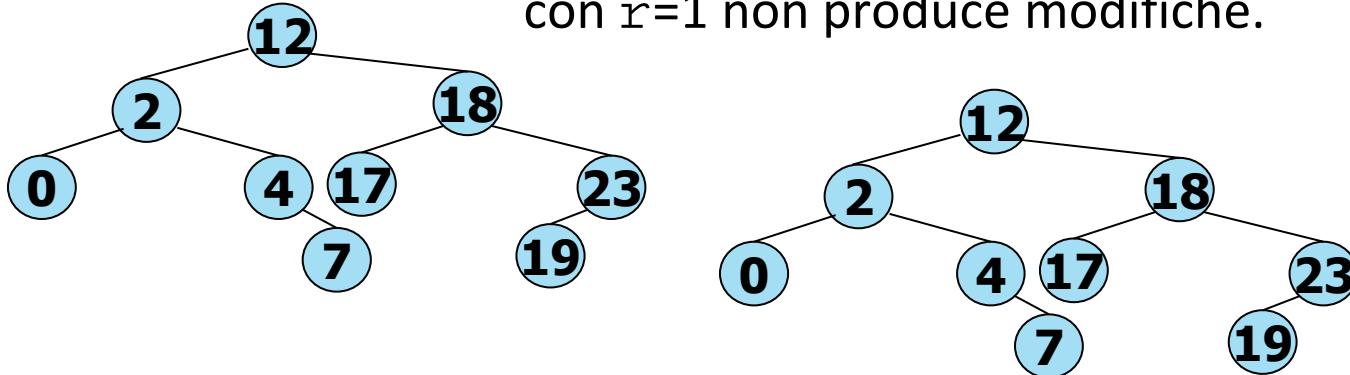


Ricorsione sul sottoalbero sinistro di 12. La mediana inferiore è 2. partR con $r=1$ dà:



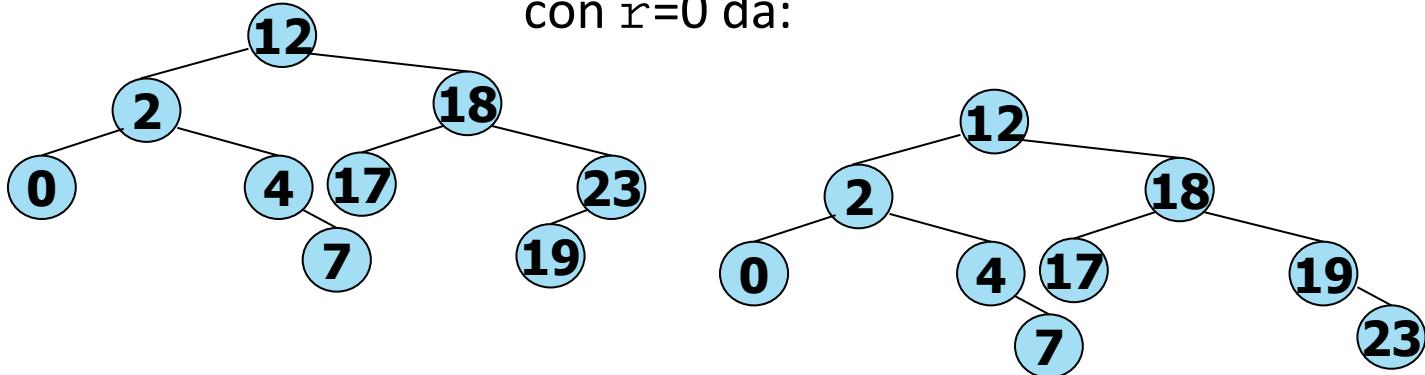
Sui sottoalberi radicati in 0, 4 e 7 la ricorsione non produce modifiche.

Ricorsione sul sottoalbero destro di 12. La mediana inferiore è 18. partR con $r=1$ non produce modifiche.



Sul sottoalbero radicato in 17 la ricorsione non produce modifiche.

Ricorsione sul sottoalbero destro di 18. La mediana inferiore è 19. partR con $r=0$ dà:



Sul sottoalbero radicato in 23 la ricorsione non produce modifiche.

Estensioni delle strutture dati

Prima di sviluppare una nuova struttura dati è bene valutare se si possano «estendere» strutture esistenti con informazioni opportune.

Procedura:

1. identificare la struttura dati candidata
2. identificare le informazioni supplementari
3. verificare di poter mantenere le informazioni supplementari senza alterare la complessità delle operazioni esistenti
4. sviluppare nuove operazioni.

Esempio: Order-Statistic BST

Procedura:

1. identificare la struttura dati candidata
2. identificare le informazioni supplementari
3. verificare di poter mantenere le informazioni supplementari senza alterare la complessità delle operazioni esistenti
4. sviluppare nuove operazioni.

BST

dimensione
del sottoalbero

$O(1)$

```
Item BSTselect(BST, int);
```

Interval BST

BST la cui chiave è un intervallo chiuso: coppia ordinata di reali $[t_1, t_2]$, dove $t_1 \leq t_2$ e $[t_1, t_2] = \{t \in \mathbb{R} : t_1 \leq t \leq t_2\}$.

L'item intervallo $[t_1, t_2]$ può essere realizzato da una **struct** con campi **low** = t_1 e **high** = t_2 .

BST con inserzione secondo
l'estremo inferiore

Procedura:

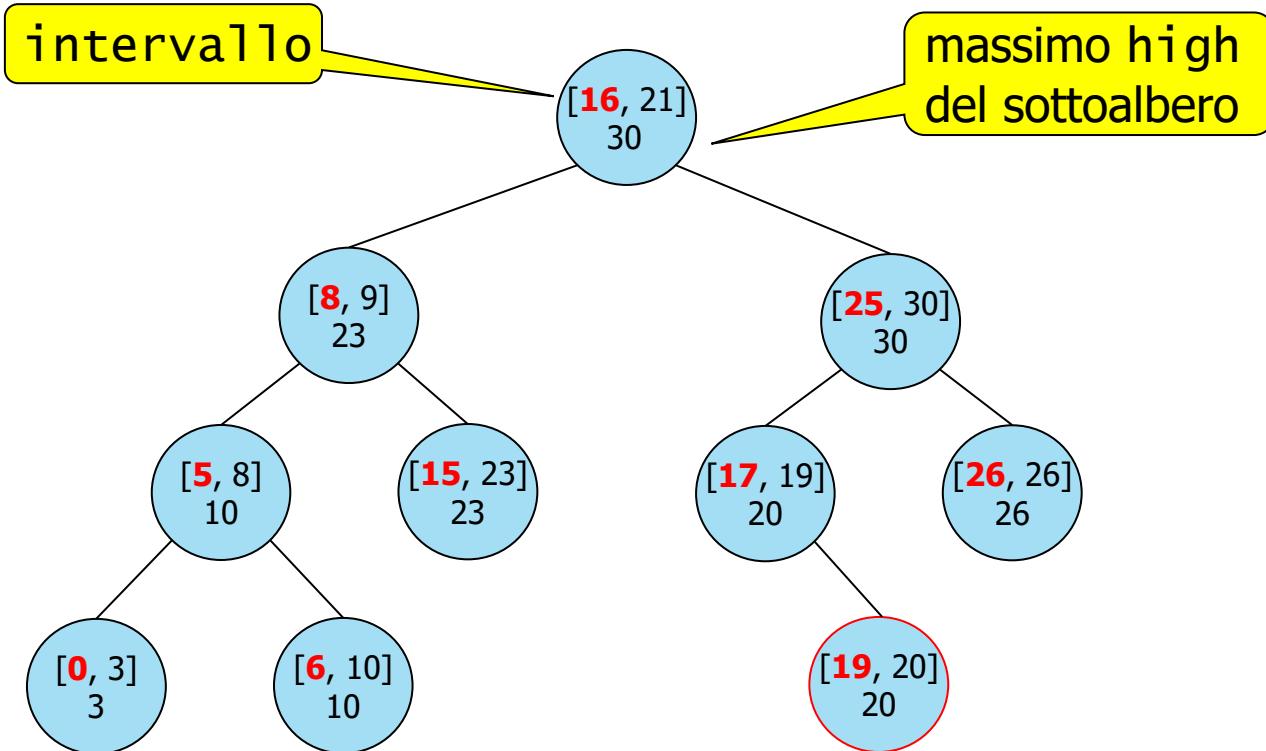
1. identificare la struttura dati candidata
2. identificare le informazioni supplementari
3. verificare di poter mantenere le informazioni supplementari
senza alterare la complessità delle operazioni esistenti
4. sviluppare nuove operazioni.

max: massimo high
del sottoalbero

O(1)

Item IBSTsearch(BST, Item);

Esempio



Riferimenti

- Alberi binari
 - Sedgewick 5.6, 5.7
- Binary Search Trees
 - Cormen 13.1, 13.2, 13.3
 - Sedgewick 12.5, 12.8, 12.9
- Order-statistic BST
 - Cormen 15.1
- Interval BST
 - Cormen 15.3
 - Lucidi di approfondimento



POLITECNICO
DI TORINO

Dipartimento
di Automatica e Informatica

Gli Interval BST

Paolo Camurati e Gianpiero Cabodi

Interval BST

Intervallo chiuso: coppia ordinata di reali $[t_1, t_2]$, dove $t_1 \leq t_2$ e $[t_1, t_2] = \{t \in \mathbb{R} : t_1 \leq t \leq t_2\}$.

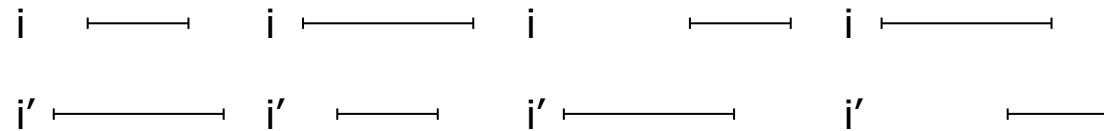
L'item intervallo $[t_1, t_2]$ può essere realizzato da una struct con campi `low = t1` e `high = t2`. Gli intervalli i e i' hanno intersezione se e solo se:

$$\text{low}[i] \leq \text{high}[i'] \quad \& \quad \text{low}[i'] \leq \text{high}[i].$$

$\forall i, i'$ vale la seguente tricotomia:

- a. i e i' hanno intersezione
- b. $\text{high}[i] \leq \text{low}[i']$
- c. $\text{high}[i'] \leq \text{low}[i]$

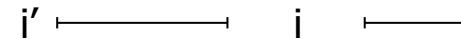
caso a



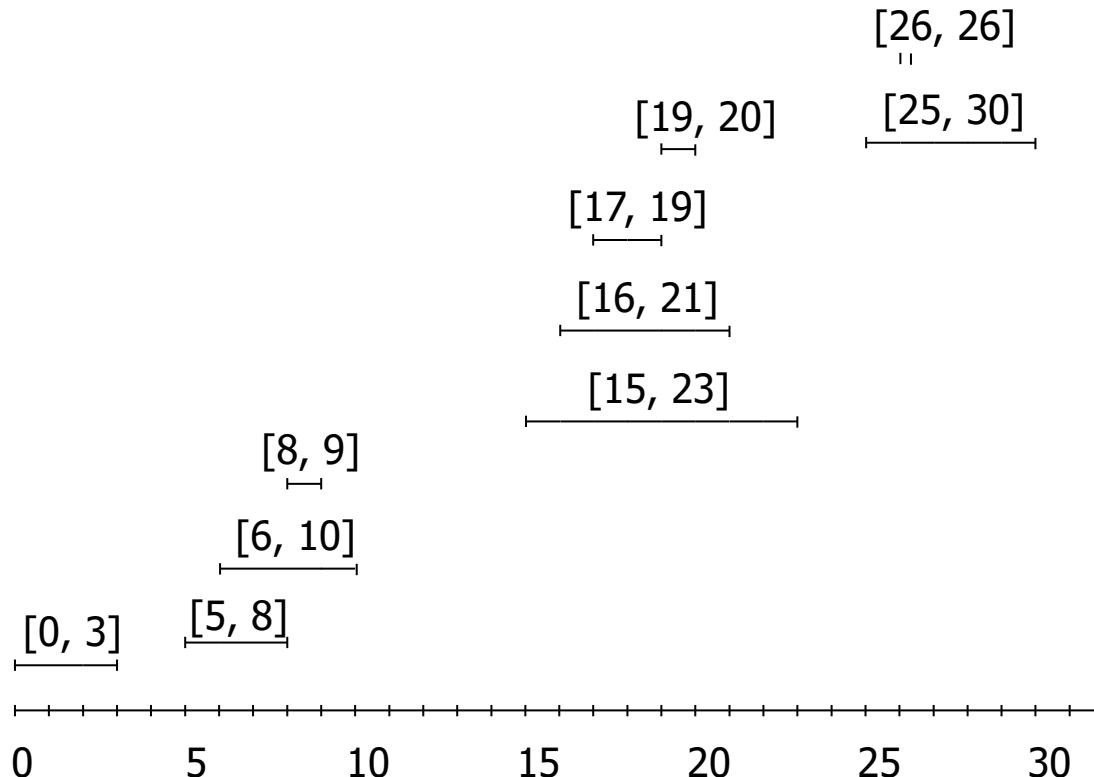
caso b



caso c



Esempio



Procedura

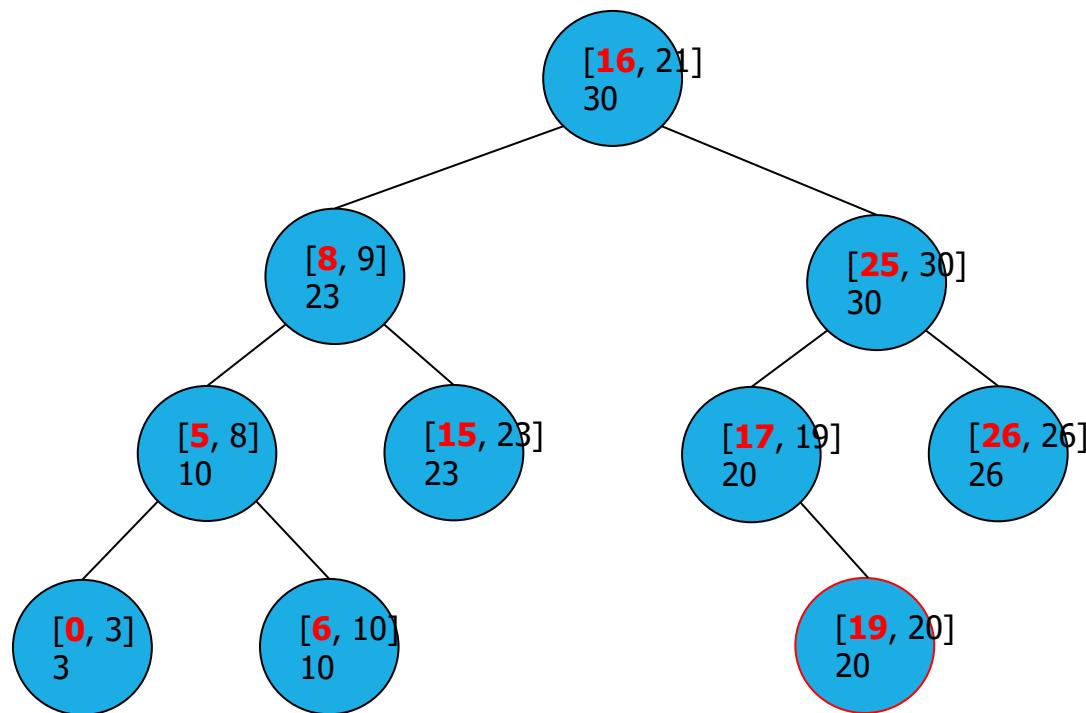
1. identificare la struttura dati candidata
2. identificare le informazioni supplementari
3. verificare di poter mantenere le informazioni supplementari senza alterare la complessità delle operazioni esistenti
4. sviluppare nuove operazioni.

BST con inserzione secondo l'estremo inferiore

max: massimo high del sottoalbero

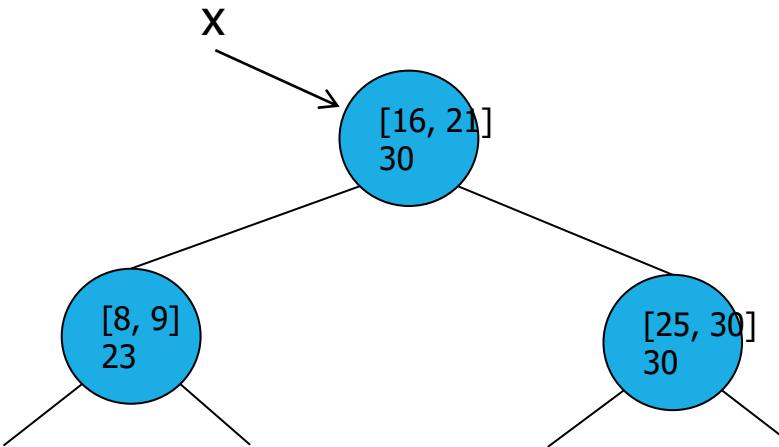
```
Item IBSTsearch(BST, Item);
```

$O(1)$



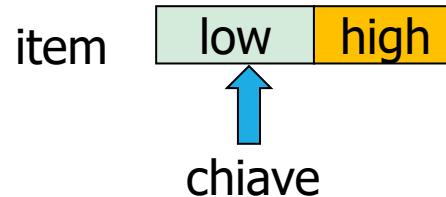
Il calcolo di max è di complessità $\Theta(1)$:

$$x\rightarrow\max = \max (\text{high}(x), x\rightarrow\text{left}\rightarrow\max, x\rightarrow\text{right}\rightarrow\max)$$



Quasi ADT Item

Tipologia 1



Item.h

```
typedef struct item { int low; int high; } Item;
```

```
Item    ITEMscan();
Item    ITEMsetVoid();
int     ITEMcheckVoid(Item val),
void   ITEMstore(Item val);
int     ITEMhigh();
int     ITEMlow();
int     ITEMoverlap(Item val1, Item val2);
int     ITEMeq(Item val1, Item val2);
int     ITEMlt(Item val1, Item val2);
int     ITEMlt_int(Item val1, int val2);
```

estremo superiore

estremo inferiore

intersezione

= in inserzione

< in inserzione

< in ricerca

Item.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "Item.h"

Item ITEMscan() {
    Item val;
    printf("low = "); scanf("%d", &val.low);
    printf("high = "); scanf("%d", &val.high);
    return val;
}

void ITEMstore(Item val) {
    printf("[%d, %d] ", val.low, val.high);
}

Item ITEMsetVoid() {
    Item val = {-1, -1};
    return val;
}
```

```
int ITEMcheckvoid(Item val) {
    if ((val.low == -1) && (val.high == -1))
        return 1;
    return 0;
}

int ITEMhigh(Item val) { return val.high; }

int ITEMlow(Item val) { return val.low; }

int ITEMoverlap(Item val1, Item val2) {
    if ((val1.low <= val2.high) && (val2.low<= val1.high))
        return 1;
    return 0;
}
```

```
int ITEMeq(Item val1, Item val2) {
    if ((val1.low == val2.low) && (val1.high == val2.high))
        return 1;
    return 0;
}

int ITEMlt(Item val1, Item val2) {
    if ((val1.low < val2.low))
        return 1;
    return 0;
}

int ITEMlt_int(Item val1, int val2) {
    if ((val1.low < val2))
        return 1;
    return 0;
}
```

Operazioni

Item IBSTsearch(IBST ibst , Item x);

cerca un item (intervallo) nell'Interval BST e ritorna il primo intervallo che lo interseca

void IBSTinsert(IBST ibst , Item x);

inserisci un item (intervallo) nell'Interval BST

void IBSTdelete(IBST ibst , Item x);

cancella un item (intervallo) dall'Interval BST

ADT di classe Interval BST

IBST.h

```
typedef struct intervalbinarysearchtree *IBST;

void IBSTinit(IBST ibst);
void IBSTfree(IBST ibst);
void BSTinsert(IBST ibst, Item x);
void IBSTdelete(IBST ibst, Item x);
Item IBSTsearch(IBST ibst, Item x);
int IBSTcount(IBST ibst);
int IBSTempty(IBST ibst);
void IBSTvisit(IBST ibst, int strategy);
```

nuove funzioni
funzioni modificate

IBST.c

```
#include <stdlib.h>
#include <stdio.h>
#include "Item.h"
#include "IBST.h"

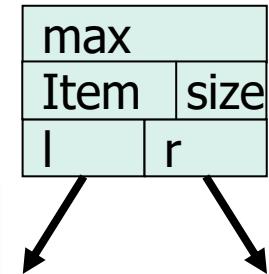
typedef struct IBSTnode *link;

struct IBSTnode {Item item; link l, r; int N; int max;};

struct intervalbinarysearchtree {link root;int size;link z;};

static link NEW(Item item, link l, link r, int N, int max) {
    link x = malloc(sizeof *x);
    x->item = item;
    x->l = l; x->r = r; x->N = N;
    x->max = max;
    return x;
}
```

IBSTnode



max: massimo high
del sottoalbero

size: dimensione
del sottoalbero

```

static void NODEshow(link x) {
    ITEMstore(x->item); printf("max = %d\n", x->max);
}
IBST IBSTinit( ) {
    IBST ibst = malloc(sizeof *ibst) ;
    ibst->N = 0;
    ibst->root=(ibst->z=NEW(ITEMsetNull(),NULL,NULL,0,-1));
    return ibst;
}
void IBSTfree(IBST ibst) {
    if (ibst == NULL) return;
    treeFree(ibst->root, ibst->z);
    free(ibst->z); free(ibst);
}
static void treeFree(link h, link z) {
    if (h == z) return;
    treeFree(h->l, z); treeFree(h->r, z);
    free(h);
}

```

```
int IBSTcount(IBST ibst) { return ibst->size; }
int IBSTempty(IBST ibst) {
    if (IBSTcount(ibst) == 0) return 1;
    return 0;
}
static void treePrintR(link h, link z, int strategy) {
    if (h == z) return;
    if (strategy == PREORDER)
        NODEshow(h);
    treePrintR(h->l, z, strategy);
    if (strategy == INORDER)
        NODEshow(h);
    treePrintR(h->r, z, strategy);
    if (strategy == POSTORDER)
        NODEshow(h);
}
void IBSTvisit(IBST ibst, int strategy) {
    if (IBSTempty(ibst)) return;
    treePrintR(ibst->root, ibst->z, strategy);
}
```

Insert

```
link insertR(link h, Item item, link z) {
    if (h == z)
        return NEW(item, z, z, 1, ITEMhigh(item));
    if (ITEMlt(item, h->item)) {
        h->l = insertR(h->l, item, z);
        h->max = max(h->max, h->l->max, h->r->max);
    }
    else {
        h->r = insertR(h->r, item, z);
        h->max = max(h->max, h->l->max, h->r->max);
    }
    (h->N)++;
    return h;
}

void IBSTinsert(IBST ibst, Item item) {
    ibst->root = insertR(ibst->root, item, ibst->z);
    ibst->size++;
}
```

rotL/rotR

```
link rotL(link h) {
    link x = h->r;
    h->r = x->l;
    x->l = h;
    x->N = h->N;
    h->N = h->l->N + h->r->N +1;
    h->max = max(ITEMhigh(h->item), h->l->max, h->r->max);
    x->max = max(ITEMhigh(x->item), x->l->max, x->r->max);
    return x;
}
link rotR(link h) {
    link x = h->l;
    h->l = x->r;
    x->r = h;
    x->N = h->N;
    h->N = h->r->N + h->l->N +1;
    h->max = max(ITEMhigh(h->item), h->l->max, h->r->max);
    x->max = max(ITEMhigh(x->item), x->l->max, x->r->max);
    return x;
```

partR/joinLR

```
link partR(link h, int r) {
    int t = h->l->N;
    if (t > r) {
        h->l = partR(h->l, r);
        h = rotR(h);
    }
    if (t < r) {
        h->r = partR(h->r, r-t-1);
        h = rotL(h);
    }
    return h;
}
link joinLR(link a, link b, link z) {
    if (b == z) return a;
    b = partR(b, 0);
    b->l = a;
    b->N = a->N + b->r->N +1;
b->max = max(ITEMhigh(b->item), a->max, b->r->max);
    return b;
}
```

Delete

```
link deleteR(link h, Item item, link z) {
    link x;
    if (h == z) return z;
    if (ITEMlt(item, h->item)) {
        h->l = deleteR(h->l, item, z);
        h->max = max(ITEMhigh(h->item), h->l->max, h->r->max);
    }
    if (ITEMlt(h->item, item)) {
        h->r = deleteR(h->r, item, z);
        h->max = max(ITEMhigh(h->item), h->l->max, h->r->max);
    }
    (h->N)--;
    if (ITEMeq(item, h->item)) {
        x = h; h = joinLR(h->l, h->r, z); free(x);
    }
    return h;
}
void IBSTdelete(IBST ibst, Item item) {
    ibst->root=deleteR(ibst->root,item,ibst->z);
    ibst->size--;
}
```

Search

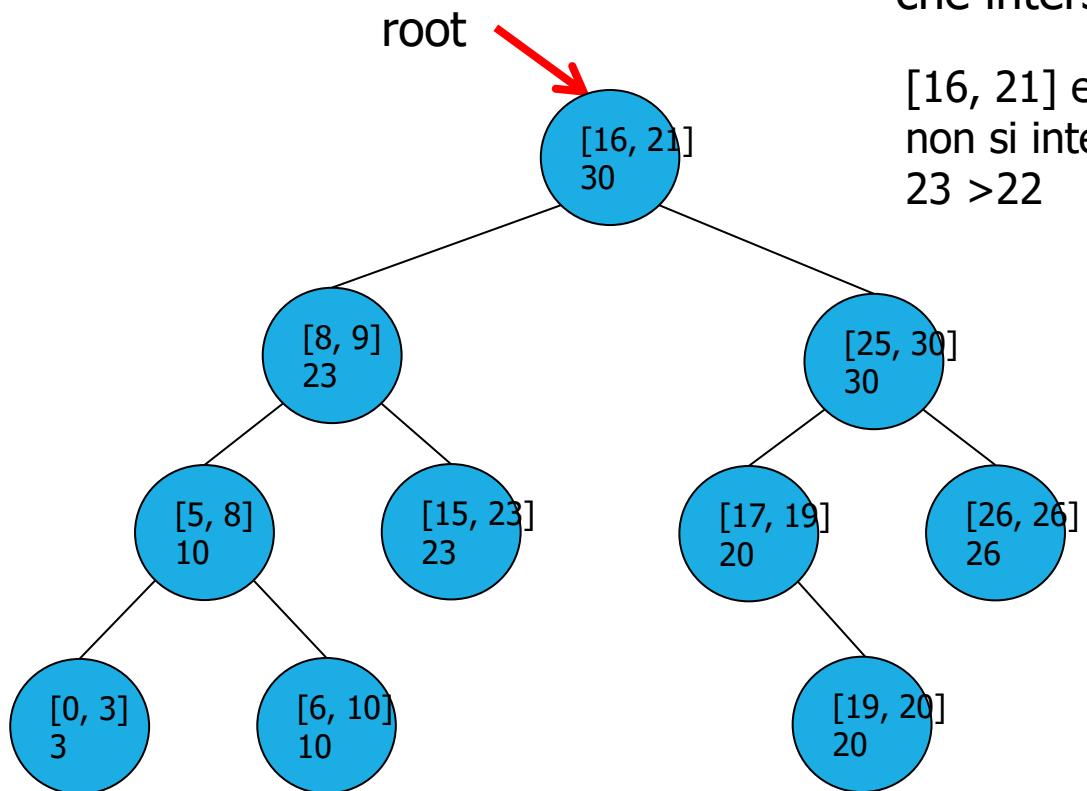
Ricerca di un nodo h con intervallo che interseca l'intervallo i :

- percorrimento dell'albero dalla radice
- terminazione: trovato intervallo che interseca i oppure si è giunti ad un albero vuoto
- ricorsione: dal nodo h
 - su sottoalbero sinistro se
 $h->l->\max \geq low[i]$
 - su sottoalbero destro se
 $h->l->\max < low[i]$

```
Item searchR(link h, Item item, link z) {
    if (h == z)
        return ITEMsetNull();
    if (ITEMoverlap(item, h->item))
        return h->item;
    if (ITEMlt_int(item, h->l->max))
        return searchR(h->l, item, z);
    else
        return searchR(h->r, item, z);
}

Item IBSTsearch(IBST ibst, Item item) {
    return searchR(ibst->root, item, ibst->z);
```

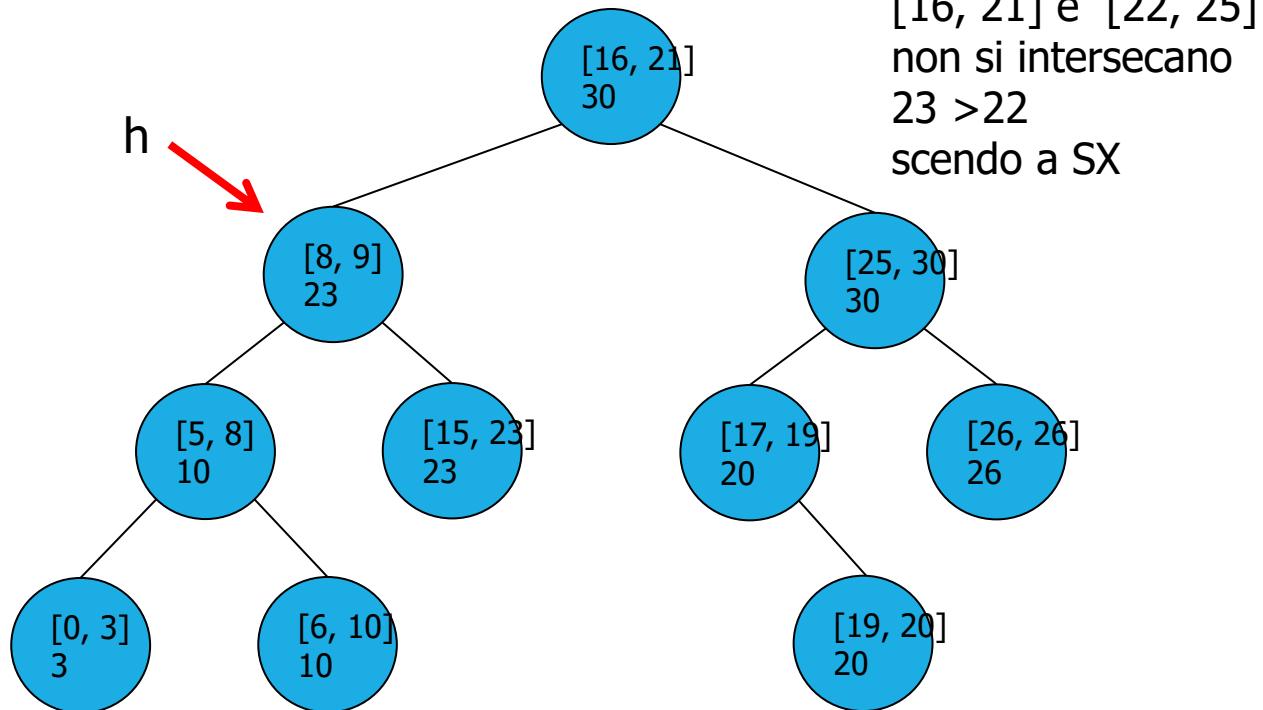
Esempio



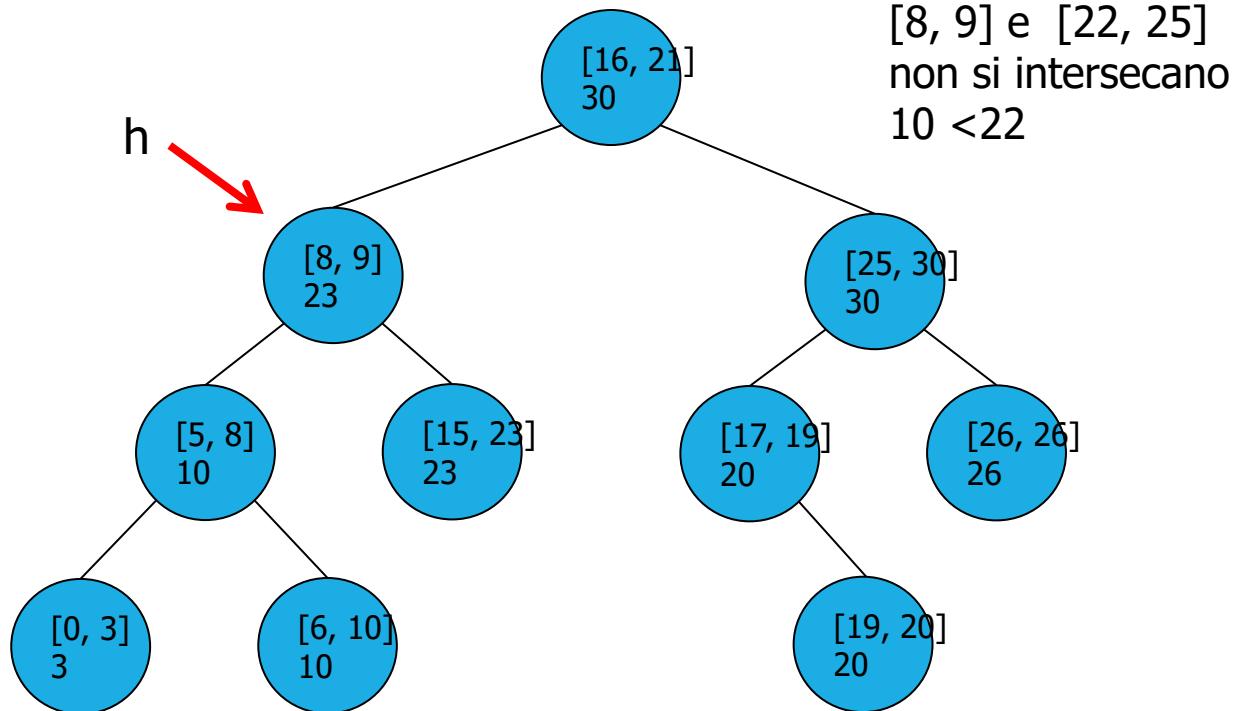
Ricerca di un intervallo
che interseca $[22, 25]$

$[16, 21]$ e $[22, 25]$
non si intersecano
 $23 > 22$

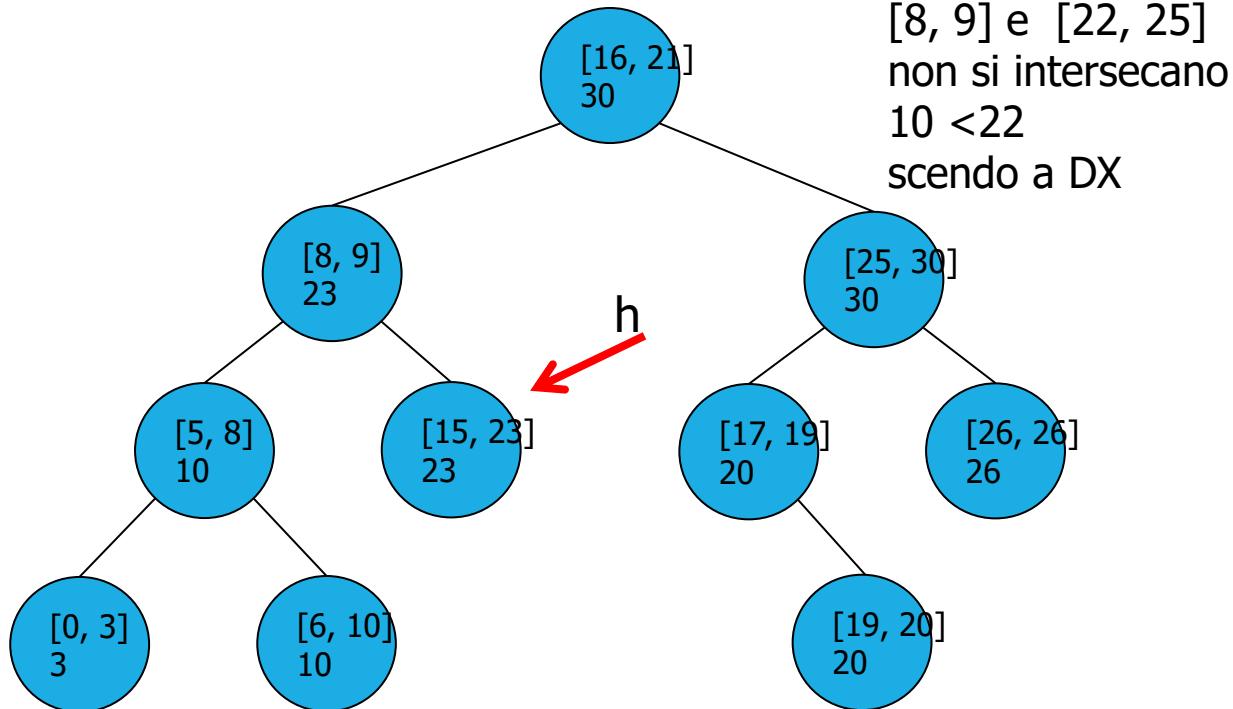
Ricerca di un intervallo
che interseca [22, 25]



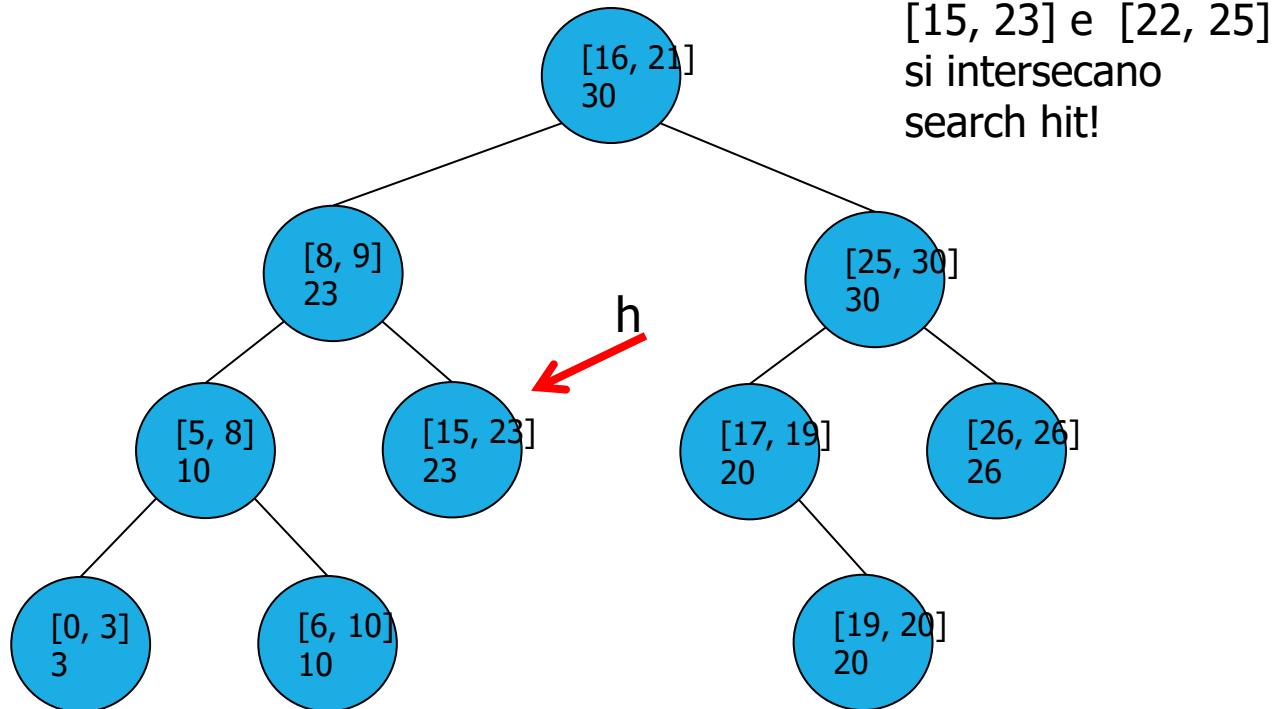
Ricerca di un intervallo
che interseca $[22, 25]$



Ricerca di un intervallo
che interseca [22, 25]

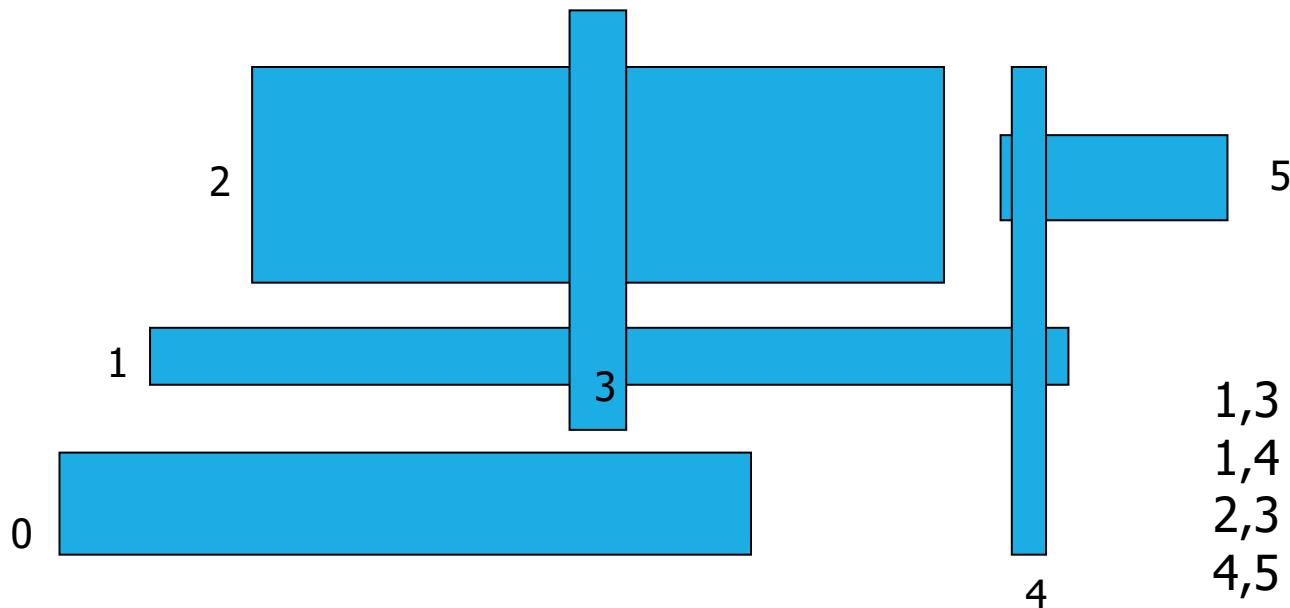


Ricerca di un intervallo
che interseca [22, 25]



Applicazioni degli I-BST

Dati N rettangoli disposti parallelamente agli assi ortogonali, determinare tutte le coppie che si intersecano:



Applicazione al CAD elettronico

Verificare se le piste si intersecano in un circuito elettronico.

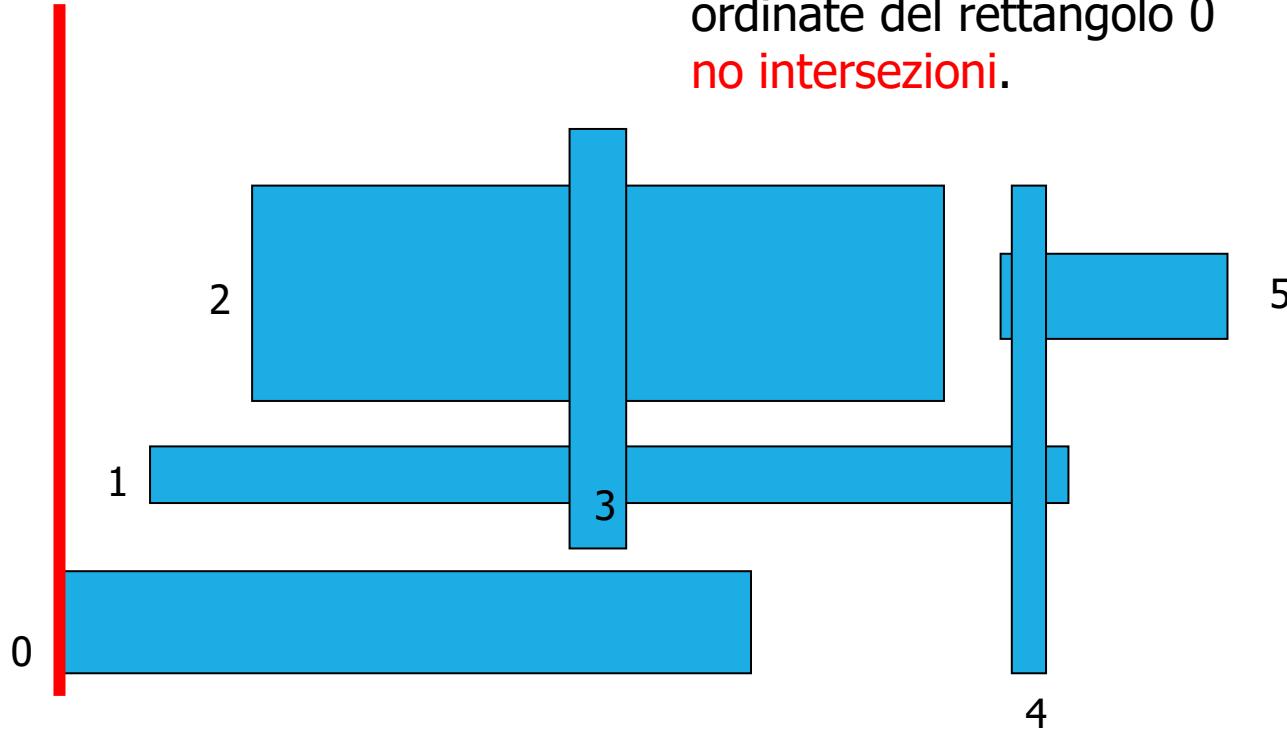
Algoritmo banale: controllare l'intersezione tra tutte le coppie di rettangoli, complessità $O(N^2)$.

Algoritmo efficiente

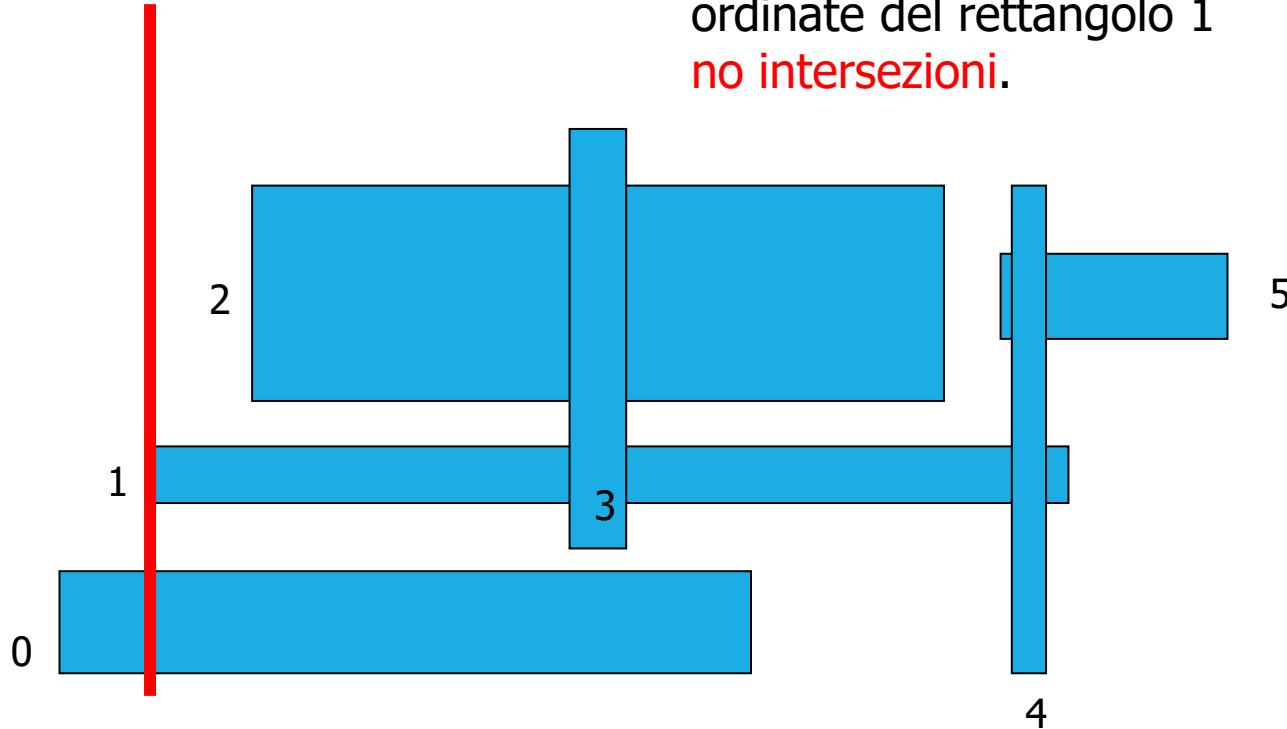
Complessità $O(N \log N)$, applicabilità a VLSI e oltre:

- ordina i rettangoli per ascisse dell'estremo sinistro crescenti
- itera sui rettangoli per ascisse crescenti:
 - quando incontri l'estremo sinistro, inserisci in un I-BST l'intervallo delle ordinate e controlla l'intersezione
 - quando incontri l'estremo destro, rimuovi l'intervallo delle ordinate dall'I-BST.

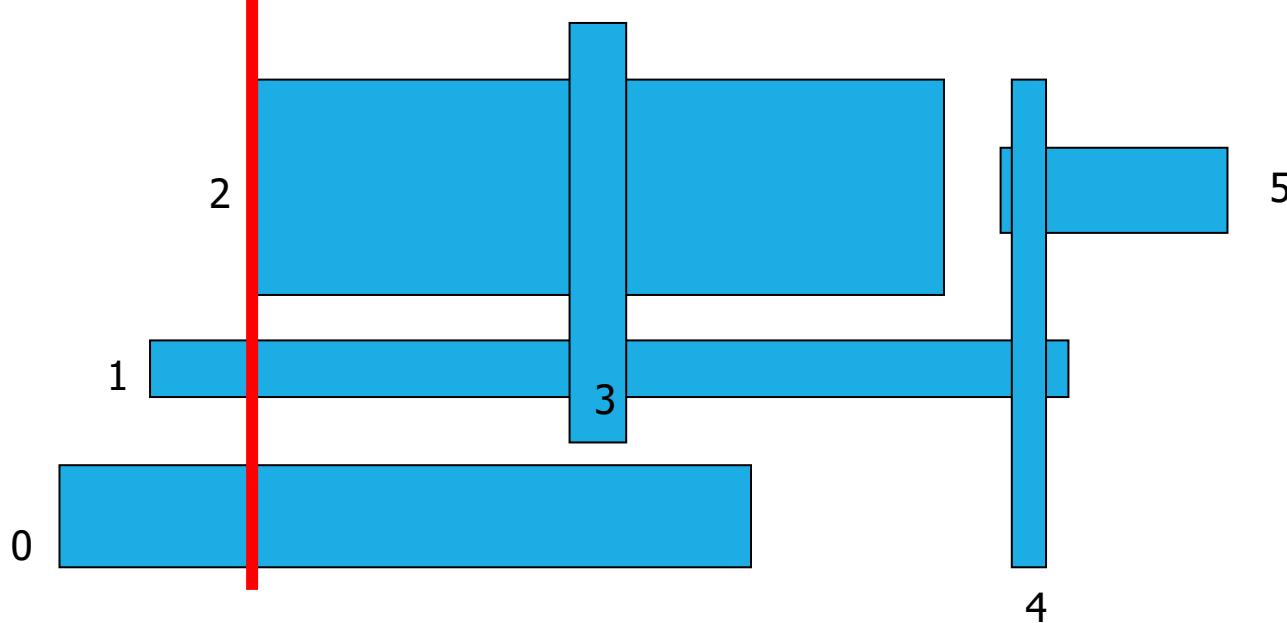
inserisci
intervallo
ordinate del rettangolo 0
no intersezioni.



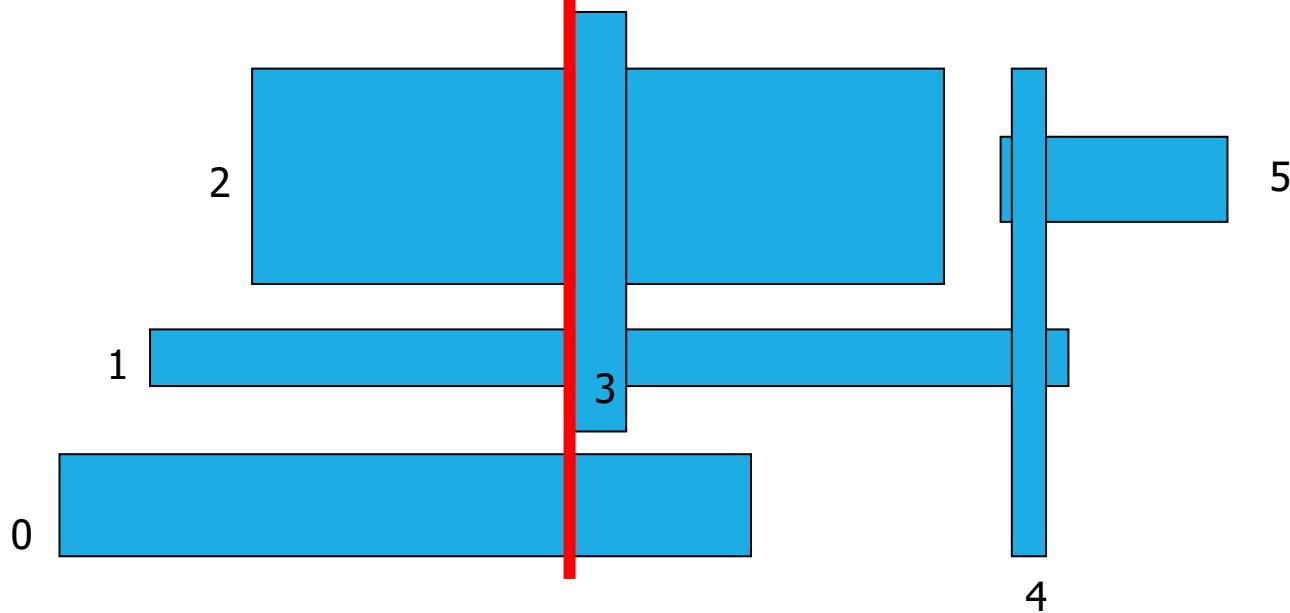
inserisci
intervallo
ordinate del rettangolo 1
no intersezioni.



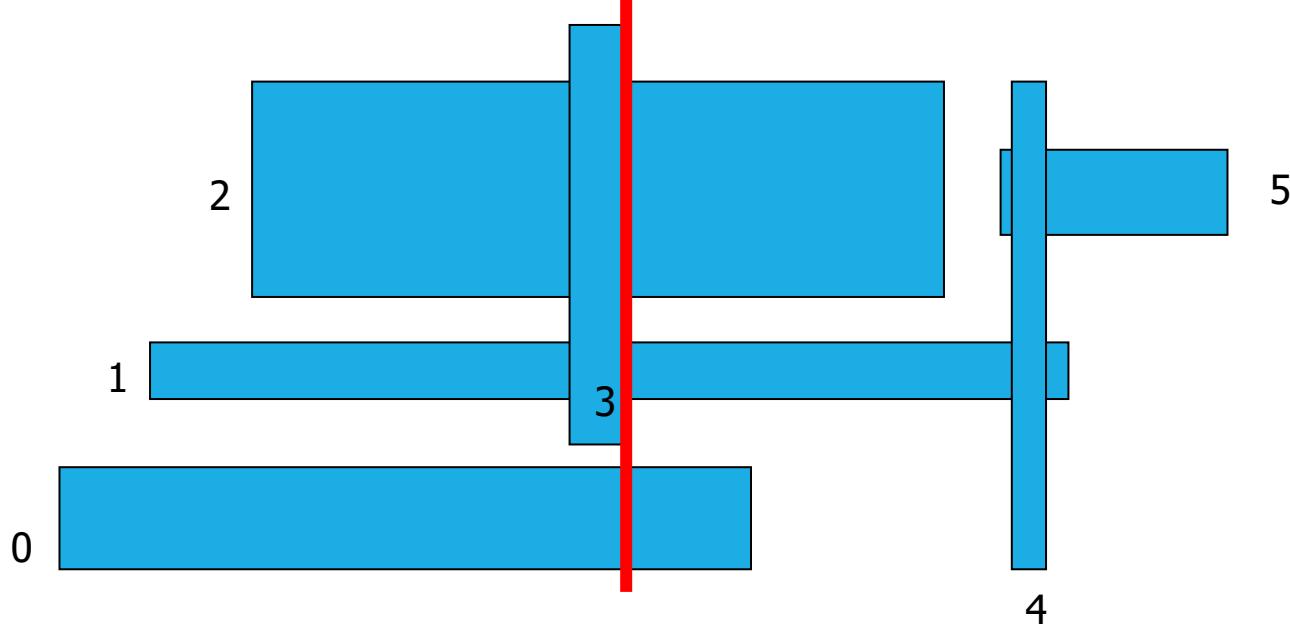
inserisci
intervallo
ordinate del rettangolo 2
no intersezioni.



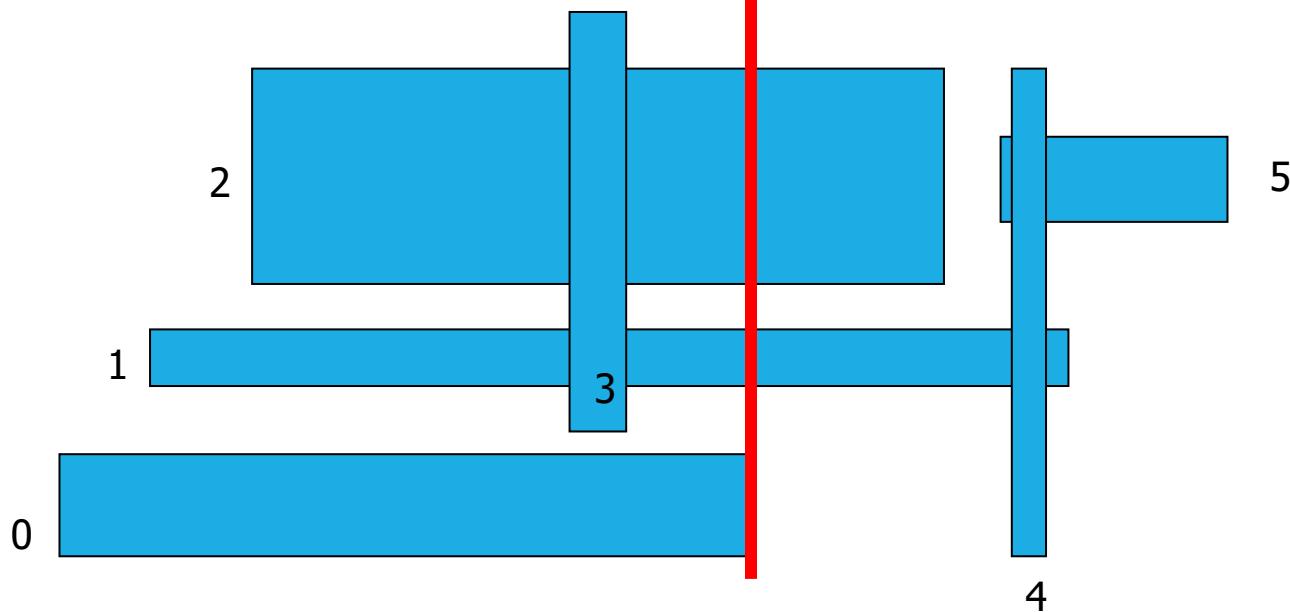
inserisci intervallo
ordinate del rettangolo 3
intersezioni con 1 e 2.



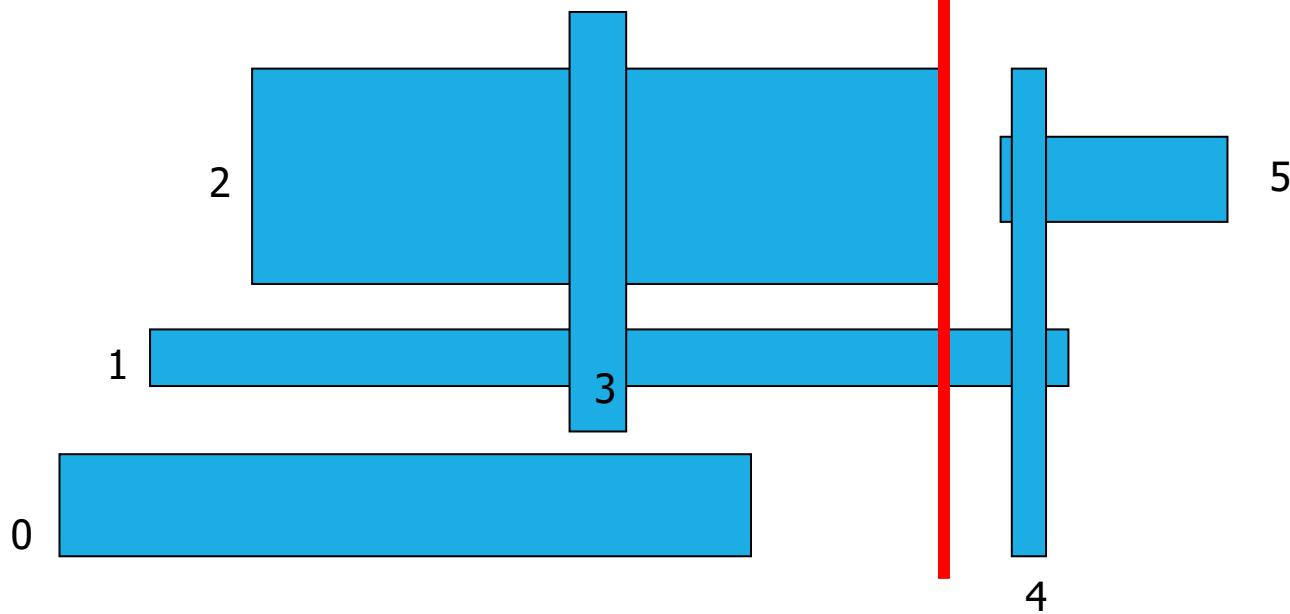
rimuovi
intervallo
ordinate del rettangolo 3.



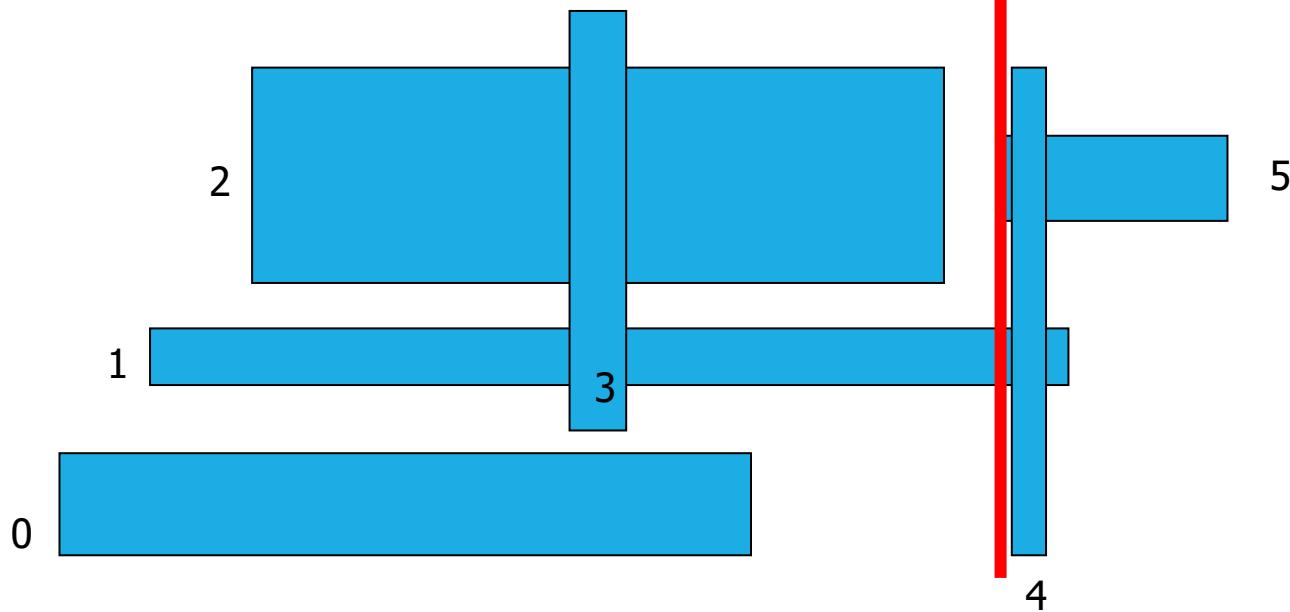
rimuovi
intervallo
ordinate del rettangolo 0.



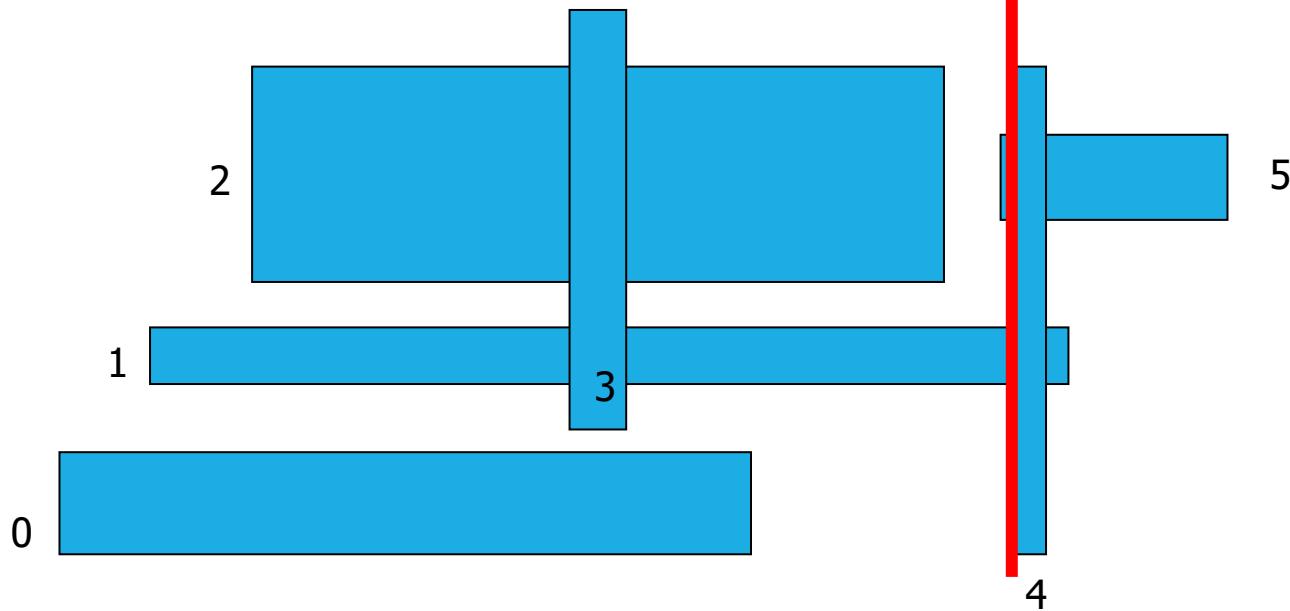
rimuovi
intervallo
ordinate del rettangolo 2.



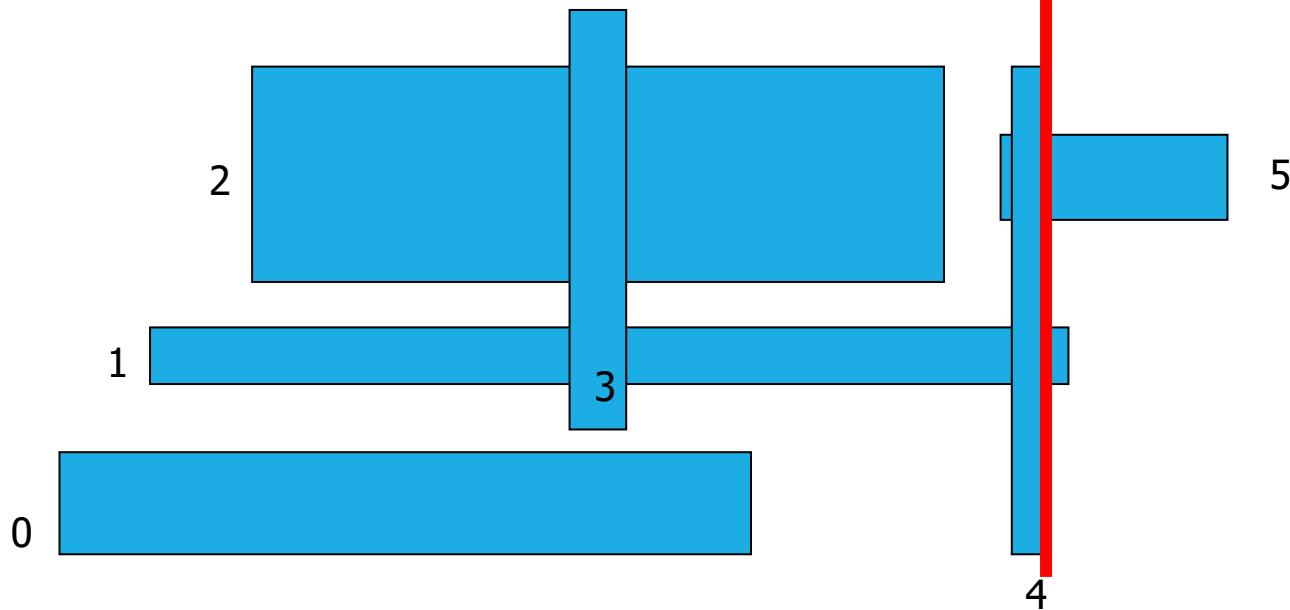
inserisci
intervallo
ordinate del rettangolo 5
no intersezioni.



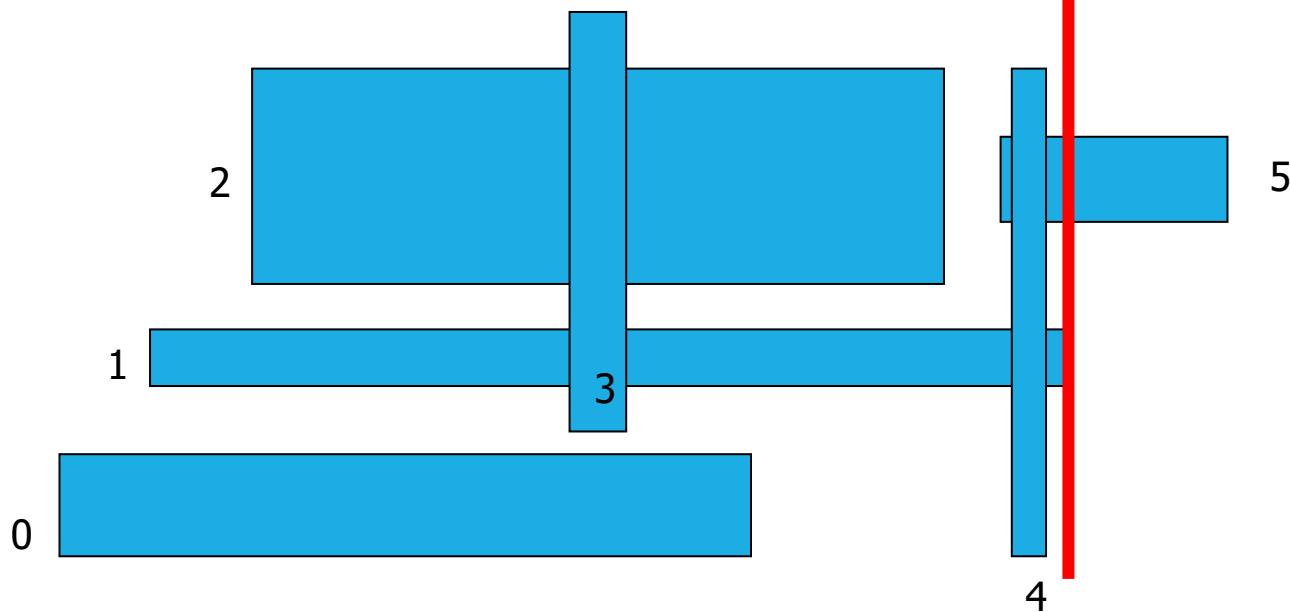
inserisci
intervallo
ordinate del rettangolo 4
intersezioni con 1 e 5.



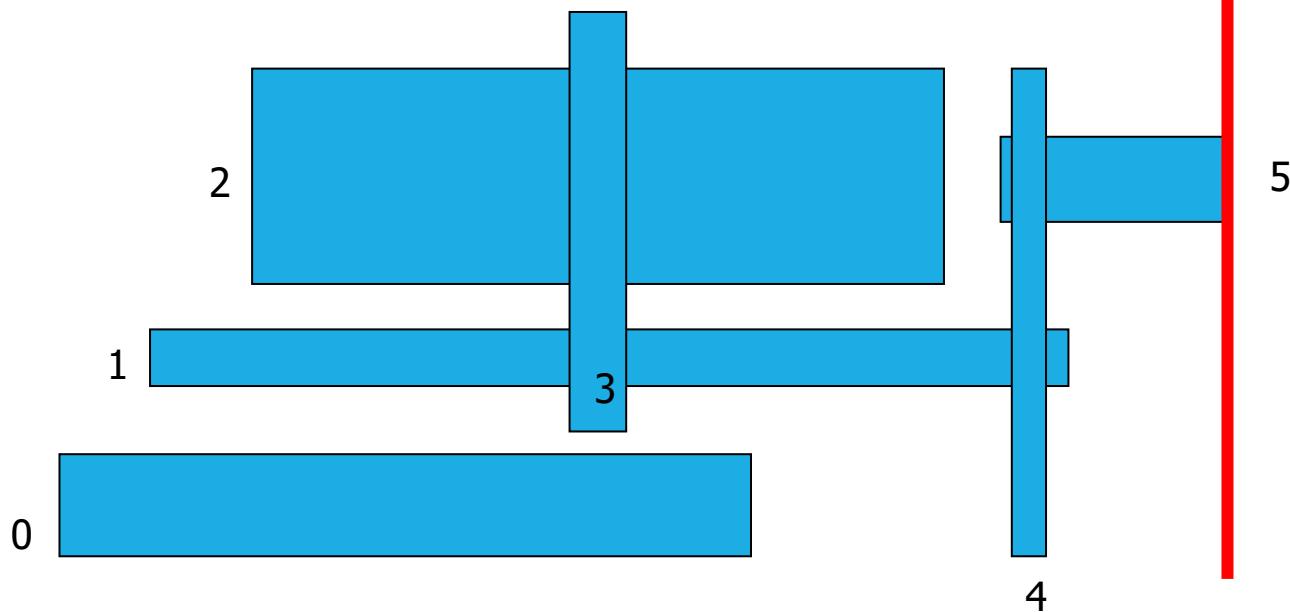
rimuovi
intervallo
ordinate del rettangolo 4.



rimuovi
intervallo
ordinate del rettangolo 1.



rimuovi
intervallo
ordinate del rettangolo 5.



Analisi

Ordinamento: $T(n) = O(N \log N)$

Se l'IBST è bilanciato:

- ogni inserzione/cancellazione di intervallo o ricerca del primo intervallo che interseca uno dato costa $T(n) = O(\log N)$,
- la ricerca di tutti gli intervalli che intersecano un intervallo dato costa $T(n) = O(R \log N)$ se R è il numero di intersezioni.

Riferimenti

- Alberi binari
 - Sedgewick 5.6, 5.7
- Binary Search Tree
 - Cormen 13.1, 13.2, 13.3
 - Sedgewick 12.5, 12.8, 12.9
- Order-statistic BST
 - Cormen 15.1
- Interval BST
 - Cormen 15.3



POLITECNICO
DI TORINO

Dipartimento
di Automatica e Informatica

Le tabelle di hash

Gianpiero Cabodi e Paolo Camurati

Tabelle di hash

Finora gli algoritmi di ricerca si erano basati sul confronto.

Eccezione: tavelle ad accesso diretto dove la chiave $k \in U = \{0, 1, \dots, \text{card}(U)-1\}$ funge da **indice** di un array $st[0, 1, \dots, \text{card}(U)-1]$.

Limiti delle tavelle ad accesso diretto:

- $|U|$ grande (vettore st non allocabile)
- $|K| << |U|$ (spreco di memoria).

Tabella di hash: è un ADT con occupazione di spazio $O(|K|)$ e tempo medio di accesso $O(1)$.

La funzione di **hash** trasforma la chiave di ricerca in un indice della tabella.

La funzione di hash non può essere perfetta  **collisione**.

Usate per inserzione, ricerca, cancellazione, non per ordinamento e selezione.

ADT di classe ST

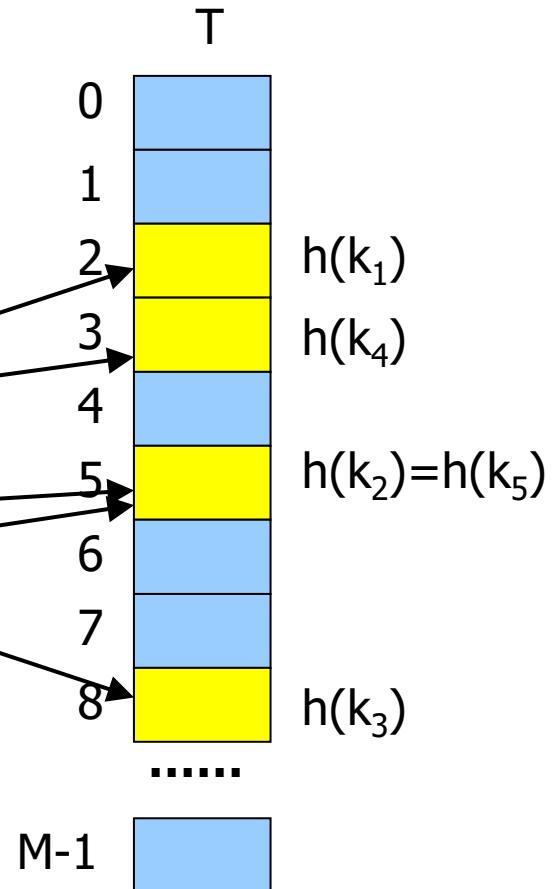
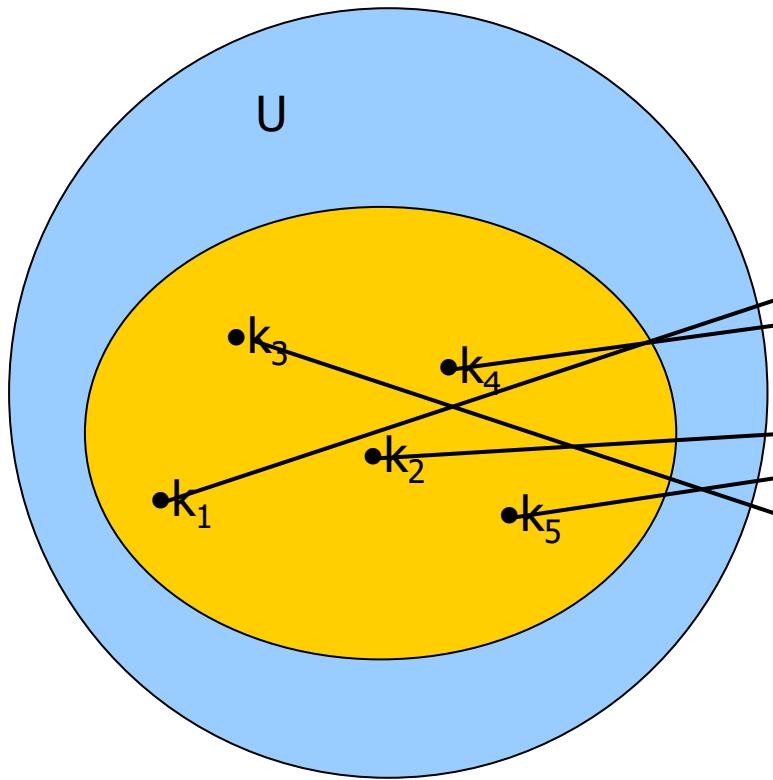
ST.h

```
typedef struct symboltable *ST;

ST      STinit(int maxN, float r) ;
void    STinsert(ST st, Item val);
Item   STsearch(ST st, Key k) ;
void   STdelete(ST st, Key k) ;
void   STdisplay(ST st) ;
void   STfree(ST st);
int    STcount(ST st);
int    STempty(ST st);
```

Funzione di hash

- La tabella di hash ha dimensione M e contiene $|K|$ elementi ($|K| \ll |U|$)
- La tabella di hash ha indirizzi nell'intervallo $[0 \dots M-1]$
- La funzione di **hash** h mette in corrispondenza una chiave k con un indirizzo della tabella $h(k)$
$$h: U \rightarrow \{0, 1, \dots, M-1\}$$
- L'elemento x viene memorizzato all'indirizzo $h(k)$ dato dalla sua chiave k (attenzione alla gestione delle collisioni!).



Progetto della funzione di hash

Funzione ideale: **hashing uniforme semplice**:

- chiavi k equiprobabili \Rightarrow valori di $h(k)$ devono essere equiprobabili.

In pratica

- le chiavi k non sono equiprobabili
- chiavi diverse k_i, k_j sono correlate.

Per rendere i **valori di $h(k)$ equiprobabili** occorre:

- rendere $h(k_i)$ scorrelato da $h(k_j)$
 - “amplificare” le differenze
 - scorrelare $h(k)$ da k
- distribuire gli $h(k)$ in modo uniforme:
 - usare tutti i bit della chiave
 - moltiplicare per un numero primo.

Tipologie di funzioni di hash

Metodo moltiplicativo:

chiavi: numeri in virgola mobile in un intervallo prefissato ($s \leq k < t$):

$$h(k) = (k - s) / (t - s) * M$$

```
int hash(float k, int M, float s, float t) {  
    return ((k-s)/(t-s))*M;  
}
```

Esempio:

$$M = 97, s = 0.0, t = 1.0$$

$$k = 0.513870656$$

$$h(k) = (0.513870656 - 0) / (1 - 0) * 97 = 49$$

Metodo modulare:

chiavi: numeri interi; M numero primo

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

```
int hash(int k, int M){  
    return (k%M);  
}
```

Esempio:

M = 19

k = 31

$h(k) = 31 \% 19 = 12$

M numero primo evita:

- di usare solo gli ultimi n bit di k se $M = 2^n$
- di usare solo le ultime n cifre decimali di k se $M = 10^n$.

Metodo moltiplicativo-modulare

- chiavi: numeri interi:
 - data costante $0 < A < 1$
$$A = \phi = (\sqrt{5} - 1) / 2 = 0.6180339887$$
 - $h(k) = \lfloor k \cdot A \rfloor \% M$

Metodo modulare

- chiavi: stringhe alfanumeriche corte come interi derivati dalla valutazione di polinomi in una data base
 - M numero primo
 - $h(k) = k \% M$

Esempio

$$\begin{aligned}\text{stringa now} &= 'n'*128^2 + 'o'*128 + 'w' \\ &= 110*128^2 + 111*128 + 119 \\ k &= 1816567\end{aligned}$$

$$k = 1816567 \quad M = 19$$

$$h(k) = 1816567 \% 19 = 15$$

Metodo modulare:

chiavi: stringhe alfanumeriche lunghe come interi derivati dalla valutazione di polinomi in una data base con il metodo di Horner: ad esempio

$$\begin{aligned}P_7(x) &= p_7x^7 + p_6x^6 + p_5x^5 + p_4x^4 + p_3x^3 + p_2x^2 + p_1x + p_0 \\&= (((((p_7x + p_6)x + p_5)x + p_4)x + p_3)x + p_2)x + p_1)x + p_0\end{aligned}$$

Come prima:

M numero primo

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

Esempio

stringa averylongkey con base 128 (ASCII)

$$k = 97 \cdot 128^{11} + 118 \cdot 128^{10} + 101 \cdot 128^9 + 114 \cdot 128^8 + 121 \cdot 128^7 + 108 \cdot 128^6 + \\ 111 \cdot 128^5 + 110 \cdot 128^4 + 103 \cdot 128^3 + 107 \cdot 128^2 + 101 \cdot 128^1 + 121 \cdot 128^0$$

Ovviamente k non è rappresentabile su un numero ragionevole di bit.

Con il metodo di Horner:

$$k = (((((((((97 \cdot 128 + 118) \cdot 128 + 101) \cdot 128 + 114) \cdot 128 + 121) \cdot 128 + 108) \cdot 128 + 111) \cdot 128 + 110) \cdot \\ 128 + 103) \cdot 128 + 107) \cdot 128 + 101) \cdot 128 + 121$$

Anche con il metodo di Horner k non è rappresentabile su un numero ragionevole di bit.

È possibile però ad ogni passo eliminare i multipli di M, anziché farlo dopo in fase di applicazione del metodo modulare, ottenendo la seguente funzione di hash per stringhe con base 128 per l'ASCII:

```
int hash (char *v, int M){  
    int h = 0, base = 128;  
    for (; *v != '\0'; v++)  
        h = (base * h + *v) % M;  
    return h;  
}
```

In realtà anche per stringhe ASCII non si usa 128 come base, bensì:

- un numero primo (ad esempio 127)
- numero pseudocasuale diverso per ogni cifra della chiave (hash universale)

con lo scopo di ottenere una distribuzione abbastanza uniforme (probabilità di collisione tra 2 chiavi diverse prossima a $1/M$).

Funzione di hash per chiavi stringa con base prima:

```
int hash (char *v, int M) {  
    int h = 0, base = 127;  
    for (; *v != '\0'; v++)  
        h = (base * h + *v) % M;  
    return h;  
}
```

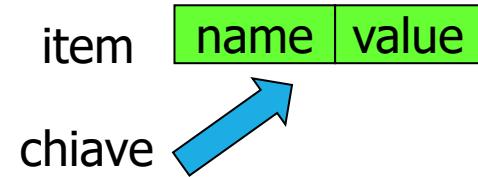
Funzione di hash per chiavi stringa con hash universale:

```
int hashU( char *v, int M) {  
    int h, a = 31415, b = 27183;  
    for ( h = 0; *v != '\0'; v++, a = a*b % (M-1))  
        h = (a*h + *v) % M;  
    return h;  
}
```

Item

- Quasi ADT Item
- Dati:
 - Nome (stringa), valore (intero)
 - Chiave = nome
 - Tipologia 3

Negli esempi stringhe da 1
carattere e non visualizzato
l'intero



Collisioni

Definizione:

collisione: $h(k_i) = h(k_j)$ per $k_i \neq k_j$

Le collisioni sono inevitabili, occorre:

- minimizzarne il numero (buona funzione di hash):
- gestirle:
 - linear chaining
 - open addressing.

Linear chaining

Più elementi possono risiedere nella stessa locazione della tabella \Rightarrow lista concatenata.

Operazioni:

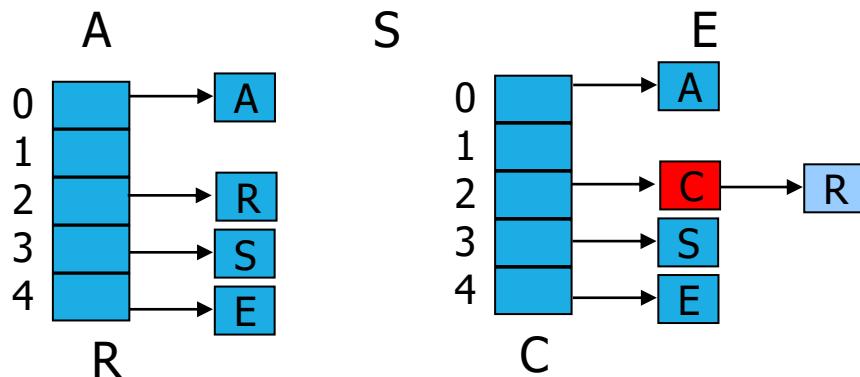
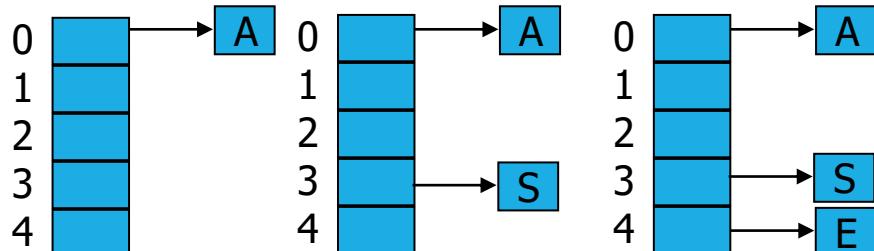
- inserimento in testa alla lista
- ricerca nella lista
- cancellazione dalla lista.

Determinazione della dimensione M della tabella:

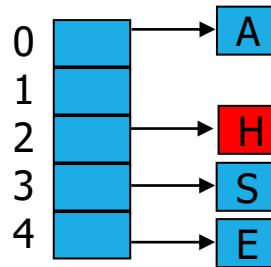
- il più piccolo primo $M \geq \text{numero di chiavi max} / r$ così che la lunghezza media delle liste sia r .

Esempio

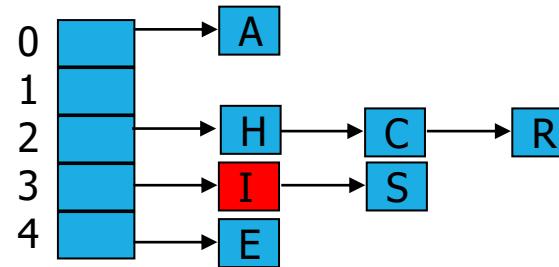
A S E R C H I N G X M P L
 $h(k) = 0 \ 3 \ 4 \ 2 \ 2 \ 2 \ 3 \ 3 \ 1 \ 3 \ 2 \ 0 \ 1$ $M = 5$



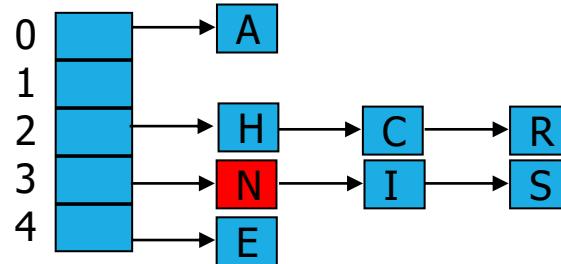
```
int hash (key k, int M) {  
    int h = 0, base = 127;  
    for (; *k != '\0'; k++)  
        h = (base * h + *k) % M;  
    return h;  
}
```



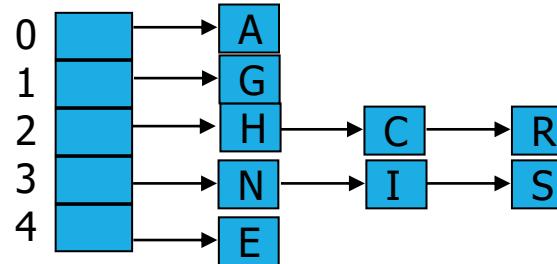
H



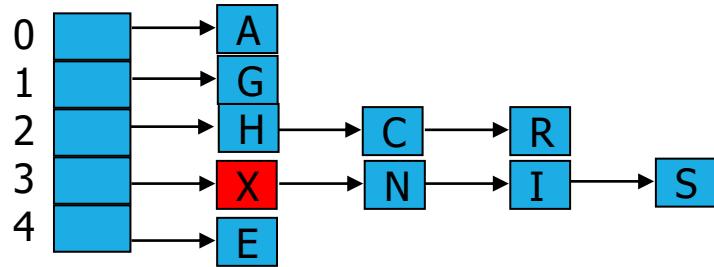
I



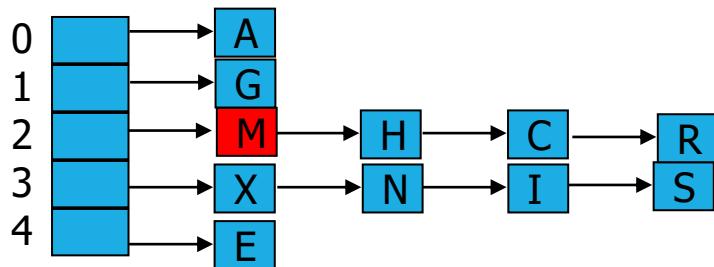
N



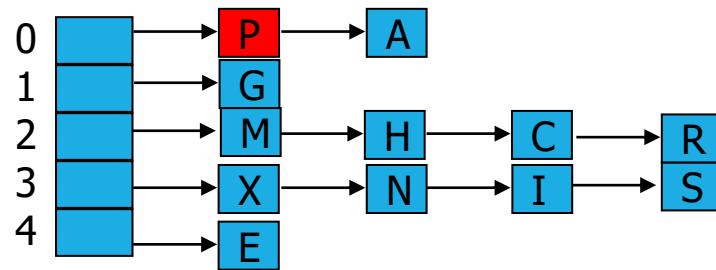
G



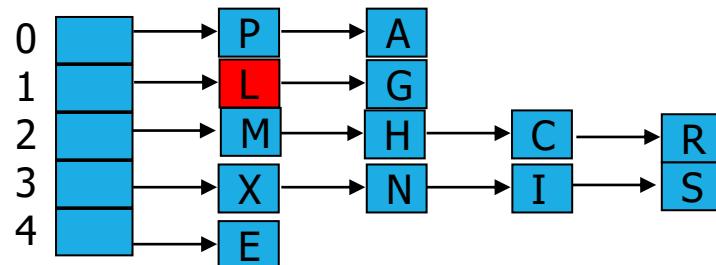
X



M



P



L

Linear chaining

ST.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Item.h"
#include "ST.h"

typedef struct STnode* link;

struct STnode { Item item; link next; } ;

struct symbtab { link *heads; int N; int M; link z; };

static link NEW( Item item, link next)
{
    link x = malloc(sizeof *x);
    x->item = item;
    x->next = next;
    return x;
}
```

vettore di liste con nodo sentinella in coda

nodo sentinella

dimensione tabella

numero di chiavi

```
ST STinit(int maxN, float r) {
    int i;
    ST st;

    st = malloc(sizeof(*st));
    st->N = 0;
    st->M = STsizeSet(maxN, r);
    st->heads = malloc(st->M*sizeof(link));
    st->z = NEW(ITEMsetNull(), NULL);

    for (i=0; i < st->M; i++)
        st->heads[i] = st->z;

    return st;
}
```

maxN ≤ 53

```
static int STsizeSet(int maxN, float r) {
    int primes[16]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53};
    int i = 0;
    int size;
    size = maxN /r;
    if (size < primes[15]) {
        for (i = 0; i<16; i++)
            if (size <= primes[i])
                return primes[i];
    }
    else
        printf("Too many entries!\n");
    return -1;
}
```

```
void STfree(ST st) {
    int i;
    link t,u;
    for(i=0; i<st->M; i++)
        for (t = st->heads[i]; t != st->z; t = u){
            u = t->next;
            free(t);
        }
    free(st->z);
    free(st->heads);
    free(st);
}

int STcount(ST st) {
    return st->N;
}

int STempty(ST st) {
    if (STcount(st) == 0)
        return 1;
    return 0;
}
```

```
static int hash(Key v, int M) {
    int h = 0, base = 127;
    for ( ; *v != '\0'; v++)
        h = (base * h + *v) % M;
    return h;
}

static int hashU(Key v, int M) {
    int h, a = 31415, b = 27183;
    for ( h = 0; *v != '\0'; v++, a = a*b % (M-1))
        h = (a*h + *v) % M;
    return h;
}

void STinsert (ST st, Item val) {
    int i;
    i = hash(KEYget(&val), st->M);
    st->heads[i] = NEW(val, st->heads[i]);
}
```

```

Item STsearch(ST st, Key k) {
    return searchR(st->heads[hash(k, st->M)], k, st->z);
}
Item searchR(link t, Key k, link z) {
    if (t == z) return ITEMsetNull();
    if ((KEYcmp(KEYget(&t->item), k))==0) return t->item;
    return searchR(t->next, k, z);
}

void STdelete(ST st, Key k) {
    int i = hash(k, st->M);
    st->heads[i] = deleteR(st->heads[i], k);
}

link deleteR(link x, Key k) {
    if (x == NULL) return NULL;
    if ((KEYcmp(KEYget(&x->item), k))==0) {
        link t = x->next; free(x); return t;
    }
    x->next = deleteR(x->next, k);
    return x;
}

```

```
void STdelete(ST st, Key k) {
    int i = hash(k, st->M);
    st->heads[i] = deleteR(st->heads[i], k);
}

void visitR(link h, link z) {
    if (h == z) return;
    ITEMstore(h->item);
    visitR(h->next, z);
}

void STdisplay(ST st)  {
    int i;
    for (i=0; i < st->M; i++) {
        printf("st->heads[%d] = ", i);
        visitR(st->heads[i], st->z);
        printf("\n");
    }
}
```

Complessità

Ipotesi:

Liste non ordinate:

- $N = |K|$ = numero di elementi memorizzati
- M = dimensione della tabella di hash

Hashing semplice uniforme:

$h(k)$ ha egual probabilità di generare gli M valori di uscita.

Definizione

fattore di carico $\alpha=N/M$ ($>, = o < 1$)

- Inserimento: $T(n) = O(1)$
- Ricerca:
 - caso peggiore $T(n) = \Theta(N)$
 - caso medio $T(n) = O(1+\alpha)$
- Cancellazione:
 - $T(n) = O(1)$ se disponibile il puntatore ad x e la lista è doppiamente linkata
 - come la ricerca se disponibile il valore di x , oppure il valore della chiave k , oppure la lista è semplicemente linkata

Open addressing

- Ogni cella della tabella può contenere un solo elemento
 - Tutti gli elementi sono memorizzati in tabella
 - Collisione: ricerca di cella non ancora occupata mediante **probing**:
 - generazione di una permutazione delle celle = ordine di ricerca della cella libera.
 - Concettualmente:
$$h(k, t) : U \times \{ 0, 1, \dots, M-1 \} \rightarrow \{ 0, 1, \dots, M-1 \}$$


chiave tentativo (0...M-1)
- $N \leq M$
 $\alpha \leq 1$

Open addressing

ST.c

```
...
struct symboltable { Item *a; int N; int M;};

ST STinit(int maxN, float alpha) {
    int i;
    ST st = malloc(sizeof(*st));
    st->N = 0;
    st->M = STsizeSet(maxN, alpha);
    if (st->M == -1)
        st = NULL;
    else {
        st->a = malloc(st->M * sizeof(Item) );
        for (i = 0; i < st->M; i++)
            st->a[i] = ITEMsetNull();
    }
    return st;
}
```

```
static int STsizeSet(int maxN, float alpha) {
    int primes[16]={2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53};
    int i = 0;
    if (maxN < primes[15]*alpha) {
        for (i = 0; i<16; i++)
            if (maxN <= primes[i]*alpha)
                return primes[i];
    }
    else
        printf("Too many entries!\n");
    return -1;
}
```

Funzioni di probing

- Linear probing
- Quadratic probing
- Double hashing

Un problema dell'open addressing è il **clustering**, cioè il raggruppamento di posizioni occupate contigue.

Linear probing

Insert:

- calcola $i = h(k)$
- se libero, inserisci chiave, altrimenti incrementa i di 1 modulo M
- ripeti fino a cella vuota.

```
void STinsert(ST st, Item item) {
    int i = hash(KEYget(&item), st->M);
    while (full(st, i))
        i = (i+1)%st->M;
    st->a[i] = item;
    st->N++;
}
int full(ST st, int i) {
    if (ITEMcheckNull(st->a[i])) return 0;
    return 1;
}
```

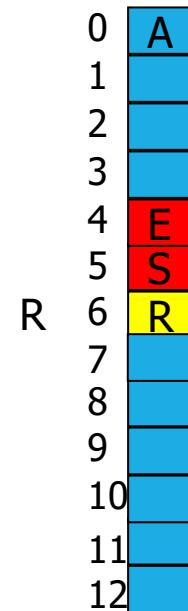
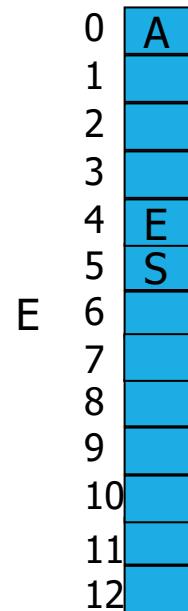
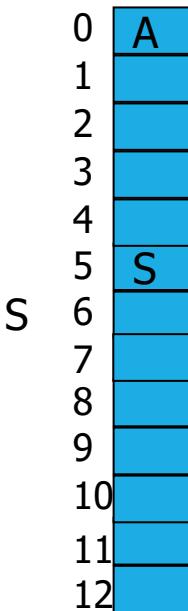
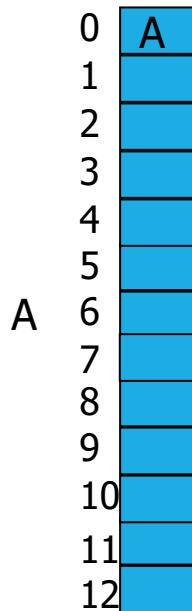
Search:

- calcola $i = h(k)$
- se trovata chiave, termina con successo
- incrementa i di 1 modulo M
- ripeti fino a cella vuota (insuccesso).

```
Item STsearch(ST st, Key k) {
    int i = hash(k, st->M);
    while (full(st, i))
        if (KEYcmp(k, KEYget(&st->a[i]))==0)
            return st->a[i];
        else
            i = (i+1)%st->M;
    return ITEMsetNull();
}
```

Esempio

A S E R C H I N G X M P
 $h(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$

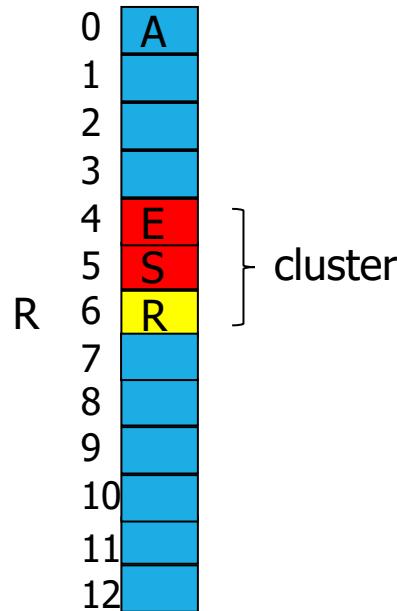


```
static int hash (key k,int M){  
    int h = 0, base = 127;  
    for (; *k != '\0'; k++)  
        h = (base * h + *k) % M;  
    return h;  
}
```

} cluster

NB: non si
rispetta
il vincolo
 $\alpha < 1/2$

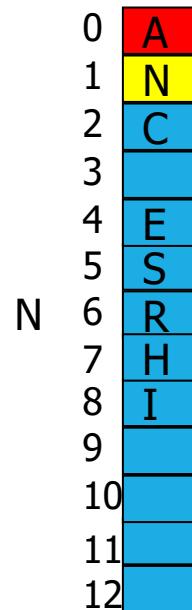
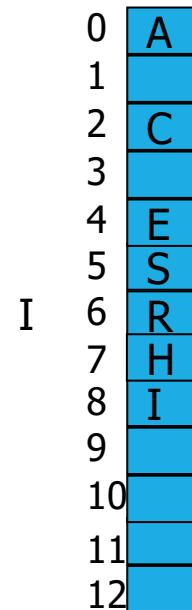
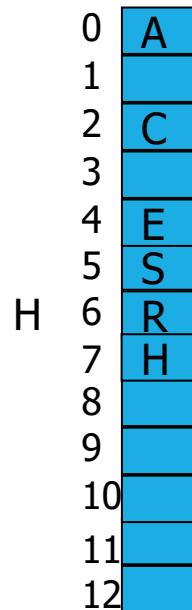
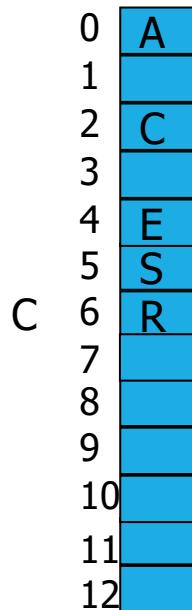
A S E R C H I N G X M P
 $h(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$



$$i = h('R') = 82 \% 13 = 4 \text{ collisione}$$
$$i = (4+1) \% 13 = 5 \text{ collisione}$$
$$i = (5+1) \% 13 = 6$$

A S E R C H I N G X M P

$$h(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$$

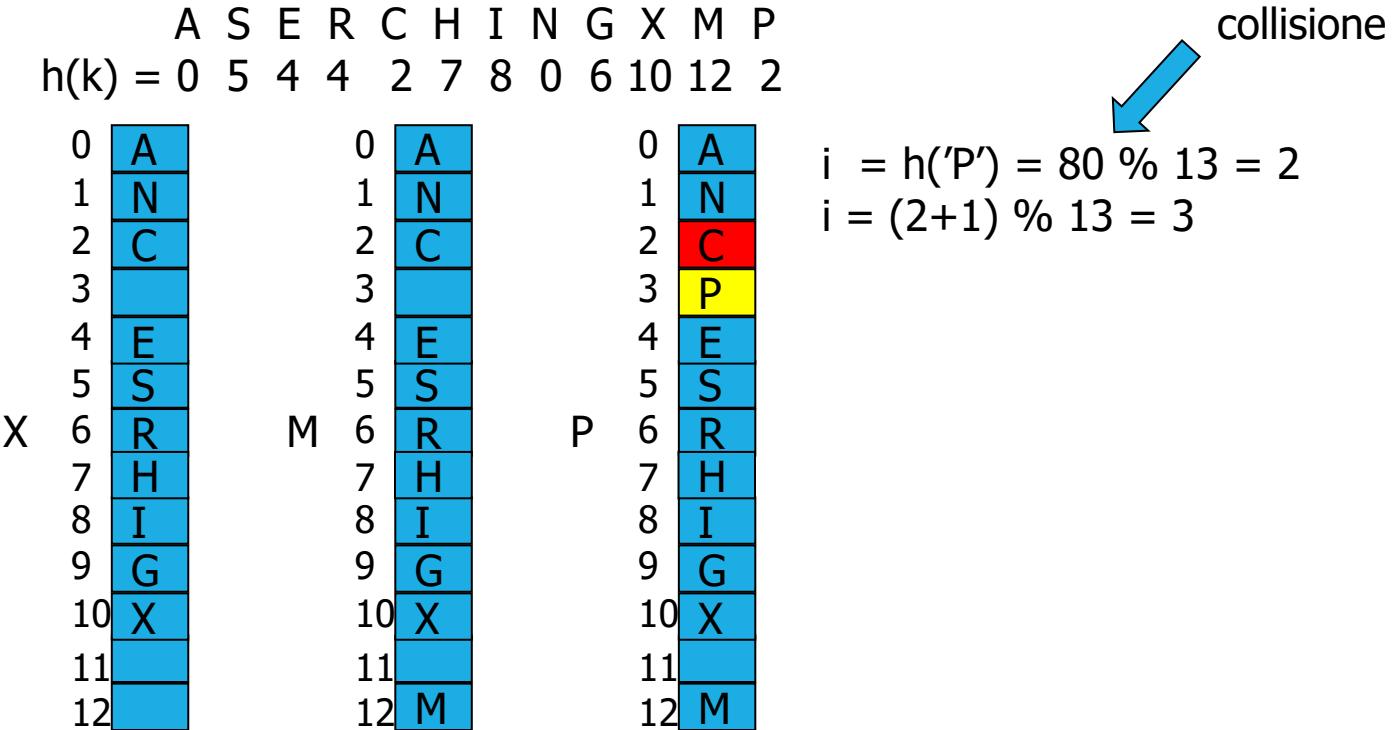


	A	S	E	R	C	H	I	N	G	X	M	P
$h(k) =$	0	5	4	4	2	7	8	0	6	10	12	2
	0	A										
	1		N									
	2			C								
	3											
	4			E								
	5				S							
N	6					R						
	7						H					
	8							I				
	9											
	10											
	11											
	12											

$i = h('N') = 78 \% 13 = 0$ collisione
 $i = (0+1) \% 13 = 1$

A S E R C H I N G X M P
 $h(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$

	0	A	
	1	N	$i = h('G') = 71 \% 13 = 6$ collisione
	2	C	$i = (6+1) \% 13 = 7$ collisione
	3		$i = (7+1) \% 13 = 8$ collisione
	4	E	$i = (8+1) \% 13 = 9$
	5	S	
G	6	R	
	7	H	
	8	I	
	9	G	
	10		
	11		
	12		



Delete

operazione complessa che interrompe le catene di collisione.

L'open addressing è in pratica utilizzato solo quando non si deve mai cancellare.

Soluzioni:

1. sostituire la chiave cancellata con una chiave sentinella che conta come piena in ricerca e vuota in inserzione
2. reinserire le chiavi del cluster sottostante la chiave cancellata

Soluzione 1

Nell'ADT si introduce un vettore status di interi: 0 se la cella è vuota, 1 se è occupata, -1 se cancellata.

La funzione CheckFull controlla se la cella i è piena (status=1).
La funzione CheckDeleted controlla se la cella è cancellata (status=-1).

```
struct symboltable { Item *a; int *status; int N; int M;};
static int CheckFull(ST st, int i);
static int CheckDeleted(ST st, int i);
```

```
static int CheckFull(ST st, int i) {
    if (st->status[i] == 1) return 1;
    return 0;
}

static int CheckDeleted(ST st, int i){
    if (st->status[i] == -1) return 1;
    return 0;
}

void STinsert(ST st, Item item) {
    int i = hash(KEYget(&item), st->M);
    while (CheckFull(st, i))
        i = (i+1)%st->M;
    st->a[i] = item;
    st->status[i] = 1;
    st->N++;
}
```

```
Item STsearch(ST st, Key k) {
    int i = hash(k, st->M);
    while (CheckFull(st, i)==1 || CheckDeleted(st, i)==1)
        if (KEYcmp(k, KEYget(&st->a[i]))==0) return st->a[i];
        else i = (i+1)%st->M;
    return ITEMsetNull();
}

void STdelete(ST st, Key k){
    int i = hash(k, st->M);
    while (CheckFull(st, i)==1 || CheckDeleted(st, i)==1)
        if (KEYcmp(k, KEYget(&st->a[i]))==0) break;
        else i = (i+1) % st->M;
    if (ITEMcheckNull(st->a[i])) return;
    st->a[i] = ITEMsetNull();
    st->N--;
    st->status[i]=-1;
}
```

Esempio

Cancellare E, ricordando che c'era stata collisione tra E e R.

0	A	1
1	0	
2	C	1
3	0	
4	E	1
5	S	1
6	R	1
7	H	1
8	0	
9	0	
10	0	
11	0	
12	0	

vettore status



0	A	1
1	0	
2	C	1
3	0	
4		-1
5	S	1
6	R	1
7	H	1
8	0	
9	0	
10	0	
11	0	
12	0	

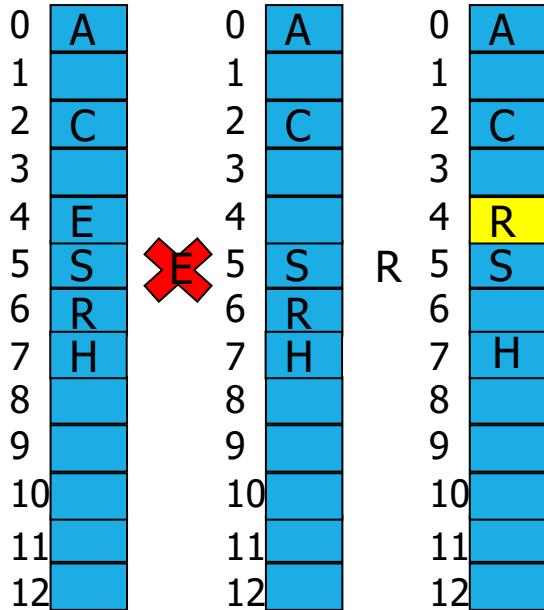
vettore status

Soluzione 2

```
void STdelete(ST st, Key k) {
    int j, i = hash(k, st->M);
    Item tmp;
    while (full(st, i))
        if (KEYcmp(k, KEYget(&st->a[i]))==0)
            break;
        else
            i = (i+1) % st->M;
    if (ITEMcheckNull(st->a[i]))
        return;
    st->a[i] = ITEMsetNull();
    st->N--;
    for (j = i+1; full(st, j); j = (j+1)%st->M, st->N--) {
        tmp = st->a[j];
        st->a[j] = ITEMsetNull();
        STinsert(st, tmp);
    }
}
```

Esempio

Cancellare E, ricordando che c'era stata collisione tra E e R.



Complessità

Complessità con l'ipotesi di:

- hashing semplice uniforme
- probing uniforme.

Tentativi in media di “probing” per la ricerca:

- search hit: $1/2(1 + 1/(1-\alpha))$
- search miss: $1/2(1 + 1/(1-\alpha)^2)$

α	1/2	2/3	3/4	9/10	
hit	1.5	2.0	3.0	5.5	
miss	2.5	5.0	8.5	55.5	

Quadratic probing

Insert:

- i è il contatore dei tentativi (all'inizio 0)
- $\text{index} = (\text{h}'(\text{k}) + c_1 i + c_2 i^2) \% M$
- se libero, inserisci chiave, altrimenti incrementa i e ripeti fino a cella vuota.

```
#define c1 1
#define c2 1
void STinsert(ST st, Item item) {
    int i = 0, start = hash(KEYget(&item), st->M), index=start;
    while (full(st, index)) {
        i++;
        index = (start + c1*i + c2*i*i)%st->M;
    }
    st->a[index] = item;
    st->N++;
}
```

Search

```
Item STsearch(ST st, Key k) {
    int i=0, start = hash(k, st->M), index = start;
    while (full(st, index))
        if (KEYcmp(k, KEYget(&st->a[index]))==0)
            return st->a[index];
        else {
            i++;
            index = (start + c1*i + c2*i*i)%st->M;
        }
    return ITEMsetNull();
}
```

Delete (soluzione 2)

```
void STdelete(ST st, Key k) {
    int i=0, start = hash(k, st->M), index = start;
    Item tmp;
    while (full(st, index))
        if (KEYcmp(k, KEYget(&st->a[index]))==0) break;
        else { i++; index = (start + c1*i + c2*i*i)%st->M;
        }
    if (ITEMcheckNull(st->a[index])) return;
    st->a[index] = ITEMsetNull();
    st->N--; i++;
    index = (start + c1*i + c2*i*i)%st->M;
    while(full(st, index)) {
        tmp = st->a[index];
        st->a[index] = ITEMsetNull();
        st->N--; i++;
        STinsert(st, tmp);
        index = (start + c1*i + c2*i*i)%st->M;
    }
}
```

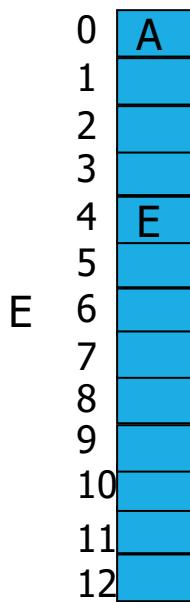
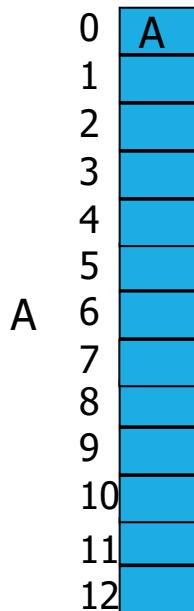
Scelta di c_1 e c_2

- se $M = 2^K$, scegliere $c_1 = c_2 = \frac{1}{2}$ per garantire che siano generati tutti gli indici tra 0 e $M-1$:
- se M è primo, se $\alpha < \frac{1}{2}$ i seguenti valori
 - $c_1 = c_2 = \frac{1}{2}$
 - $c_1 = c_2 = 1$
 - $c_1 = 0, c_2 = 1$.

garantiscono che, con inizialmente $start = h(k)$ e poi $index = (start + c_1 i + c_2 i^2) \text{ modulo } M$ si abbiano valori distinti per $1 \leq i \leq (M-1)/2$.

Esempio

$$h(k) = \begin{matrix} & A & E & R & C & N & P \\ k & 0 & 4 & 4 & 2 & 0 & 2 \end{matrix}$$

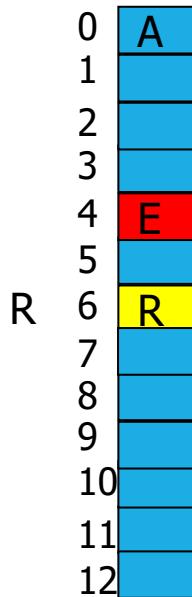


```
static int hash (Key k, int M){  
    int h = 0, base = 127;  
    for (; *k != '\0'; k++)  
        h = (base * h + *k) % M;  
    return h;  
}
```

Funzione di quadratic probing
 $c_1=1$ $c_2=1$
 $i + i^2$

$$\alpha = 6/13 < 1/2$$

$$h(k) = \begin{matrix} & A & E & R & C & N & P \\ 0 & 0 & 4 & 4 & 2 & 0 & 2 \end{matrix}$$



start = $h('R') = 82 \% 13 = 4$ collisione
 index = $(4+1+1^2) \% 13 = 6$

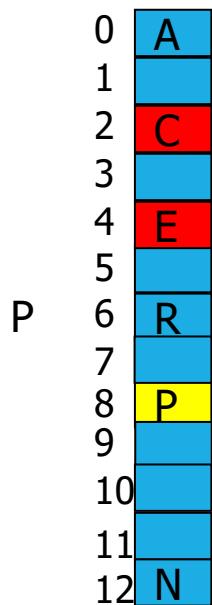
$$h(k) = \begin{matrix} & A & E & R & C & N & P \\ 0 & 0 & 4 & 4 & 2 & 0 & 2 \end{matrix}$$

C	0	A
	1	
	2	C
	3	
	4	E
	5	
	6	R
	7	
	8	
	9	
	10	
	11	
	12	N

N	0	A
	1	
	2	C
	3	
	4	E
	5	
	6	R
	7	
	8	
	9	
	10	
	11	
	12	N

start = $h('N') = 78 \% 13 = 0$ collisione
 index = $(0+1+1^2) \% 13 = 2$ collisione
 index = $(0+2+2^2) \% 13 = 6$ collisione
 index = $(0+3+3^2) \% 13 = 12$

$$h(k) = \begin{matrix} & A & E & R & C & N & P \\ 0 & 0 & 4 & 4 & 2 & 0 & 2 \end{matrix}$$



start = $h('P') = 80 \% 13 = 2$ collisione
index = $(2+1+1^2) \% 13 = 4$ collisione
index = $(2+2+2^2) \% 13 = 8$

Double hashing

Insert:

- calcola $i = h_1(k)$
- se posizione libera, inserisci chiave, altrimenti calcola $j = h_2(k)$ e prova in $i = (i + j) \% M$
- ripeti fino a cella vuota. Ricordare che, se $M = 2^{\alpha} \text{max}$, $\alpha < 1$

Importante: bisogna che il nuovo valore

$$i = (i + j) \% M = (h_1(k) + h_2(k)) \% M$$

sia diverso dal vecchio valore di i , altrimenti si entra in un ciclo infinito. Per evitarlo:

- h_2 non deve mai ritornare 0
- $h_2 \% M$ non deve mai ritornare 0

Esempi di h_1 e h_2 :

$$h_1(k) = k \% M \text{ e } M \text{ primo}$$

$$h_2(k) = 1 + k \% 97$$

$h_2(k)$ non ritorna mai 0 e $h_2 \% M$ non ritorna mai 0 se $M > 97$.

```
static int hash1(Key k, int M) {
    int h = 0, base = 127;
    for ( ; *k != '\0'; k++) h = (base * h + *k) % M;
    return h;
}
static int hash2(Key k, int M) {
    int h = 0, base = 127;
    for ( ; *k != '\0'; k++) h = (base * h + *k);
    h = ((h % 97) + 1)%M;
    if (h==0) h=1;
    return h;
}
void STinsert(ST st, Item item) {
    int i = hash1(KEYget(&item), st->M);
    int j = hash2(KEYget(&item), st->M);
    while (full(st, i))
        i = (i+j)%st->M;
    st->a[i] = item;
    st->N++;
}
```

Search

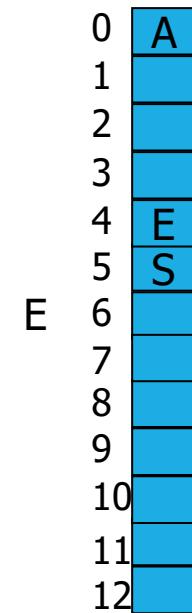
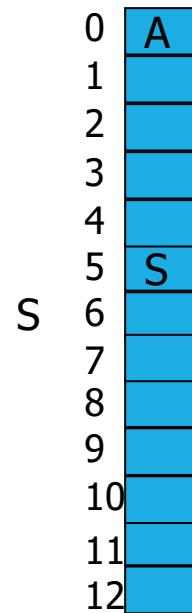
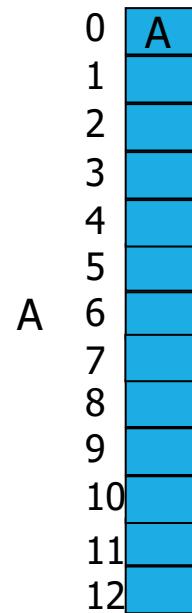
```
Item STsearch(ST st, Key k) {
    int i = hash1(k, st->M);
    int j = hash2(k, st->M);
    while (full(st, i))
        if (KEYcmp(k, KEYget(&st->a[i]))==0)
            return st->a[i];
        else
            i = (i+j)%st->M;
    return ITEMsetNull();
}
```

Delete (soluzione 2)

```
void STdelete(ST st, Key k) {
    int i = hash1(k, st->M), j = hash2(k); Item tmp;
    while (full(st, i))
        if (KEYcmp(k, KEYget(&st->a[i]))==0) break;
        else i = (i+j) % st->M;
    if (ITEMcheckNull(st->a[i]))
        return;
    st->a[i] = ITEMsetNull();
    st->N--;
    i = (i+j) % st->M;
    while(full(st, i)) {
        tmp = st->a[i];
        st->a[i] = ITEMsetNull();
        st->N--;
        STinsert(st, tmp);
        i = (i+j) % st->M;
    }
}
```

Esempio

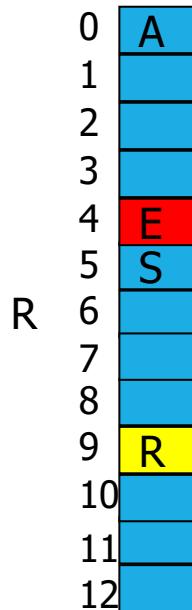
A S E R C H I N G X M P
 $h_1(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$



NB: non si rispetta
il vincolo $\alpha < 1/2$

A S E R C H I N G X M P

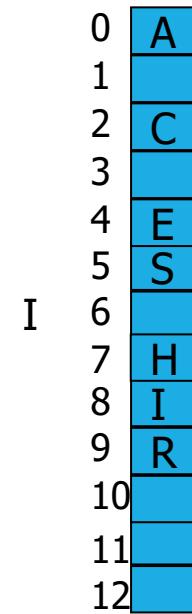
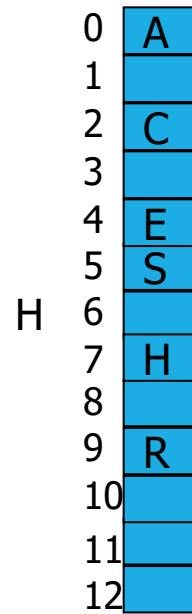
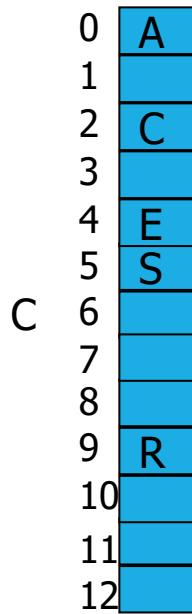
$h_1(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$



$$\begin{aligned} i &= h('R') = 82 \% 13 = 4 \text{ collisione} \\ j &= (82 \% 97 + 1) \% 13 = 5 \\ i &= (4 + 5) \% 13 = 9 \end{aligned}$$

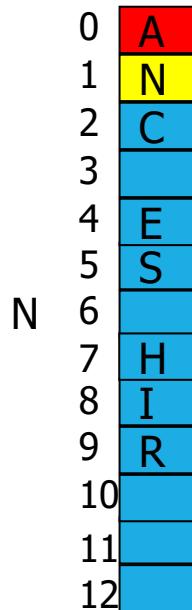
A S E R C H I N G X M P

$h_1(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$



A S E R C H I N G X M P

$h_1(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$



$$i = h('N') = 78 \% 13 = 0 \text{ collisione}$$
$$j = (78 \% 97 + 1 \% 13) = 1$$
$$i = (0 + 1) \% 13 = 1$$

A S E R C H I N G X M P

$h_1(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$

0	A
1	N
2	C
3	
4	E
5	S
6	G
7	H
8	I
9	R
10	
11	
12	

0	A
1	N
2	C
3	
4	E
5	S
6	G
7	H
8	I
9	R
10	X
11	
12	

0	A
1	N
2	C
3	
4	E
5	S
6	G
7	H
8	I
9	R
10	X
11	
12	M

G

X

M

A S E R C H I N G X M P
 $h_1(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$

P	0	A
	1	N
	2	C
	3	
	4	E
	5	S
	6	G
	7	H
	8	I
	9	R
	10	X
	11	P
	12	M

$$i = h('P') = 80 \% 13 = 2 \text{ collisione}$$

$$j = (80 \% 97 + 1) \% 13 = 3$$

$$i = (2 + 3) \% 13 = 5 \text{ collisione}$$

$$i = (5 + 3) \% 13 = 8 \text{ collisione}$$

$$i = (8 + 3) \% 13 = 11$$

A S E R C H I N G X M P

$$h_1(k) = 0 \ 5 \ 4 \ 4 \ 2 \ 7 \ 8 \ 0 \ 6 \ 10 \ 12 \ 2$$

P	0	A
	1	N
	2	C
	3	
	4	E
	5	S
	6	G
	7	H
	8	I
	9	R
	10	X
	11	P
	12	M

In inserzione c'è stata collisione tra 'P' e 'C'. Non ci sono state altre collisioni.
Se si cancella 'C', 'P' prende il suo posto.

P	0	A
	1	N
	2	P
	3	
	4	E
	5	S
	6	G
	7	H
	8	I
	9	R
	10	X
	11	
	12	M

Complessità del double hashing

Ipotesi:

- hashing semplice uniforme
- probing uniforme.

Tentativi di “probing” per la ricerca:

- search miss: $1/(1-\alpha)$
- search hit: $1/\alpha \ln (1/(1-\alpha))$

α	1/2	2/3	3/4	9/10	
hit	1.4	1.6	1.8	2.6	
miss	1.5	2.0	3.0	5.5	

Confronto tra alberi e tabelle di hash

Tabelle di hash:

- più facili da realizzare
- unica soluzione per chiavi senza relazione d'ordine
- più veloci per chiavi semplici

Alberi (BST e loro varianti):

- meglio garantite le prestazioni (per alberi bilanciati)
- permettono operazioni su insiemi con relazione d'ordine.

Riferimenti

- Tabelle di hash
 - Cormen 12.1, 12.2, 12.3, 12.4
 - Sedgewick 14.1, 14.2, 14.3, 14.4

Esercizi di teoria

■ 6. Tabelle di hash

- 6.1 Hashing
 - 6.2 Linear chaining
 - 6.3 Open addressing con linear probing
 - 6.3 Open addressing con quadratic probing
 - 6.3 Open addressing con double hashing



L'ADT Grafo

Gianpiero Cabodi e Paolo Camurati



Perché la Teoria dei Grafi?

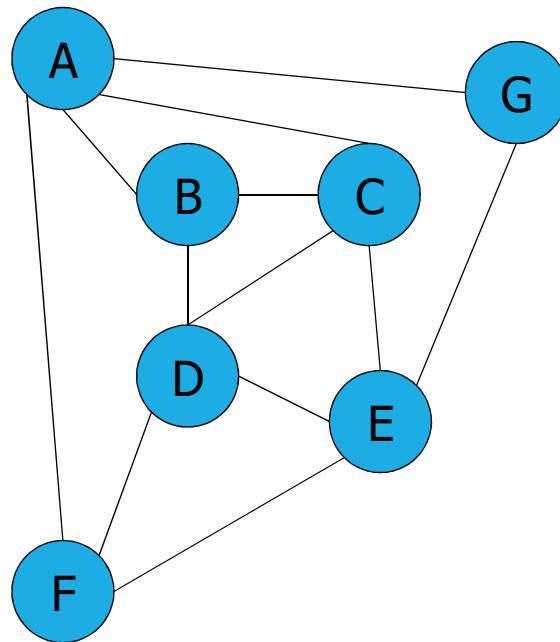
- Moltissime applicazioni pratiche
- Centinaia di algoritmi
- Astrazione interessante e utilizzabile in molti domini diversi
- Ricerca attiva in Informatica e Matematica discreta.

Complessità dei problemi

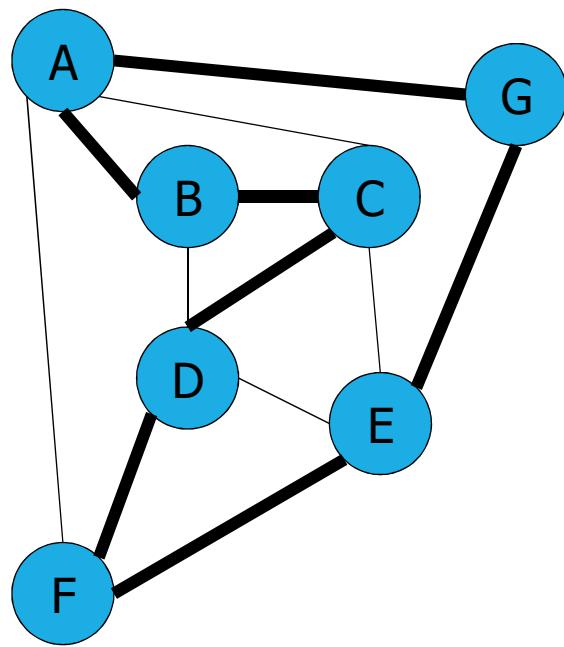
- problemi facili:
 - determinare se un grafo è connesso
 - determinare la presenza di un ciclo
 - individuare le componenti fortemente connesse
 - individuare gli alberi minimi ricoprenti
 - calcolare i cammini minimi
 - determinare se un grafo è bipartito
 - trovare un cammino di Eulero
- problemi trattabili:
 - planarità di un grafo
 - matching

- problemi intrattabili:
 - cammini a lunghezza massima
 - colorabilità
 - clique massimale
 - ciclo di Hamilton
 - problema del commesso viaggiatore
- problemi di complessità ignota: isomorfismo di due grafi F e G :
 - isomorfismo fra G e H : applicazione biiettiva f dai vertici di G ai vertici di H tale che vi sia un arco dal vertice u al vertice v in G se e solo se c'è un arco dal vertice $f(u)$ al vertice $f(v)$ in H
 - non esiste un algoritmo polinomiale, ma non è stato provato che il problema è NP-completo.

Ciclo di Hamilton



Dato un grafo non orientato $G = (V, E)$, esiste un ciclo semplice che visita ogni vertice una e una sola volta?



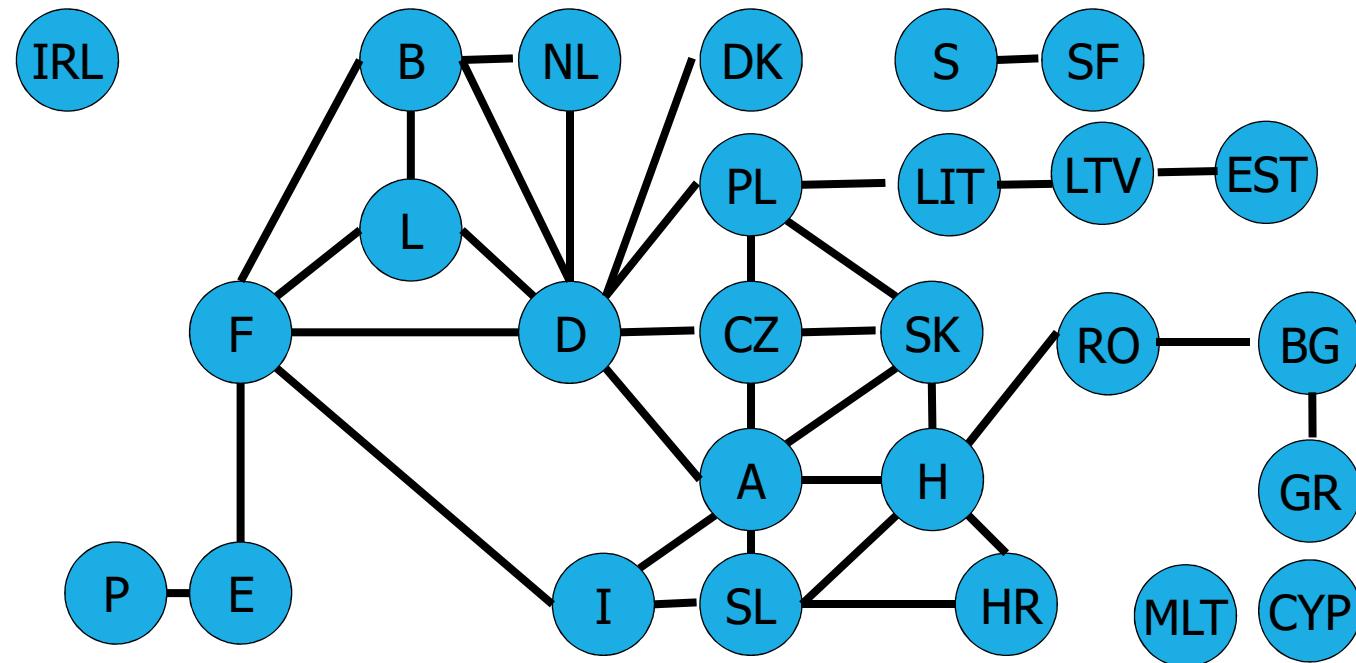
Colorabilità

Dato un grafo non orientato $G = (V, E)$, quale è il minimo numero di colori k necessario affinché nessun vertice abbia lo stesso colore di un vertice ad esso adiacente?

Si dice «planare» un grafo che, se disegnato su di un piano, non ha archi che si intersecano.

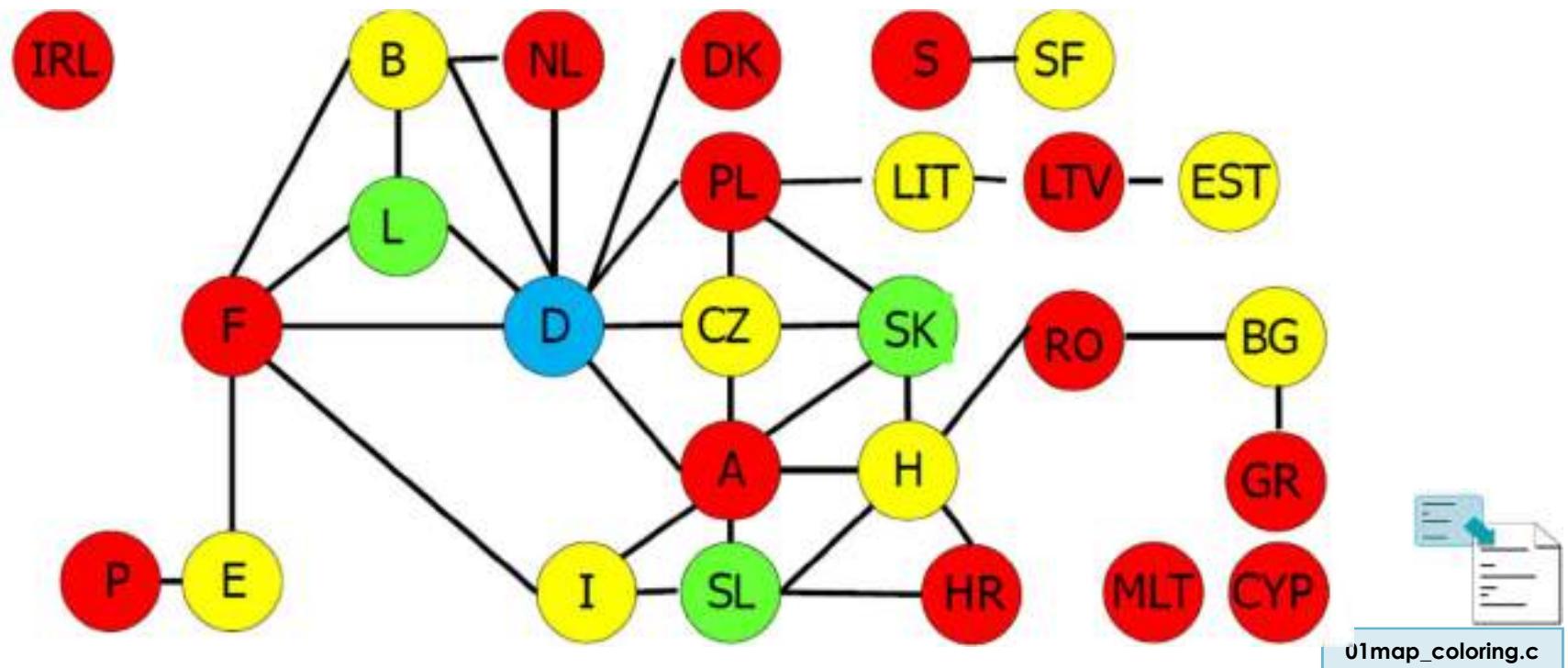
Le mappe geografiche sono modellate come grafi planari.

L'UE a 27 stati



Colorabilità: k=4

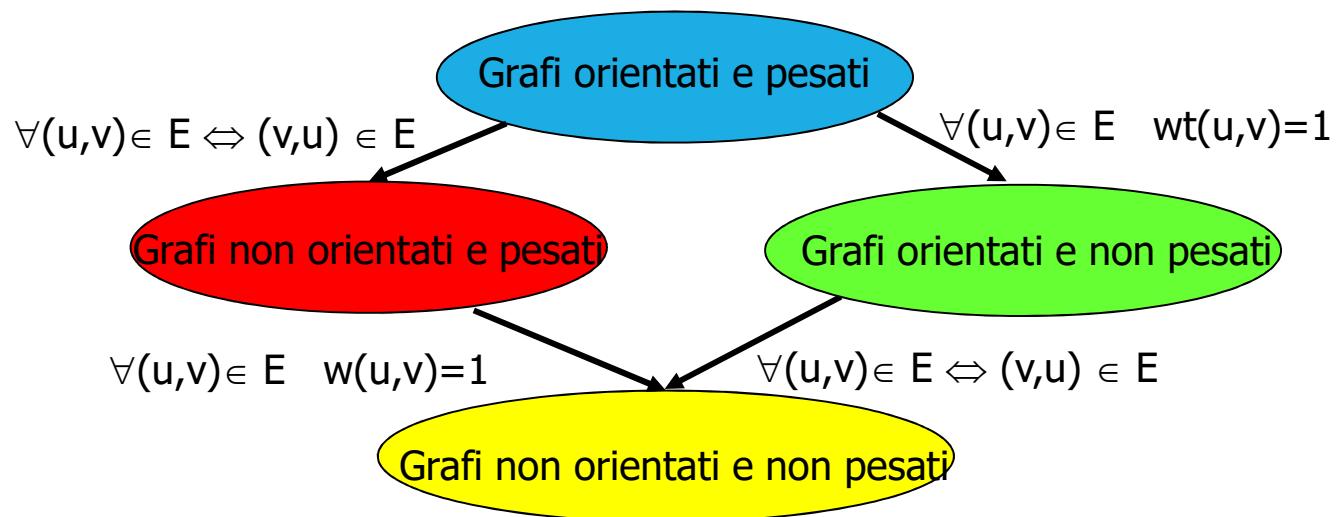
Per i grafi planari si può dimostrare che servono al più 4 colori.



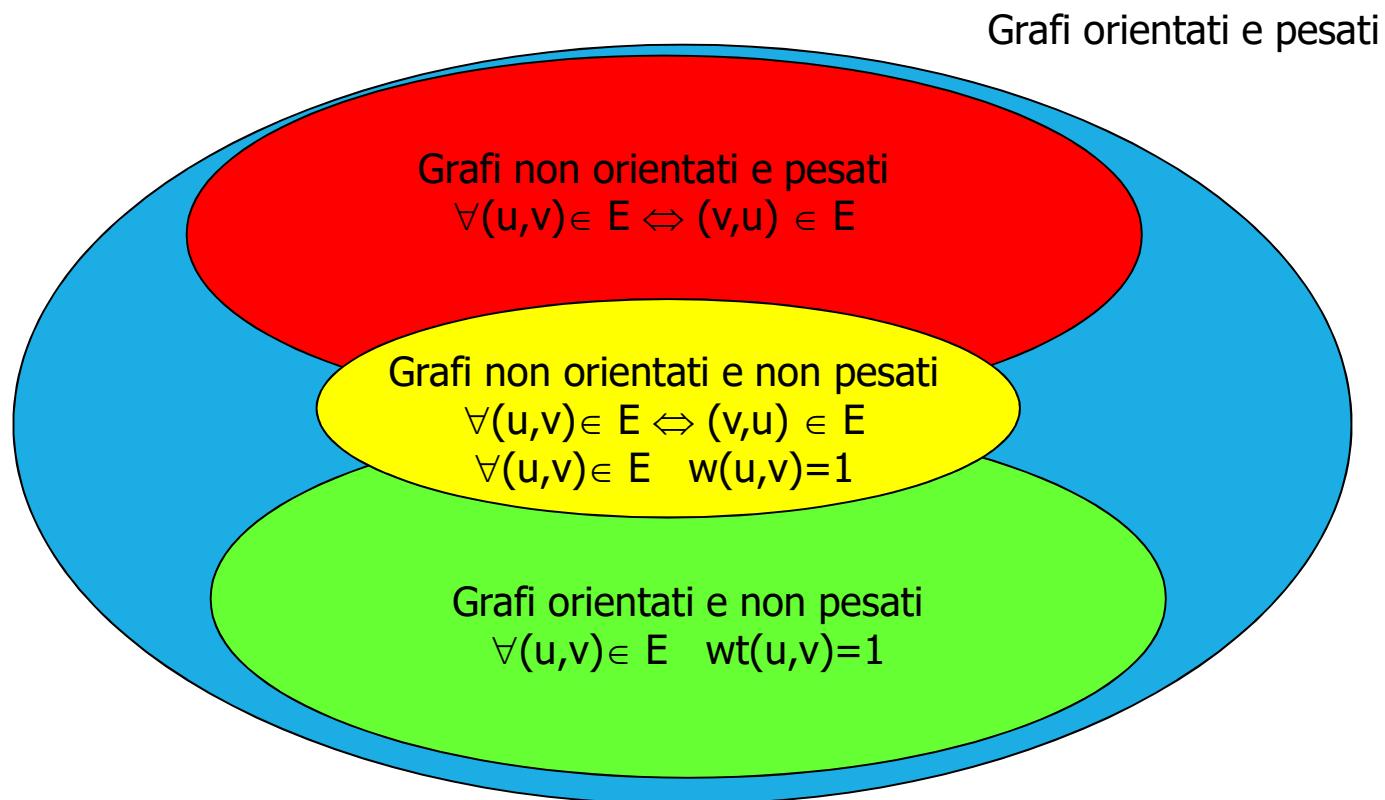
L'ADT Grafo

- Ipotesi
- Interfaccia
- Lettura da file
- Rappresentazione:
 - matrice delle adiacenze
 - lista delle adiacenze
 - (elenco di archi)

Tipologie (visione a grafo)



Tipologie (visione a insieme)



Ipotesi

- Il modello più generale è il grafo orientato e pesato. Per efficienza si implementano ADT specifici per le 4 tipologie
- Dopo l'inizializzazione:
 - grafi statici: non si aggiungono né si cancellano né vertici né archi
 - grafi semi-statici: non si aggiungono né si cancellano vertici, si possono aggiungere o cancellare archi
 - grafi dinamici: si possono aggiungere e cancellare sia vertici, sia archi
- Nel Corso si considerano solo grafi semi-statici, in cui i vertici vengono cancellati «logicamente», aggiungendo un campo per marcare se è cancellato o meno.

Vertici: informazioni rilevanti:

- interi per identificarli ai fini dell'algoritmo
 - stringhe per identificarli ai fini dell'utente memorizzate in tabelle di simboli:
 - esterne al grafo
 - interne al grafo
- cui si accede mediante 2 chiavi (intero e stringa).

Tabella di simboli con funzioni:

- STsearch da chiave (nome) a intero (indice)
- STsearchByIndex da chiave (indice) a stringa (nome)

Possibili soluzioni:

- estensione della tabella di simboli se implementata con vettore con la STsearchByIndex
- tabella standard (BST, hash table) e vettore a lato per la ricerca da indice a chiave

- possibili implementazioni:
 - tabella di simboli estesa: vettore non ordinato di stringhe: l'indice del vettore coincide con l'indice del vertice che non è memorizzato esplicitamente. Ricerca inversa con scansione lineare
 - BST o tabella di hash: l'indice del vertice è memorizzato esplicitamente. STsearchByIndex con scansione lineare di un vettore di corrispondenza indice-chiave
- negli esempi: tabella di simboli estesa interna all'ADT grafo realizzata come vettore non ordinato con indice del vertice coincidente con quello del vettore
- nei laboratori: tutte le tipologie.

- Il numero di vertici $|V|$ che serve per inizializzare grafo e tabella di simboli può essere:
 - noto per lettura o perché compare come intero sulla prima riga del file da cui si legge
 - ignoto, ma sovrastimabile: se il grafo è dato come elenco di archi
 - con una prima lettura si determina il numero di archi e $|V|$ si sovrasta come 2 volte il numero di archi, ipotizzando che ogni arco connetta vertici distinti
 - con una seconda lettura si inseriscono i vertici su cui insistono gli archi nella tabella di simboli, ottenendo il numero di vertici distinti $|V|$ esatto e la corrispondenza nome del vertice – indice.

- Formato del file di input:
 - numero V di vertici sulla prima riga
 - V righe con ciascuna il nome di un vertice
 - numero indefinito di righe con coppie vertice-vertice per gli archi (con peso se grafo pesato).
- con file in formato standard:
 - GRAPHload/GRAHstore di libreria
 - altrimenti fileRead ad hoc nel main.

- Il grafo è un ADT di 1 classe
- Gli archi sono definiti con `typedef` in `Graph.h` e sono quindi visibili anche al client
- La tabella di simboli è un ADT di 1 classe
- Altre eventuali collezioni di dati sono ADT di 1 classe (code, code a priorità, etc.)

ADT di I classe Grafo

Graph.h

```
typedef struct edge { int v; int w; int wt; } Edge;
typedef struct graph *Graph;
Graph GRAPHinit(int v);
void GRAPHfree(Graph G);
void GRAPHload(FILE *fin);
void GRAPHstore(Graph G, FILE *fout);
int GRAPHgetIndex(Graph G, char *label);
void GRAPHinsertE(Graph G, int id1, int id2; int wt; );
void GRAPHremoveE(Graph G, int id1, int id2);
void GRAPHedges(Graph G, Edge *a);
int GRAPHpath(Graph G, int id1, int id2);
void GRAPHpathH(Graph G, int id1, int id2);
void GRAPHbfs(Graph G, int id);
void GRAPHdfs(Graph G, int id);
int GRAPHcc(Graph G);
int GRAPHscc(Graph G);
```

grafo pesati

indice dato nome

grafo pesati

grafo non orientato

grafo orientato

Lettura/scrittura di un grafo da/su file

- Se il file contiene la lista degli archi in formato «standard» ha senso offrire GRAPHload/ GRAPHstore. In alternativa il client implementa la sua funzione di lettura/scrittura
- è già noto il numero di vertici $|V|$
- lettura dei vertici e inserzione nella tabella di simboli
- lettura degli archi e inserzione nel grafo

- Scrittura: si scandiscono gli archi:
 - con GRAPHedges si accumulano gli archi in un vettore
 - per ciascuno dei vertici su cui l'arco insiste, tramite la tabella di simboli, dato l'indice si ricava il nome.

```
Graph GRAPHload(FILE *fin) {
    int v, i, id1, id2, wt;
    char label1[MAXC], label2[MAXC];
    Graph G;
    fscanf(fin, "%d", &v);
    G = GRAPHinit(v);
    for (i=0; i<v; i++) {
        fscanf(fin, "%s", label1);
        STinsert(G->tab, label1, i);
    }
    while(fscanf(fin,"%s %s %d", label1, label2, &wt) == 3) {
        id1 = STsearch(G->tab, label1);
        id2 = STsearch(G->tab, label2);
        if (id1 >= 0 && id2 >=0)
            GRAPHinsertE(G, id1, id2, wt);
    }
    return G;
}
```

The diagram illustrates the flow of data from the input file through the program's variables to the final graph structure. A yellow box labeled "grafo pesati" (heavy graph) has three green arrows pointing to specific locations in the code:

- An arrow points to the variable `wt` in the first line of the function definition.
- An arrow points to the `label1` parameter in the `fscanf` statement within the `while` loop.
- An arrow points to the `wt` parameter in the final call to `GRAPHinsertE`.

```

void GRAPHstore(Graph G, FILE *fout) {
    int i;
    Edge *a;

    a = malloc(G->E * sizeof(Edge));

    GRAPHedges(G, a); ← dipende dalla rappresentazione

    fprintf(fout, "%d\n", G->V);
    for (i = 0; i < G->V; i++)
        fprintf(fout, "%s\n", STsearchByIndex(G->tab, i));
    for (i = 0; i < G->E; i++)
        fprintf(fout, "%s %s %d\n",
                STsearchByIndex(G->tab, a[i].v),
                STsearchByIndex(G->tab, a[i].w), a[i].wt);
}

```

grafi pesati

Inserzione/rimozione archi

dipende dalla rappresentazione

```
void GRAPHinsertE(Graph G, int id1, int id2, int wt) {  
    insertE(G, EDGEcreate(id1, id2, wt));  
}  
  
void GRAPHremoveE(Graph G, int id1, int id2)  
    removeE(G, EDGEcreate(id1, id2, 0));  
}
```

grafi pesati

Rappresentazione

Matrice di adiacenza

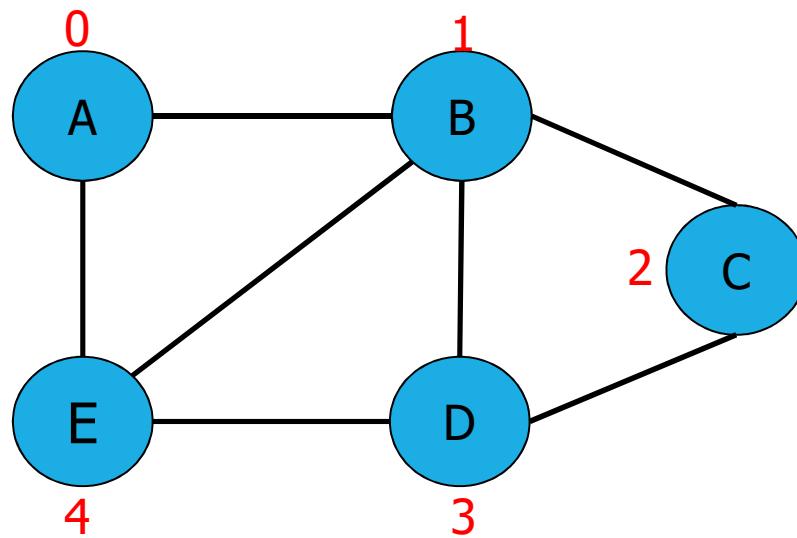
Dato $G = (V, E)$, la matrice di adiacenza è:

- matrice adj di $|V| \times |V|$ elementi

$$\text{adj}[i,j] = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{se } (i, j) \notin E \end{cases}$$

- grafi non orientati: adj simmetrica
- grafi pesati: $\text{adj}[i,j]=$ peso dell'arco (i,j) se esiste, 0 non è un peso ammesso.

Non orientato non pesato



Lista archi

ST
A B
B C
B D
A E
C D
B E
D E

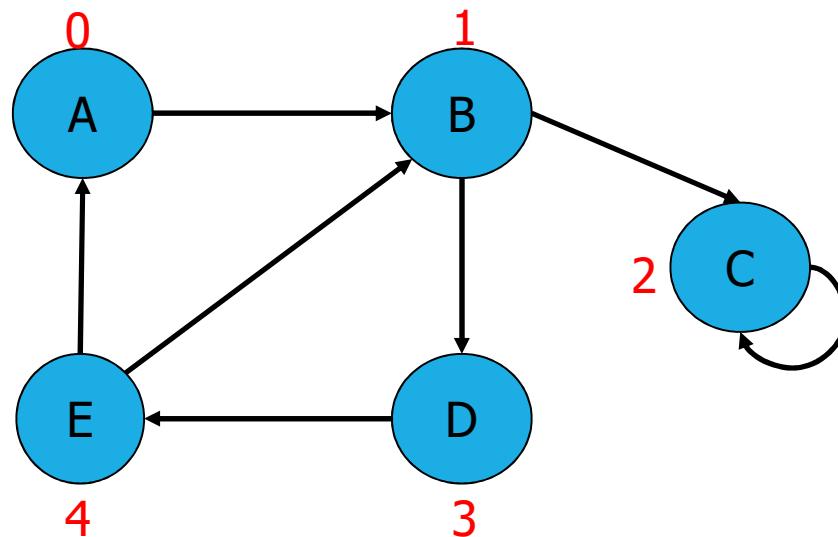
ST

0	A
1	B
2	C
3	D
4	E

G->adj

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Orientato non pesato



Lista archi

A	B
B	C
C	C
D	D
E	E

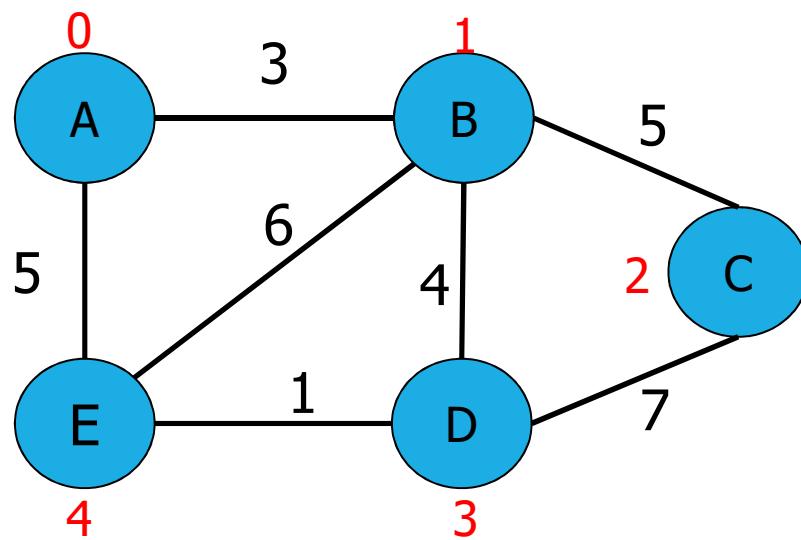
ST

0	A
1	B
2	C
3	D
4	E

G->adj

0	1	2	3	4
0	0	1	0	0
1	0	0	1	1
2	0	0	1	0
3	0	0	0	0
4	1	1	0	0

Non orientato pesato



Lista archi

A	B	3
B	C	5
B	D	4
A	E	5
C	D	7
B	E	6
D	E	1

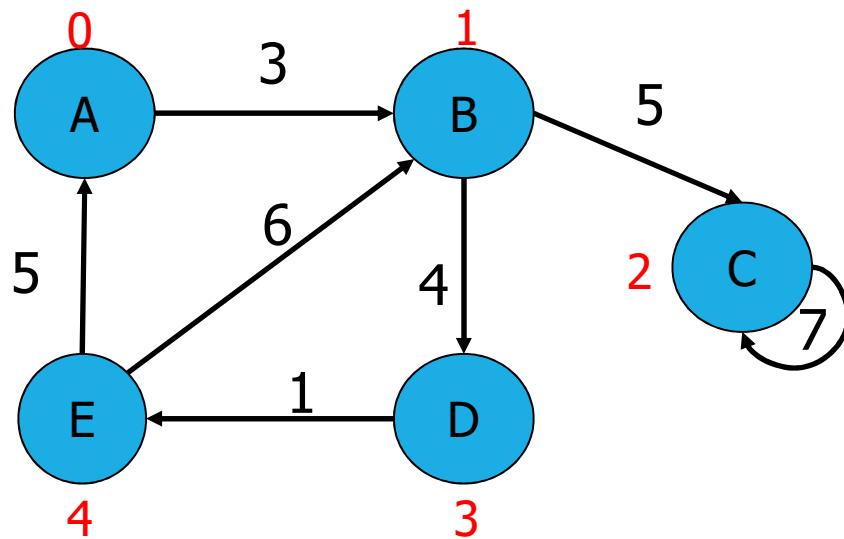
ST

0	A
1	B
2	C
3	D
4	E

G->adj

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

Orientato pesato



Lista archi

A	B	3
B	C	5
B	D	4
E	A	5
E	B	6
C	C	7
D	E	1

ST

0	A
1	B
2	C
3	D
4	E

G->adj

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

Graph.c

numero
di vertici

```
... numero di archi
struct graph {int v; int E; int **madj; ST tab;};
static int **MATRIXint(int r, int c, int val) {
    int i, j;
    int **t = malloc(r * sizeof(int *));
    for (i=0; i < r; i++) t[i] = malloc(c * sizeof(int));
    for (i=0; i < r; i++)
        for (j=0; j < c; j++)
            t[i][j] = val;
    return t;
}
static Edge EDGEcreate(int v, int w, int wt) {
    Edge e;
    e.v = v; e.w = w; e.wt = wt;
    return e;
}
```

matrice di adiacenza

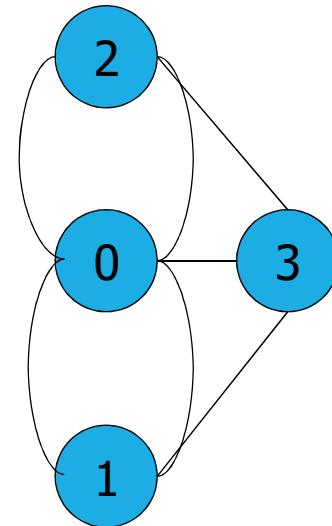
tabella di simboli

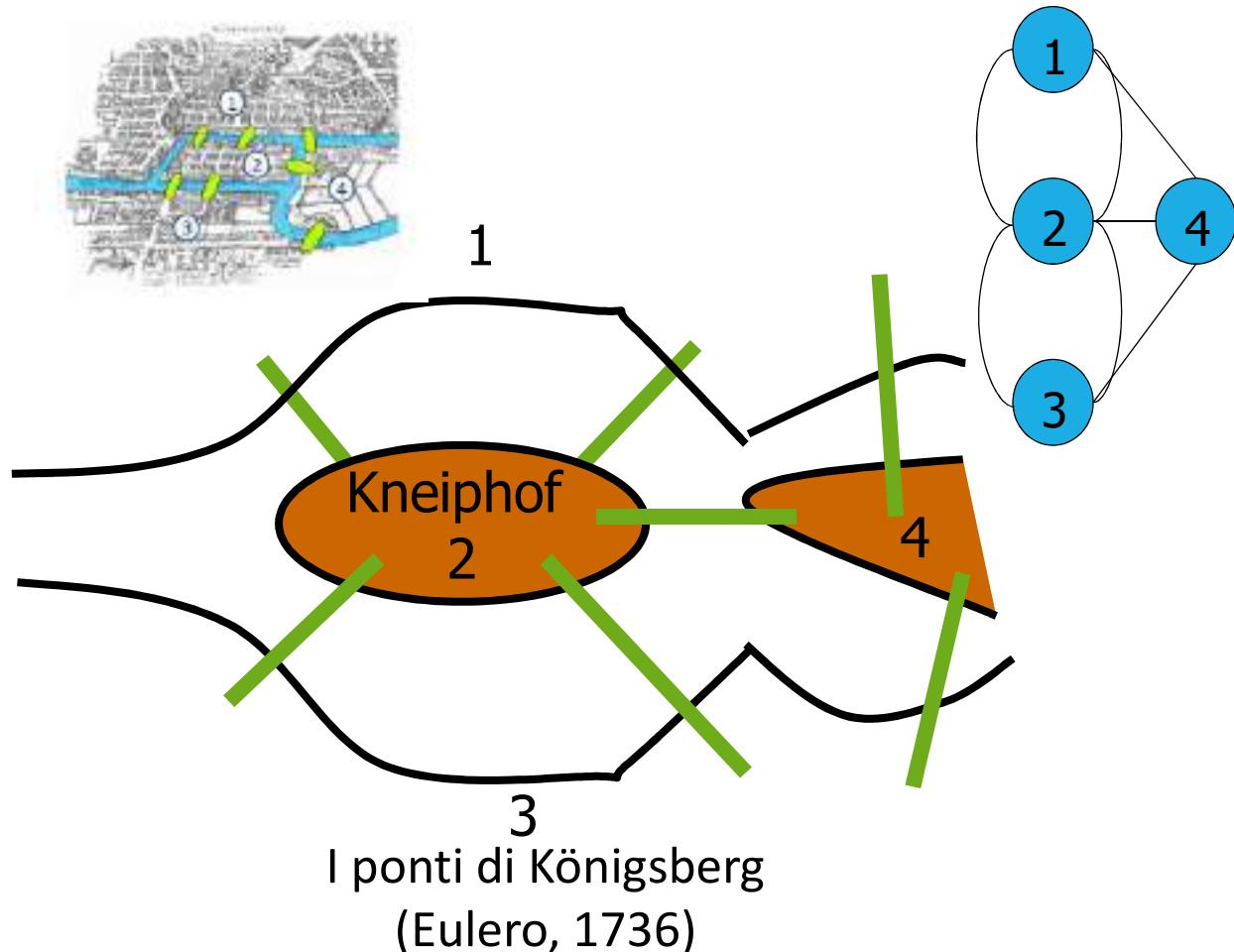
```
Graph GRAPHinit(int v) {
    Graph G = malloc(sizeof *G);
    G->V = V;
    G->E = 0;
    G->madj = MATRIXint(v, V, 0);
    G->tab = STinit(V);
    return G;
}

void GRAPHfree(Graph G) {
    int i;
    for (i=0; i<G->V; i++)
        free(G->madj[i]);
    free(G->madj);
    STfree(G->tab);
    free(G);
}
```

Il multigrafo

Multigrafo: archi multipli che connettono la stessa coppia di vertici.
Si può generare se in fase di inserzione degli archi non si scartano
quelli già esistenti.





grafi non orientati

```
static void insertE(Graph G, Edge e) {
    int v = e.v, w = e.w, wt = e.wt;
    if (G->madj[v][w] == 0)
        G->E++;
    G->madj[v][w] = 1; G->madj[v][w] = wt;
    G->madj[w][v] = 1; G->madj[w][v] = wt;
}
int GRAPHgetIndex(Graph G, char *label) {
    int id;
    id = STsearch(G->tab, label);
    if (id == -1) {
        id = STcount(G->tab);
        STinsert(G->tab, label, id);
    }
    return id;
}
```

grafi pesati

grafi non orientati pesati

Attenzione:

- si possono generare cappi!
- non si possono generare multigrafi!

```

static void removeE(Graph G, Edge e) {
    int v = e.v, w = e.w;
    if (G->madj[v][w] != 0)
        G->E--;
    G->madj[v][w] = 0;
    G->madj[w][v] = 0; ← grafi non orientati
}

void GRAPHedges(Graph G, Edge *a) {
    int v, w, E = 0;
    for (v=0; v < G->V; v++)
        for (w=v+1; w < G->V; w++) ← grafi orientati
            for (w=0; w < G->V; w++) ← grafi pesati
                if (G->madj[v][w] !=0)
                    a[E++] = EDGEcreate(v, w, G->madj[v][w]);
    return;
}

```

Vantaggi/svantaggi

- Complessità spaziale
 $S(n) = \Theta(|V|^2)$ \Rightarrow vantaggiosa SOLO per grafi densi
- No costi aggiuntivi per i pesi di un grafo pesato
- Accesso efficiente ($O(1)$) alla topologia del grafo (adiacenza di 2 vertici).

Rappresentazione

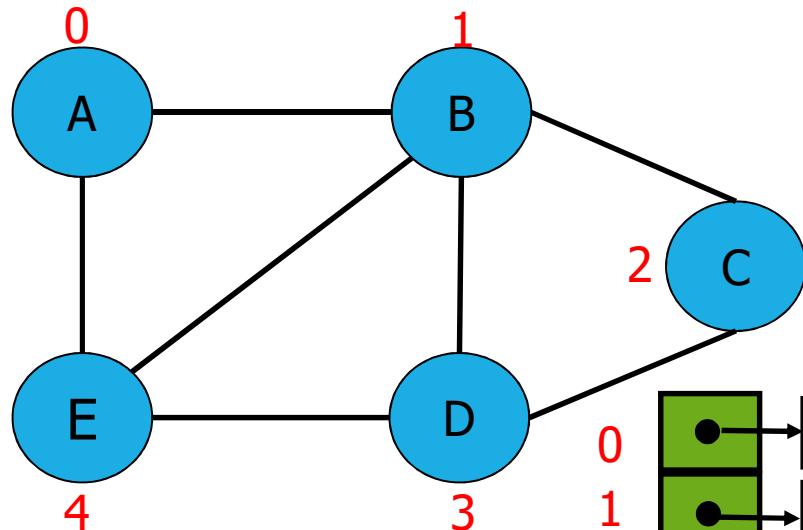
Lista di adiacenza

Dato $G = (V, E)$, la lista di adiacenza è:

- un vettore A di $|V|$ elementi. Il vettore è vantaggioso per via dell'accesso diretto
- $A[i]$ contiene il puntatore alla lista dei vertici adiacenti a i .

Conoscendo il numero di vertici, si può implementare con un vettore di liste, altrimenti lista di liste

Non orientato non pesato



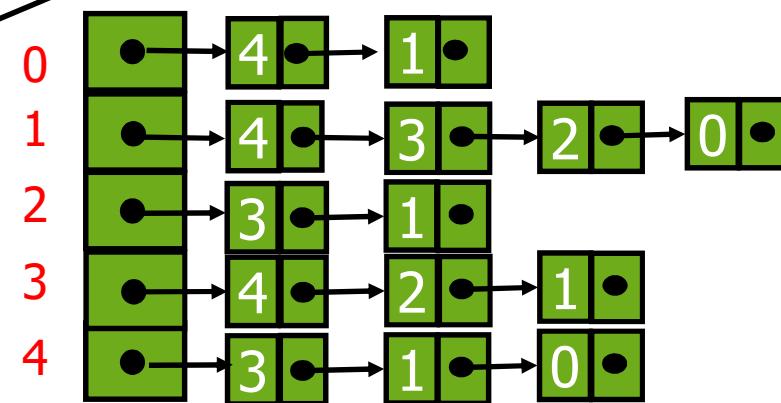
Lista archi

ST	A B
0	B C
1	B D
2	A E
3	B E
4	C D
	D E

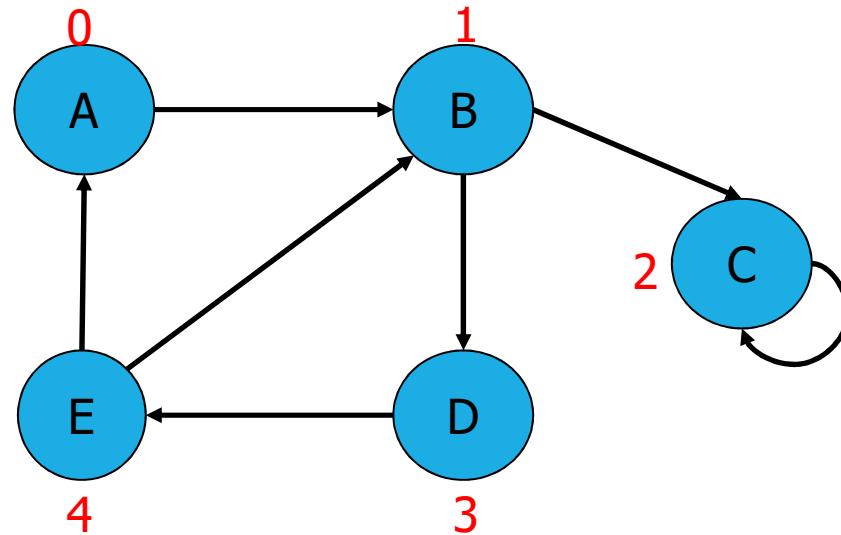
ST

0	A
1	B
2	C
3	D
4	E

G->adj



Orientato non pesato



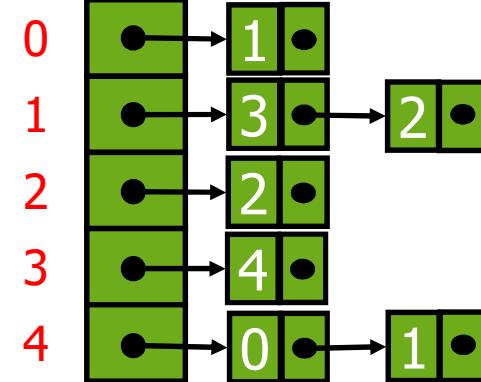
Lista archi

ST
A B
B C
B D
C C
D E
E B
E A

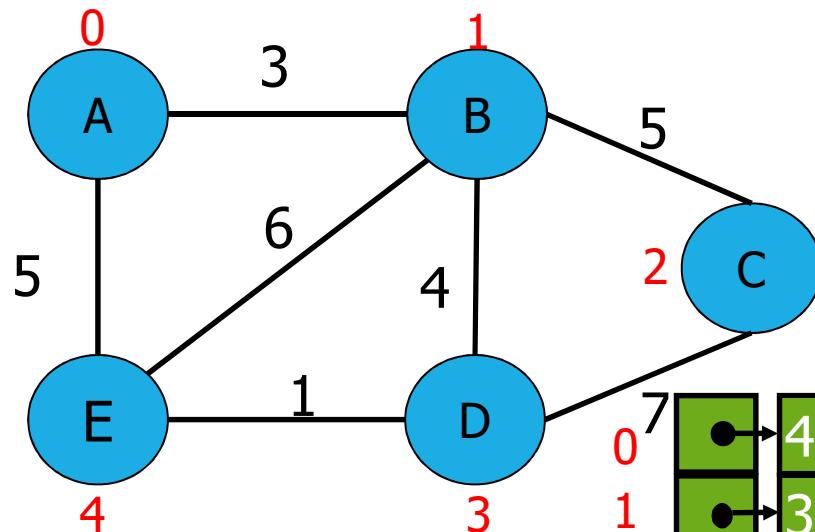
ST

0	A
1	B
2	C
3	D
4	E

G->adj



Non orientato pesato



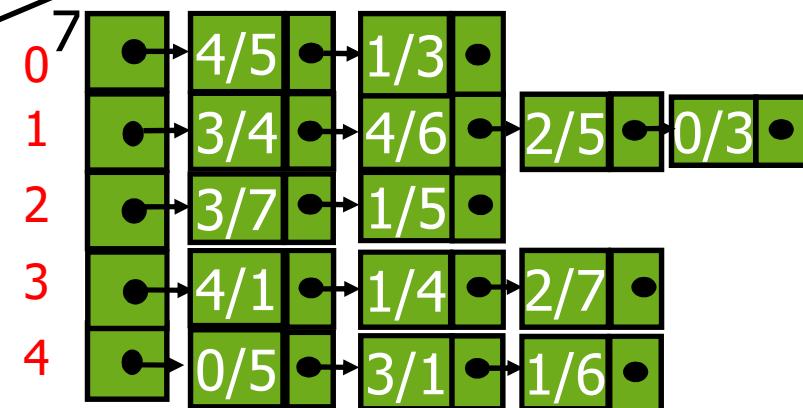
Lista archi

ST	0	1	2	3	4
A	B	C	D	E	
B					
C					
D					
E					

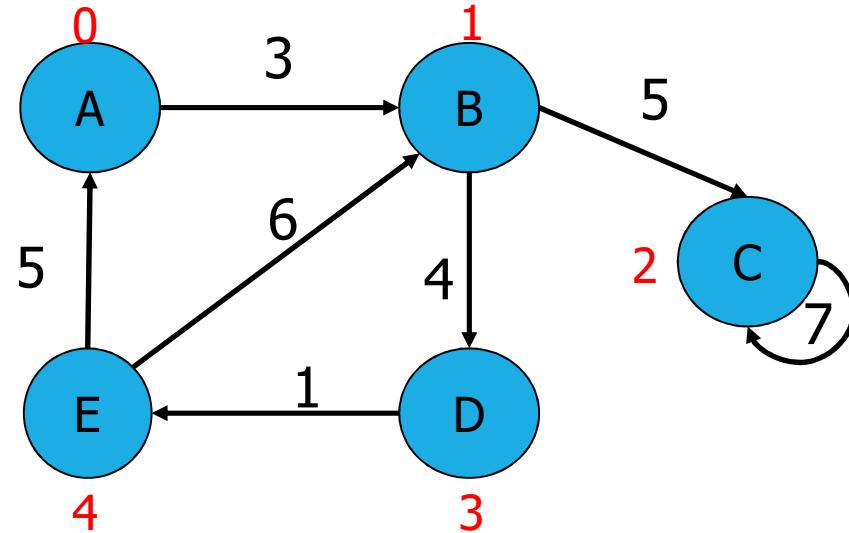
ST

0
1
2
3
4

G->adj



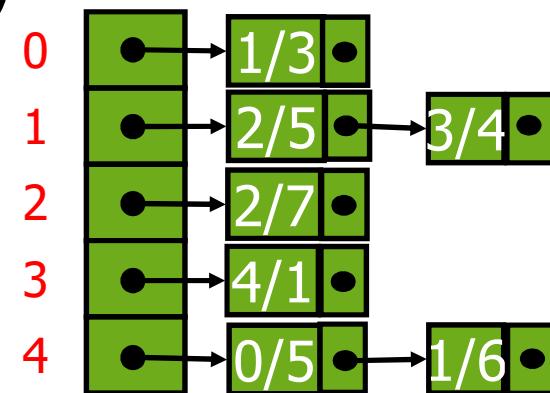
Orientato pesato



Lista archi ST

A	0
B	1
C	2
D	3
E	4

G->adj



Graph.c

```

typedef struct node link;
struct node { int v; int wt; link next; } ;
struct graph{int v; int E; link *ladj; ST tab; link z;} ;

static link NEW(int v, int wt, link next) {
    link x = malloc(sizeof *x),
    x->v = v; x->wt = wt; x->next = next;
    return x;
}

static Edge EDGEcreate(int v, int w, int wt) {
    Edge e;
    e.v = v; e.w = w; e.wt = wt;
    return e;
}

```

```
Graph GRAPHinit(int v) {
    int v;
    Graph G = malloc(sizeof *G);
    G->V = V;
    G->E = 0;
    G->z = NEW(-1, -1, NULL);
    G->ladj = malloc(G->V*sizeof(link));
    for (v = 0; v < G->V; v++)
        G->ladj[v] = G->z;
    G->tab = STinit(v);
    return G;
}
```

```

void GRAPHfree(Graph G) {
    int v;
    link t, next;
    for (v=0; v < G->V; v++)
        for (t=G->ladj[v]; t != G->z; t = next) {
            next = t->next;
            free(t);
        }
    STfree(G->tab);
    free(G->ladj); free(G->z); free(G);
}

void GRAPHedges(Graph G, Edge *a) {
    int v, E = 0;
    link t;
    for (v=0; v < G->V; v++)
        for (t=G->ladj[v]; t != G->z; t = t->next)
            if (v < t->v) a[E++] = EDGEcreate(v, t->v, t->wt);
}

```

grafi pesati



grafi non orientati



Attenzione: si possono generare cappi!

grafi pesati

```
static void insertE(Graph G, Edge e) {  
    int v = e.v, w = e.w, wt = e.wt;  
    G->1adj[v] = NEW(w, wt, G->1adj[v]);  
    G->1adj[w] = NEW(v, wt, G->1adj[w]);  
    G->E++;  
}
```

grafi non orientati (pesati)

```

static void removeE(Graph G, Edge e) {
    int v = e.v, w = e.w;  link x, p;
    for (x = G->1adj[v], p = NULL; x != G->z; p = x, x = x->next) {
        if (x->v == w) {
            if (x == G->1adj[v]) G->1adj[v] = x->next;
            else p->next = x->next;
            break;
        }
    }
    for (x = G->1adj[w], p = NULL; x != G->z; p = x, x = x->next) {
        if (x->v == v) {
            if (x == G->1adj[w]) G->1adj[w] = x->next;
            else p->next = x->next;
            break;
        }
    }
    G->E--;    free(x);
}

```

grafo non orientato



Vantaggi/svantaggi

Vantaggi:

- Grafi non orientati:
 - elementi complessivi nelle liste = $2|E|$
- Grafi orientati:
 - elementi complessivi nelle liste = $|E|$
- Complessità spaziale
 - $S(n) = O(\max(|V|, |E|)) = O(|V+E|)$ \Rightarrow vantaggioso per grafi sparsi

Svantaggi:

- verifica dell'adiacenza di 2 vertici v e w mediante scansione della lista di adiacenza di v
- uso di memoria per i pesi dei grafi pesati.

Generazione dei grafi

In generale i grafi sono modelli di situazioni reali e vengono forniti come dati di ingresso. In caso contrario, si possono generare dei grafi senza alcuna relazione ad un problema specifico.

Tecnica 1: archi casuali

- Vertici come interi tra 0 e $|V|-1$
- generazione di un grafo casuale a partire da E coppie casuali di archi (interi tra 0 e $|V|-1$)
 - possibili archi ripetuti (multigrafo) e cappi
 - grafo con $|V|$ vertici e $|E|$ archi (inclusi cappi e archi ripetuti)

grafi non orientati

Tecnica 2: archi con probabilità p

- si considerano tutti i possibili archi $|V| * (|V| - 1)/2$
- tra questi si selezionano quelli con probabilità p
- p è calcolato in modo che sia

$$|E| = p * (|V| * (|V| - 1)/2)$$

quindi

$$p = 2 * |E| / (|V| * (|V| - 1))$$

- si ottiene un grafo con in media $|E|$ archi
- non ci sono archi ripetuti.

```

int randv(Graph G) {
    return G->v * (rand() / (RAND_MAX + 1.0));
}

Graph GRAPHrand1(Graph G, int v, int E) {
    while (G->E < E)
        GRAPHinsertE(G, randv(G), randv(G));
    return G;
}

Graph GRAPHrand2(Graph G, int v, int E) {
    int i, j; double p = 2.0 * E / (v * (v-1));
    for (i = 0; i < v; i++)
        for (j = i+1; j < v; j++)
            if (rand() < p * RAND_MAX)
                GRAPHinsertE(G, i, j);
    return G;
}

```

Cammino semplice

Dato un grafo non orientato $G = (V, E)$ e 2 suoi vertici v e w , esiste un cammino semplice che li connette? **Non è richiesto trovarli tutti.**

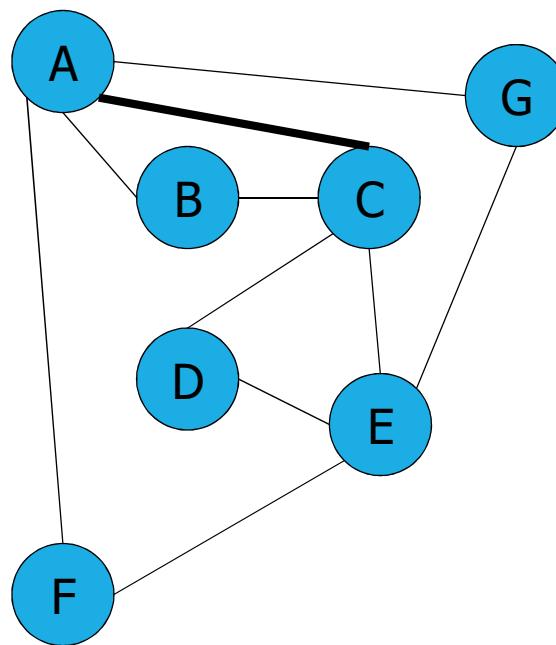
Se il grafo non orientato è connesso \Rightarrow il cammino esiste per definizione, basta trovarne uno qualsiasi senza altri vincoli se non essere semplice. Non serve backtrack.

Se il grafo non orientato non è connesso \Rightarrow il cammino esiste per definizione se i vertici sono nella stessa componente connessa, altrimenti non esiste. Non serve backtrack.

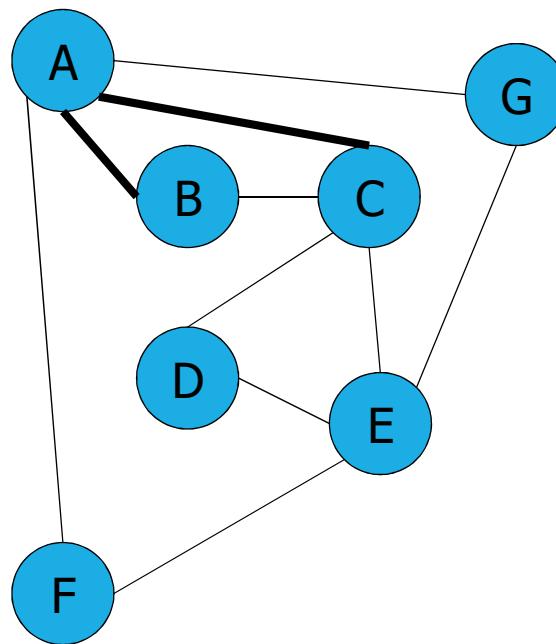
$\exists p: v \rightarrow_p w$

- \forall vertice t adiacente al vertice corrente v, determinare ricorsivamente se esiste un cammino semplice da t a w
- array `visited[maxV]` per marcare i nodi già visitati
- cammino visualizzato in ordine inverso
- complessità $T(n) = O(|V+E|)$

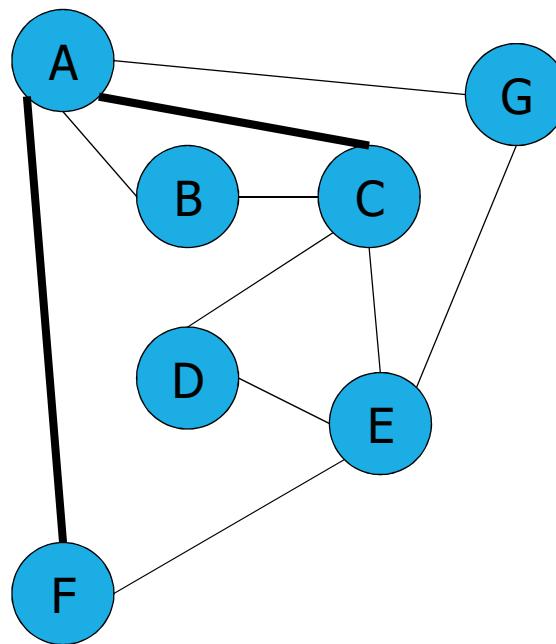
Esempio



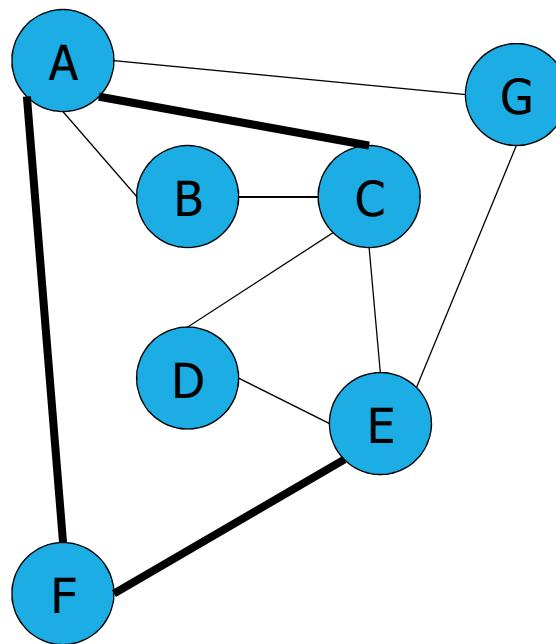
$\exists p: C \xrightarrow{p} G?$
passo 1:
vertici adiacenti a C
non ancora visitati:
A, B, D, E
seleziono A



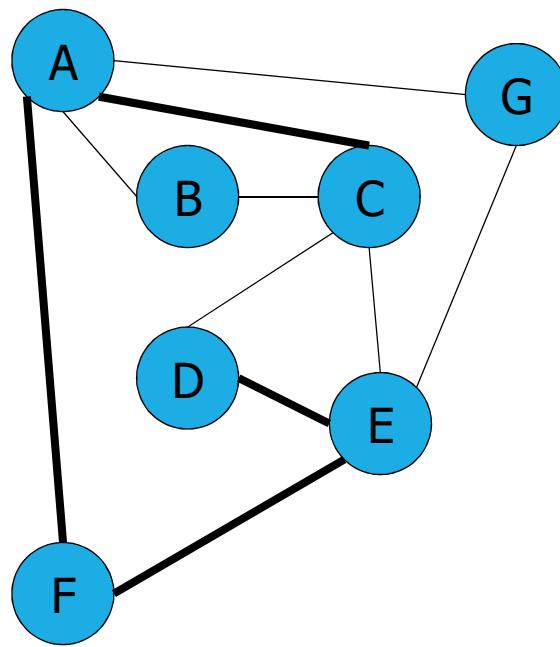
$\exists p: A \xrightarrow{p} G?$
passo 2:
vertici adiacenti a A
non ancora visitati:
B, F, G
seleziono B



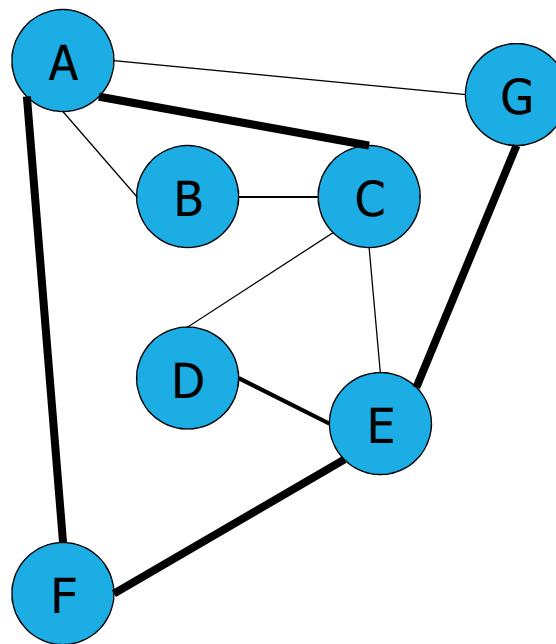
$\exists p: B \xrightarrow{p} G?$
passo 3:
vertici adiacenti a B
non ancora visitati:
nessuno
La ricorsione torna a A
e seleziona F



$\exists p: F \rightarrow_p G?$
passo 4:
vertici adiacenti a F
non ancora visitati:
E
seleziono E



$\exists p: E \rightarrow_p G?$
passo 5:
vertici adiacenti a E
non ancora visitati:
D, G
seleziono D



$\exists p: D \rightarrow_p G?$

passo 6:

vertici adiacenti a D

non ancora visitati:

nessuno

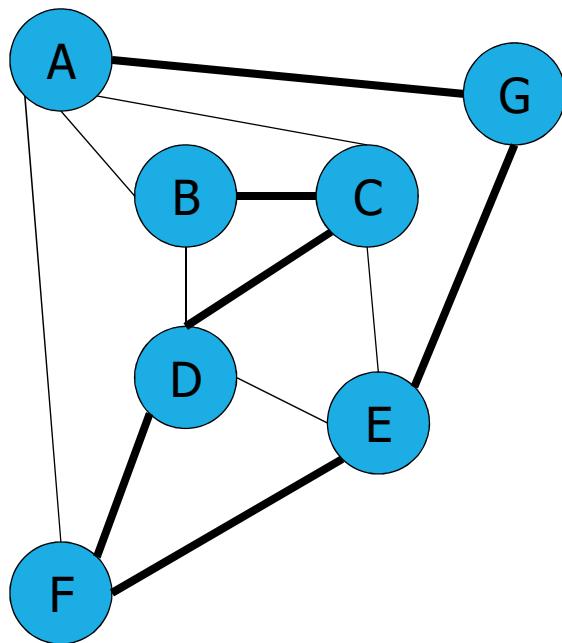
La ricorsione torna a E e
seleziona G

Il Cammino di Hamilton

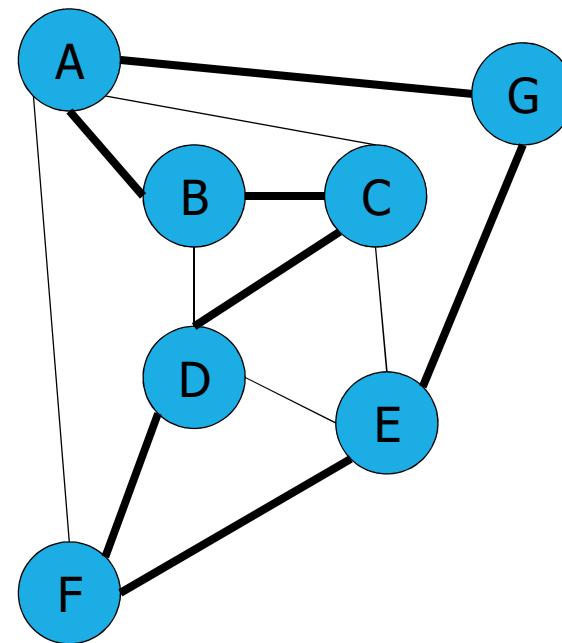
Dato un grafo non orientato $G = (V, E)$ e 2 suoi vertici v e w , se esiste un cammino semplice che li connette visitando ogni vertice una e una sola volta, questo si dice **cammino di Hamilton**.

Se v coincide con w , si parla di **ciclo di Hamilton**.

Esempio



Cammino di Hamilton tra A e B



Ciclo di Hamilton

Algoritmo

Cammino di Hamilton tra v e w:

- \forall vertice t adiacente al vertice corrente v, determinare ricorsivamente se esiste un cammino semplice da t a w
- ritorno con successo se e solo se la lunghezza del cammino è $|V|-1$
- set della cella dell'array **visited** per marcare i nodi già visitati
- reset della cella dell'array **visited** quando ritorna con insuccesso (backtrack)
- complessità **esponenziale!**

```
void GRAPHpath/GRAphpathH(Graph G, int id1, int id2) {  
    int t, found, *visited;  
  
    visited = malloc(G->v*sizeof(int));  
    for (t=0; t<G->V; t++)  
        visited[t]=0;  
  
    if (id1 < 0 || id2 < 0)  
        return;  
    found = pathR/pathRH(G, id1, id2, G->v-1, visited);  
    if (found == 0)  
        printf("\n Path not found!\n");  
}
```

Attenzione: per
sinteticità si viola
la sintassi del C

```

static int pathR(Graph G, int v, int w, int *visited) {
    int t;
    if (v == w)
        return 1;
    visited[v] = 1;
    for (t = 0 ; t < G->v ; t++)
        if (G->madj[v][t] == 1)
            if (visited[t] == 0)
                if (pathR(G, t, w, visited)) {
                    printf("(%s, %s) in path\n",
                           STsearchByIndex(G->tab, v),
                           STsearchByIndex(G->tab, t));
                    return 1;
                }
    return 0;
}

```

matrice delle
adiacenze

stampa gli archi
del cammino in
ordine inverso

```

static int pathRH(Graph G, int v, int w, int d, int *visited) {
    int t;
    if (v == w) {
        if (d == 0) return 1;
        else return 0;
    }
    visited[v] = 1;
    for (t = 0 ; t < G->V ; t++)
        if (G->adj[v][t] == 1)
            if (visited[t] == 0)
                if (pathRH(G, t, w, d-1, visited)) {
                    printf("(%s, %s) in path \n", STsearchByIndex(G->tab, v),
                           STsearchByIndex(G->tab, t));
                    return 1;
                }
    visited[v] = 0;
    return 0;
}

```

matrice delle
adiacenze

intero che indica
quanto manca a un
cammino lungo $|V|-1$

stampa gli archi
del cammino in
ordine inverso

backtrack

Il Cammino di Eulero

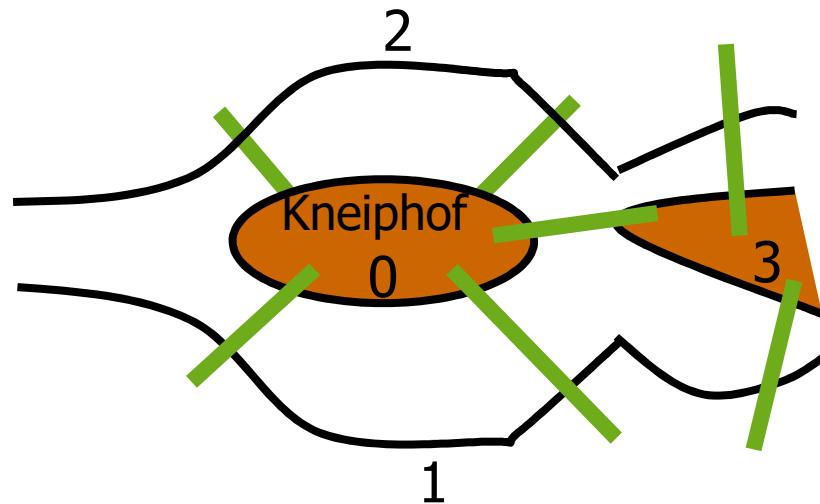
Dato un grafo non orientato $G = (V, E)$ e 2 suoi vertici v e w , si dice **cammino di Eulero** un cammino (anche non semplice) che li connette attraversando ogni arco una e una sola volta.
Se v coincide con w , si parla di **ciclo di Eulero**.



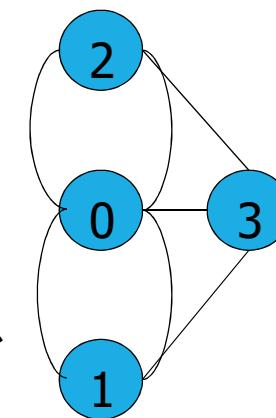
Lemmi

- Un grafo non orientato ha un **ciclo di Eulero** se e solo se è connesso e tutti i suoi vertici sono di grado pari
- Un grafo non orientato ha un **cammino di Eulero** se e solo se è connesso e se esattamente due vertici hanno grado dispari.

I ponti di Königsberg

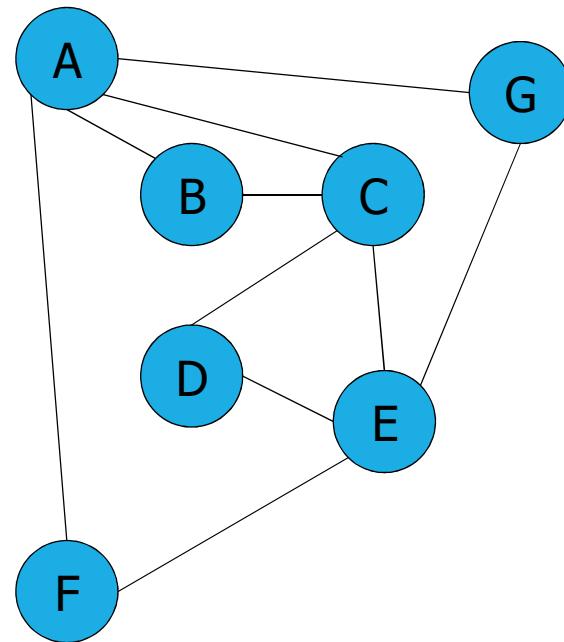


\nexists né cammini, né cicli di Eulero



Multigrafo: archi multipli che connettono la stessa coppia di vertici

Esempio: verifica dell'esistenza del ciclo di Eulero



nodo	deg
A	4
B	2
C	4
D	2
E	4
F	2
G	2

Ciclo di Eulero:
(A, B), (B, C), (C, A), (A, G), (G, E)
(E, D), (D, C), (C, E), (E, F), (F, A)

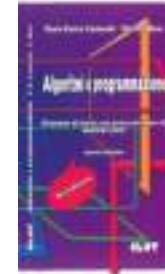
Algoritmo di complessità $O(|E|)$

Riferimenti

- L'ADT grafo non orientato:
 - Sedgewick Part 5: 17.2
- Rappresentazione dei grafi:
 - Sedgewick Part 5: 17.3, 17.4
 - Cormen 23.1
- Generazione di grafi:
 - Sedgewick Part 5: 17.6
- Cammini semplici, di Hamilton, di Eulero:
 - Sedgewick Part 5: 17.6
 - Cormen 36.2, 36.5.4

Esercizi di teoria

- 10. Visite dei grafi e applicazioni
 - 10.1 Rappresentazioni



Gli algoritmi di visita dei grafi

Gianpiero Cabodi e Paolo Camurati



Algoritmi di visita

Visita di un grafo $G=(V, E)$:

- a partire da un vertice dato, seguendo gli archi con una certa strategia, elencare i vertici incontrati, eventualmente aggiungendo altre informazioni.

Algoritmi:

- in profondità (depth-first search, DFS)
- in ampiezza (breadth-first search, BFS).

Visita in profondità (versione base)

Dato un grafo (connesso o non connesso), a partire da un vertice s , visita **tutti** i vertici del grafo (raggiungibili da s e non).

Principio base della visita in profondità: espandere l'ultimo vertice scoperto che ha ancora vertici non ancora scoperti adiacenti.

Scoperta di un vertice: prima volta che si incontra nella visita (discesa ricorsiva, visita in pre-order). Il vettore `pre` gioca il ruolo del vettore `mark` nel Calcolo Combinatorio, ma contiene i tempi crescenti di scoperta.

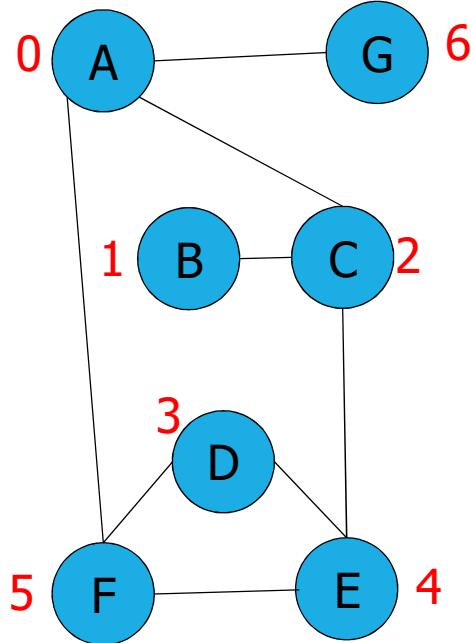
Algoritmo

wrapper

- GRAPHsimpleDfs: funzione che, a partire da un vertice dato, visita tutti i vertici di un grafo, richiamando la procedura ricorsiva SimpleDfsR. Termina quando tutti i vertici sono stati visitati.
- SimpleDfsR: funzione che visita in profondità a partire da un vertice `id` identificato con un arco fittizio EDGEcreate(`id, id`) utile in fase di visualizzazione. Termina quando ha visitato in profondità tutti i nodi raggiungibili da `id`.

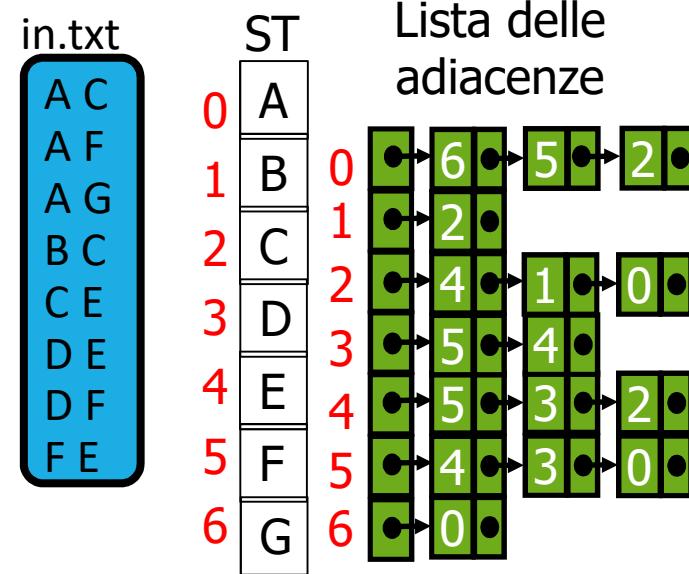
B: alcuni autori chiamano visita in profondità la sola SimpleDfsR.

Esempio

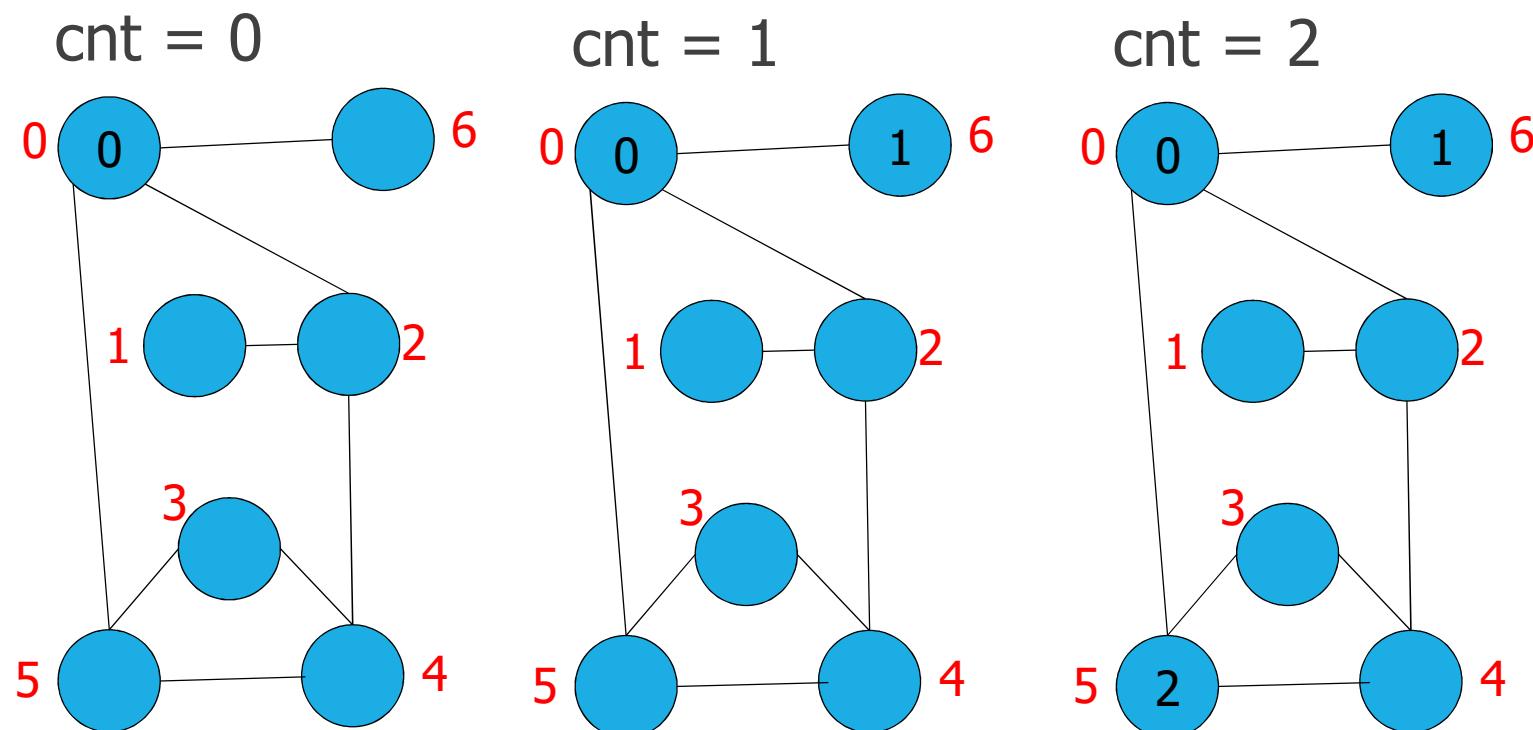


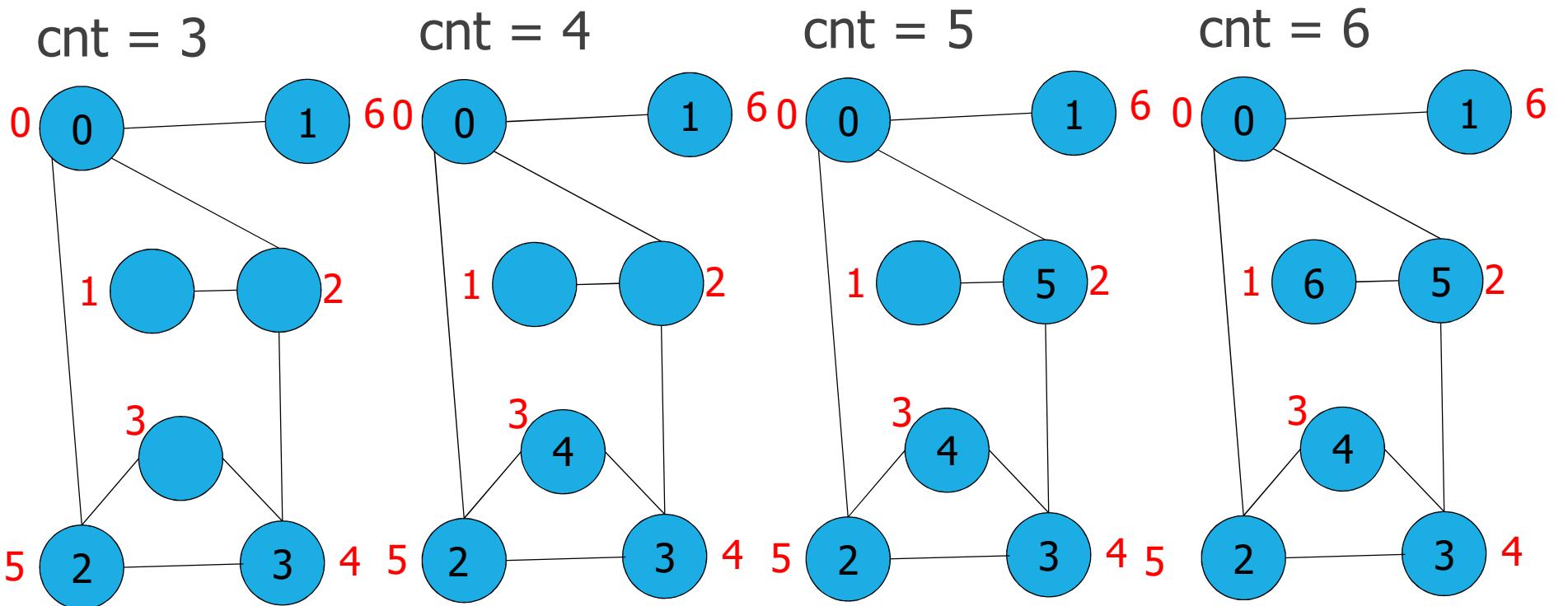
in.txt

ST	A C
0	A F
1	A G
2	B C
3	C E
4	D E
5	D F
6	F E



Esempio





Strutture dati

- grafo non pesato come lista delle adiacenze
- vettore `pre[i]` dove per ciascun vertice si registra il tempo di scoperta (numerazione in preordine dei vertici)
- contatore `cnt` per tempi di scoperta

`cnt` e `*pre` sono locali alla funzione `GRAPHsimpleDfs` e passati by reference alla funzione ricorsiva `SimpleDfsR`.

```

void GRAPHsimpleDfs(Graph G, int id) {
    int v, cnt=0, *pre;
    pre = malloc(G->V * sizeof(int));
    if ((pre == NULL)) return;
    for (v=0; v<G->V; v++) pre[v]=-1; visita a partire da id
        simpleDfsR(G, EDGEcreate(id,id), &cnt, pre);

    for (v=0; v < G->V; v++) visita dei nodi non ancora visitati
        if (pre[v]== -1)
            simpleDfsR(G, EDGEcreate(v,v), &cnt, pre);

    printf("discovery time labels \n");
    for (v=0; v < G->V; v++)
        printf("vertex %s : %d \n", STsearchByIndex(G->tab, v), pre[v]);
}

```

```
static void simpleDfsR(Graph G, Edge e, int *cnt, int *pre) {  
    link t; int w = e.w;  
    pre[w] = (*cnt)++;  
    for (t = G->ladj[w]; t != G->z; t = t->next)  
        if (pre[t->v] == -1)  
            simpleDfsR(G, EDGEcreate(w, t->v), cnt, pre);  
}
```

terminazione implicita
della ricorsione

Visita in profondità (versione estesa)

Estensione:

- nodi etichettati con etichetta tempo di scoperta / tempo di completamento
- foresta di alberi della visita in profondità, memorizzata in un vettore.

Scoperta di un vertice: prima volta che si incontra nella visita (discesa ricorsiva, visita in pre-order), vettore $\text{pre}[i]$.

Completamento: fine dell'elaborazione del vertice (uscita dalla ricorsione, visita in post-order) vettore $\text{post}[i]$.

Scoperta/Completamento: tempo discreto che avanza mediante contatore time . Avanzamento quando si scopre o si completa.

Identificazione del padre nella visita in profondità: vettore $\text{st}[i]$.

Algoritmo

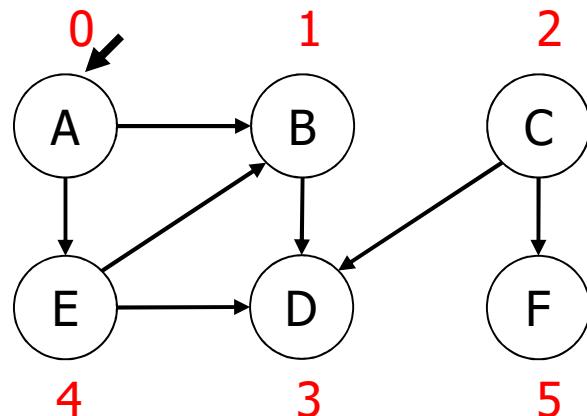
wrapper

- GRAPHextendedDfs: funzione che, a partire da un vertice dato, visita tutti i vertici di un grafo, richiamando la procedura ricorsiva ExtendedDfsR. Termina quando tutti i vertici sono stati visitati.
- ExtendedDfsR: funzione che visita in profondità a partire da un vertice `id` identificato con un arco fittizio `EDGEcreate(id, id)` utile in fase di visualizzazione. Termina quando ha visitato in profondità tutti i nodi raggiungibili da `id`.

B: alcuni autori chiamano visita in profondità la sola ExtendedDfsR.

Esempio

time = 0 st



in.txt

A	B
C	D
A	E
C	F
B	D
E	B
E	D

ST	
0	A
1	B
2	C
3	D
4	E
5	F

Lista delle
adiacenze

st

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

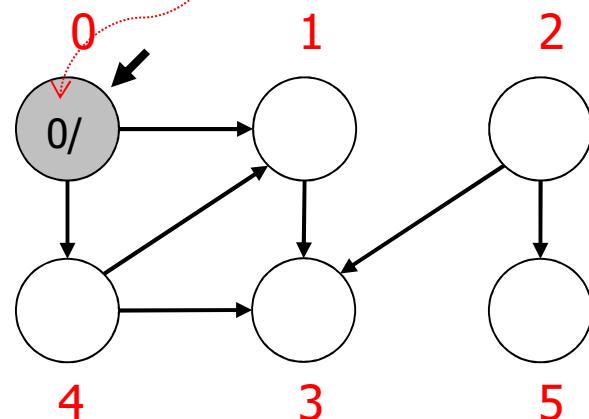
Convenzione grafica:

- nodo bianco: non ancora scoperto
- nodo grigio: scoperto ma non terminato
- nodo nero: terminato

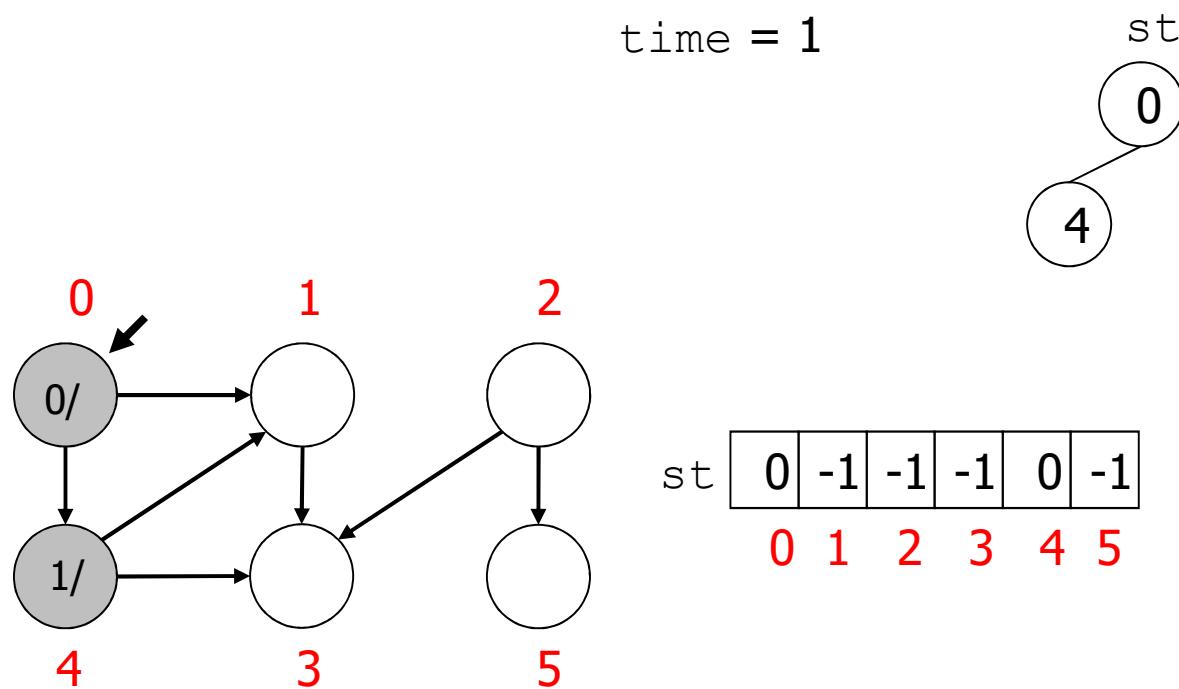
time = 0

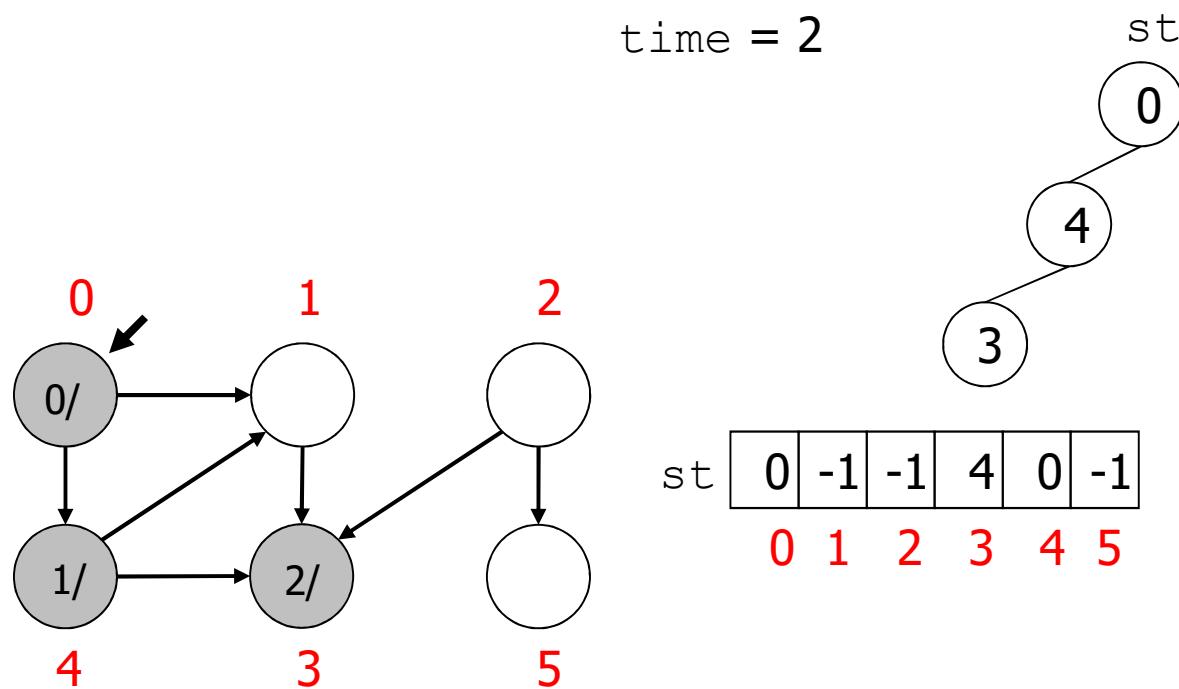
st
0

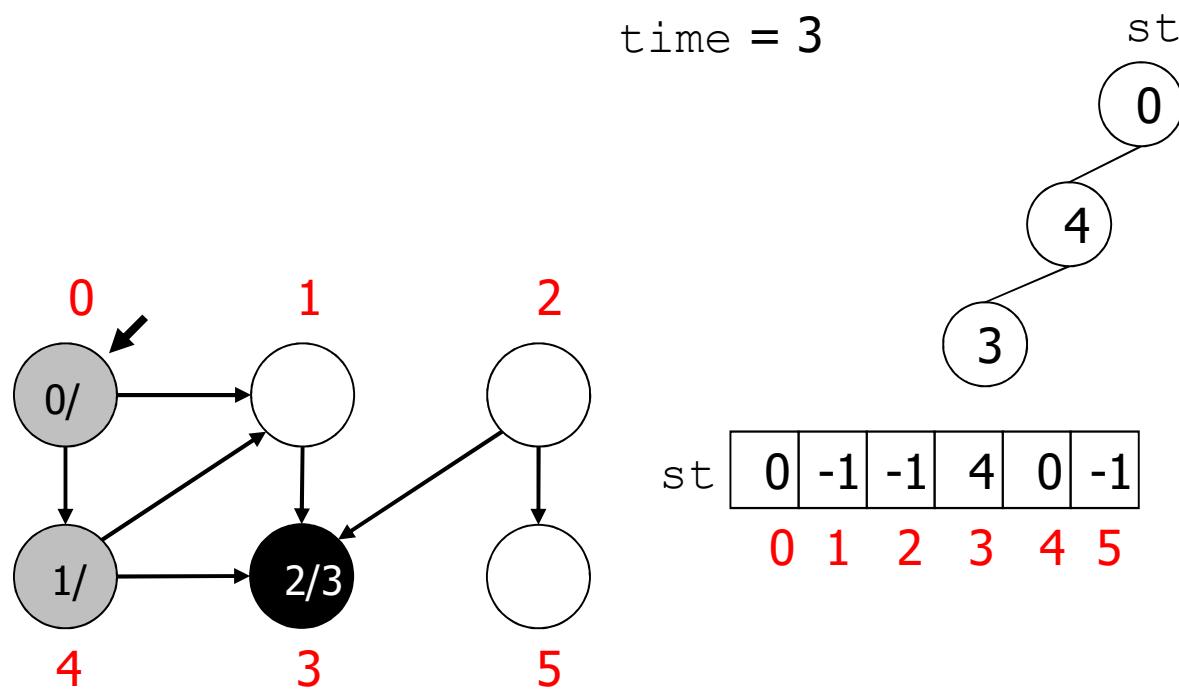
pre[i]/post[i]

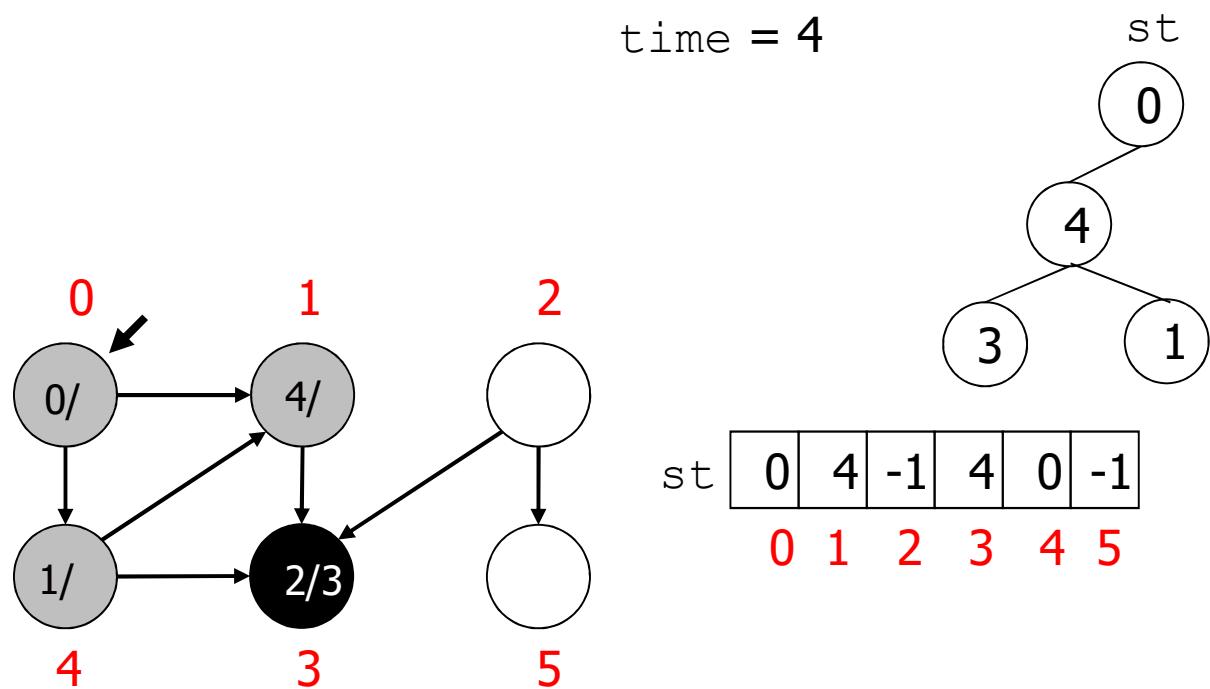


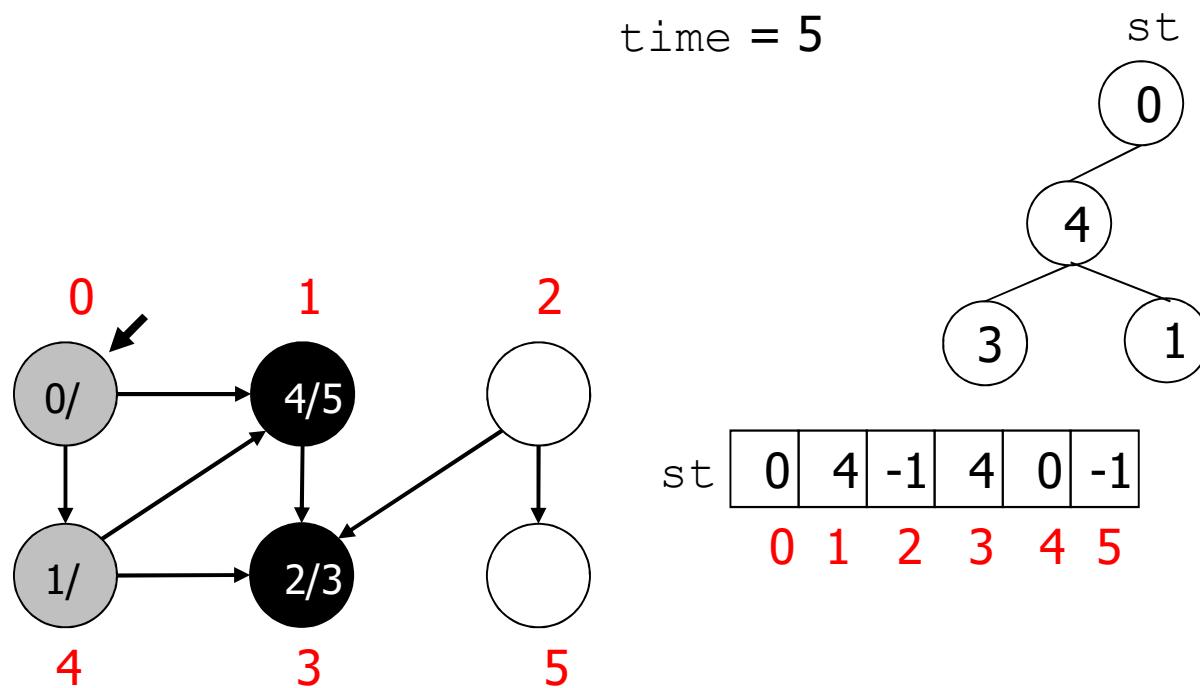
st	0	-1	-1	-1	-1	-1
	0	1	2	3	4	5

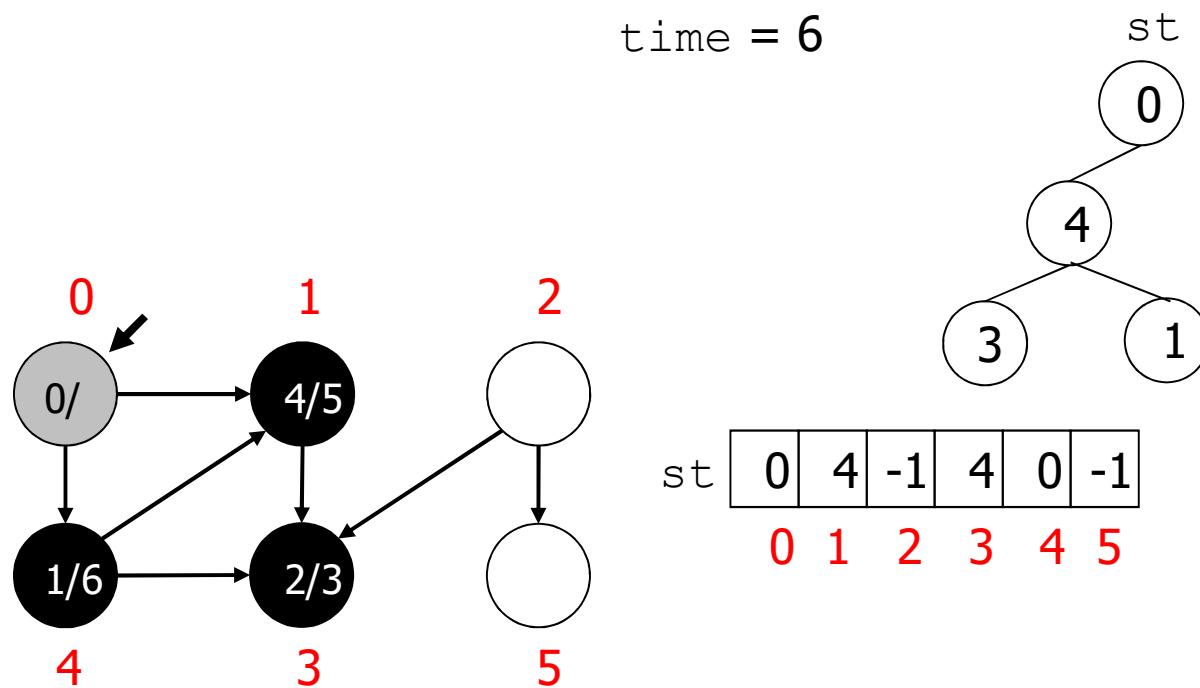


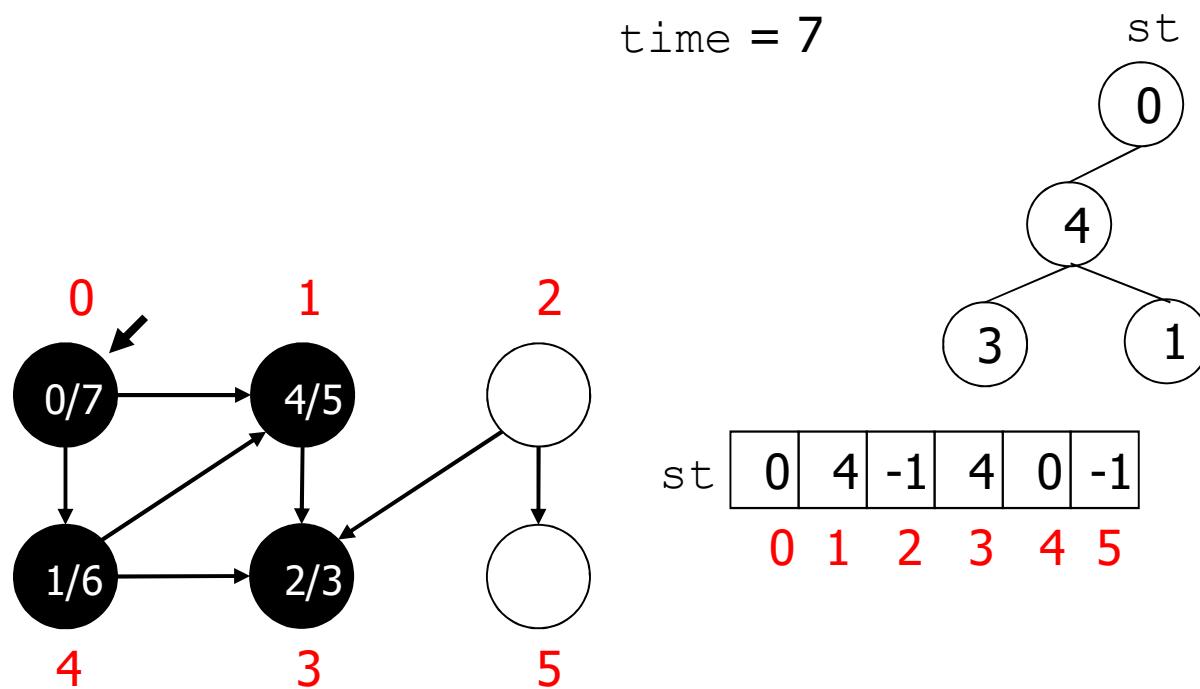


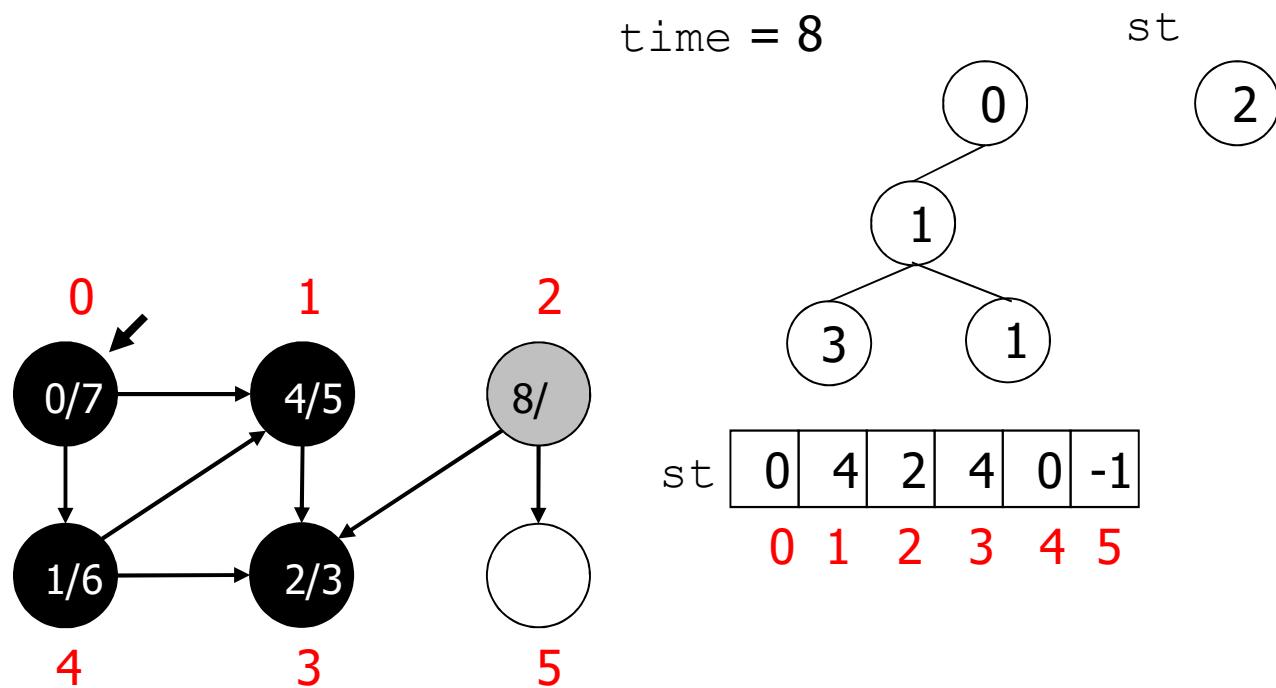


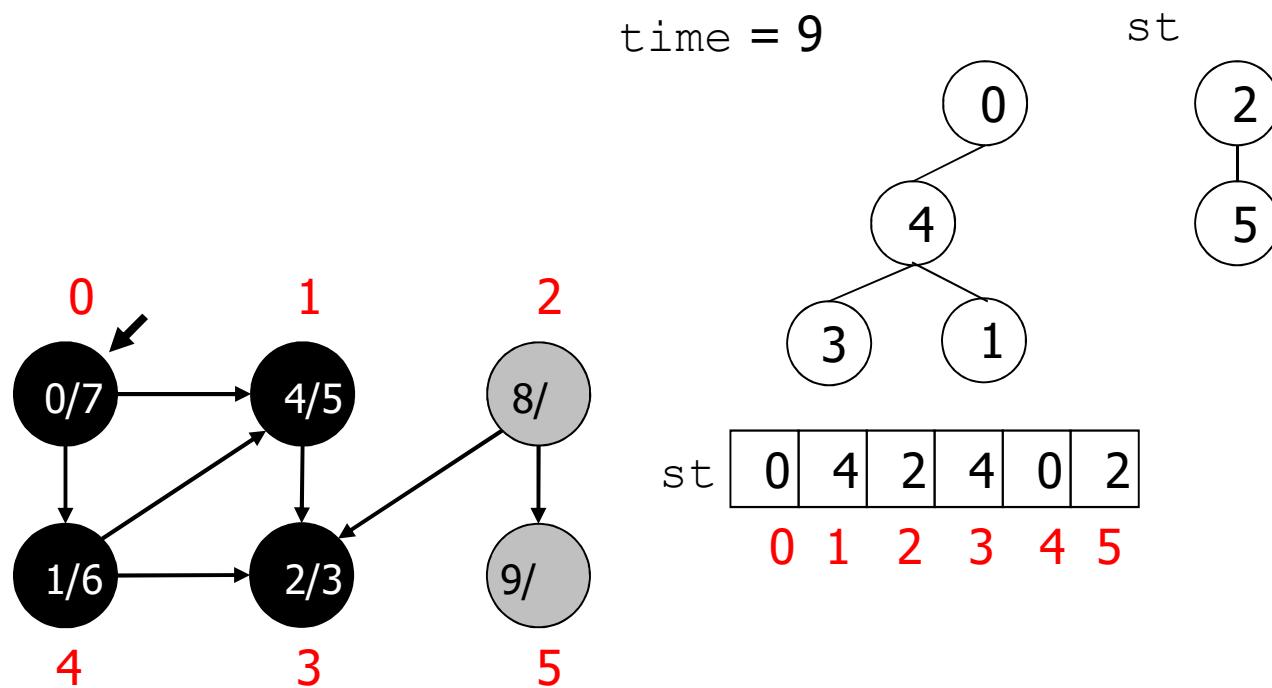


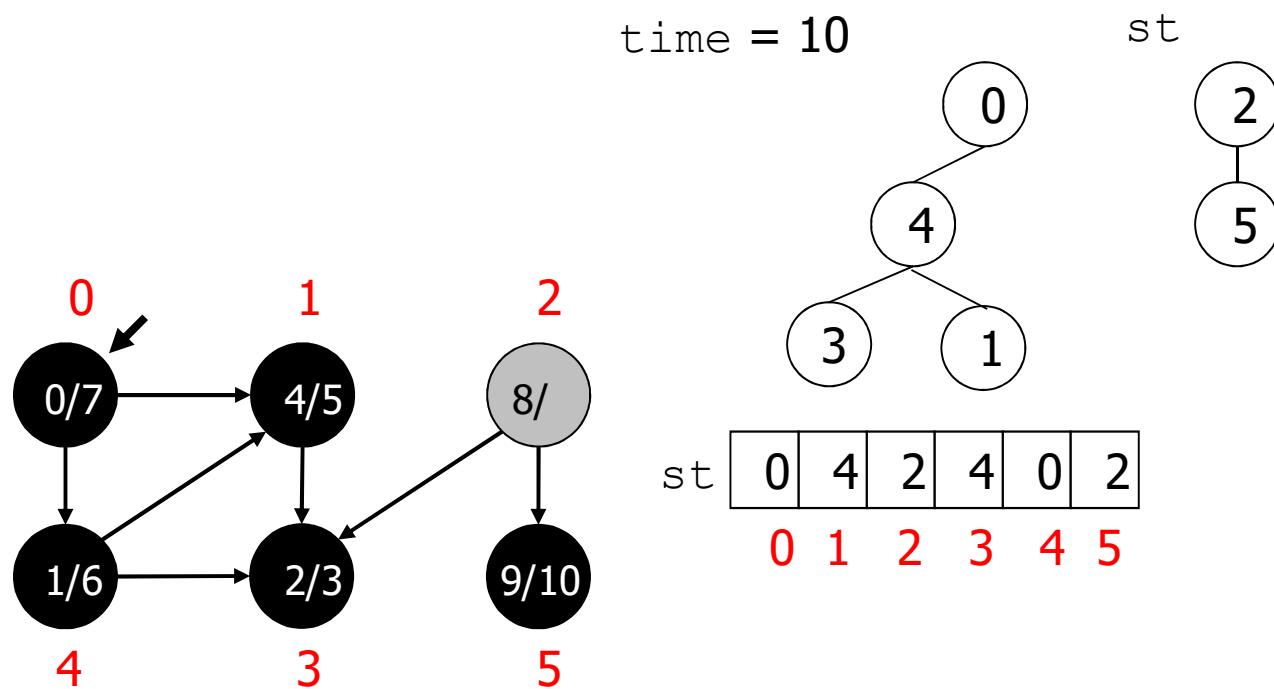


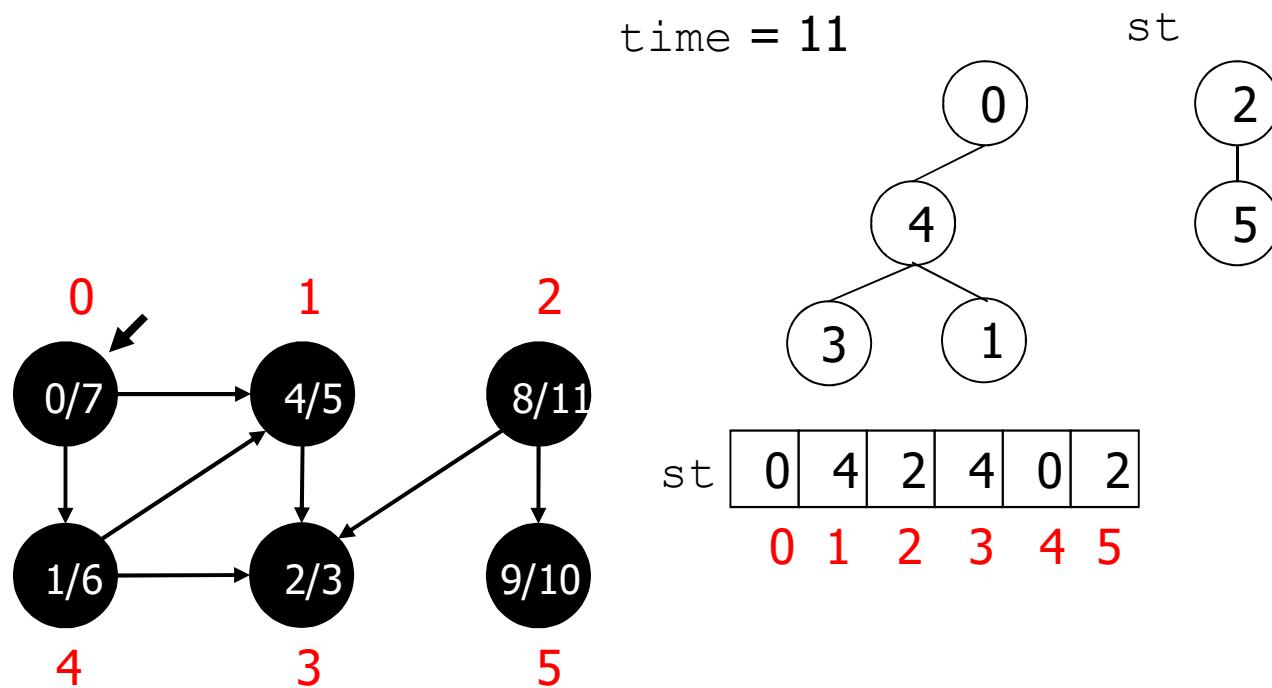












Strutture dati

- grafo non pesato come lista delle adiacenze
- vettori dove per ciascun vertice:
 - si registra il tempo di scoperta (numerazione in preordine dei vertici) `pre[i]`
 - si registra il tempo di completamento (numerazione in postordine dei vertici) `post[i]`
 - si registra il padre per la costruzione della foresta degli alberi della visita in profondità: `st[i]`
- contatore `time` per tempi di scoperta/completamento
`time`, `*pre`, `*post` e `*st` sono locali alla funzione `GRAPHextendedDfs` e passati by reference alla funzione ricorsiva `ExtendedDfsR`.

```

void GRAPHextendedDfs(Graph G, int id) {
    int v, time=0, *pre, *post, *st;
    pre/post/st = malloc(G->V * sizeof(int));
    for (v=0; v < G->V; v++) {
        pre[v]=-1; post[v]=-1; st[v]=-1;}
    extendedDfsR(G, EDGEcreate(id,id), &time, pre, post, st);
    for (v=0; v < G->V; v++)
        if (pre[v]==-1)
            extendedDfsR(G, EDGEcreate(v,v),&time,pre,post,st);
    printf("discovery/endprocessing time labels \n");
    for (v=0; v < G->V; v++)
        printf("%s:%d/%d\n",STsearchByIndex(G->tab,v),pre[v],post[v]);
    printf("resulting DFS tree \n");
    for (v=0; v < G->V; v++)
        printf("%s's parent: %s \n",STsearchByIndex (G->tab, v),
               STsearchByIndex (G->tab, st[v]));
}

```

```
static void ExtendedDfsR(Graph G, Edge e, int *time, int *pre,
                         int *post, int *st) {
    link t;
    int w = e.w;
    st[e.w] = e.v;
    pre[w] = (*time)++;
    for (t = G->ladj[w]; t != G->z; t = t->next)
        if (pre[t->v] == -1)
            ExtendedDfsR(G, EDGEcreate(w, t->v), time, pre, post, st);
    post[w] = (*time)++;
}
```

terminazione implicita
della ricorsione

Visita in profondità (versione completa)

Si etichetta ogni arco:

- grafi orientati: T(tree), **B**(backward), **F**(forward), **C**(cross)
- grafi non orientati: T(tree), **B**(backward)

Classificazione degli archi

Grafo orientato:

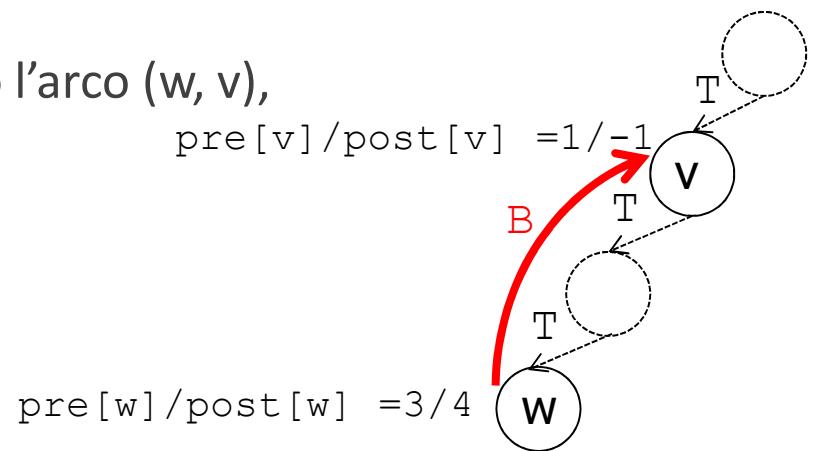
- **T**: archi dell'albero della visita in profondità
- **B**: connettono un vertice w ad un suo antenato v nell'albero:

tempo di fine elaborazione di v **sarà** > tempo di fine elaborazione di w .

Equivale a testare se, quando scopro l'arco (w, v) ,
 $\text{post}[v] == -1$

$$\text{pre}[v] / \text{post}[v] = 1 / -1$$

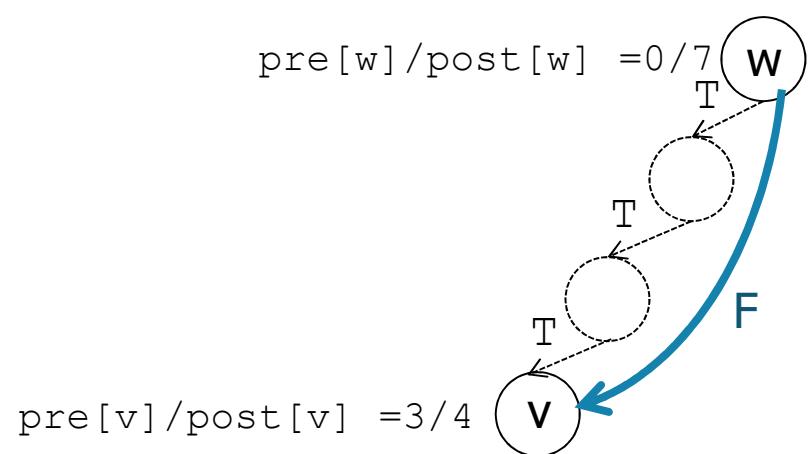
$$\text{pre}[w] / \text{post}[w] = 3 / 4$$



- **F**: connettono un vertice w ad un suo discendente v nell'albero:

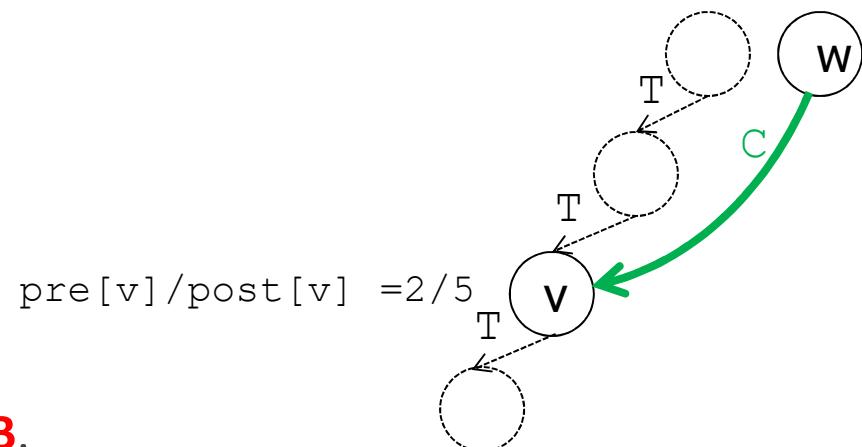
tempo di scoperta di v è > tempo di scoperta di v quando scopro l'arco (w, v)

$$\text{pre}[v] > \text{pre}[w]$$



- **C**: archi rimanenti, per cui
tempo di scoperta di w **è** > tempo di scoperta di v quando scopro
l'arco (w, v)
 $\text{pre}[w] > \text{pre}[v]$

$$\text{pre}[w]/\text{post}[w] = 8/-1$$



Grafo non orientato: solo archi **T** e **B**.

Algoritmo

wrapper

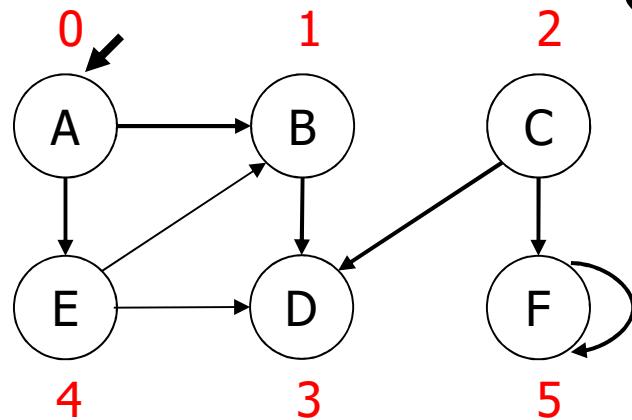
- ~~GRAPHdfs~~: funzione che, a partire da un vertice dato, visita tutti i vertici di un grafo, richiamando la procedura ricorsiva `dfsR`. Termina quando tutti i vertici sono stati visitati.
- `dfsR`: funzione che visita in profondità a partire da un vertice `id` identificato con un arco fittizio `EDGEcreate(id, id)` utile in fase di visualizzazione. Termina quando ha visitato in profondità tutti i nodi raggiungibili da `id`.

NB: alcuni autori chiamano visita in profondità la sola `dfsR`.

Strutture dati: come nella versione estesa. Codice della `GRAPHdfs` identico a quello della `GRAPHextendedDfs`.

Esempio

time = 0 st



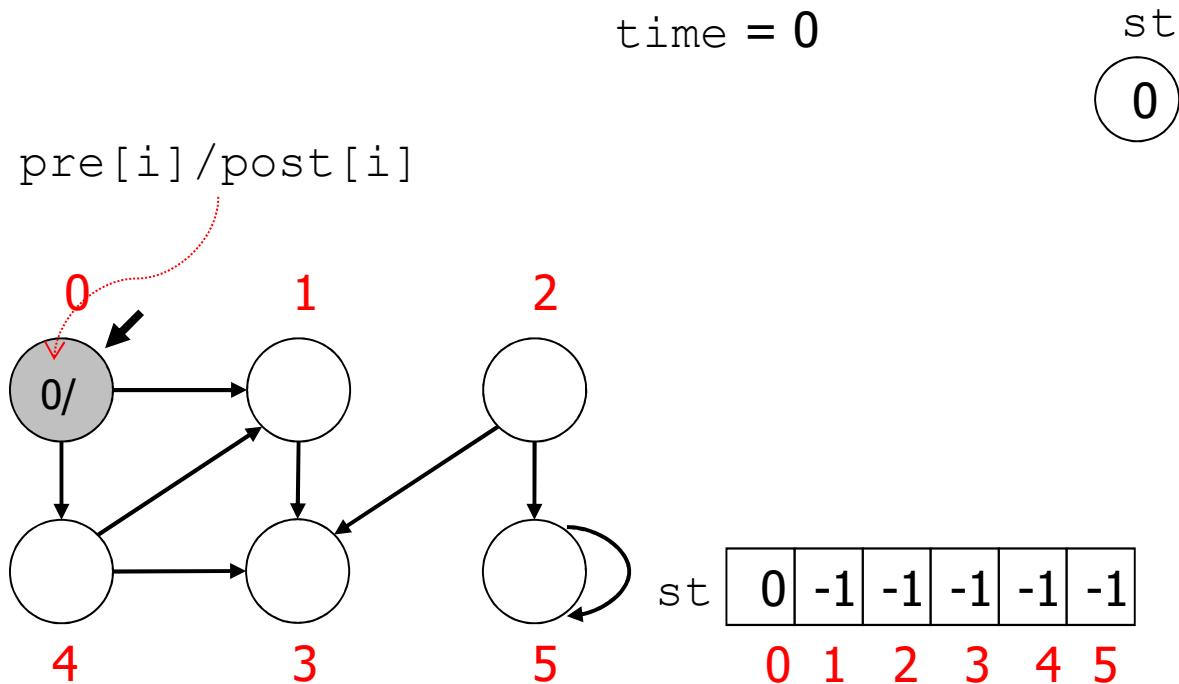
Lista archi

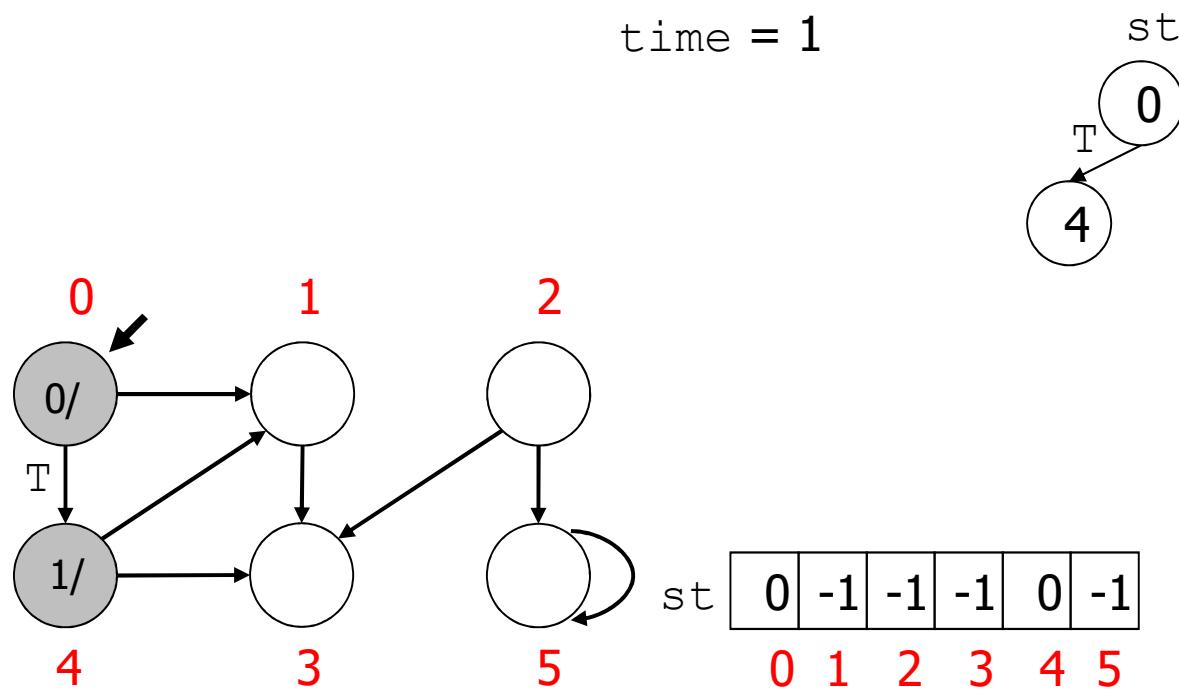
ST	A
0	B
1	C
2	E
3	F
4	B
5	D

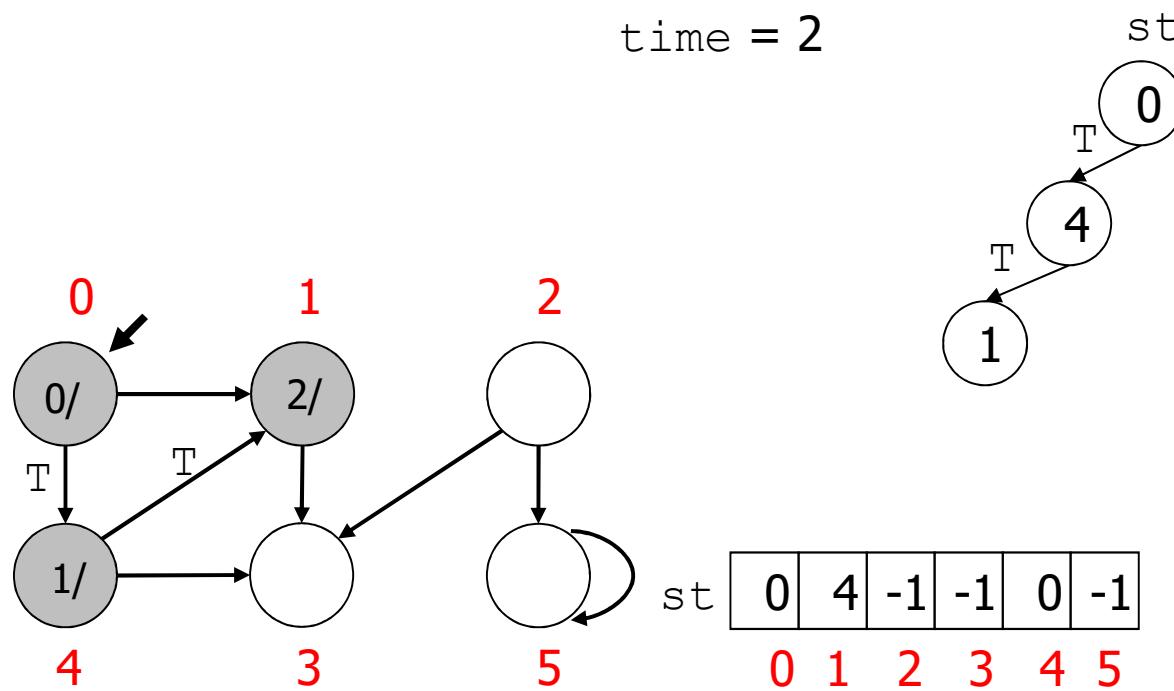
Lista delle
adiacenze

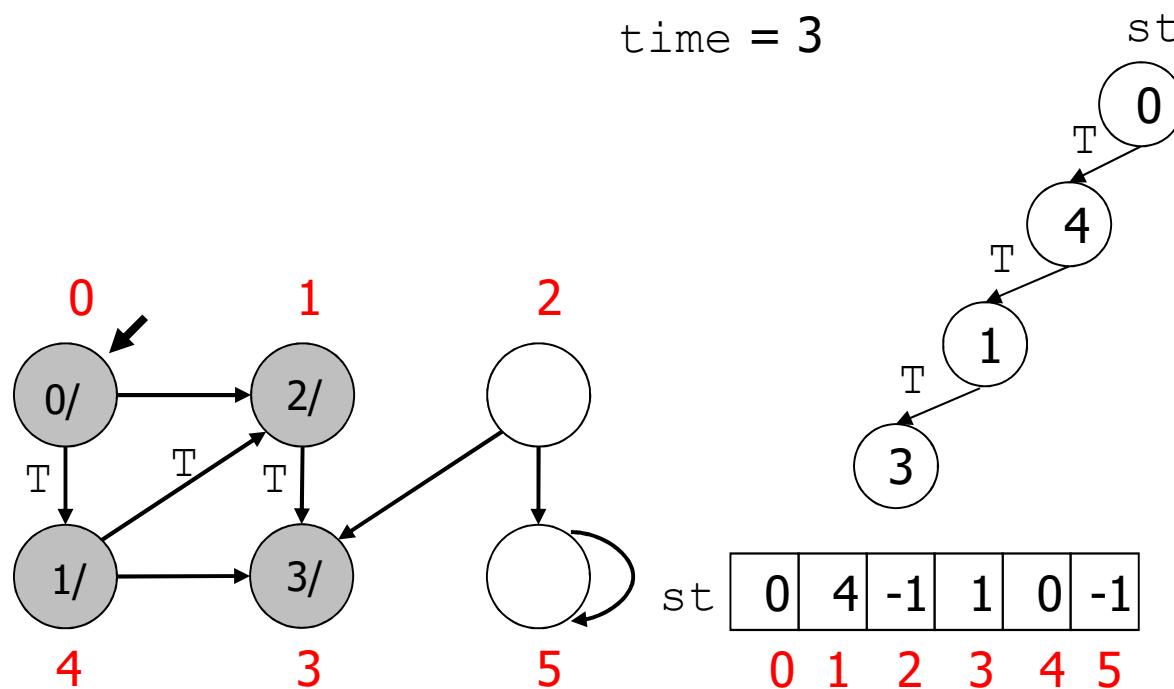
0	1	4	1
1	1	3	3
2	2	5	3
3	3	1	3
4	4	5	5

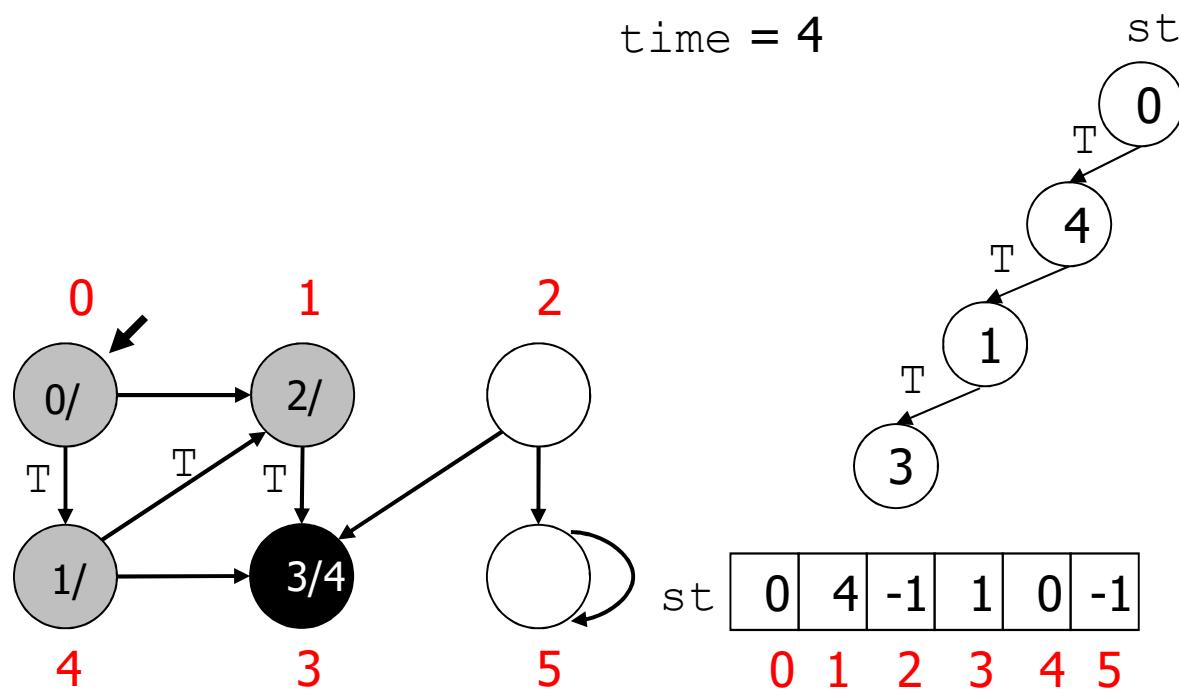
st	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5

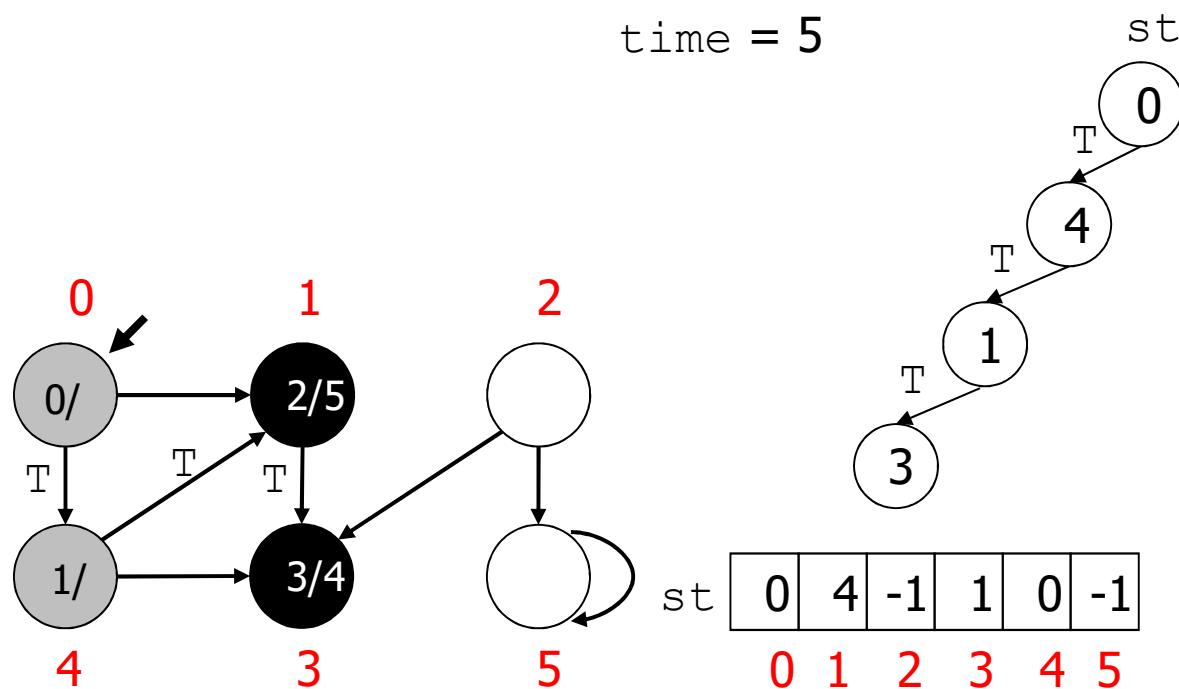


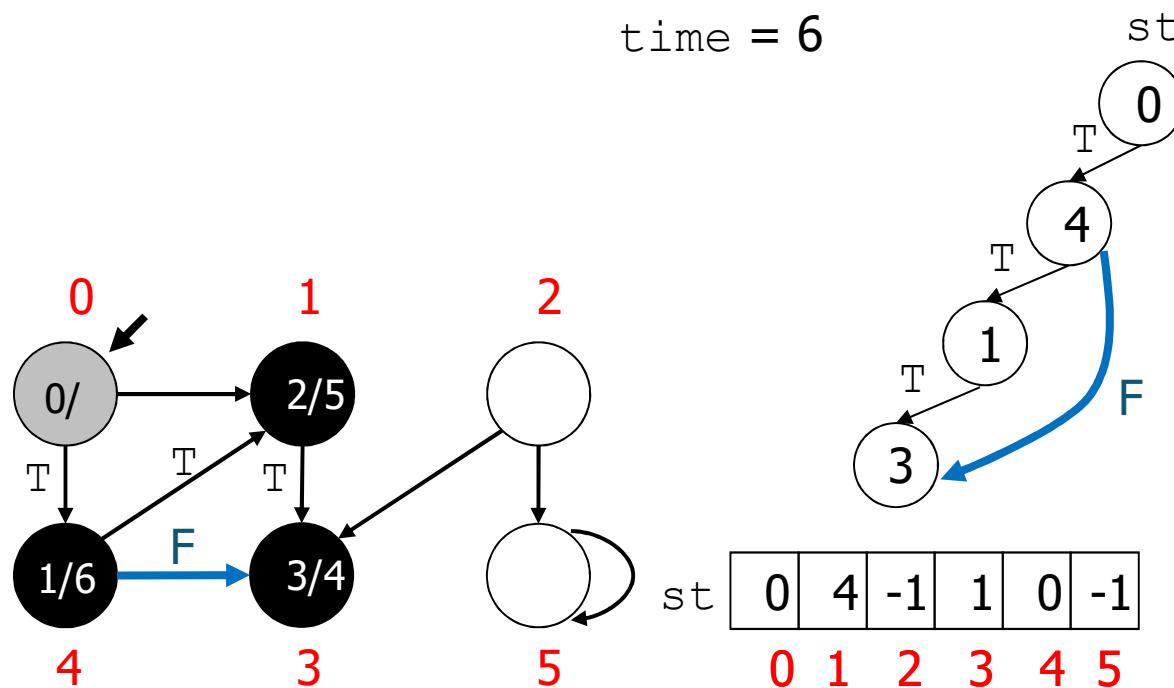


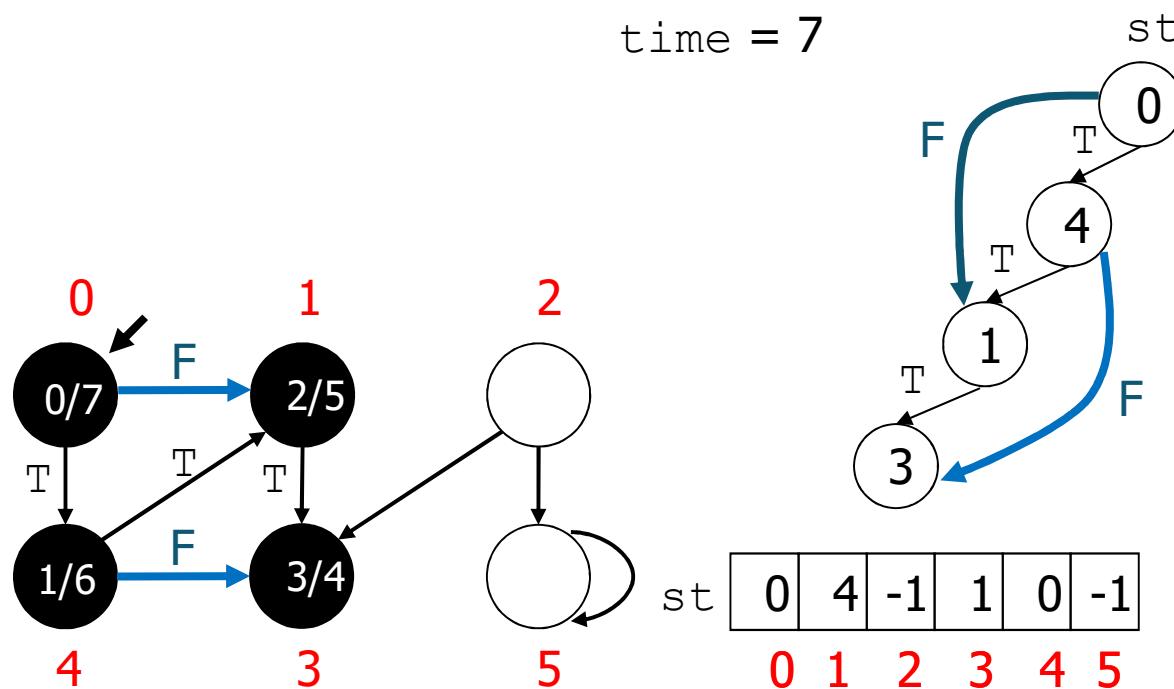


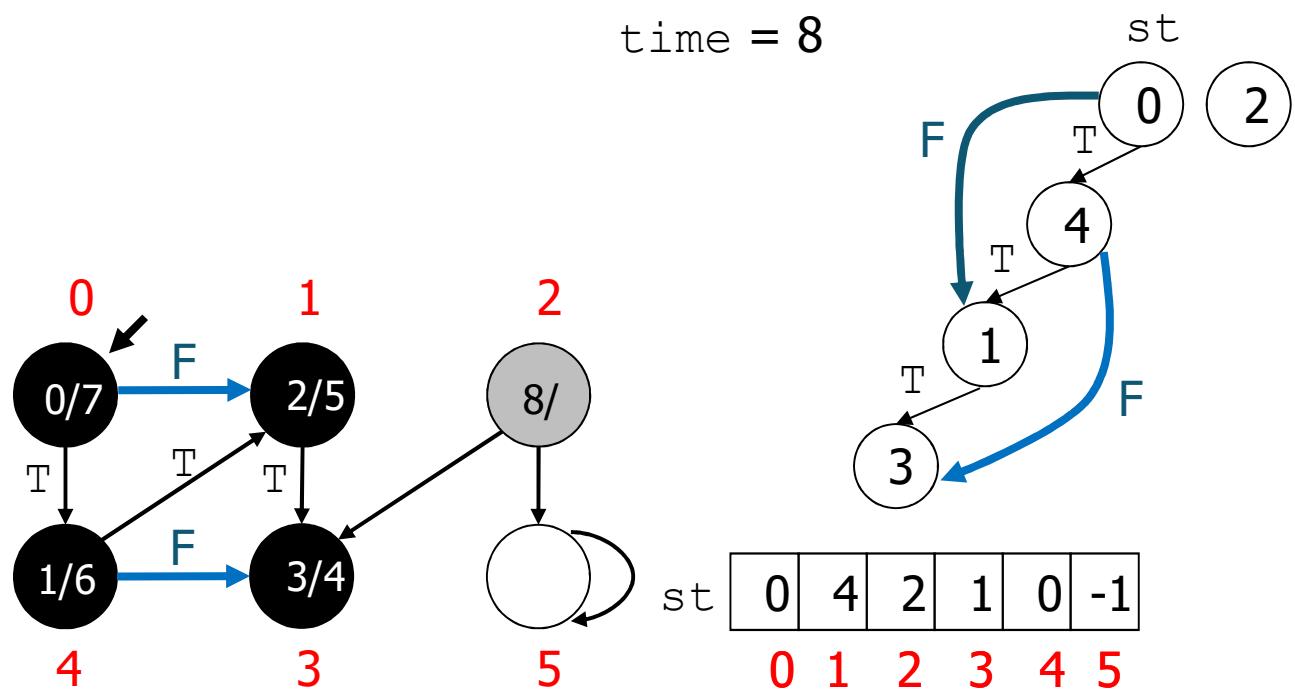


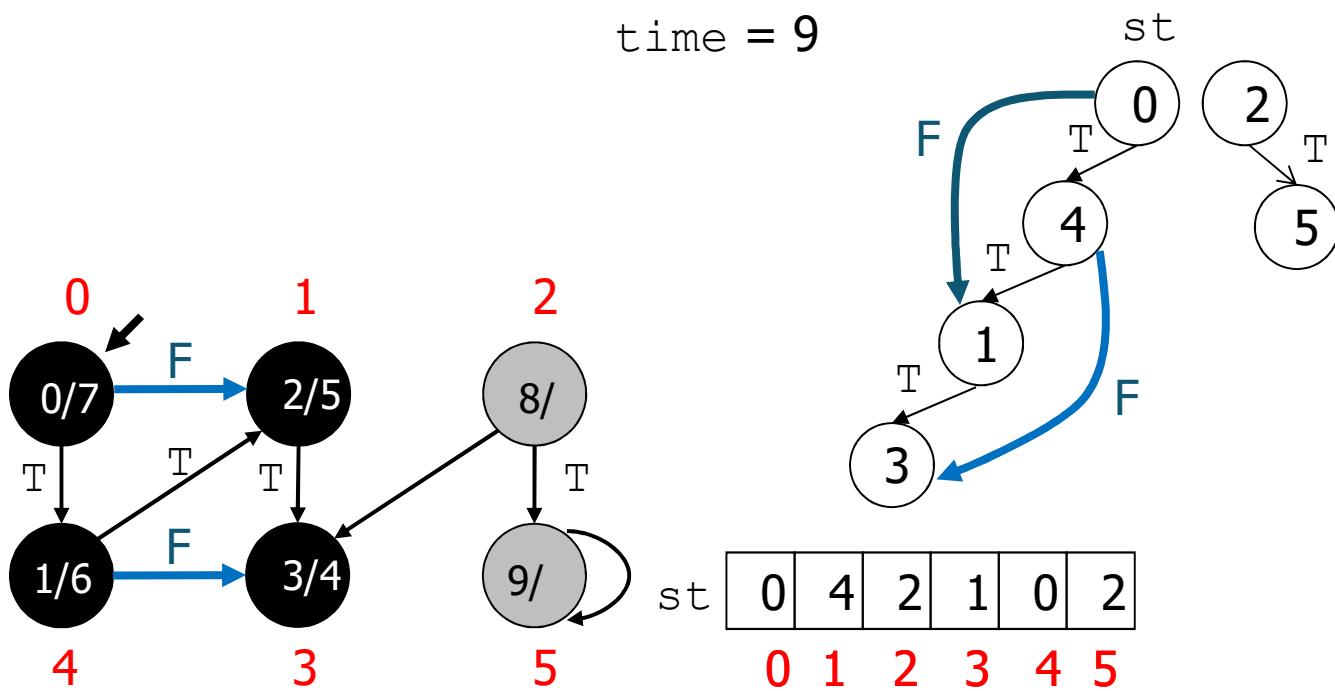


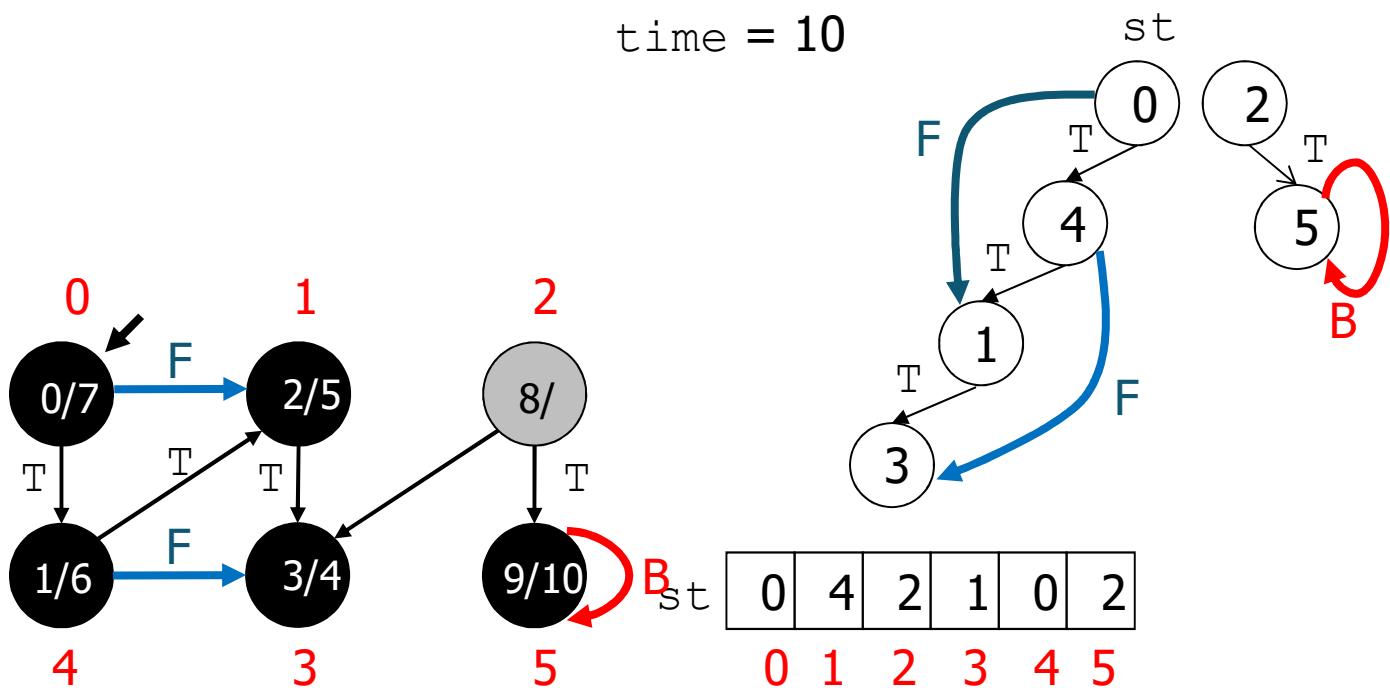


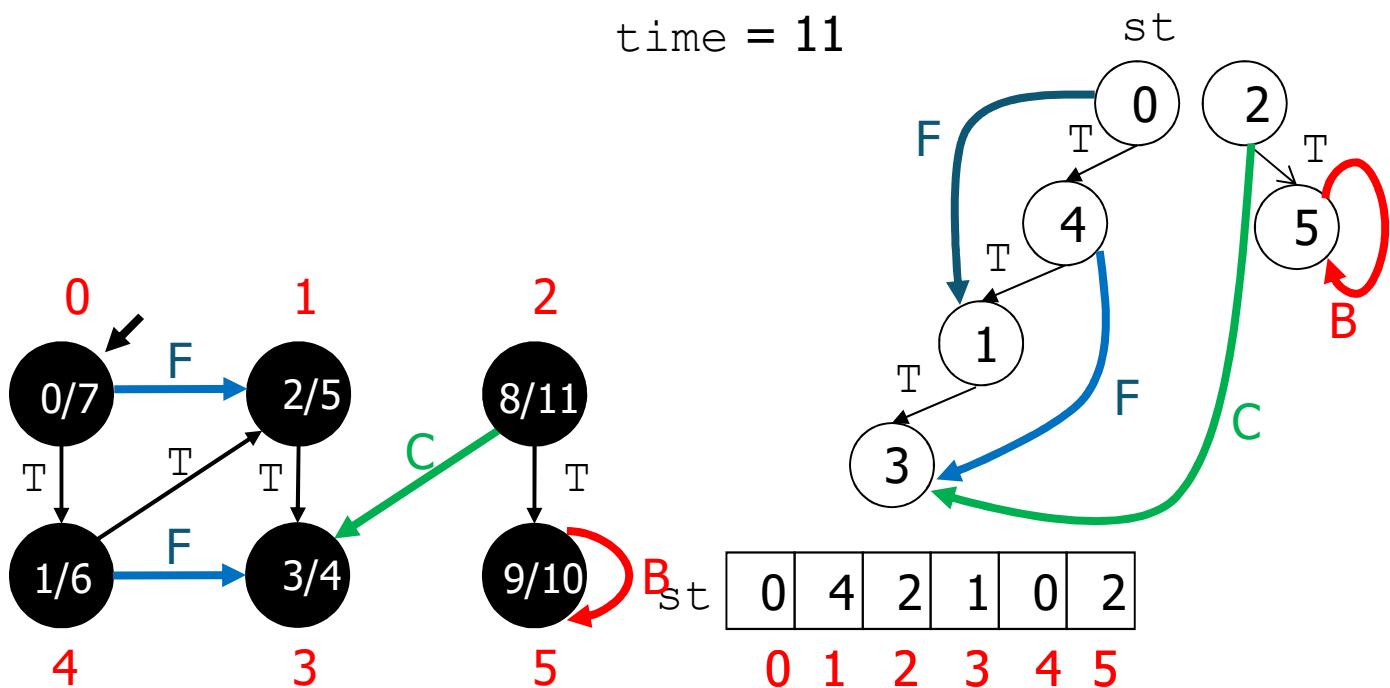








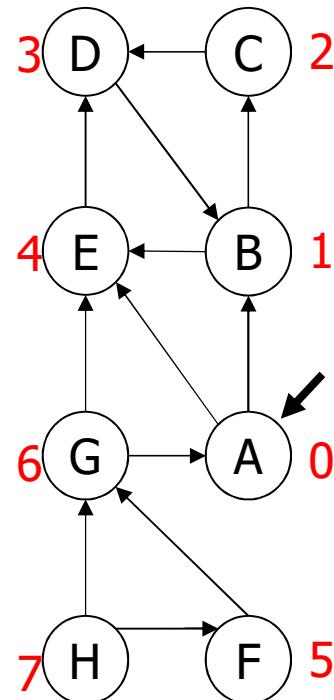




Esempio

Lista archi

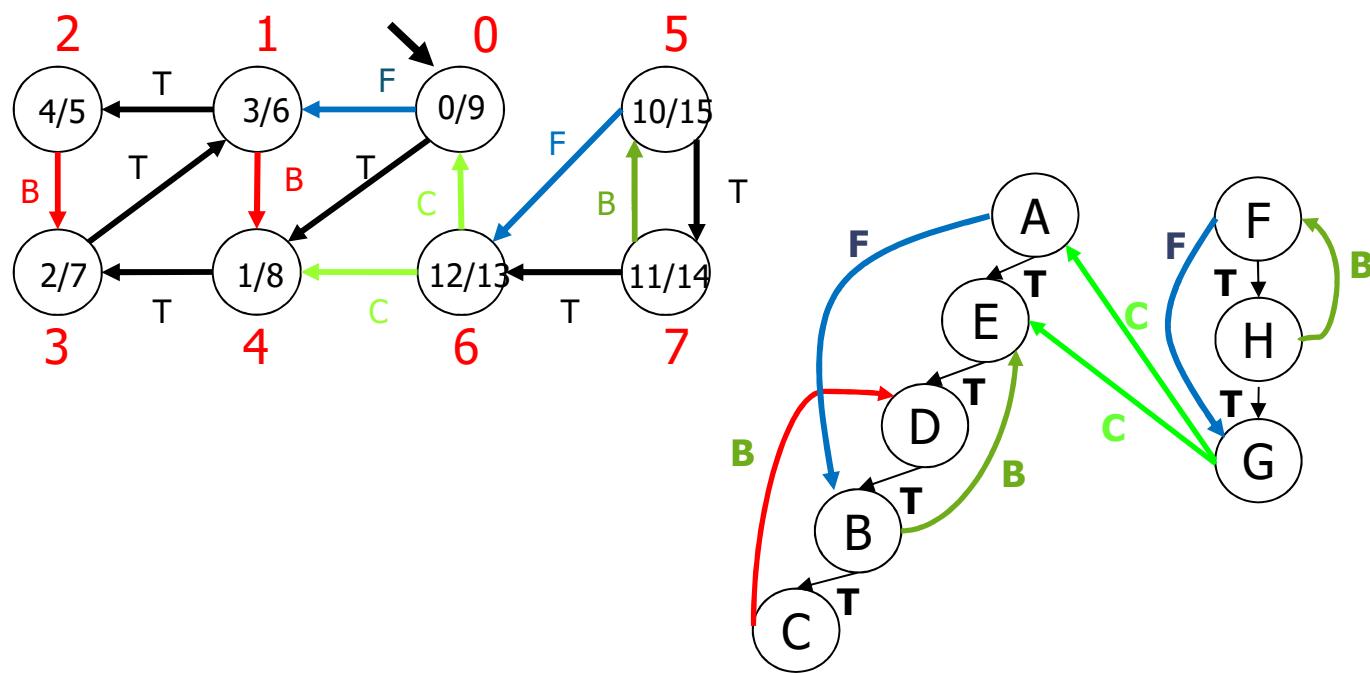
A B
B C
C D
A E
F G
F H
D B
E D
B E
G E
H G
H F
G A



ST
0
1
2
3
4
5
6
7

Lista delle
adiacenze

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



```

void dfsR(Graph G, Edge e, int *time,
          int *pre, int *post, int *st){
    link t;
    int v, w = e.w;
    Edge x;
    if (e.v != e.w) escludi arco fittizio
        printf("(%s, %s): T \n", STsearchByIndex(G->tab, e.v),
               STsearchByIndex(G->tab, e.w));
    st[e.w] = e.v;
    pre[w] = (*time)++;
    for (t = G->adj[w]; t != G->z; t = t->next) terminazione implicita  
della ricorsione
        if (pre[t->v] == -1)
            dfsR(G, EDGEcreate(w, t->v), time, pre, post, st);
        else {
            v = t->v;
            x = EDGEcreate(w, v);
    }
}

```

test per non considerare gli archi 2 volte

```
if (pre[w] < pre[v])
    printf("(%s, %s): B\n", STsearchByIndex(G->tab, x.v),
           STsearchByIndex(G->tab, x.w)) ;
if (post[v] == -1)
    printf("(%s, %s): B\n", STsearchByIndex(G->tab, x.v),
           STsearchByIndex(G->tab, x.w));
else
    if (pre[v] > pre[w])
        printf("(%s,%s):F\n",STsearchByIndex(G->tab, x.v),
               STsearchByIndex(G->tab, x.w));
    else
        printf("(%s,%s):C\n",STsearchByIndex(G->tab, x.v),
               STsearchByIndex(G->tab, x.w));
}
post[w] = (*time)++;
```

grafo non orientato

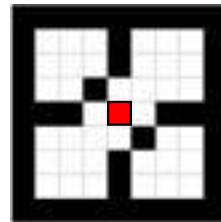
grafo orientato

Complessità (lista adiacenze)

- Inizializzazione → $\Theta(|V|)$
- visita ricorsiva da u → $\Theta(|E|)$
- $T(n) = \Theta(|V| + |E|)$.
- con la matrice delle adiacenze: $T(n) = \Theta(|V|^2)$.

Applicazione: flood fill

- Scopo: colorare un'intera area di pixel connessi con lo stesso colore (Bucket Tool)
- DFS a partire dal pixel sorgente (seed), terminazione quando si incontra una frontiera (boundary):



<http://en.wikipedia.org>

Sedgewick, Wayne, Algorithms Part I & II, www.coursera.org

Visita in ampiezza

A partire da un vertice s :

- determina tutti i vertici raggiungibili da s , quindi non visita necessariamente tutti i vertici a differenza della DFS
- calcola la distanza minima da s di tutti i vertici da esso raggiungibili.
- genera un albero della visita in ampiezza.

Aampiezza: espande tutta la frontiera tra vertici già scoperti/non ancora scoperti.

Principi base

Scoperta di un vertice: prima volta che si incontra nella visita raggiungendolo percorrendo un arco.

Il vettore $\text{pre}[v]$ registra il tempo di scoperta di v .

Dato un vertice v , il vettore $\text{st}[v]$ registra il padre di v nell'albero della visita in ampiezza.

Non appena il vertice viene scoperto, si registra il tempo di scoperta e il padre nell'albero, concludendo così l'elaborazione del vertice stesso.

Strutture dati

- grafo non pesato come matrice delle adiacenze
 - coda Q di archi $e = (v, w)$
 - vettore st dei padri nell'albero di visita in ampiezza
 - vettore pre dei tempi di scoperta dei vertici
 - contatore $time$ del tempo
- time, *pre e *st sono locali alla funzione GRAPHbfs
e passati by reference alla funzione bfs.

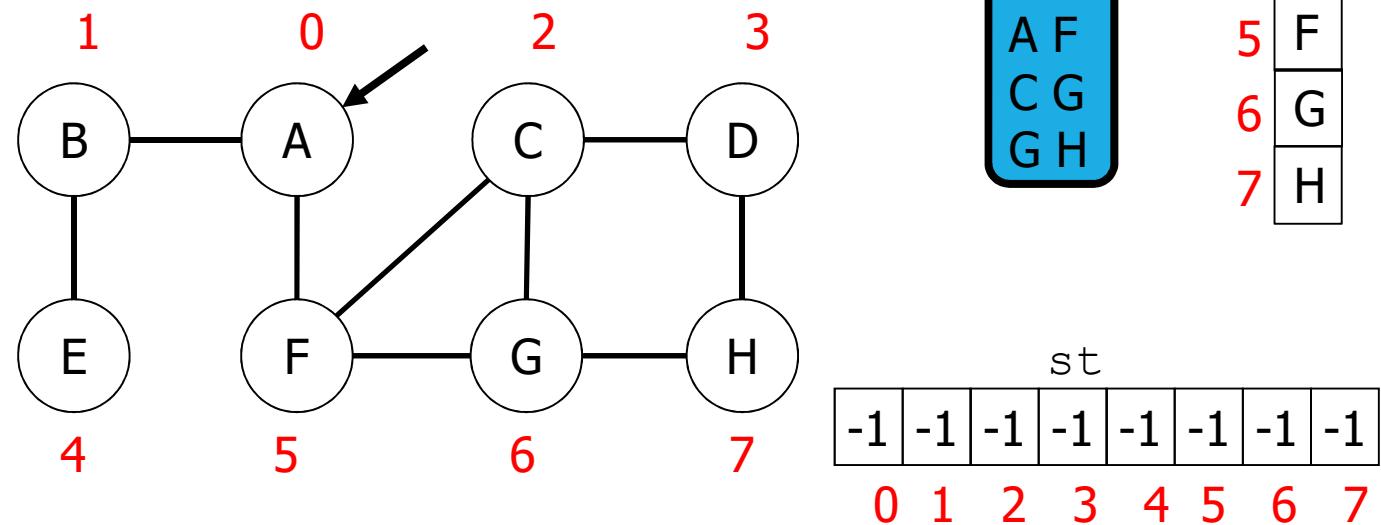
wrapper

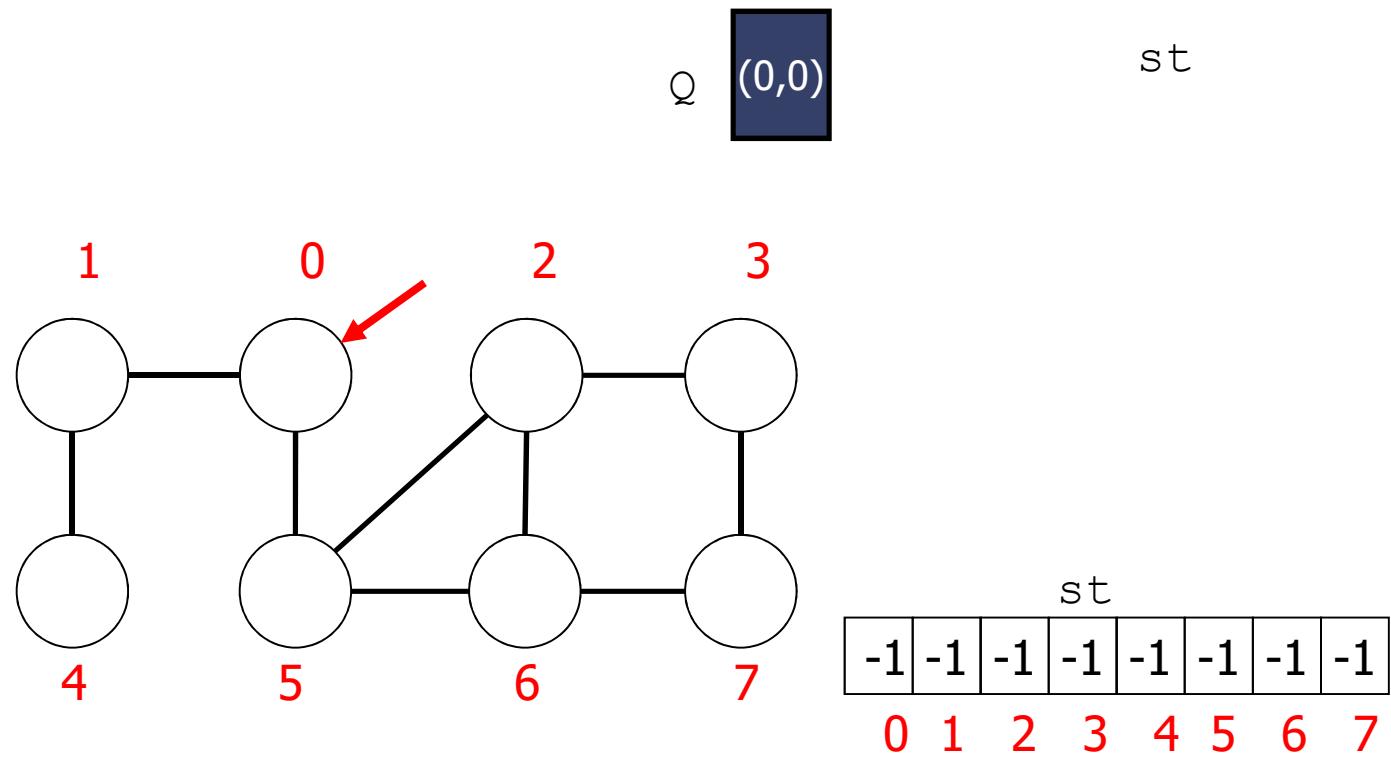
Algoritmo

- metti l'arco fittizio di partenza $e = (id, id)$ nella coda
- ripeti fintanto che la coda non si svuota
 - estrai dalla coda un arco $e = (v, w)$
 - se $e.w$ è un vertice non ancora scoperto ($\text{pre}[e.w] == -1$)
 - indica che $e.v$ è padre di $e.w$ nella BFS ($\text{st}[e.w] = e.v$)
 - marca $e.w$ come scoperto al tempo time ($\text{pre}[e.w] = \text{time}++$) e incrementa il tempo
 - trova i vertici non ancora scoperti x adiacenti a $e.w$ e metti in coda tutti gli archi che connettono $e.w$ ad essi.

bfs: funzione che visita in ampiezza a partire da un vertice di partenza id .

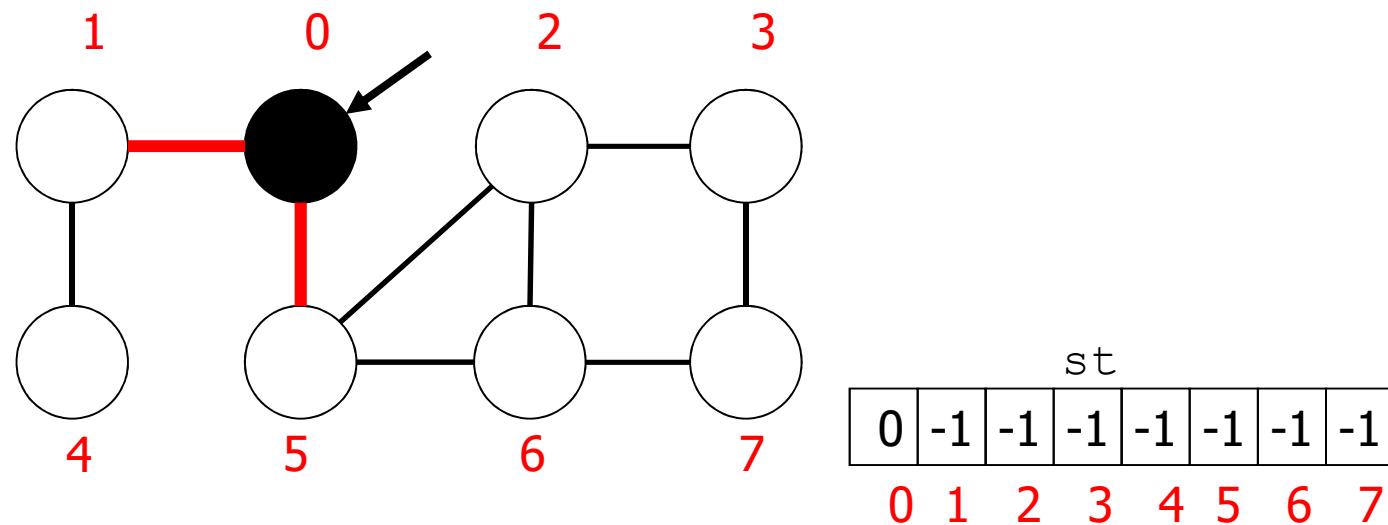
Esempio

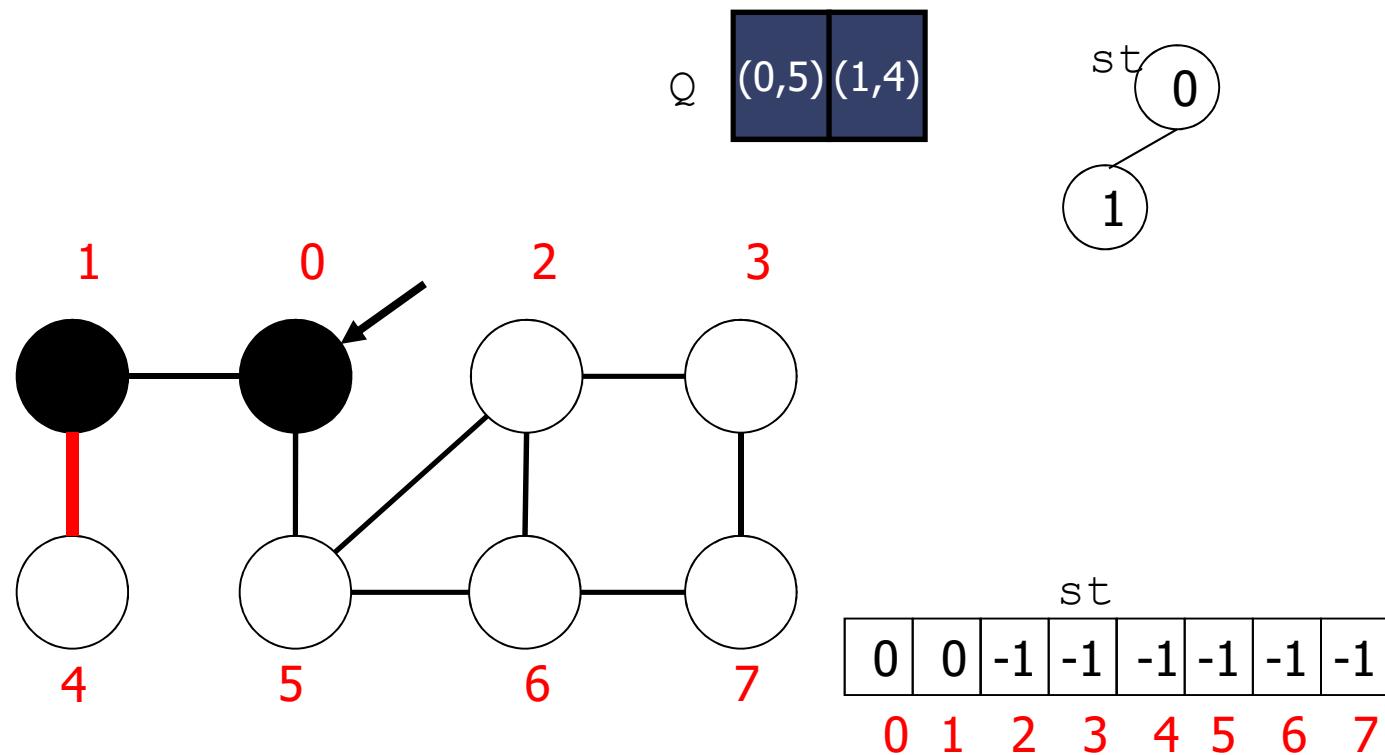


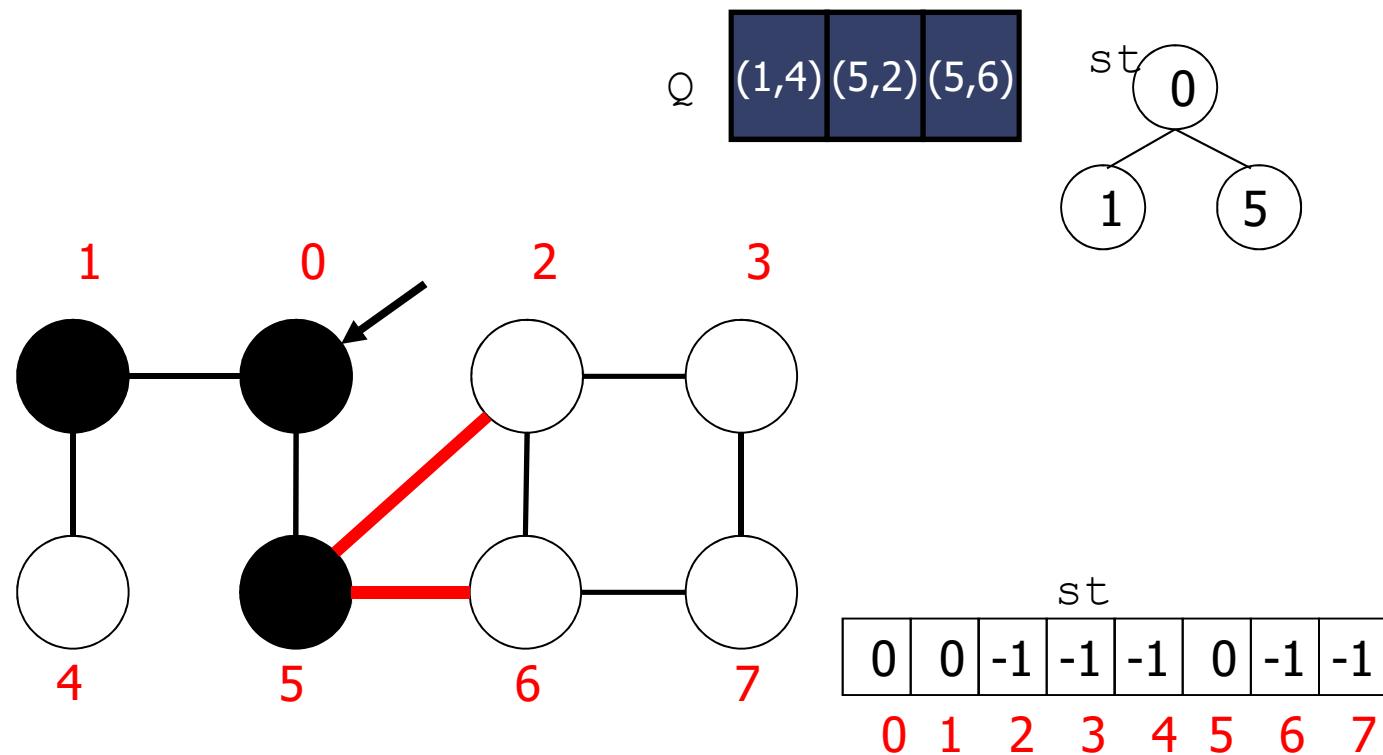


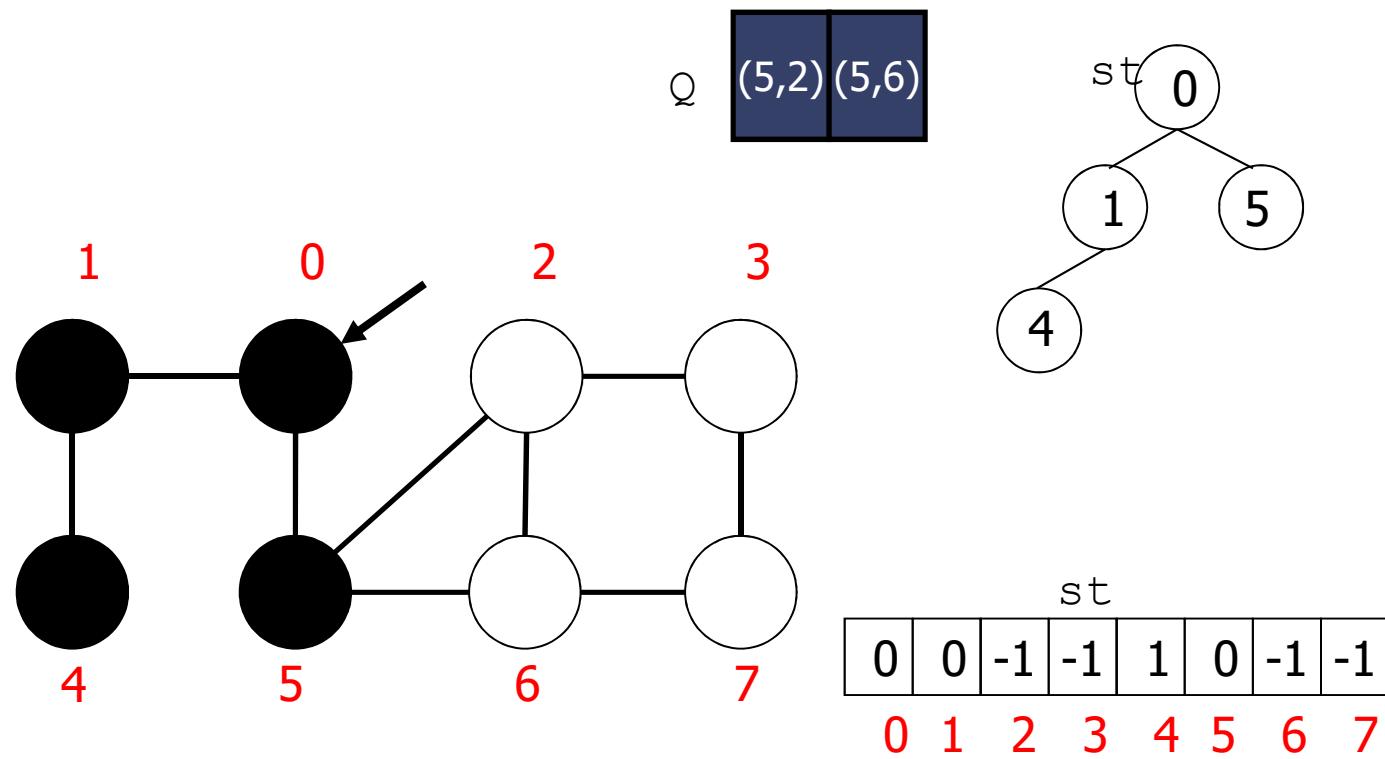
Q	$(0,1)$	$(0,5)$
-----	---------	---------

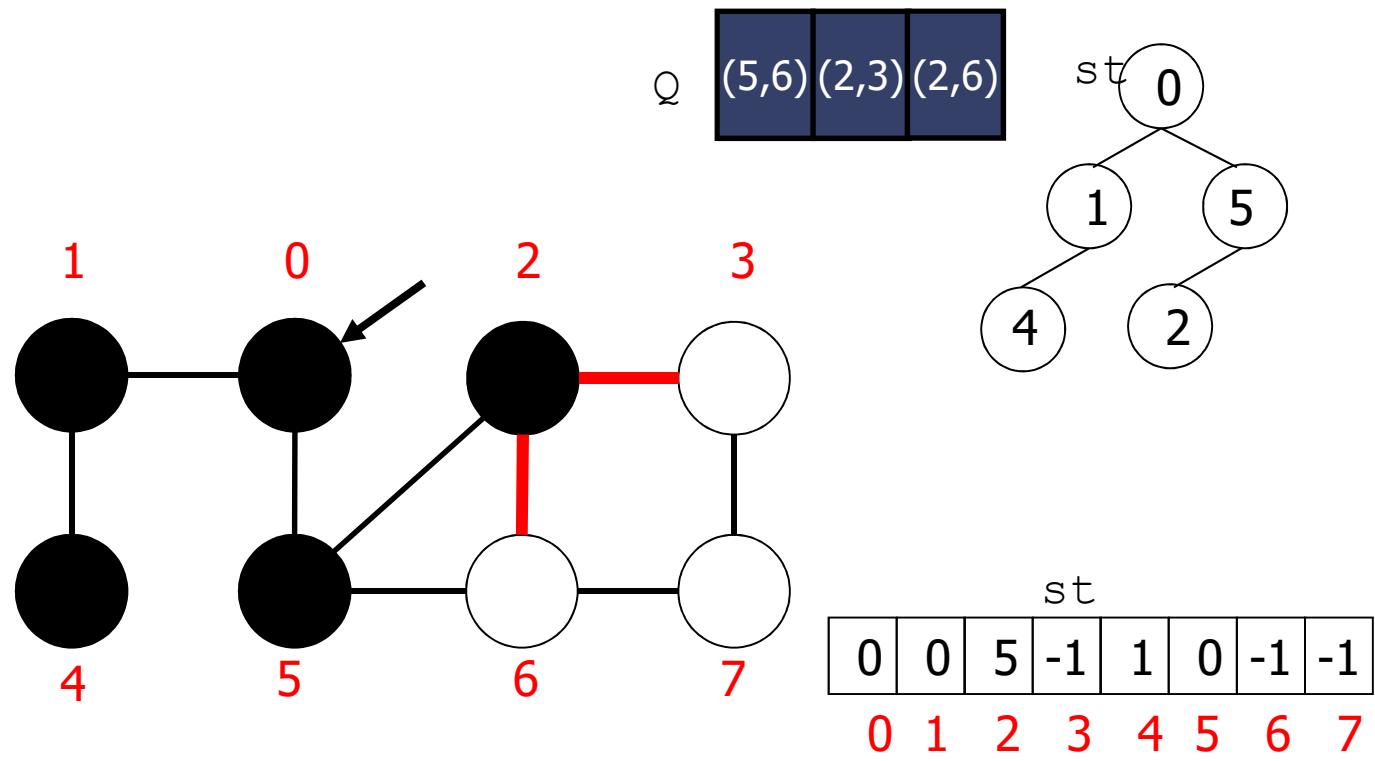
s^t 0

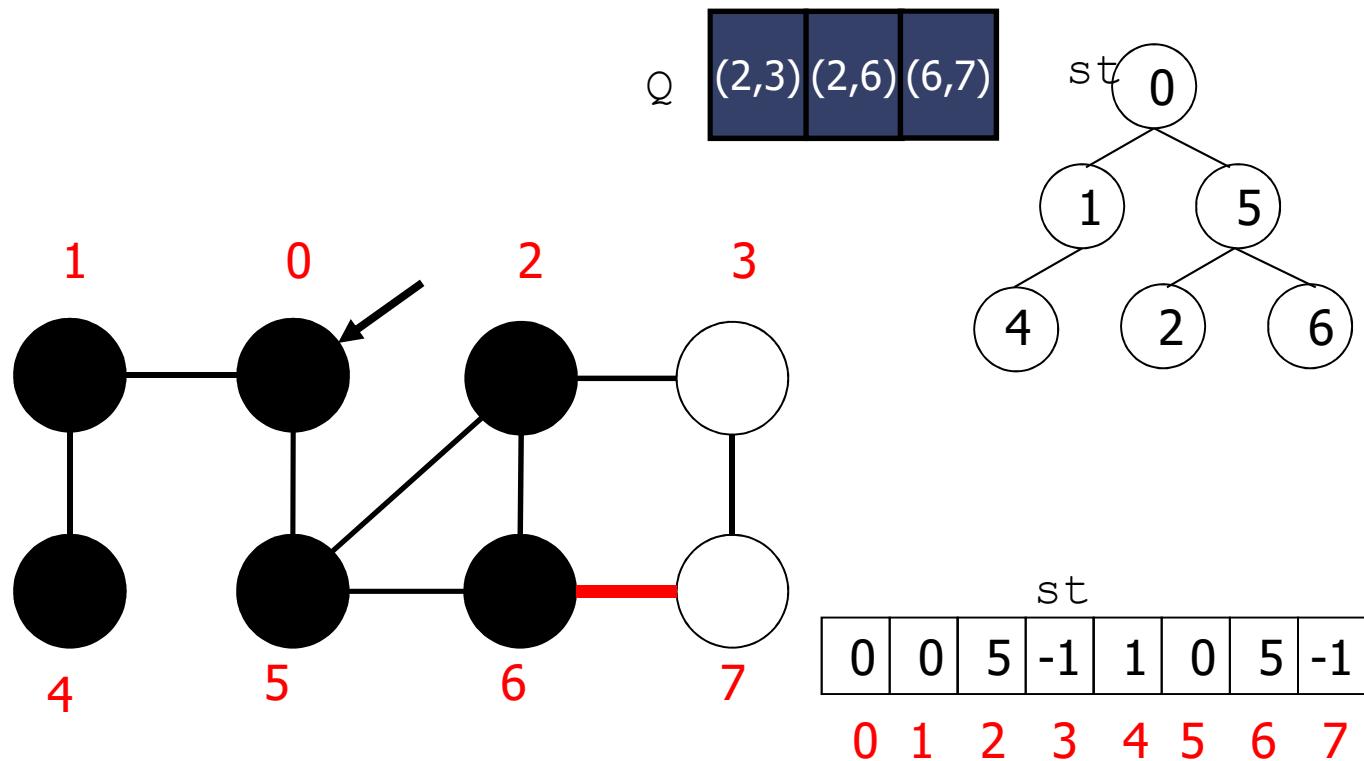


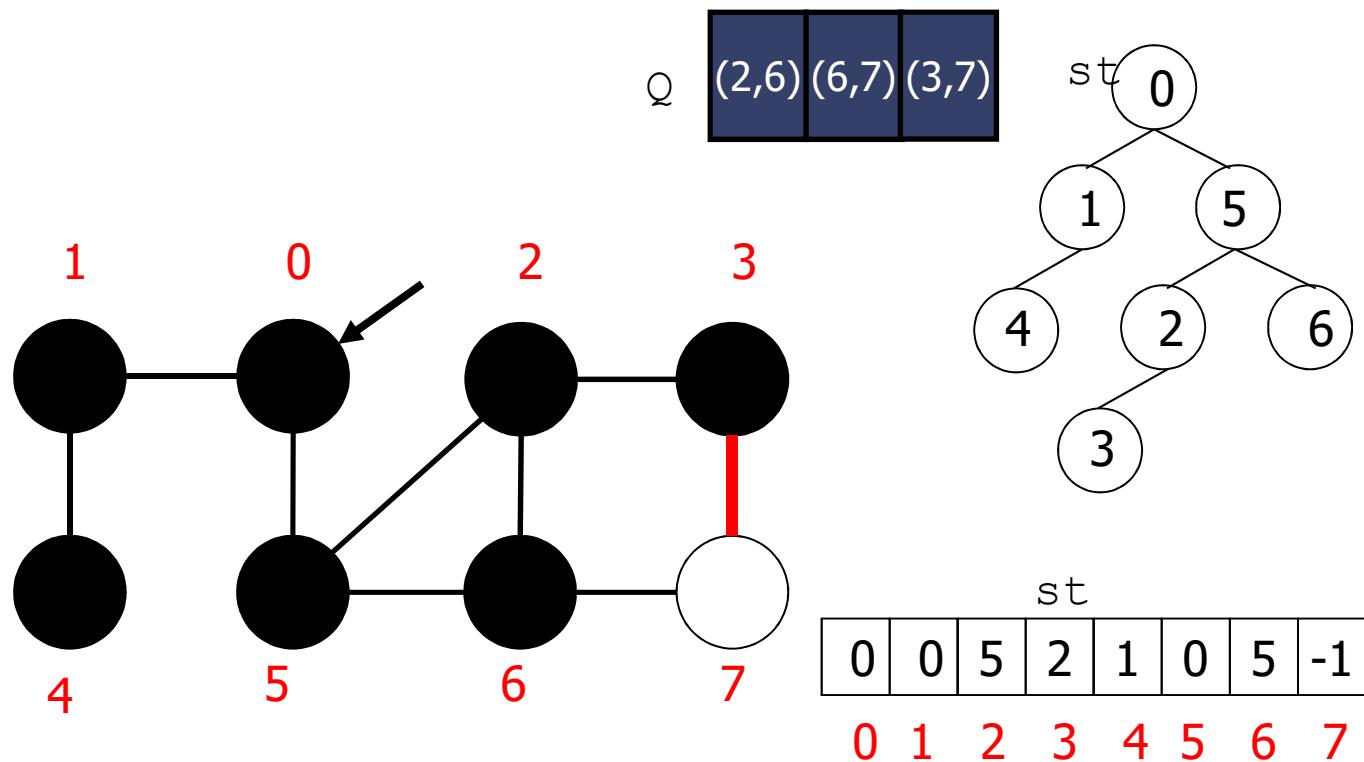


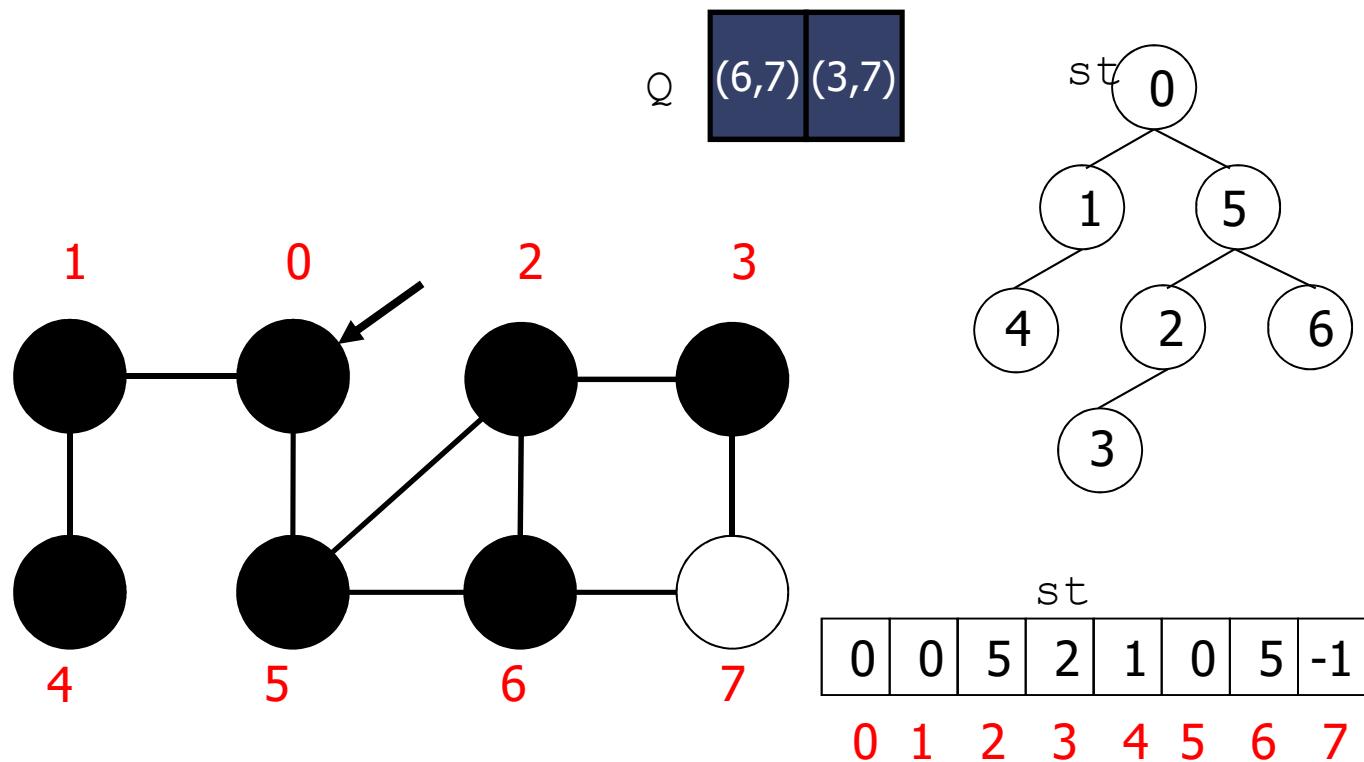


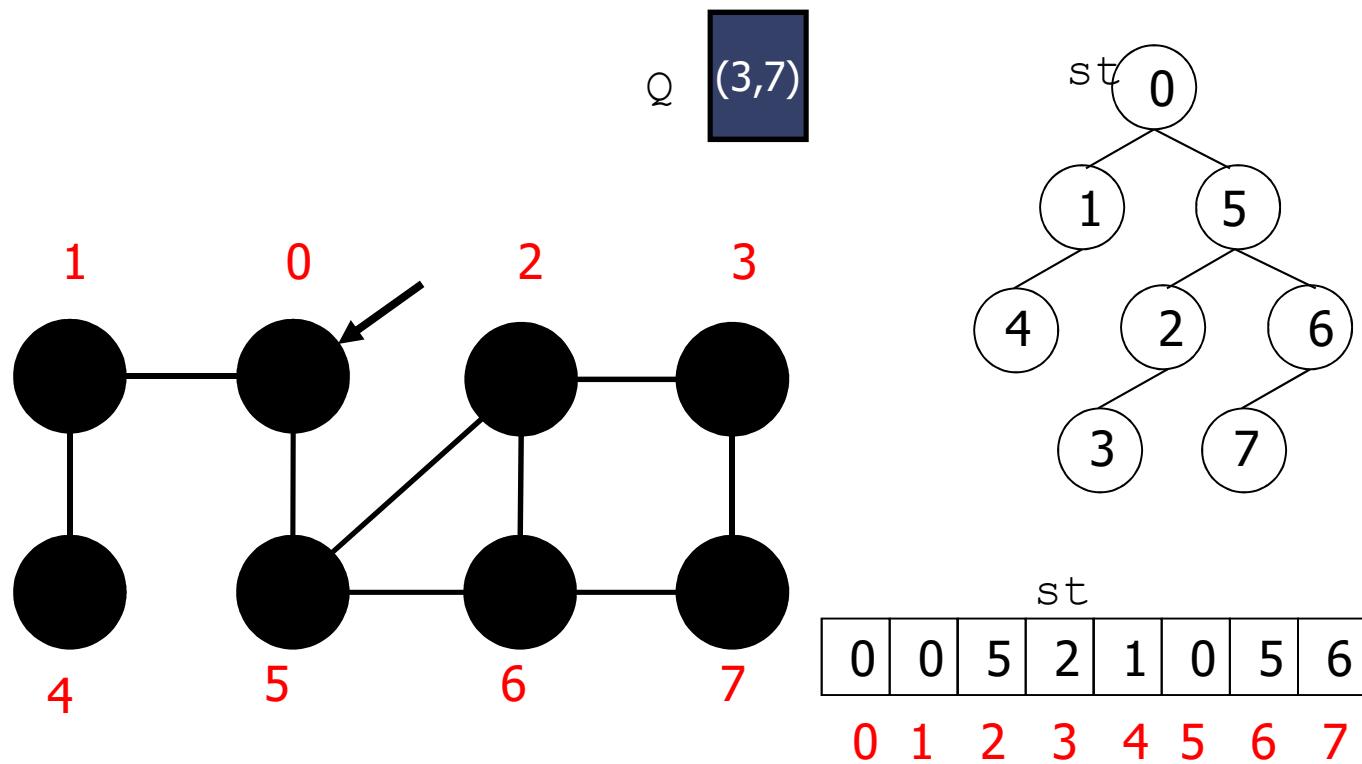


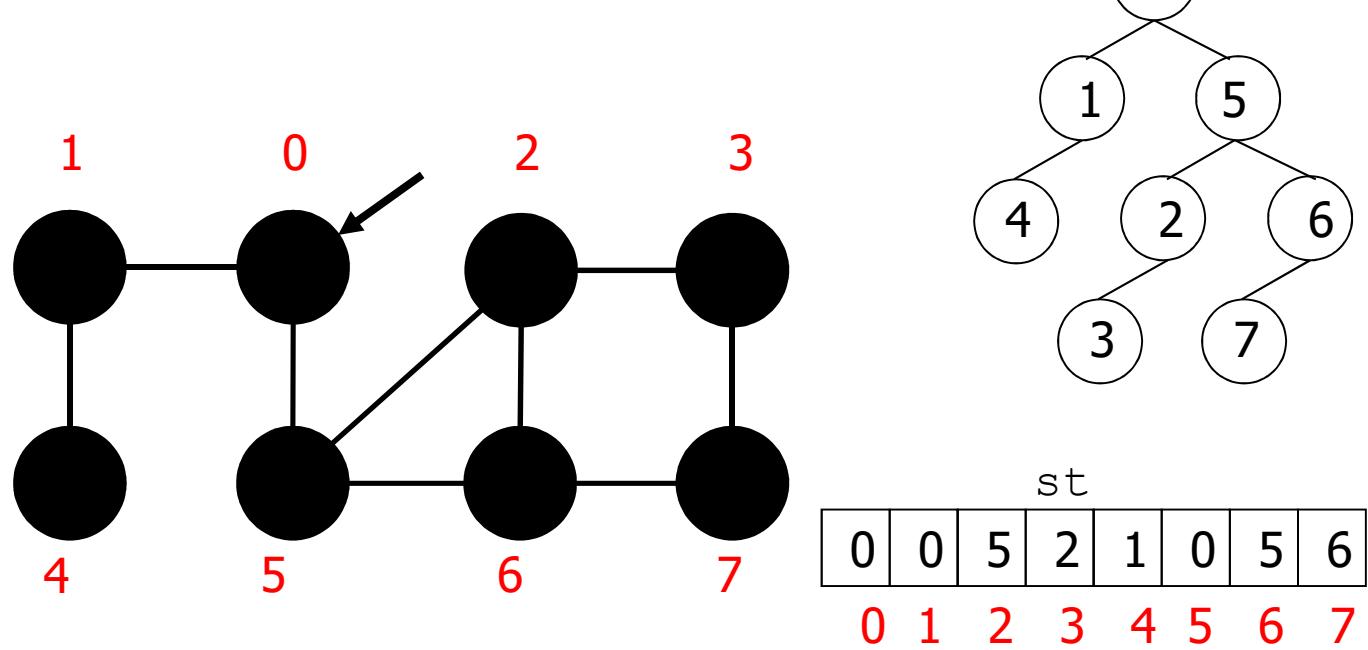












```

void GRAPHbfs(Graph G, int id) {
    int v, time=0, *pre, *st, *dist;
    /* allocazione di pre, st e dist */
    for (v=0; v < G->V; v++) {
        pre[v] = -1; st[v] = -1; dist[v] = INT_MAX;
    }
    bfs(G, EDGEcreate(id,id), &time, pre, st, dist);
    printf("\n Resulting BFS tree \n");
    for (v=0; v < G->V; v++)
        if (st[v] != -1)
            printf("%s's parent is:%s\n",STsearchByIndex(G->tab, v),
                   STsearchByIndex(G->tab, st[v]));
    printf("\n Levelizing \n");
    for (v=0; v < G->V; v++)
        if (st[v] != -1)
            printf("%s: %d \n",STsearchByIndex(G->tab,v),dist[v]);
}

```

```

void bfs(Graph G, Edge e, int *time, int *pre, int *st,
         int *dist) {
    int x;
    Q q = Qinit();
    Qput(q, e);
    dist[e.v]=-1;
    while (!Qempty(q))
        if (pre[(e = Qget(q)).w] == -1) {
            pre[e.w] = (*time)++;
            st[e.w] = e.v;
            dist[e.w] = dist[e.v]+1;
            for (x = 0; x < G->v; x++)
                if (G->adj[e.w][x] == 1)
                    if (pre[x] == -1)
                        Qput(q, EDGEcreate(e.w, x));
        }
}

```

matrice delle
adiacenze

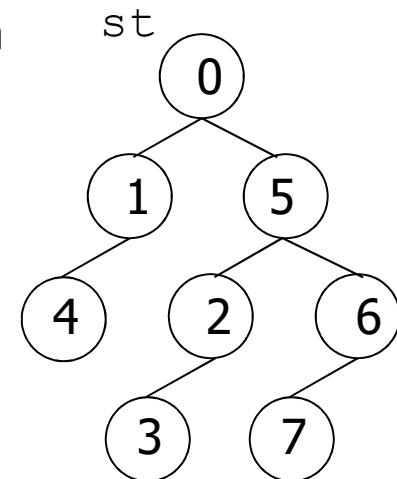
Complessità

- Operazioni sulla coda
- Scansione della matrice delle adiacenze $T(n) = \Theta(|V|^2)$.
- Con la lista delle adiacenze: $T(n) = O(|V|+|E|)$.

Proprietà

Cammini minimi: la visita in ampiezza determina la minima distanza tra s e ogni vertice raggiungibile da esso.

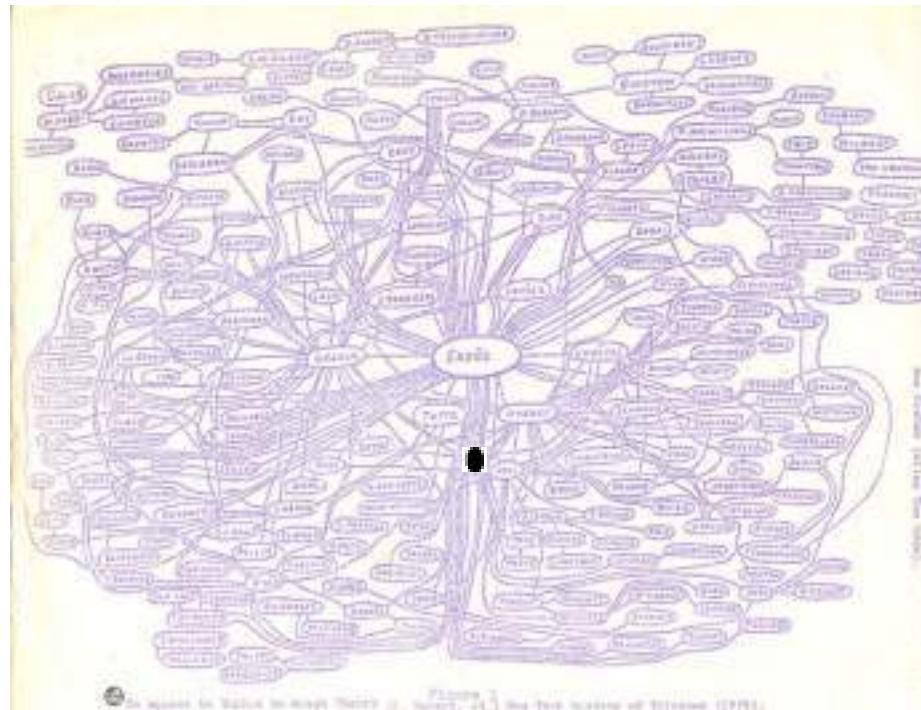
Cammino minimo da 0 a 3:
0, 5, 2, 3
lunghezza = 3



Applicazione: I numeri di Erdős



- Paul Erdős (1913-1996): matematico ungherese «itinerante»: pubblicazioni con moltissimi coautori
- Grafo non orientato:
 - vertici: matematici
 - arco: unisce 2 matematici che hanno una pubblicazione in comune
- numero di Erdős: distanza minima di ogni matematico da Erdős
- BFS



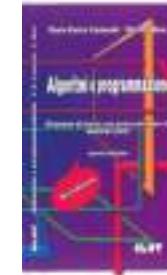
<http://www.oakland.edu/upload/images/Erdos%20Number%20Project/cgraph.jpg>
Sedgewick, Wayne, Algorithms Part I & II, www.coursera.org

Riferimenti

- Visita in profondità:
 - Sedgewick Part 5 18.2, 18.3, 18.4
 - Cormen 23.3
- Visita in ampiezza:
 - Sedgewick Part 5 18.7
 - Cormen 23.2
- Numero di Erdős:
 - Bertossi 9.5.2

Esercizi di teoria

- 10. Visite dei grafi e applicazioni
 - 10.2 Visita in ampiezza
 - 10.3 Visita in profondità



Le applicazioni degli algoritmi di visita dei grafi

Gianpiero Cabodi e Paolo Camurati



Rilevazione di cicli

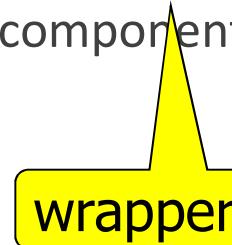
Un grafo è aciclico se e solo se in una visita in profondità non si incontrano archi etichettati **B**.

Non basta però per identificare i cicli!

Componenti connesse

In un grafo non orientato rappresentato come lista delle adiacenze:

- ogni albero della foresta della DFS è una componente连通的
- $cc[v]$ è un array locale a GRAPHCC che memorizza un intero che identifica ciascuna componente连通的. I vertici fungono da indici dell'array

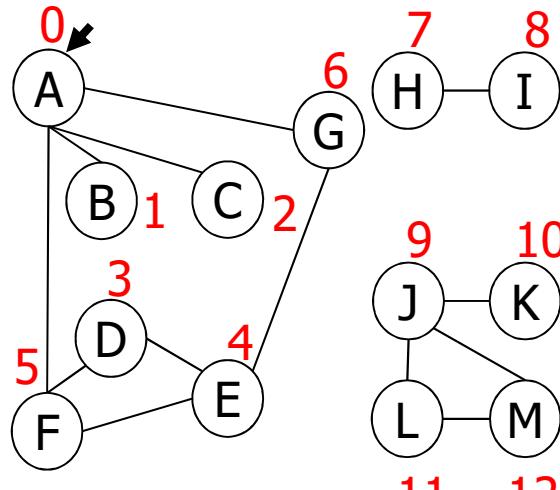


wrapper

Esempio

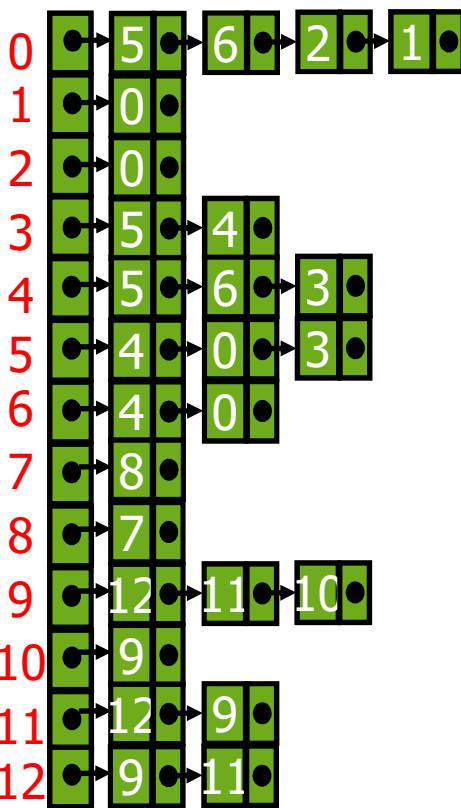
Lista archi

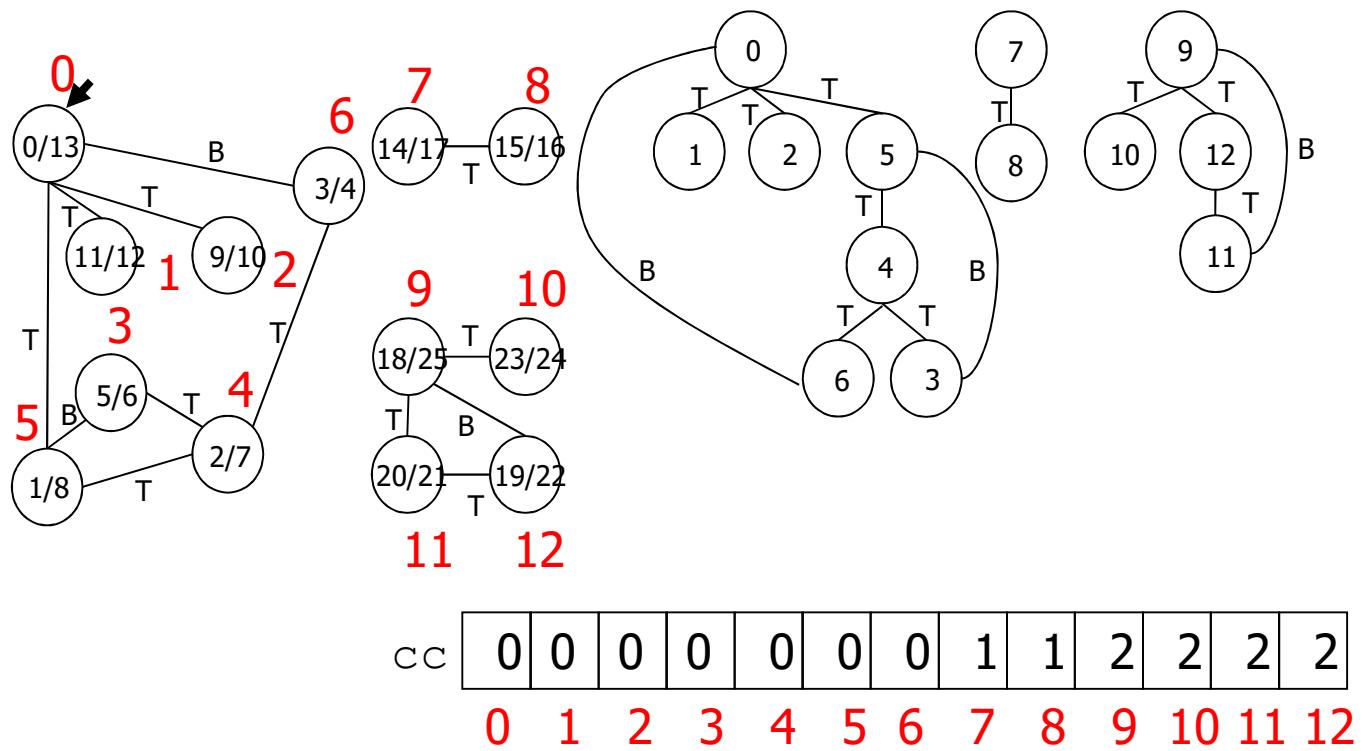
A B
A C
D E
D F
A G
H I
J K
J L
L M
G E
A F
F E
J M



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

Lista delle
adiacenze





```

void dfsRcc(Graph G, int v, int id, int *cc) {
    link t;
    cc[v] = id;
    for (t = G->ladj[v]; t != G->z; t = t->next)
        if (cc[t->v] == -1)
            dfsRcc(G, t->v, id, cc);
}
int GRAPHcc(Graph G) {
    int v, id = 0, *cc;
    cc = malloc(G->V * sizeof(int));
    for (v = 0; v < G->V; v++) cc[v] = -1;
    for (v = 0; v < G->V; v++)
        if (cc[v] == -1) dfsRcc(G, v, id++, cc);
    printf("Connected component(s) \n");
    for (v = 0; v < G->V; v++)
        printf("node %s in cc %d\n", STsearchByIndex(G->tab, v), cc[v]);
    return id;
}

```

Connettività

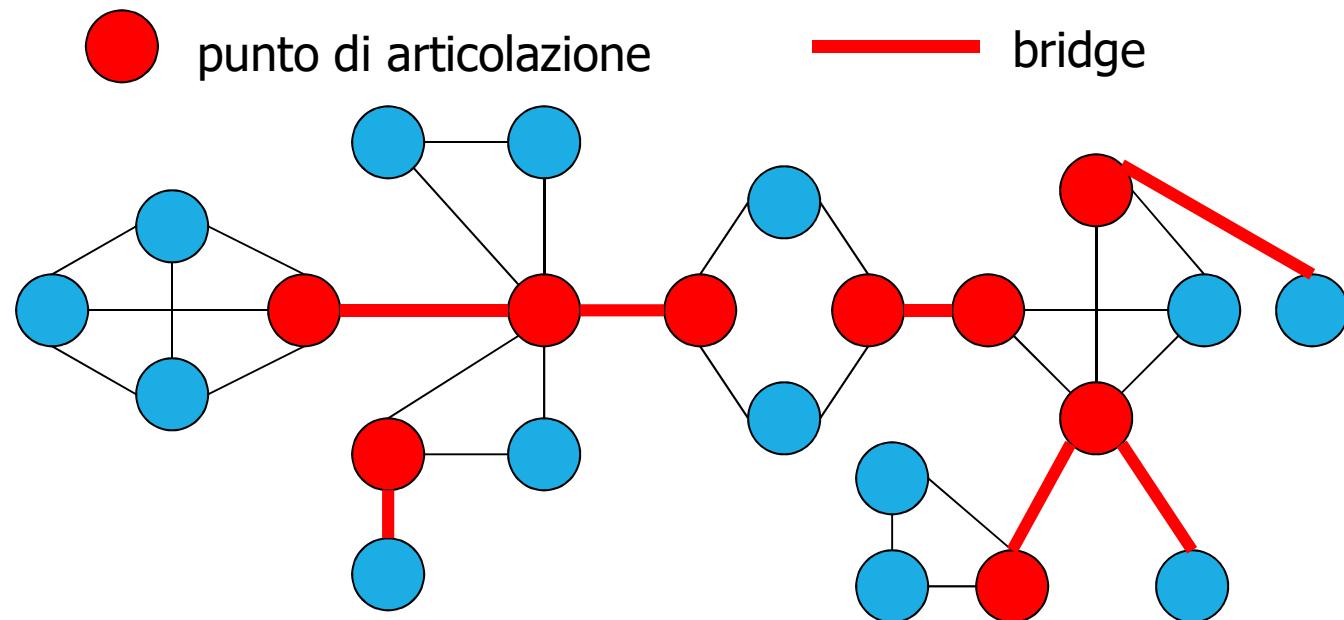
Dato un grafo non orientato e connesso, determinare se perde la proprietà di connessione a seguito della rimozione di:

- un arco
- un nodo.

Ponte (bridge): arco la cui rimozione disconnette il grafo.

Punto di articolazione: vertice la cui rimozione disconnette il grafo. Rimuovendo il vertice si rimuovono anche gli archi su di esso insistenti.

Esempio



Punto di articolazione

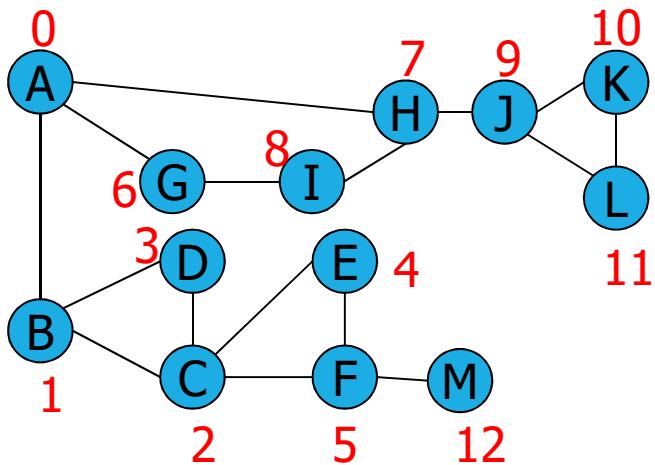
Dato un grafo non orientato e connesso G , dato l'albero G_π della visita in profondità,

- la radice di G_π è un punto di articolazione di G se e solo se ha almeno due figli
- ogni altro vertice v è un punto di articolazione di G se e solo se v ha un figlio s tale che non vi è alcun arco B da s o da un suo discendente a un antenato proprio di v .

Esempio

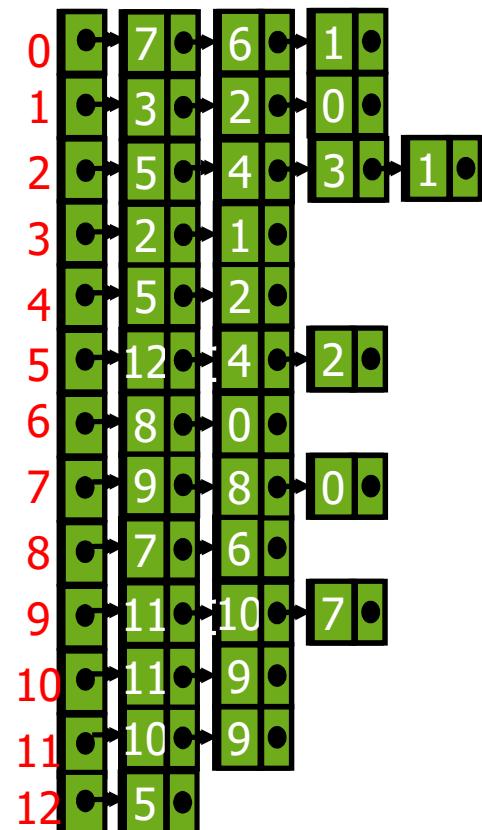
A B
B C
B D
C D
C E
C F
E F
A G
A H
G I
H I
H J
J K
J L
K L
F M

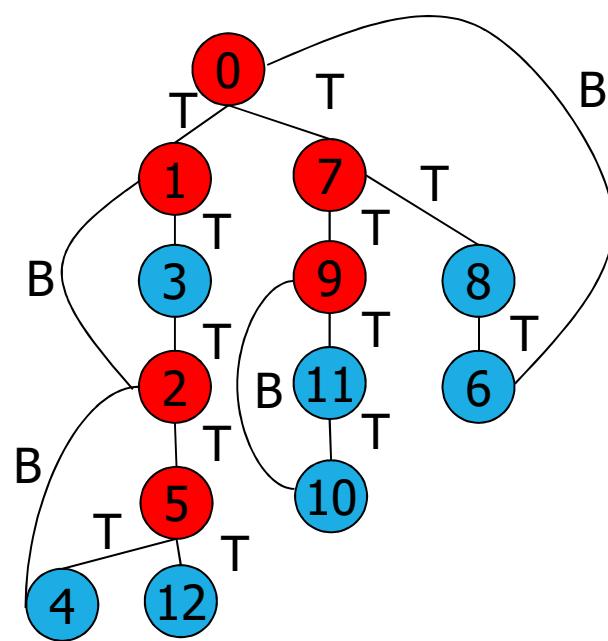
Lista archi



ST	A
0	B
1	C
2	D
3	E
4	F
5	G
6	H
7	I
8	J
9	K
10	L
11	M
12	

Lista delle
adiacenze





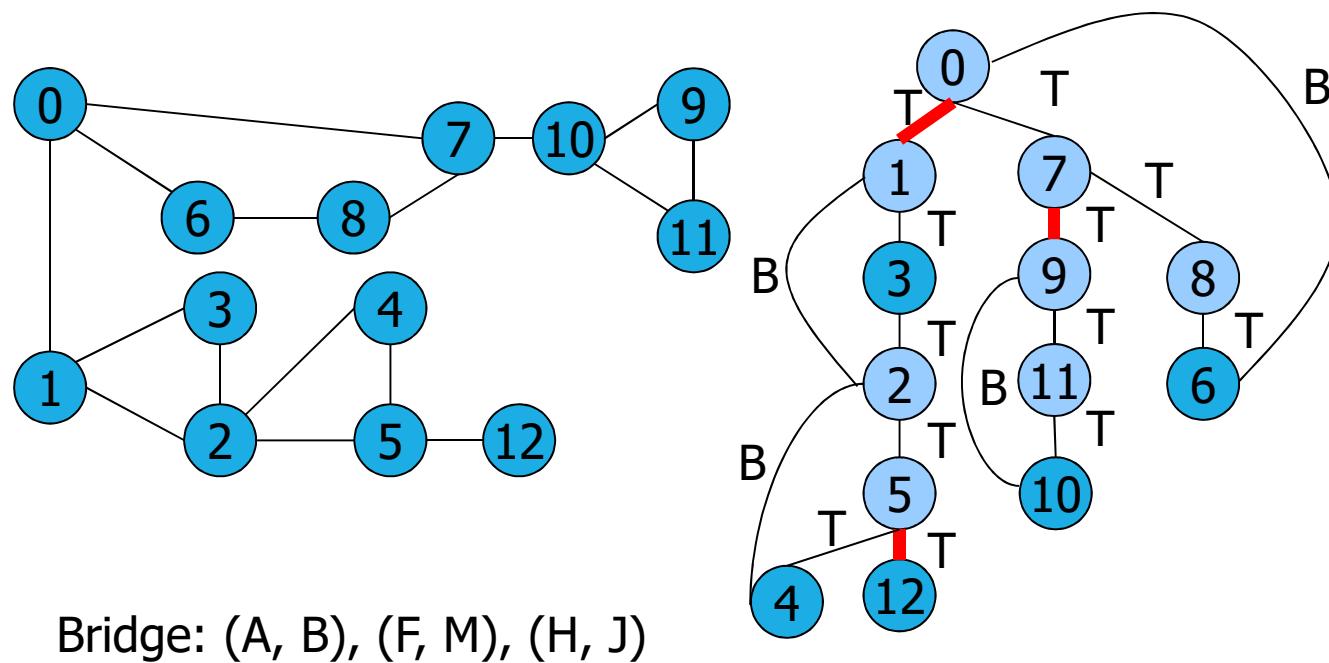
Bridge

Un arco (v,w) **Back** non può essere un ponte (i vertici v e w sono anche connessi da un cammino nell'albero della visita DFS).

Un arco (v,w) **Tree** è un ponte se e solo se non esistono archi **Back** che connettono un discendente di w a un antenato di v nell'albero della visita DFS.

Algoritmo banale: rimuovere gli archi uno alla volta e verificare se il grafo rimane连通的.

Esempio



Directed Acyclic Graph (DAG)

DAG: modelli impliciti per ordini parziali utilizzati nei problemi di scheduling.

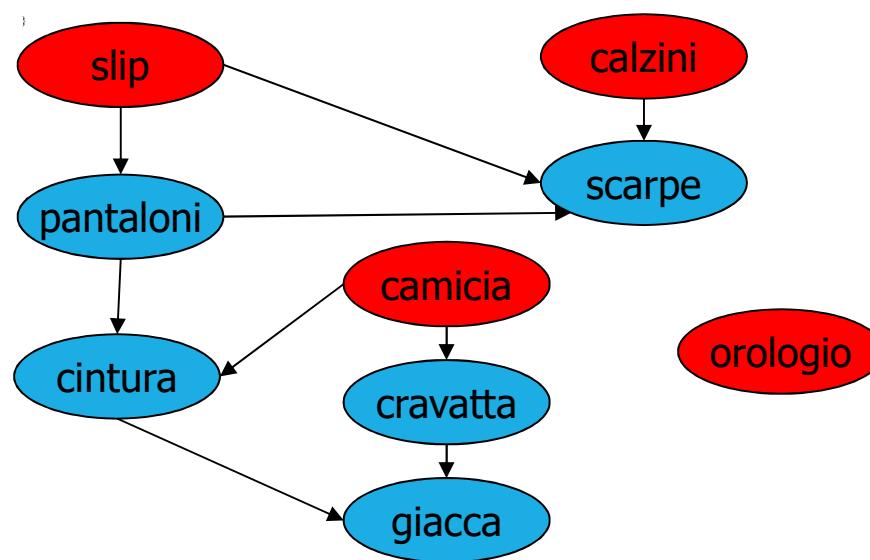
Scheduling:

- dati compiti (tasks) e vincoli di precedenza (constraints)
- come programmare i compiti in modo che siano tutti svolti rispettando le precedenze.

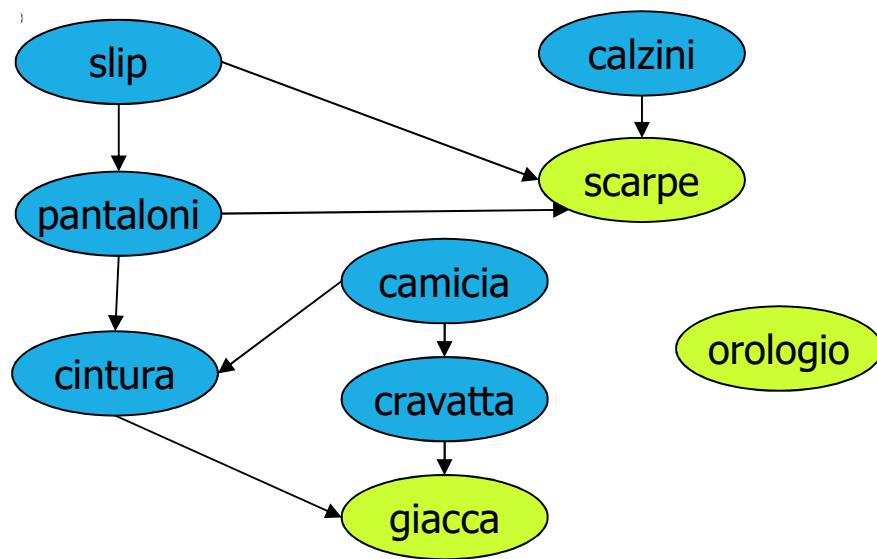
Nei DAG esistono 2 particolari classi di nodi:

- i nodi sorgente («source») che hanno in-degree=0
- i nodi pozzo o scolo («sink») che hanno out-degree=0.

Esempio



In rosso i nodi sorgente



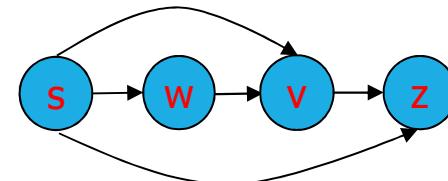
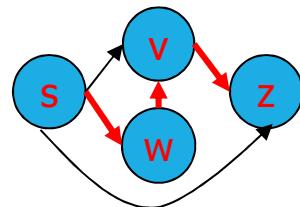
In verde i nodi pozzo

Ordinamento topologico: riordino dei vertici secondo una linea orizzontale, per cui se esiste l'arco (u, v) il vertice u compare a SX di v e gli archi vanno tutti da SX a DX.

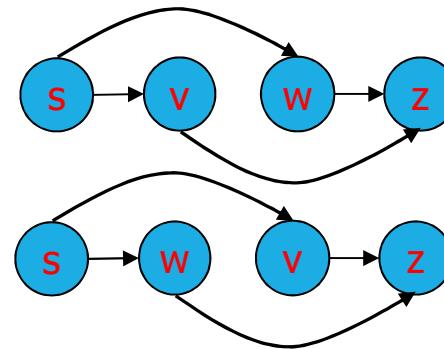
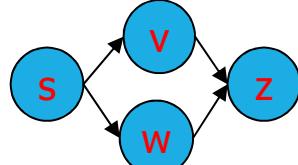
Ordinamento topologico inverso: riordino dei vertici secondo una linea orizzontale, per cui se esiste l'arco (u, v) il vertice u compare a DX di v e gli archi vanno tutti da DX a SX.

Unicità dell'ordinamento topologico

Se esiste un cammino hamiltoniano orientato, l'ordinamento topologico è unico. Tutte le coppie di vertici consecutivi sono connesse da archi.



Se \nexists cammino hamiltoniano orientato \Rightarrow l'ordinamento topologico non è unico

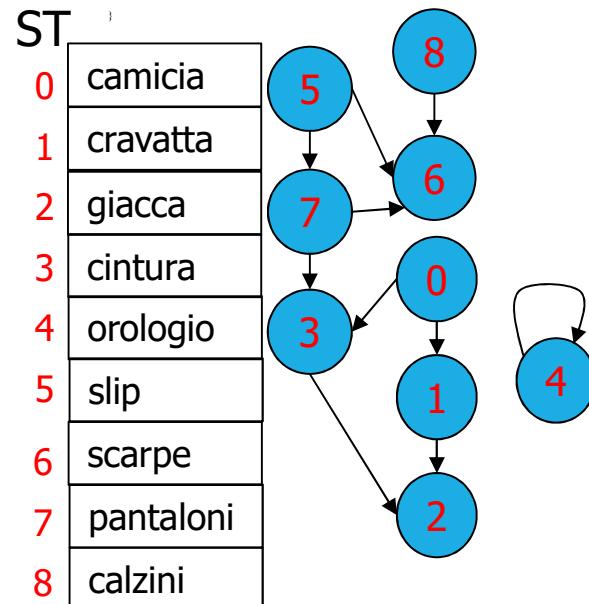


Ogni DAG ha quindi sempre almeno un ordinamento topologico.
Data la rappresentazione del grafo, il codice calcolerà un solo ordinamento topologico.

Esempio: ordinamento topologico inverso

Lista archi

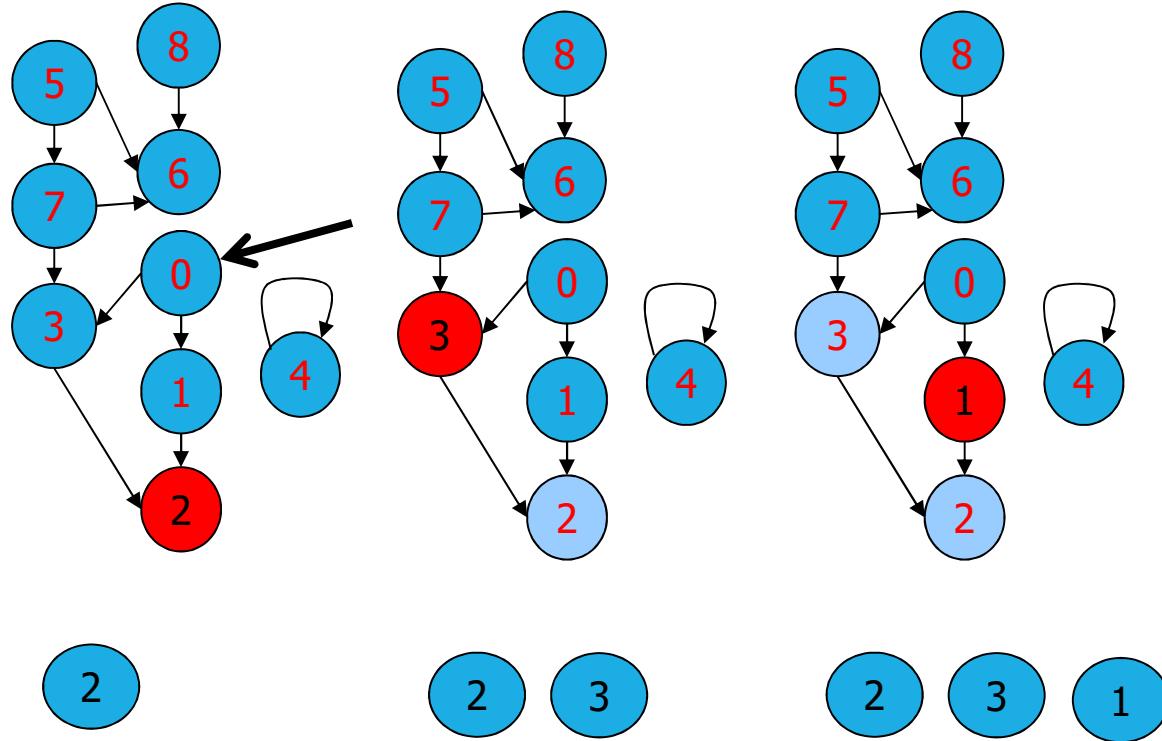
camicia cravatta
cravatta giacca
camicia cintura
cintura giacca
orologio orologio
slip scarpe
slip pantalone
calzini scarpe
pantaloni scarpe
pantaloni

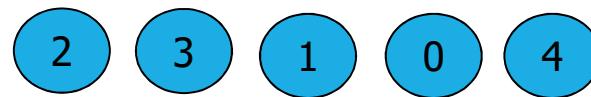
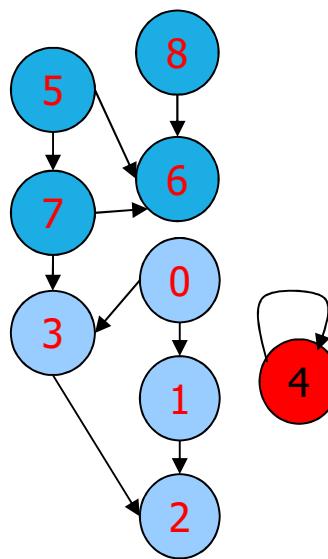
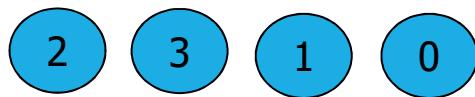
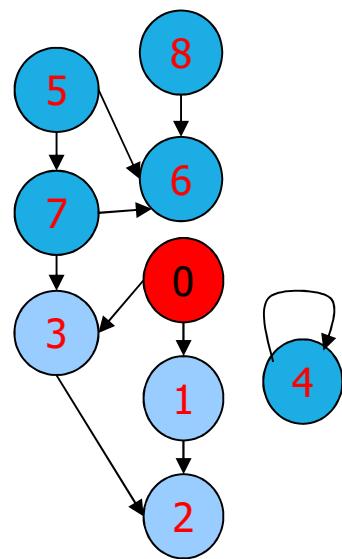


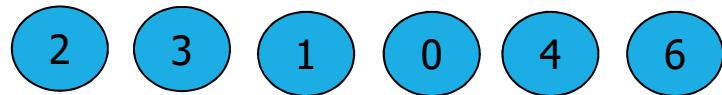
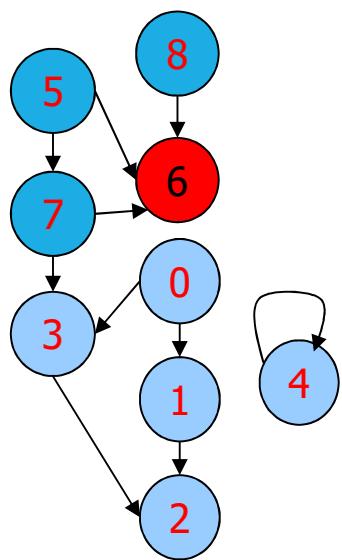
arco fittizio necessario per poter creare il nodo

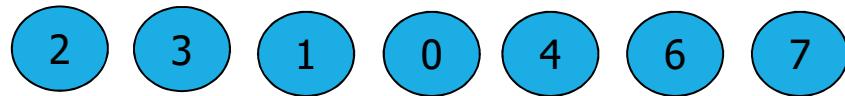
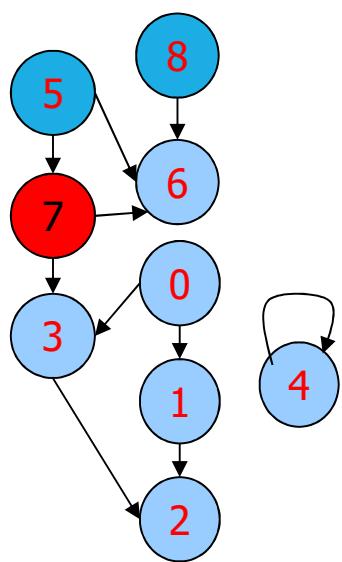
Lista delle adiacenze

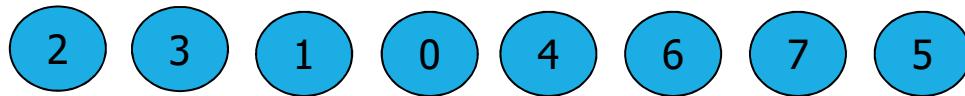
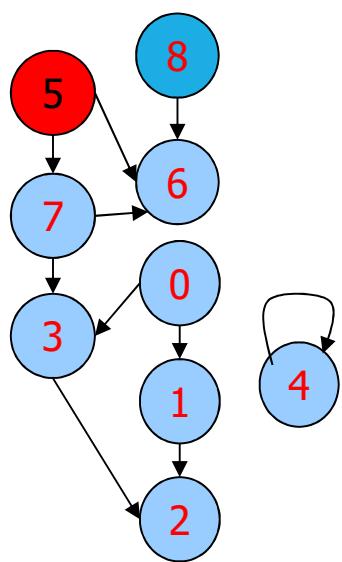
0	3	1
1	2	
2		
3	2	
4	4	
5	7	6
6		
7	3	6
8	6	

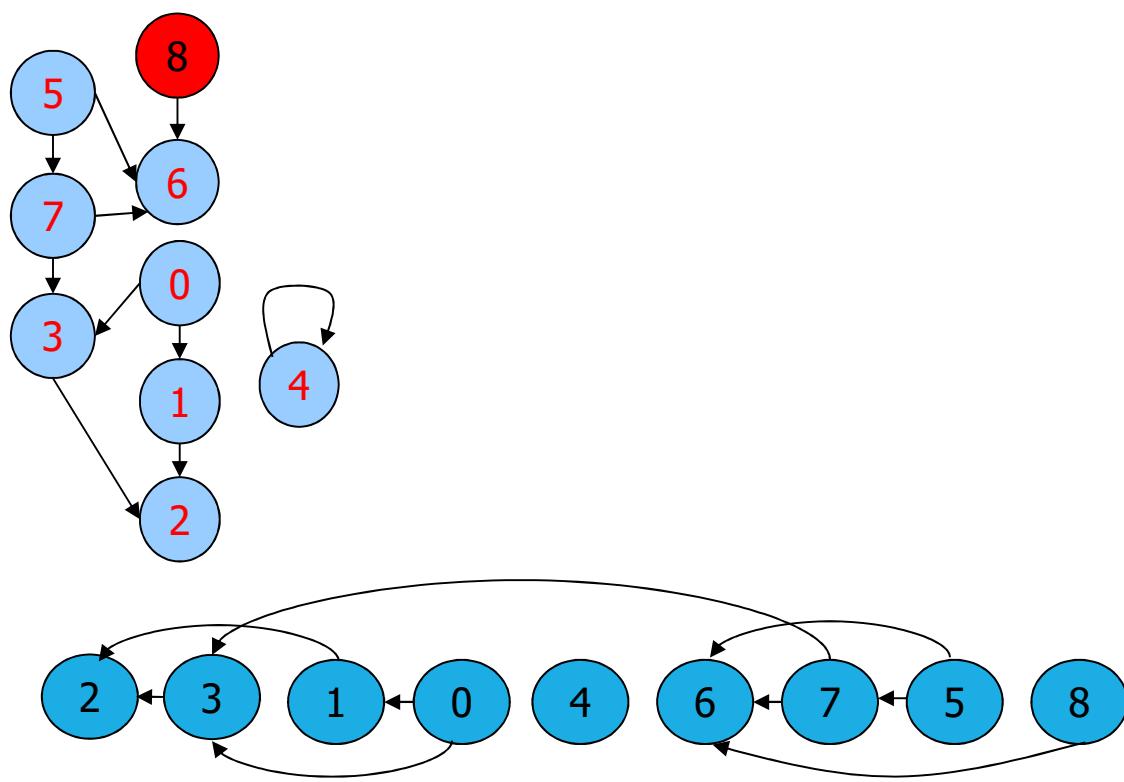












Struttura dati

- DAG come ADT di 1 classe
- rappresentazione come lista delle adiacenze
- vettori dove per ciascun vertice:
 - vettore `pre[i]` per registrare se il vertice è già stato scoperto o meno
 - vettore `ts[i]` dove per ciascun tempo si registra quale vertice è stato completato a quel tempo
- contatore `time` per tempi di completamento (avanza solo quando un vertice è completato, non scoperto)
- `time`, `*pre`, e `*ts` sono locali alla funzione `DAGrts` e passati by reference alla funzione ricorsiva `TSdfsR`.

wrapper

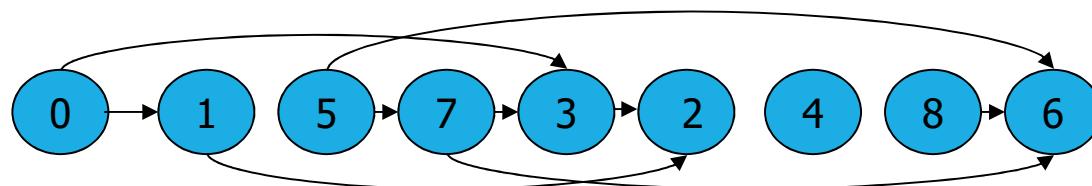
```

void TSdfsR(Dag D, int v, int *ts, int *pre, int *time) {
    link t; pre[v] = 0;
    for (t = D->ladj[v]; t != D->z; t = t->next)
        if (pre[t->v] == -1)
            TSdfsR(D, t->v, ts, pre, time);
    ts[(*time)++] = v;
}
void DAGrts(Dag D) {
    int v, time = 0, *pre, *ts;
    /* allocazione di pre e ts */
    for (v=0; v < D->V; v++) { pre[v] = -1; ts[v] = -1; }
    for (v=0; v < D->V; v++)
        if (pre[v]== -1) TSdfsR(D, v, ts, pre, &time);
    printf("DAG nodes in reverse topological order \n");
    for (v=0; v < D->V; v++)
        printf("%s ", STsearchByIndex(D->tab, ts[v]));
    printf("\n");
}

```

ordinamento topologico: con il DAG rappresentato da una matrice delle adiacenze, basta invertire i riferimenti riga-colonna (considerando gli archi incidenti):

```
void TSdfsR(Dag D, int v, int *ts, int *pre, int *time) {  
    int w;  
    pre[v] = 0;  
    for (w = 0; w < D->V; w++)  
        if (D->madj[w][v] != 0)  
            if (pre[w] == -1)  
                TSdfsR(D, w, ts, pre, time);  
    ts[(*time)++] = v;  
}
```



Componenti fortemente connesse

Algoritmo di Kosaraju (anni '80):

- trasporre il grafo
- eseguire DFS sul grafo trasposto, calcolando i tempi di scoperta e di fine elaborazione
- eseguire DFS sul grafo originale per tempi di fine elaborazione decrescenti
- gli alberi dell'ultima DFS sono le componenti fortemente connesse.

- Le SCC sono classi di equivalenza rispetto alla proprietà di mutua raggiungibilità
- Si può “estrarre” un grafo ridotto G' considerando un vertice come rappresentativo di ogni classe
- Il grafo ridotto G' è un DAG ed è detto “kernel DAG” del grafo G .

Grafo trasposto

Dato un grafo orientato $G=(V, E)$, il suo grafo trasposto $G^T=(V, E^T)$ è tale per cui

$$(u, v) \in E \Leftrightarrow (v, u) \in E^T.$$

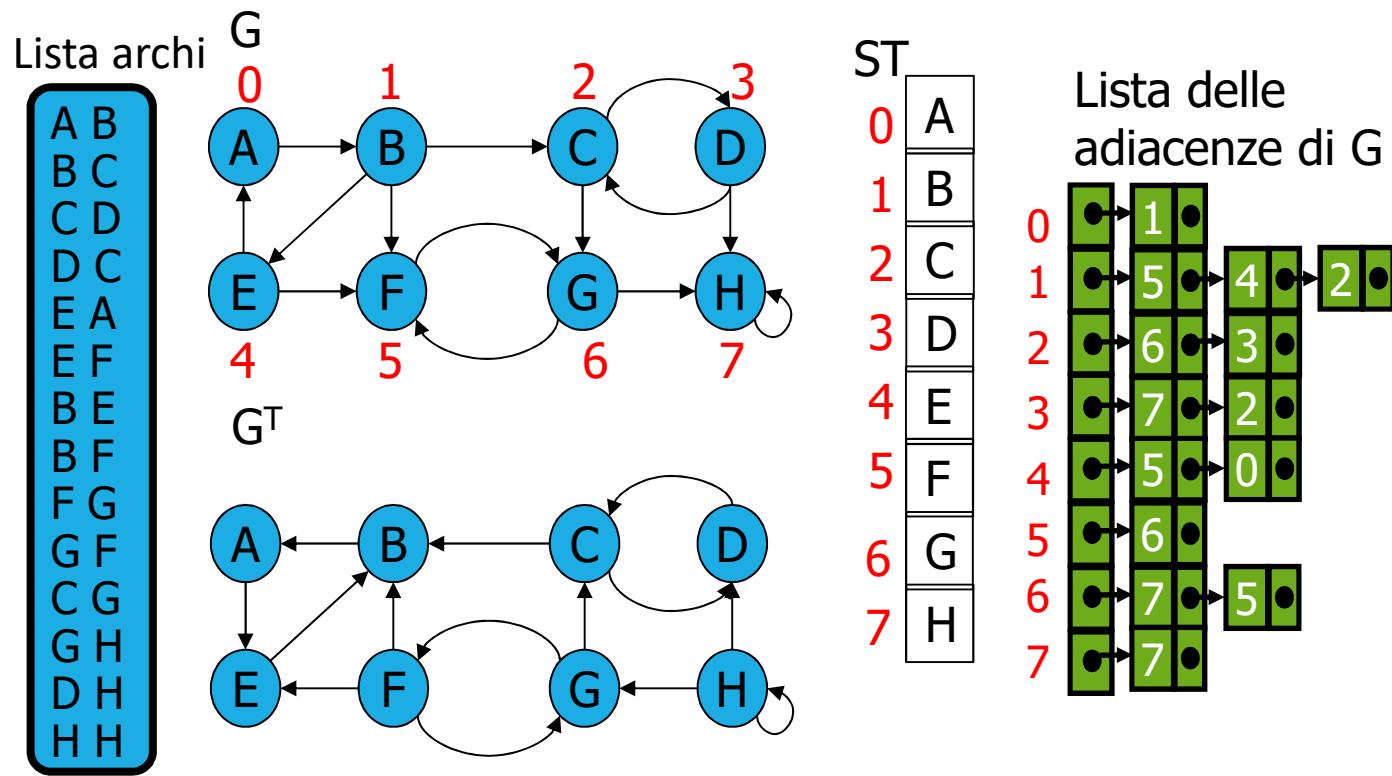
```
Graphreverse(Graph G) {
    int v;
    link t;
    Graph R = GRAPHinit(G->v);
    for (v=0; v < G->V; v++)
        for (t= G->1adj[v]; t != G->z; t = t->next)
            GRAPHinsertE(R, t->v, v);
    return R;
}
```

Algoritmo e strutture dati

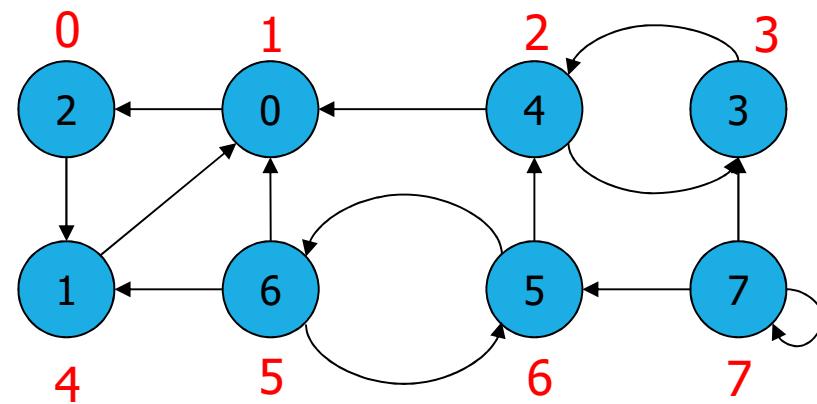
- GRAPHSCC è il wrapper con i vettori e le variabili locali passati by reference alla funzione ricorsiva SCCdfsR.
- in sccG[w] per ogni vertice si memorizza un intero che
 - identifica la componente fortemente connessa cui esso appartiene
 - marca anche se il vertice è stato visitato dalla DFS
- sccR[w] serve per marcare i vertici visitati dalla DFS del grafo trasposto
- time0 è il contatore del tempo che avanza solo quando di un vertice è terminata l'elaborazione (non serve il tempo di scoperta)

- `time1` è il contatore delle SCC
- `*postR` contiene per ogni valore del contatore di tempo `time0` quale vertice è stato terminato a quel tempo
- l'istruzione `post [(*time0) ++] =w` registra che al tempo `(*time0)` è stato terminato `w`, quindi c'è un implicito ordinamento per tempi di completamento crescenti
- percorrendo in discesa il vettore `postR` si considerano i vertici in ordine di tempo di fine elaborazione decrescente senza bisogno di un algoritmo di ordinamento
- `*postG` viene introdotto soltanto per avere un'unica versione della funzione ricorsiva `SCCdfsR` utilizzabile sia sul grafo `G`, sia sul grafo trasposto `GT`.

Esempio



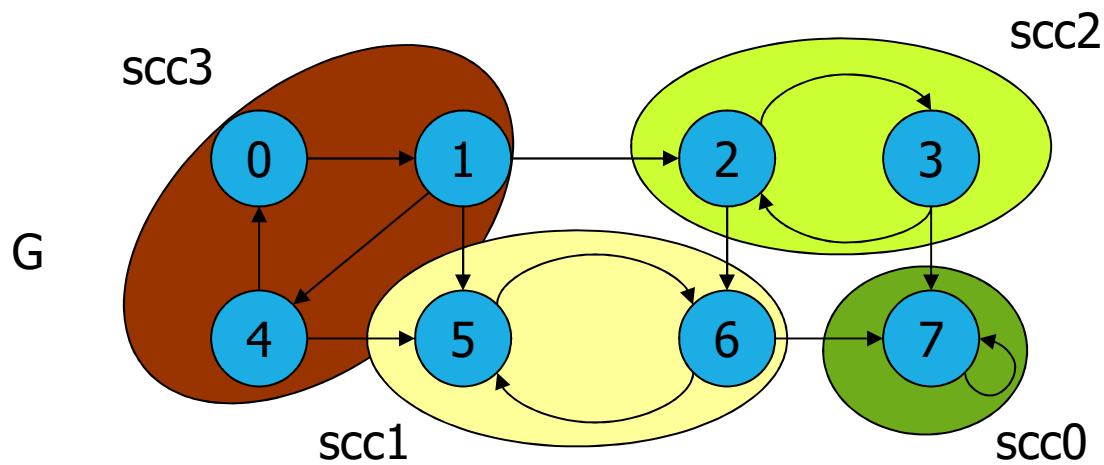
Visita DFS del grafo trasposto G^T



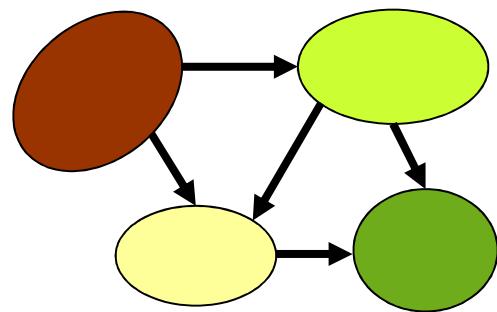
postR	1	4	0	3	2	6	5	7
0	0	1	2	3	4	5	6	7

Visita DFS del grafo secondo tempi decrescenti di fine elaborazione del grafo trasposto G^T (percorrimento in discesa di postR)

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



Kernel DAG



```

void SCCdfsR(Graph G,int w,int *scc,int *time0,int time1,int *post) {
    link t;
    scc[w] = time1;
    for (t = G->ladj[w]; t != G->z; t = t->next)
        if (scc[t->v] == -1)
            SCCdfsR(G, t->v, scc, time0, time1, post);
    post[(*time0)++]= w;
}
int GRAPHscc(Graph G) {
    int v, time0 = 0, time1 = 0, *sccG, *sccR, *postG, *postR;
    Graph R = GRAPHreverse(G);

    sccG = malloc(G->V * sizeof(int));
    sccR = malloc(G->V * sizeof(int));
    postG = malloc(G->V * sizeof(int));
    postR = malloc(G->V * sizeof(int));

```

```

for (v=0; v < G->V; v++) {
    sccG[v]=-1; sccR[v]=-1; postG[v]=-1; postR[v]=-1;
}
for (v=0; v < G->V; v++)
    if (sccR[v] == -1)
        SCCdfsR(R, v, sccR, &time0, time1, postR);
time0 = 0; time1 = 0;
for (v = G->V-1; v >= 0; v--)
    if (sccG[postR[v]]== -1){
        SCCdfsR(G,postR[v], sccG, &time0, time1, postG);
        time1++;
    }
printf("strongly connected components \n");
for (v = 0; v < G->V; v++)
    printf("node %s in scc %d\n",STsearchByIndex(G->tab,v),sccG[v]);
return time1;
}

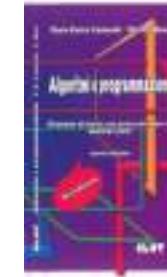
```

Riferimenti

- Componenti connesse:
 - Sedgewick Part 5 18.5
- Bridge e punti di articolazione:
 - Sedgewick Part 5 18.6
- DAG e ordinamento topologico dei DAG:
 - Sedgewick Part 5 19.5 e 19.6
 - Cormen 23.4
- Componenti fortemente connesse:
 - Sedgewick Part 5 19.8
 - Cormen 23.5

Esercizi di teoria

- 10. Visite dei grafi e applicazioni
 - 10.4 Componenti connesse
 - 10.5 Componenti fortemente connesse
 - 10.6 Punti di articolazione
 - 10.7 Ordinamento topologico dei DAG



Gli alberi ricoprenti minimi

Gianpiero Cabodi e Paolo Camurati



Alberi ricoprenti minimi

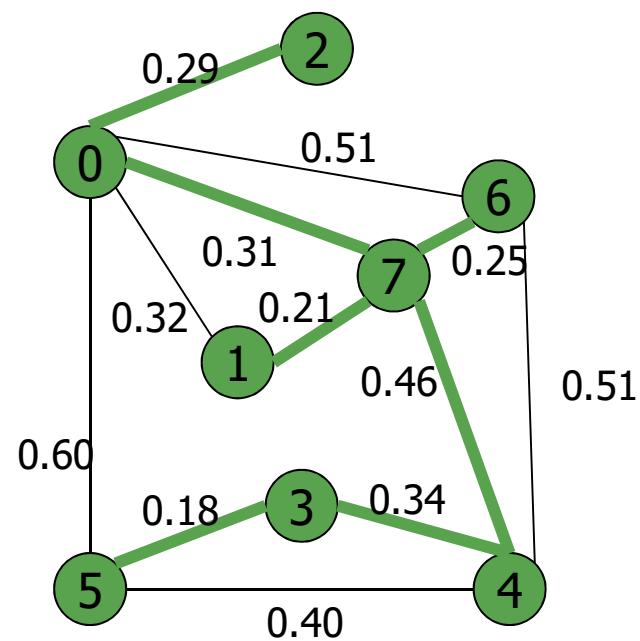
Dato $G=(V,E)$ grafo non orientato, pesato con pesi reali $w: E \rightarrow \mathbb{R}$ e connesso, estrarre da G un **albero ricoprente minimo** (Minimum-weight Spanning Tree – MST), definito come:

- grafo $G'=(V, A)$ dove $A \subseteq E$
- aciclico
- minimizza $w(A) = \sum_{(u,v) \in T} w(u,v)$.

Acicità && copertura di tutti i vertici $\Rightarrow G'$ è un albero.

L'albero MST è unico se e solo se tutti i pesi sono distinti.

Esempio



Rappresentazione

ADT grafo non orientato e pesato: estensione dell'ADT grafo non orientato:

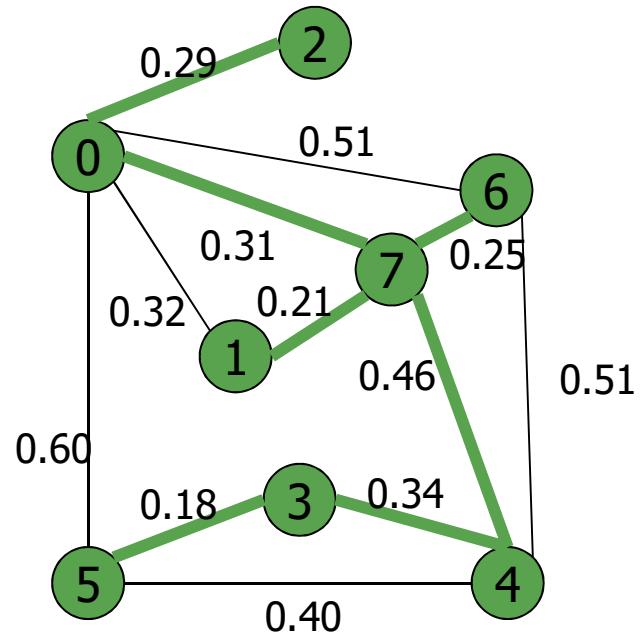
- lista delle adiacenze
- matrice delle adiacenze

Valore-sentinella per indicare l'assenza di un arco (peso inesistente):

- maxWT (idealmente $+\infty$), soluzione scelta nell'algoritmo di Prim
- 0 se non sono ammessi archi a peso 0
- -1 se non sono ammessi archi a peso negativo.

Per semplicità si considerano pesi interi e non reali.

Rappresentazione degli MST: soluzione 1



Elenco di archi, memorizzato in un vettore
di archi `mst [maxE]`

0-2 0.29

4-3 0.34

5-3 0.18

7-4 0.46

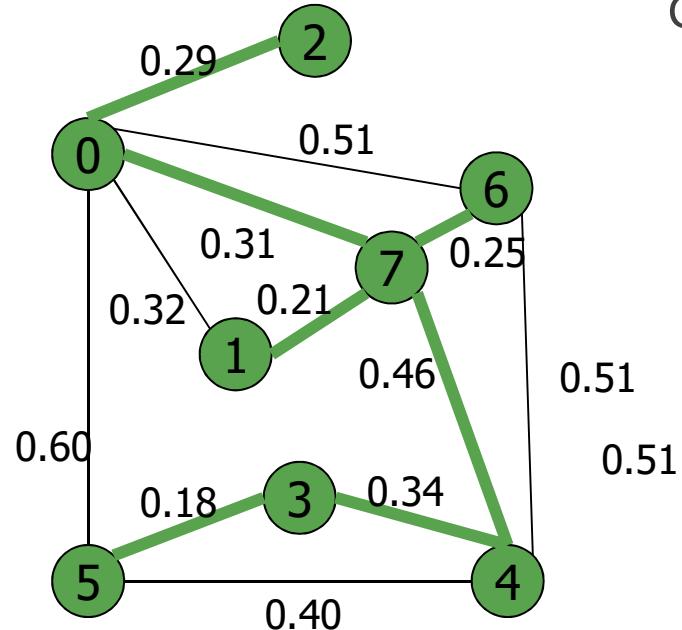
7-0 0.31

7-6 0.25

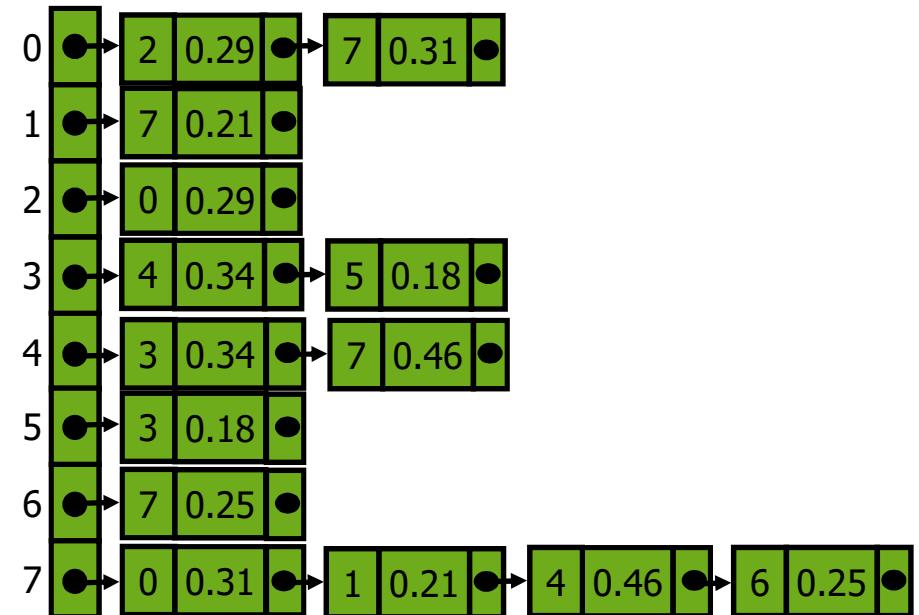
7-1 0.21

Soluzione usata nell'Algoritmo di Kruskal.

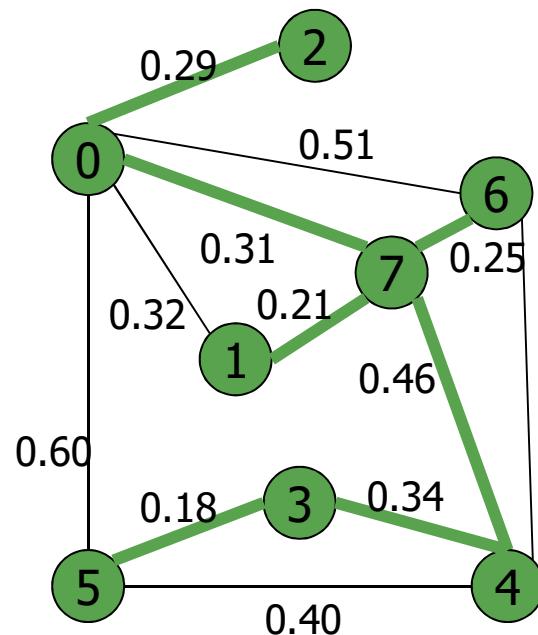
Rappresentazione degli MST: soluzione 2



Grafo come lista di adiacenze:



Rappresentazione degli MST: soluzione 3



Vettore st dei padri e wt dei pesi

	0	1	2	3	4	5	6	7
st	0	7	0	4	7	3	7	0
wt	0	.21	.29	.34	.46	.18	.25	.31

0.51 Soluzione usata nell'Algoritmo di Prim.

Approccio completo

- Dati V vertici, l'albero ricoprente avrà esattamente $V-1$ archi
- si esplorano tutte le maniere di raggruppare $V-1$ archi scelti dagli E archi del grafo
 - l'ordine non conta (combinazioni)
 - condizione di accettazione: verifica di aciclicità
 - problema di ottimizzazione: si considerano tutte le soluzioni e si sceglie la migliore
- costo: esponenziale.

Approccio greedy

In generale:

- ad ogni passo, si sceglie la soluzione localmente ottima
- non garantisce necessariamente una soluzione globalmente ottima.

Per gli MST:

- si parte da un algoritmo **incrementale, generico e greedy** dove la scelta è ottima localmente
- si può dimostrare che la soluzione è globalmente ottima.

Algoritmo incrementale, generico e greedy

- La soluzione A corrente è un sottoinsieme degli archi di un albero ricoprente minimo.
- inizialmente A è l'insieme vuoto
- ad ogni passo si aggiunge ad A un arco “sicuro”
- fino a quando A non diventa un albero ricoprente minimo.

Invarianza: l'arco (u,v) è **sicuro** se e solo se aggiunto ad un sottoinsieme di un albero ricoprente minimo produce ancora un sottoinsieme di un albero ricoprente minimo.

Un teorema e un corollario permettono di identificare gli archi sicuri..

Tagli e archi

Dato $G=(V,E)$ grafo non orientato, pesato, connesso, si definisce **taglio** una partizione di V in S e $V-S$

$$V = S \cup V-S \quad \& \quad S \cap V-S = \emptyset$$

tale che se $(u,v) \in E$ attraversa il taglio allora

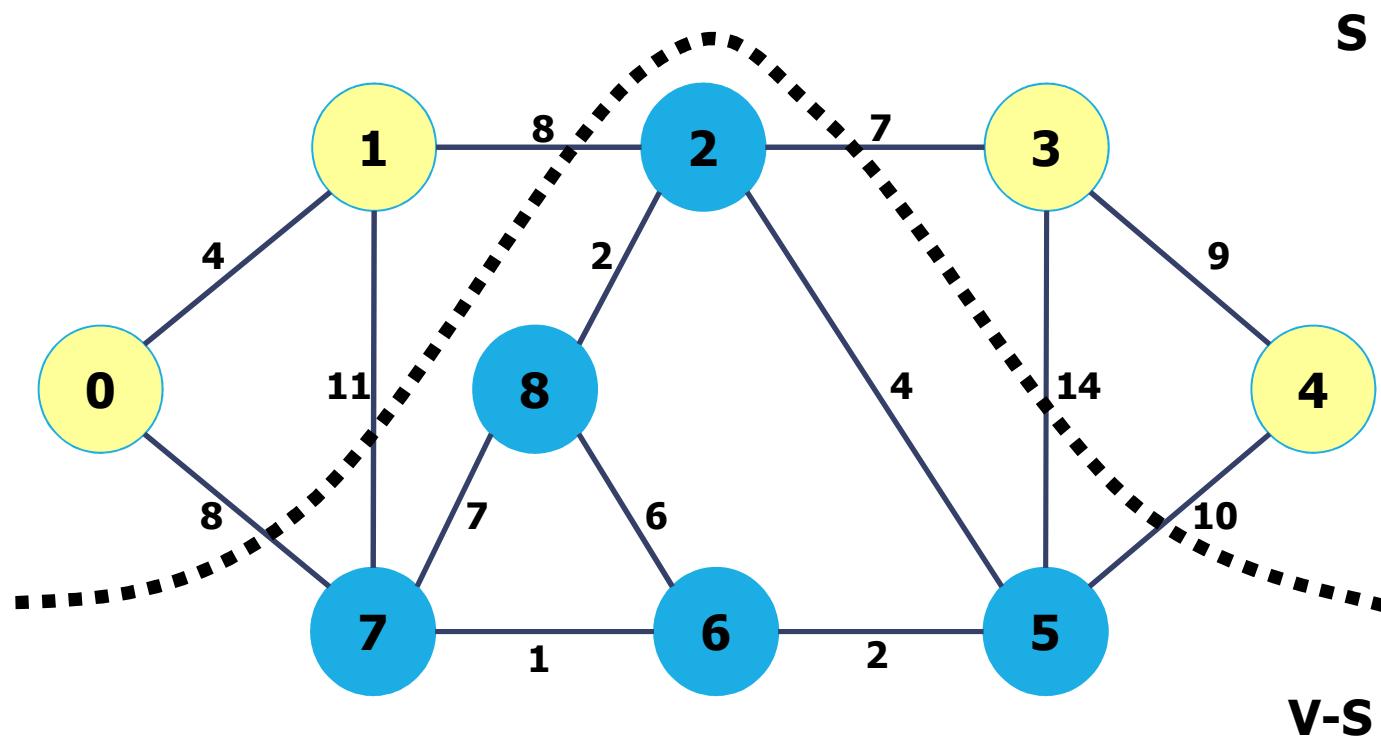
$$u \in S \quad \& \quad v \in V-S \text{ (o viceversa)}$$

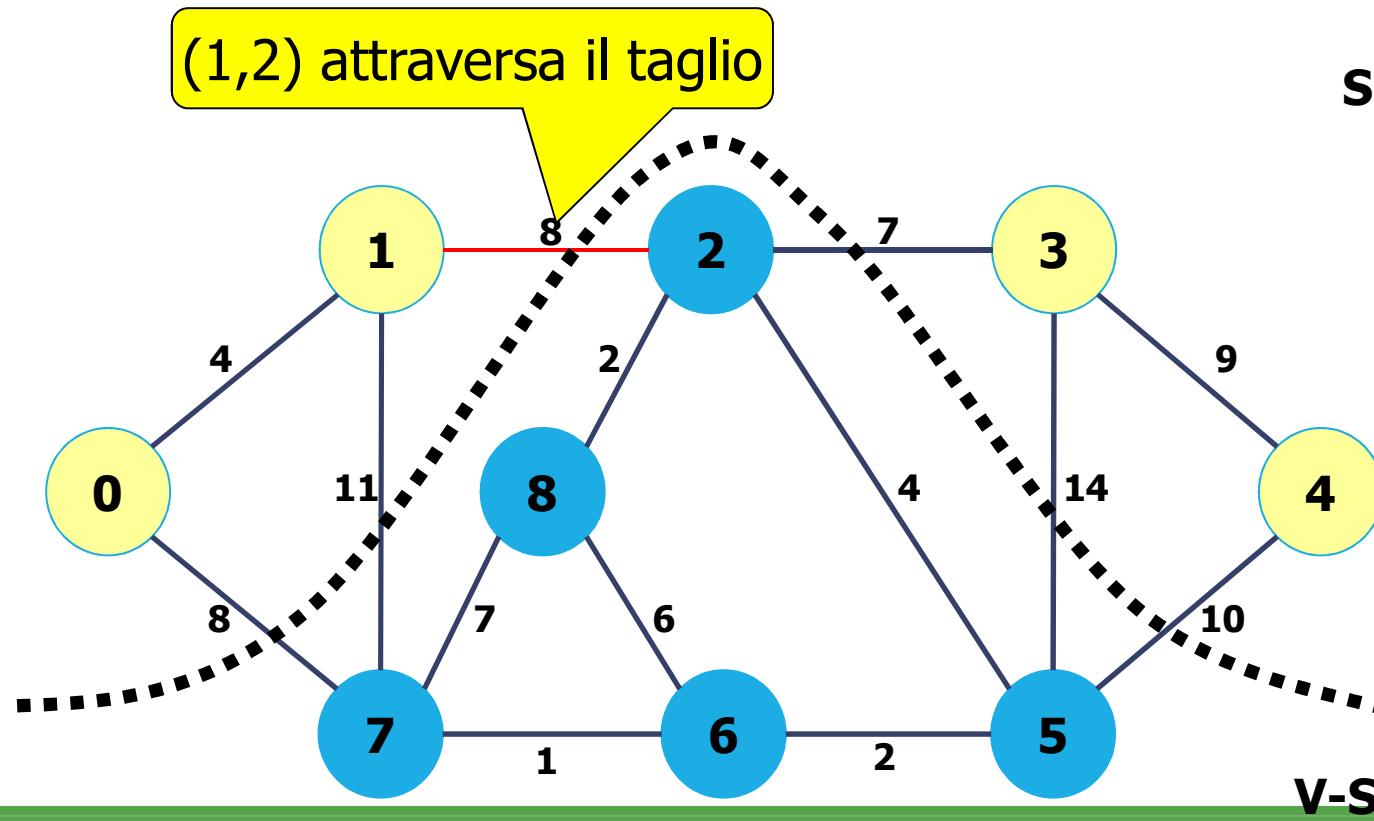
Un taglio rispetta un insieme A di archi se nessun arco di A attraversa il taglio.

Un arco si dice leggero se ha **peso minimo** tra gli archi che attraversano il taglio.

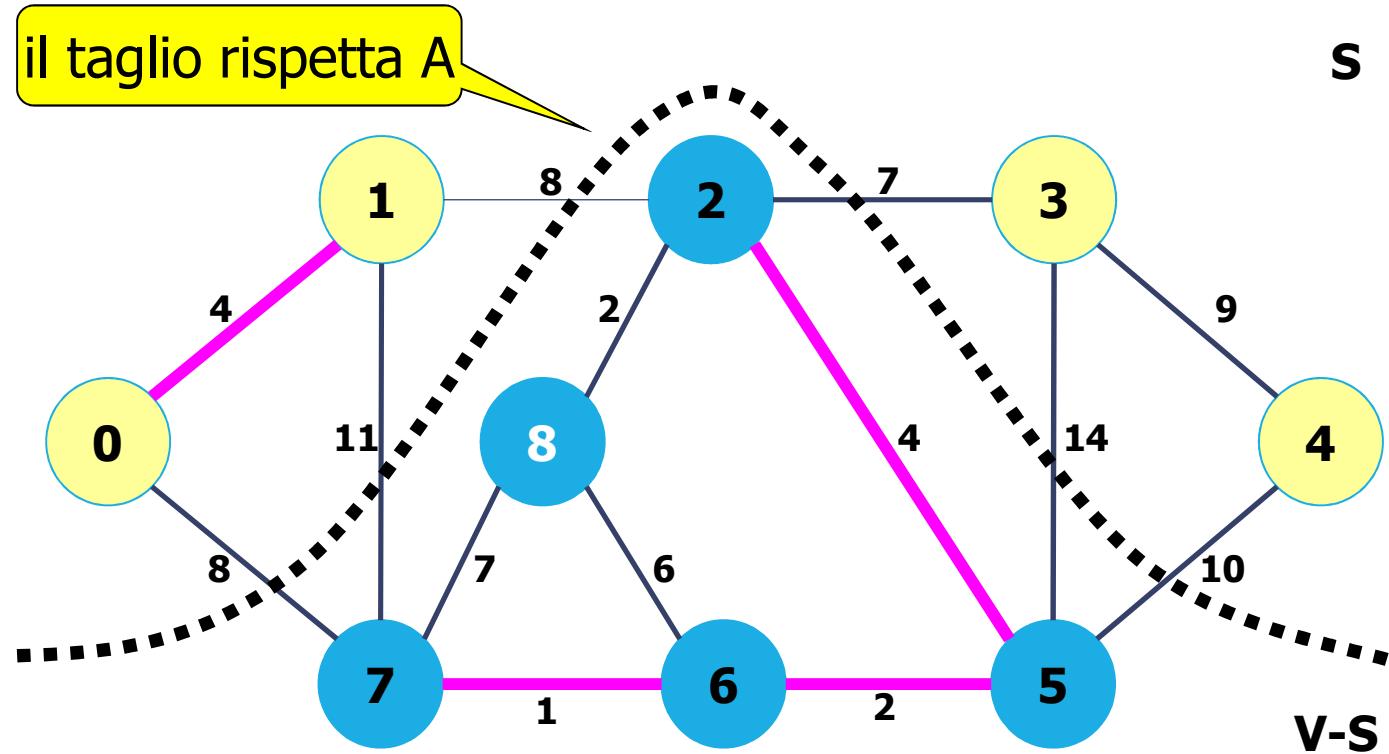
greedy: scelta localmente ottima

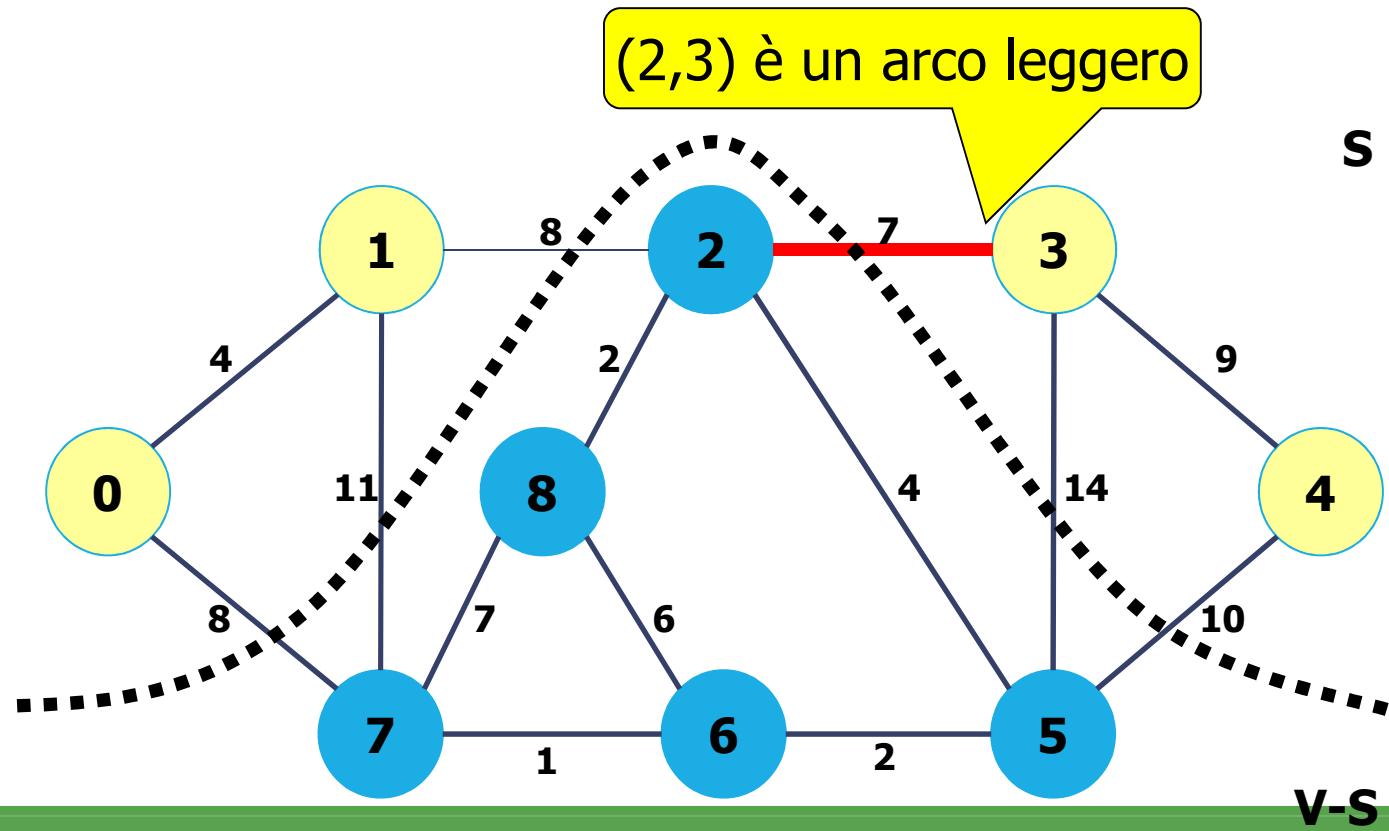
Esempio





Posto che A sia $A = \{(0,1), (2,5), (5,6), (6,7)\}$





Archi sicuri: teorema

Dati:

- $G=(V,E)$ grafo non orientato, pesato, connesso
- A sottoinsieme degli archi

se:

- $A \subseteq E$ è contenuto in un qualche albero ricoprente minimo di G . Inizialmente A è vuoto
- $(S, V-S)$ è un taglio qualunque che rispetta A
- (u,v) è un arco leggero che attraversa $(S, V-S)$

allora

(u,v) è sicuro per A .

Archi sicuri: corollario

Dati:

- $G=(V,E)$ grafo non orientato, pesato, connesso
- A sottoinsieme degli archi

se:

- $A \subseteq E$ è contenuto in un qualche albero ricoprente minimo di G . Inizialmente A è vuoto
- C è un albero nella foresta $G_A = (V,A)$
- (u,v) è un arco leggero che connette C ad un altro albero in G_A

allora

(u,v) è sicuro per A .

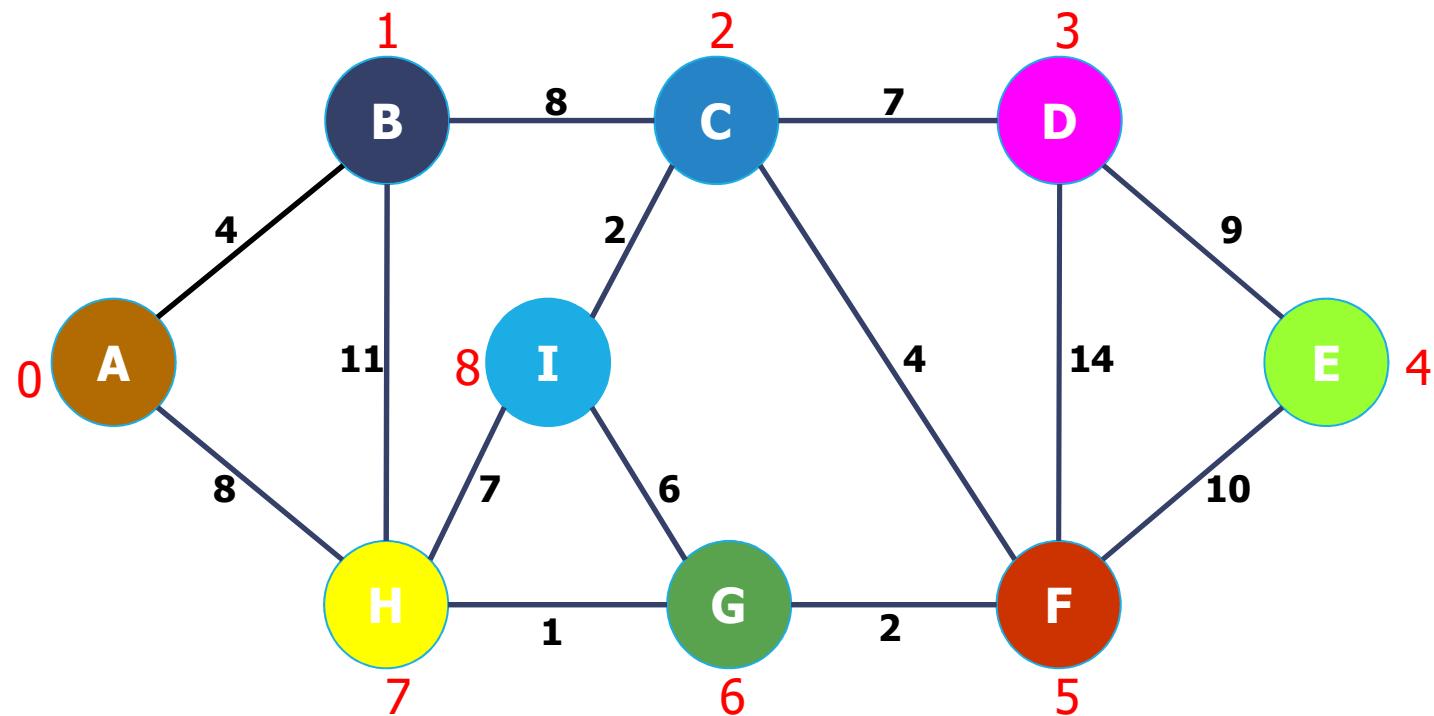
Algoritmo di Kruskal (1956)

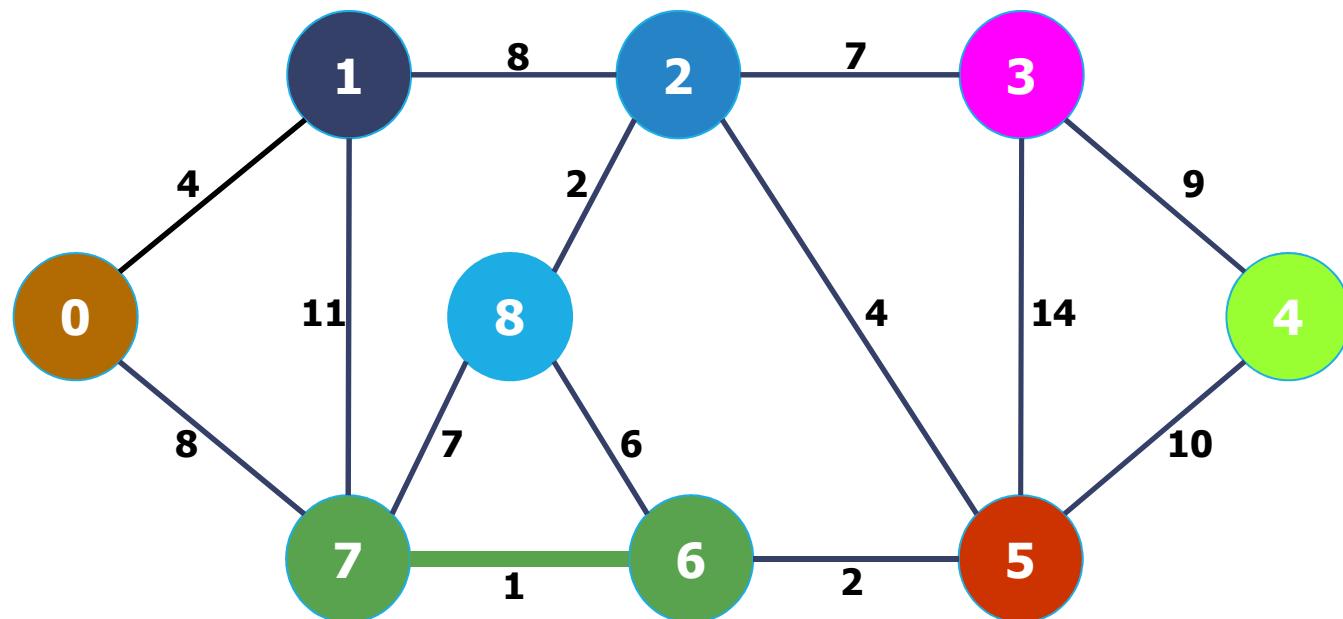
- basato su algoritmo generico
- uso del corollario per determinare l'arco sicuro:
 - si considera una foresta di alberi, inizialmente formati dai singoli vertici
 - si ordinano degli archi per pesi crescenti
 - si itera la selezione di un arco sicuro: arco di peso minimo che connette 2 alberi generando ancora un albero
 - terminazione: considerati tutti i vertici.

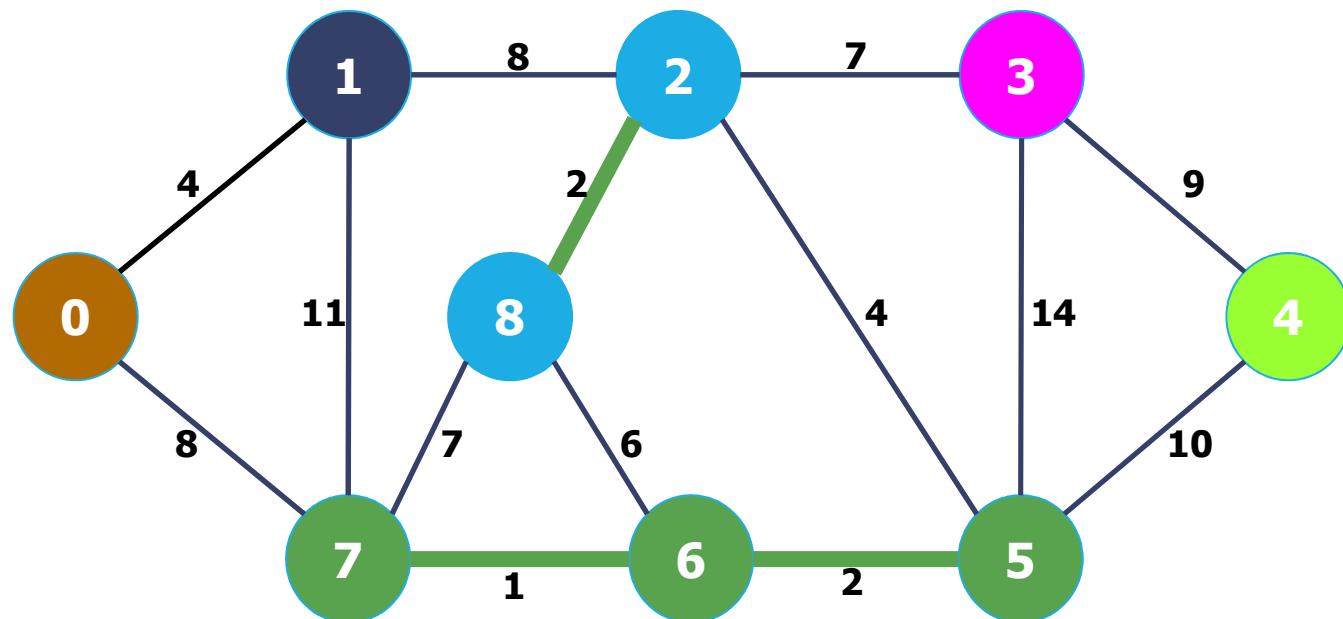


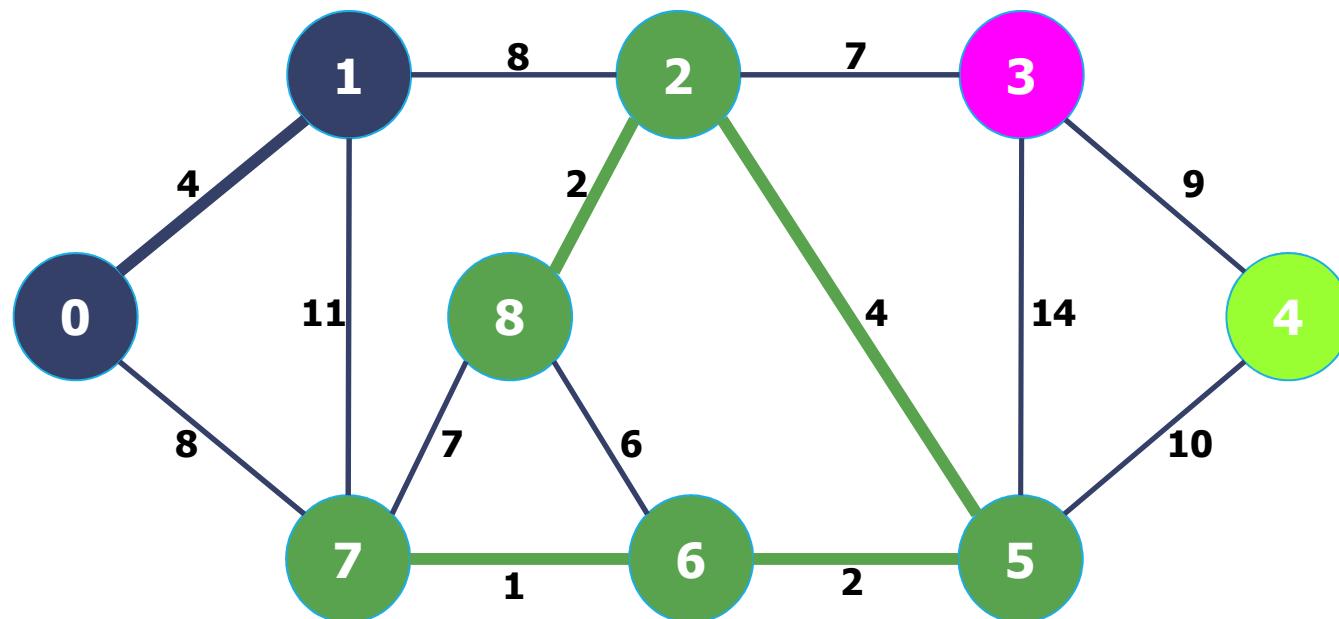
La rappresentazione degli alberi è fatta con ADT Union-Find (vedi Tecniche di Programmazione, lezione sull'Analisi della Complessità, problema dell'On-line connectivity)

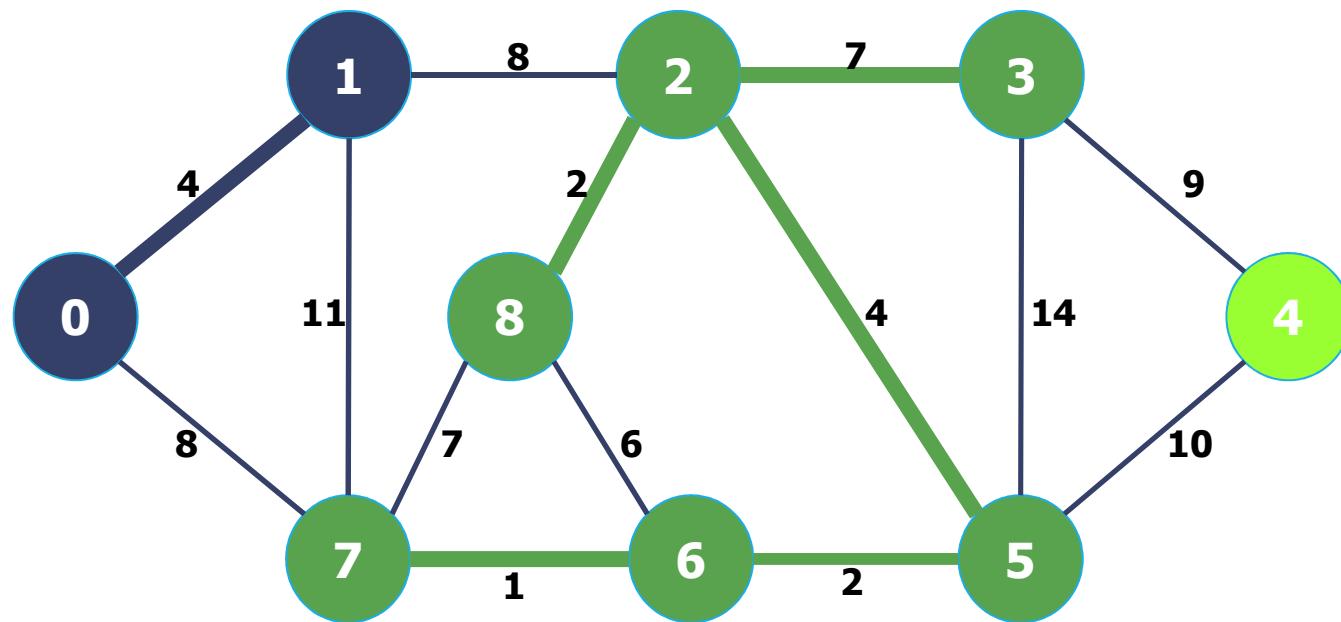
Esempio

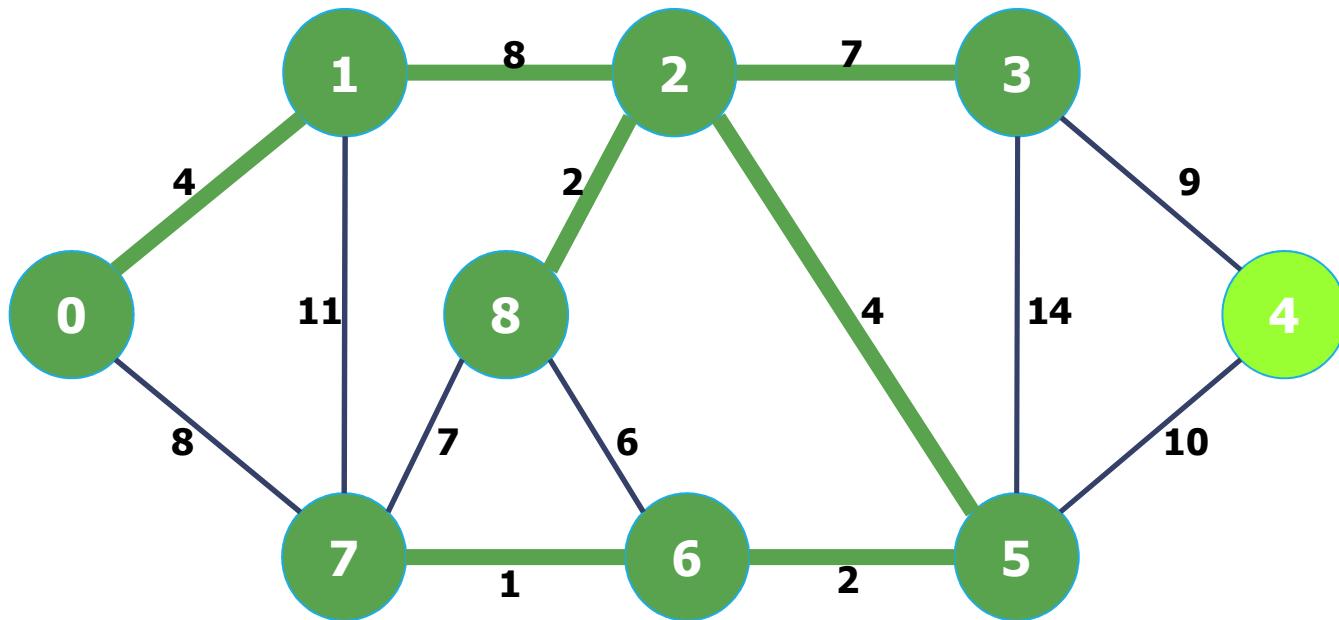


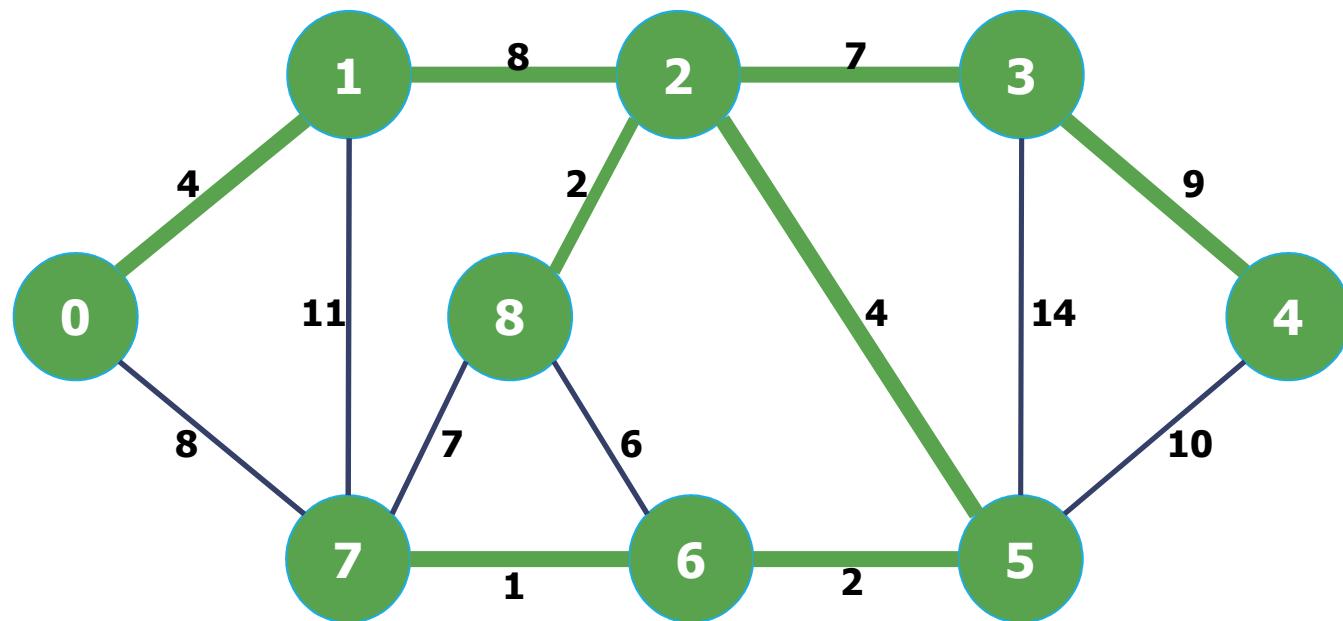












ADT Union-find (1964)

- Struttura dati per memorizzare una collezione di insiemi disgiunti, ad esempio la partizione di un insieme in più sottoinsiemi (disgiunti per definizione di partizione)
- anche nota come struttura dati **disjoint-set data** o **merge–find set**
- Operazioni:
 - **UUnion**: fusione di 2 sottoinsiemi
 - **UFfind**: verifica se 2 elementi appartengono o meno allo stesso sottoinsieme

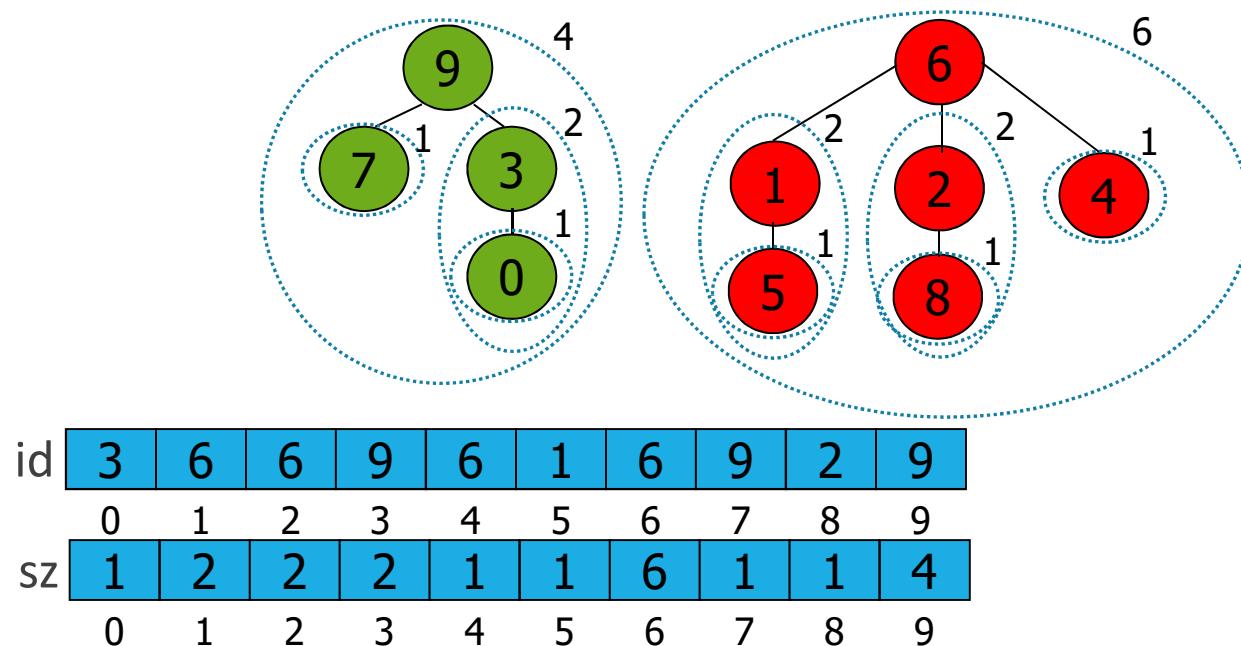
Strutture dati

Dato un insieme di N elementi denominati da 0 a N-1:

- il vettore id contiene per ogni elemento l'indice dell'elemento che lo rappresenta
- inizialmente ogni elemento rappresenta se stesso
- l'elemento che rappresenta il sottoinsieme è quello che rappresenta se stesso (indice coincide con contenuto a quell'indice)
- il vettore sz contiene la cardinalità del sottoinsieme cui appartiene ogni elemento

Esempio

L'insieme di 10 elementi denominati da 0 a 9 è partizionato in 2 sottoinsiemi $\{0, 3, 7, 9\}$ e $\{1, 2, 4, 5, 6, 8\}$ così rappresentati



ADT di I classe UF

- Versione con weighted quick-union
- **quick-union**: un elemento punta a chi lo rappresenta, quindi complessità $O(1)$
- **find**: percorrimento di una “catena” dall’elemento fino al rappresentante “ultimo” del sottoinsieme. Grazie al mantenimento dell’informazione sulla cardinalità dell’insieme che permette di fondere l’insieme a cardinalità minore in quello a cardinalità maggiore, generando un cammino di lunghezza logaritmica, la complessità è $O(\log N)$.

UF.h

```
void UFinit(int N);
int UFFind(int p, int q);
void UFunion(int p, int q);
```

UF.c

```
#include <stdlib.h>
#include "UF.h"
static int *id, *sz;
void UFinit(int N) {
    int i;
    id = malloc(N*sizeof(int));
    sz = malloc(N*sizeof(int));
    for(i=0; i<N; i++) {
        id[i] = i; sz[i] = 1;
    }
}
```

```
static int find(int x) {
    int i = x;
    while (i != id[i]) i = id[i];
    return i;
}
int UFind(int p, int q) { return(find(p) == find(q)); }

void Union(int p, int q) {
    int i = find(p), j = find(q);
    if (i == j) return;
    if (sz[i] < sz[j]) {
        id[i] = j; sz[j] += sz[i];
    }
    else {
        id[j] = i; sz[i] += sz[j];
    }
}
```

wrapper

```
void GRAPHmstK(Graph G) {
    int i, k, weight = 0;
    Edge *mst = malloc((G->V-1) * sizeof(Edge));
    Edge *a = malloc(G->E * sizeof(Edge));

    k = mstE(G, mst, a);

    printf("\nEdges in the MST: \n");
    for (i=0; i < k; i++) {
        printf("%s - %s \n", STsearchByIndex(G->tab, mst[i].v),
               STsearchByIndex(G->tab, mst[i].w));
        weight += mst[i].wt;
    }
    printf("minimum weight: %d\n", weight);
}
```

```
int mstE(Graph G, Edge *mst, Edge *a) {
    int i, k;

    GRAPHedges(G, a);
    sort(a, 0, G->E-1);
    UFinit(G->V);
    for (i=0, k=0; i < G->E && k < G->V-1; i++ )
        if (!UFFind(a[i].v, a[i].w)) {
            UFunion(a[i].v, a[i].w);
            mst[k++]=a[i];
        }

    return k;
}
```

Complessità

Con l'ADT UF:

$$T(n) = O(|E| \lg |E|) = O(|E| \lg |V|)$$

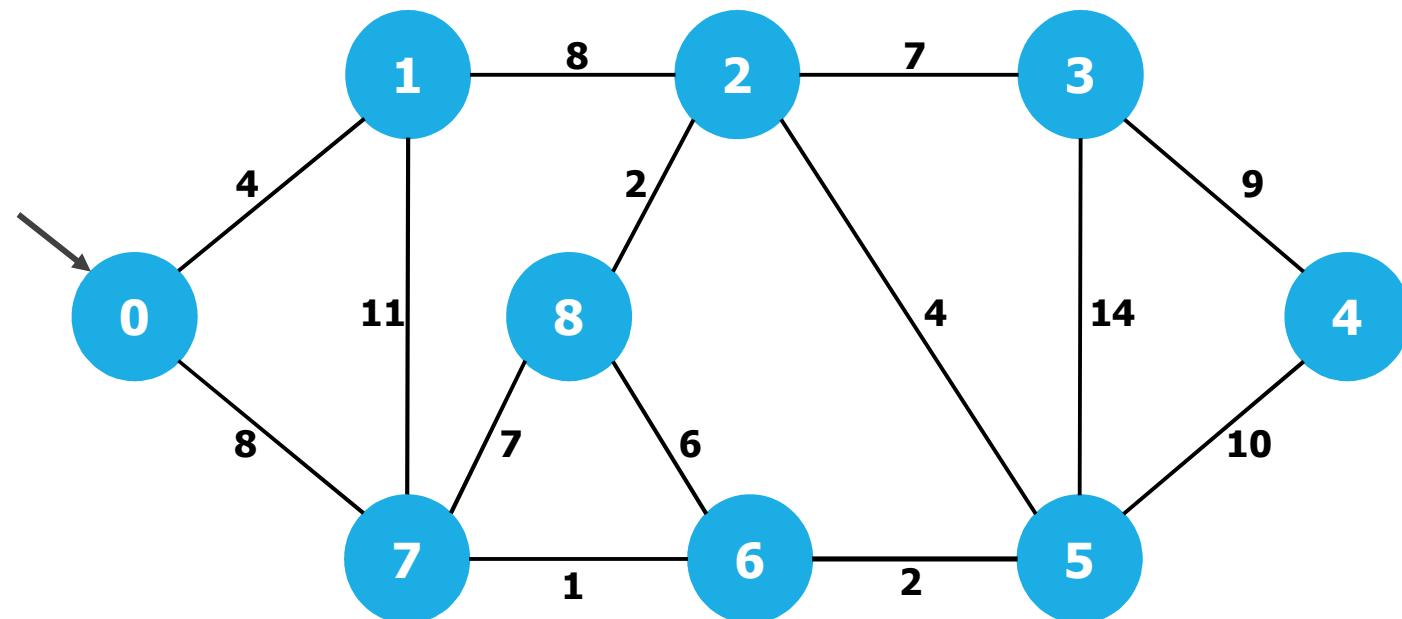
in quanto, ricordando che $|E| = O(|V|^2)$ (grafo completo),
 $\log |E| = O(\log |V|^2) = O(2\log |V|) = O(\log |V|)$.

Algoritmo di Prim (1930-1959)



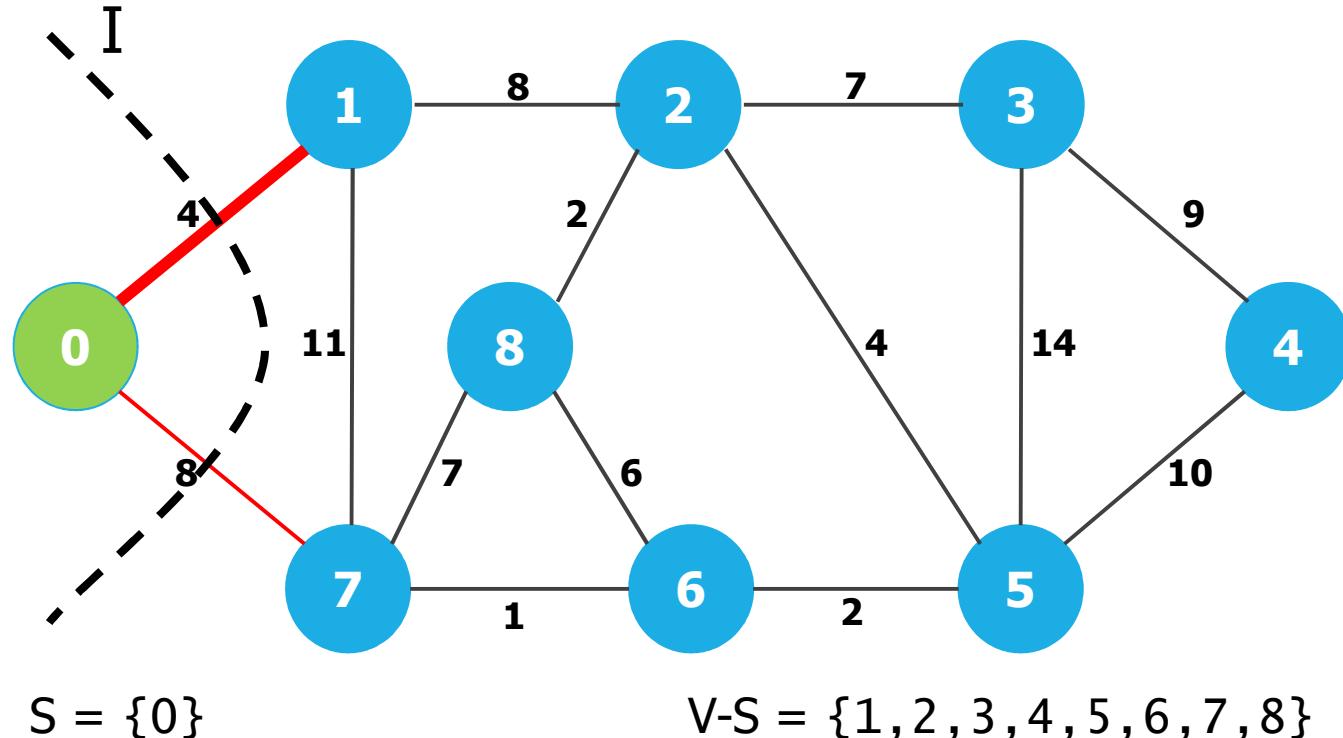
- basato su algoritmo generico
- soluzione brute-force
- uso del teorema per determinare l'arco sicuro:
 - inizialmente $S = \emptyset$, poi $S = \{\text{vertice di partenza}\}$
 - iterazione: $V-1$ passi in cui si aggiunge 1 arco alla soluzione
 - iterazione sugli archi per selezionarne 1:
 - selezionare quello di peso minimo tra gli archi che attraversano il taglio e aggiungerlo alla soluzione
 - in base al vertice in cui arriva l'arco, aggiornare S
 - terminazione: considerati tutti i vertici, quindi soluzione che contiene $V-1$ archi
 - versione semplice, ma non efficiente a causa del ciclo annidato sugli archi.

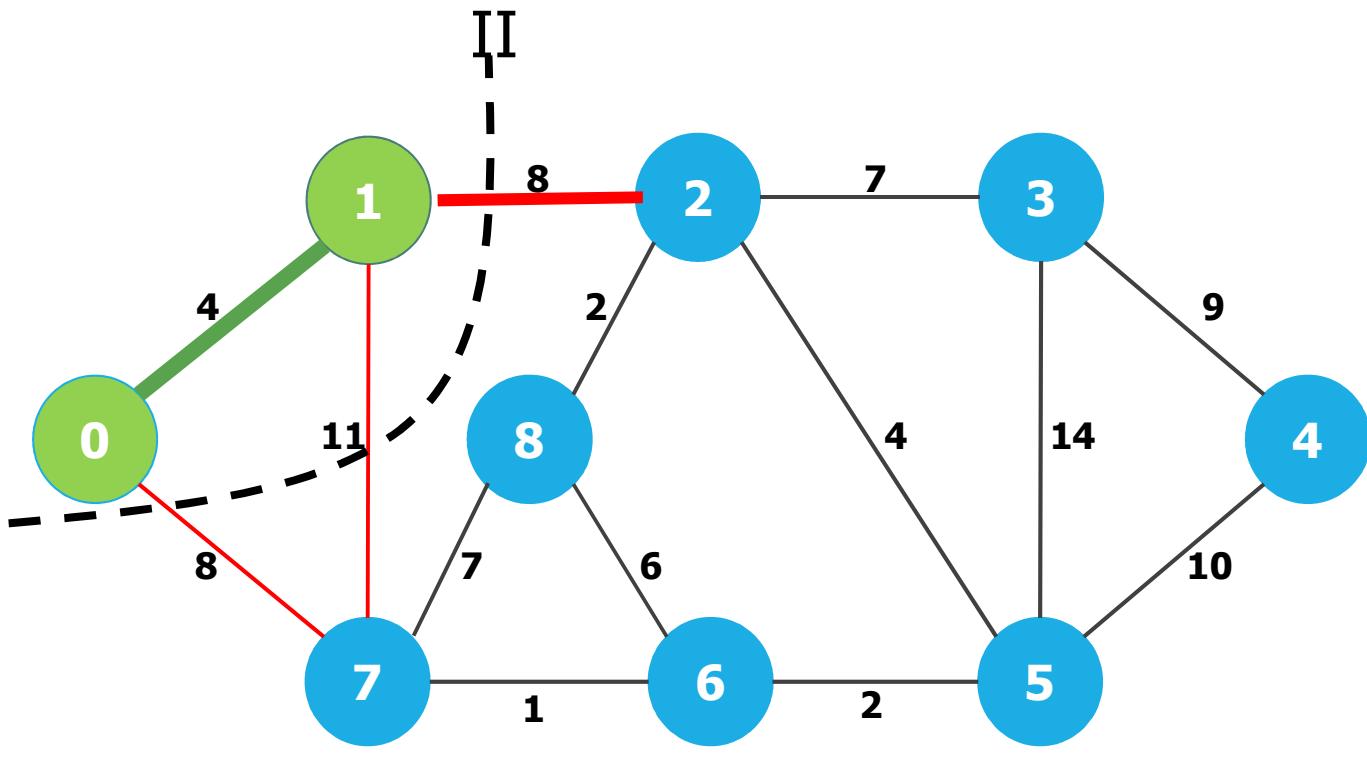
Esempio



$$S = \emptyset$$

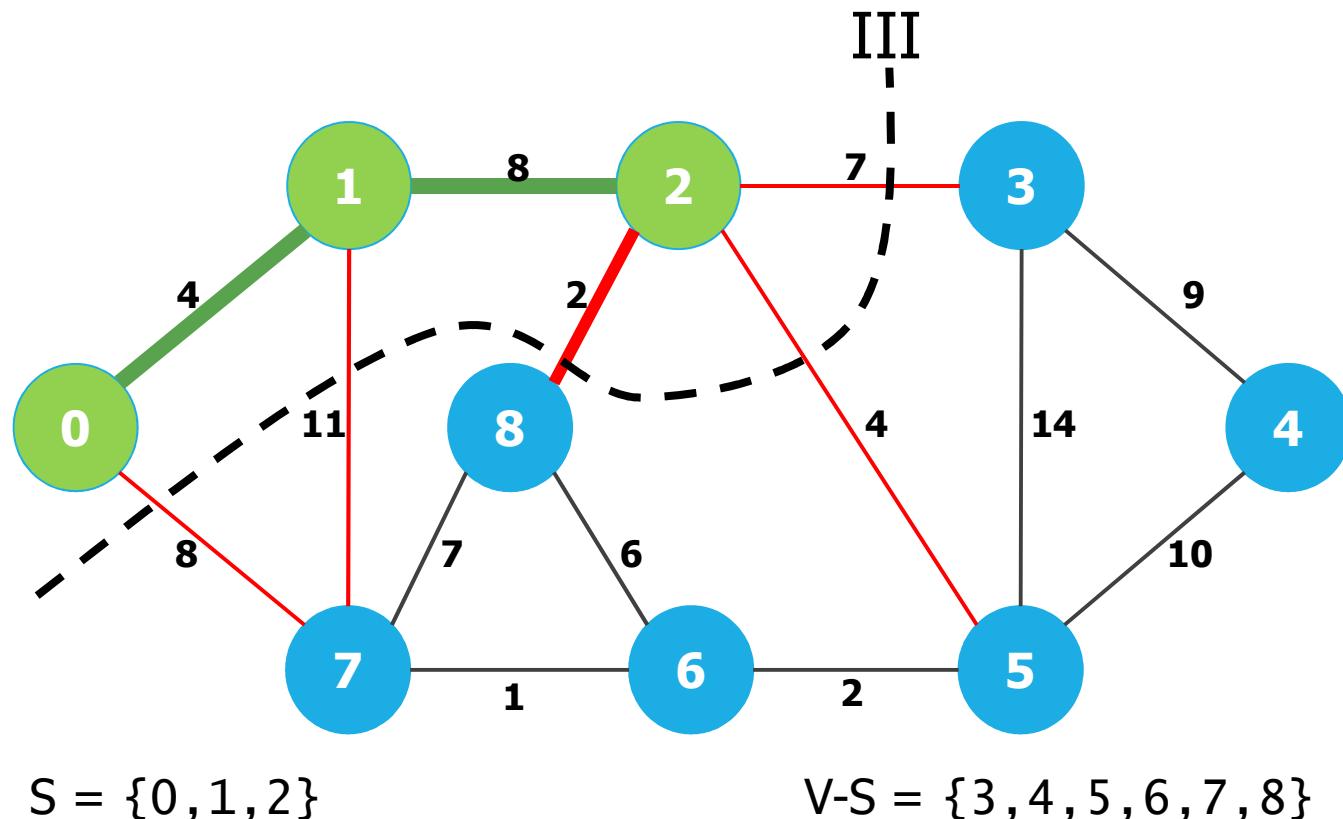
$$V-S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$$

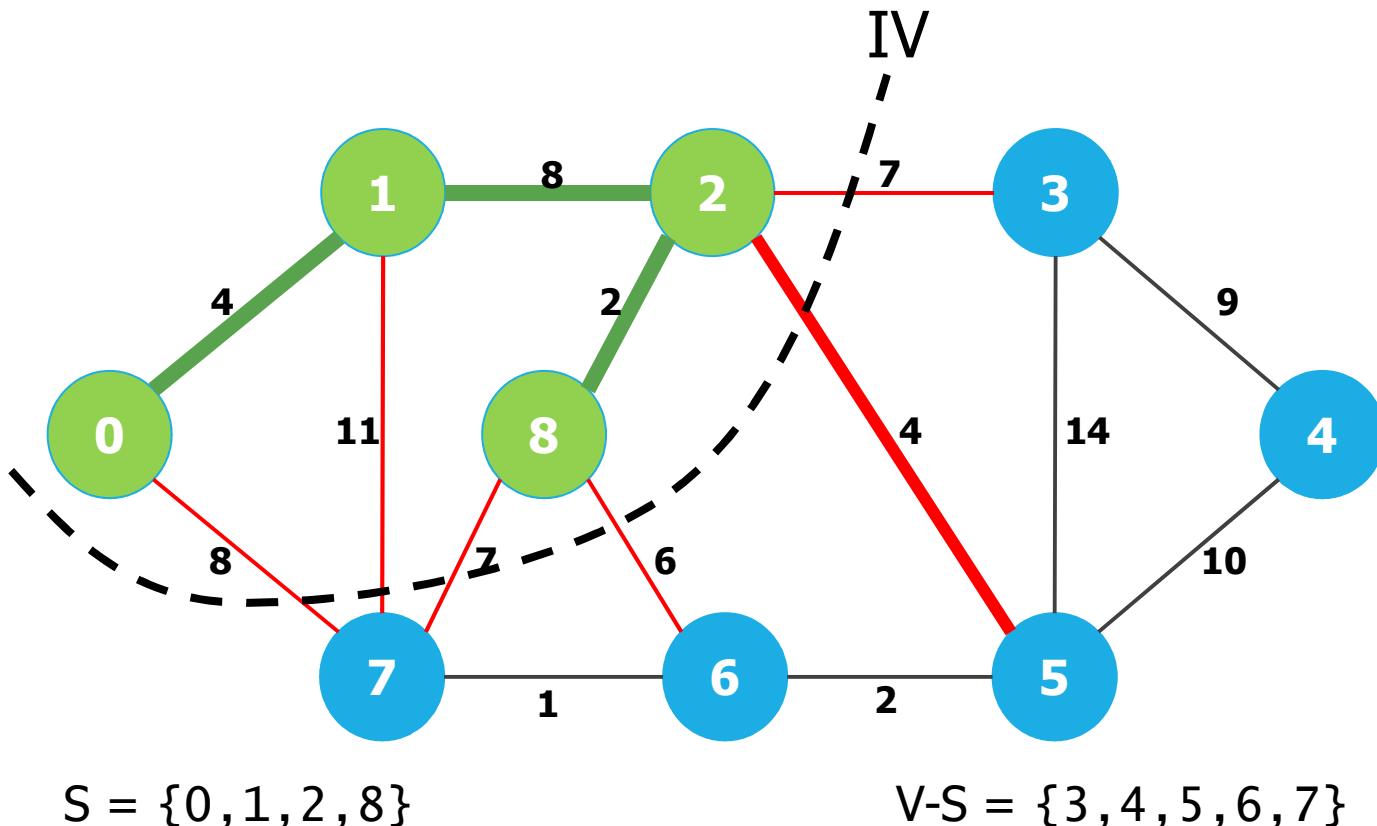


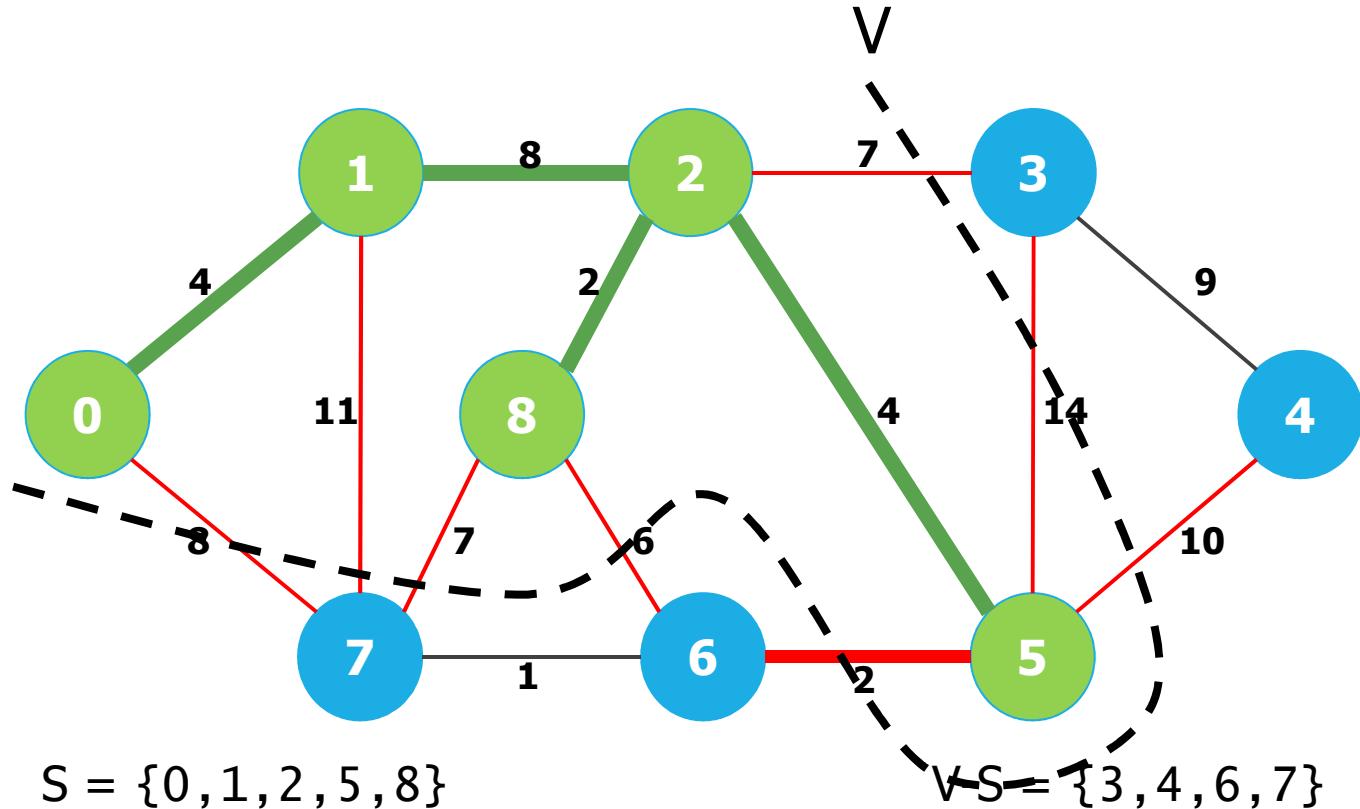


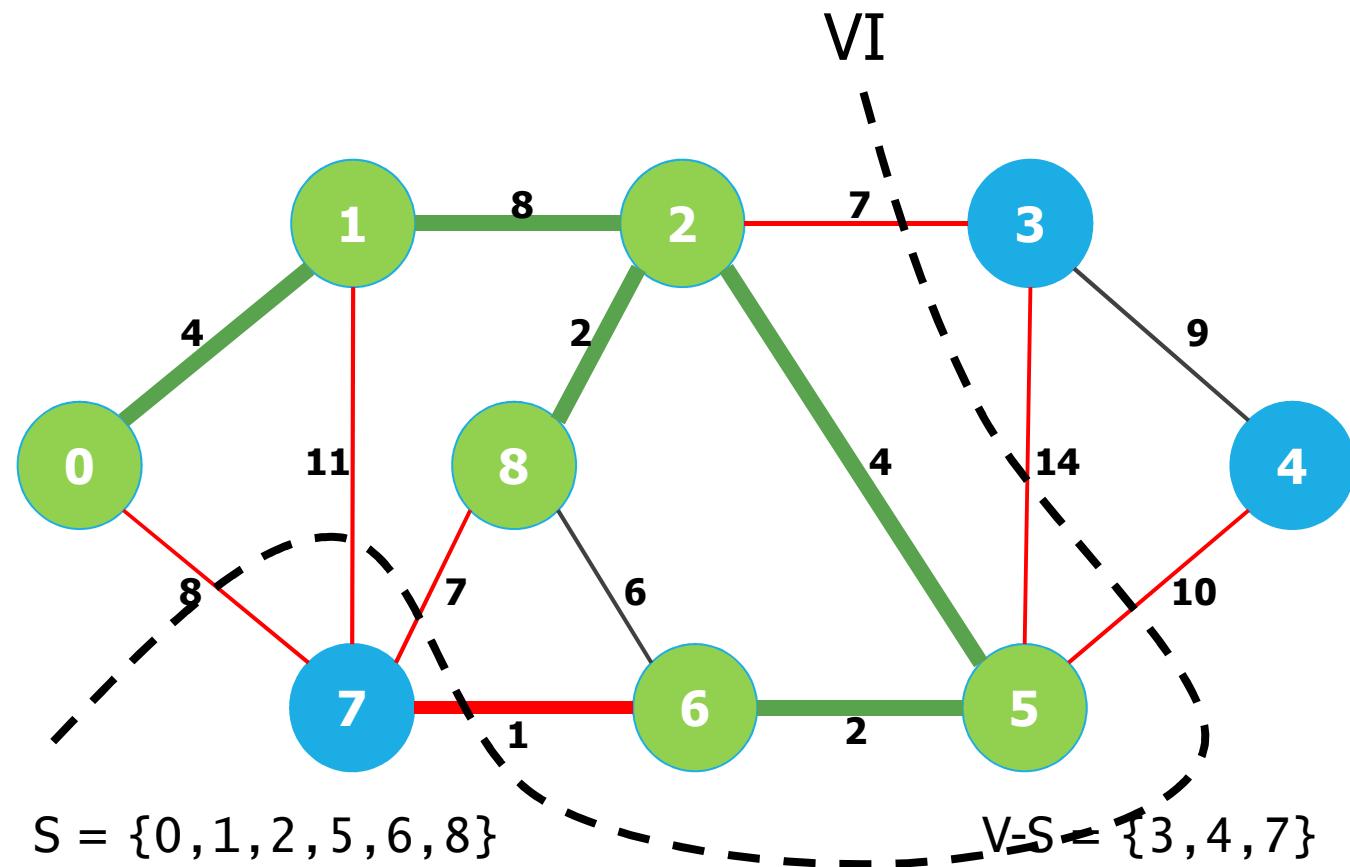
$$S = \{0, 1\}$$

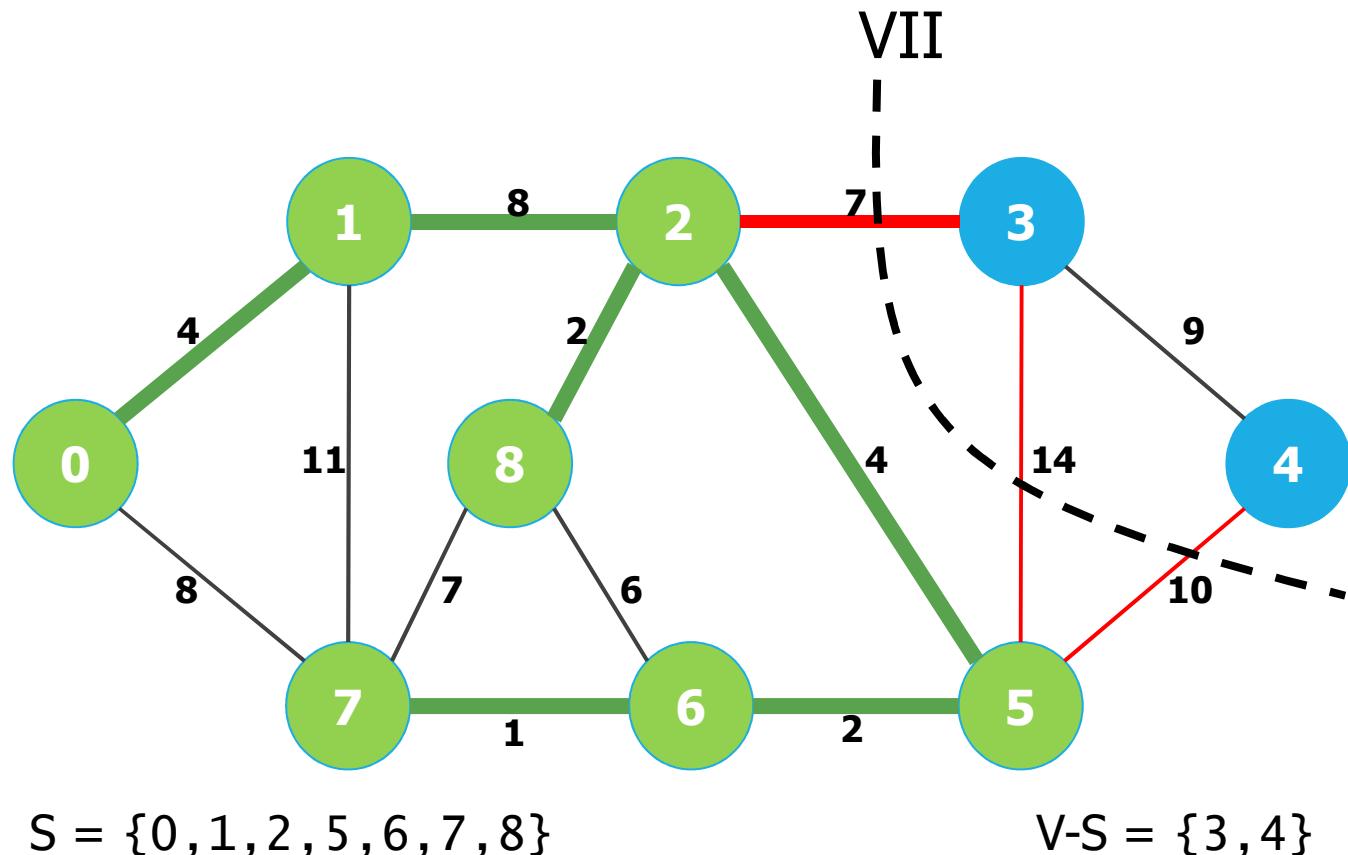
$$V-S = \{2, 3, 4, 5, 6, 7, 8\}$$

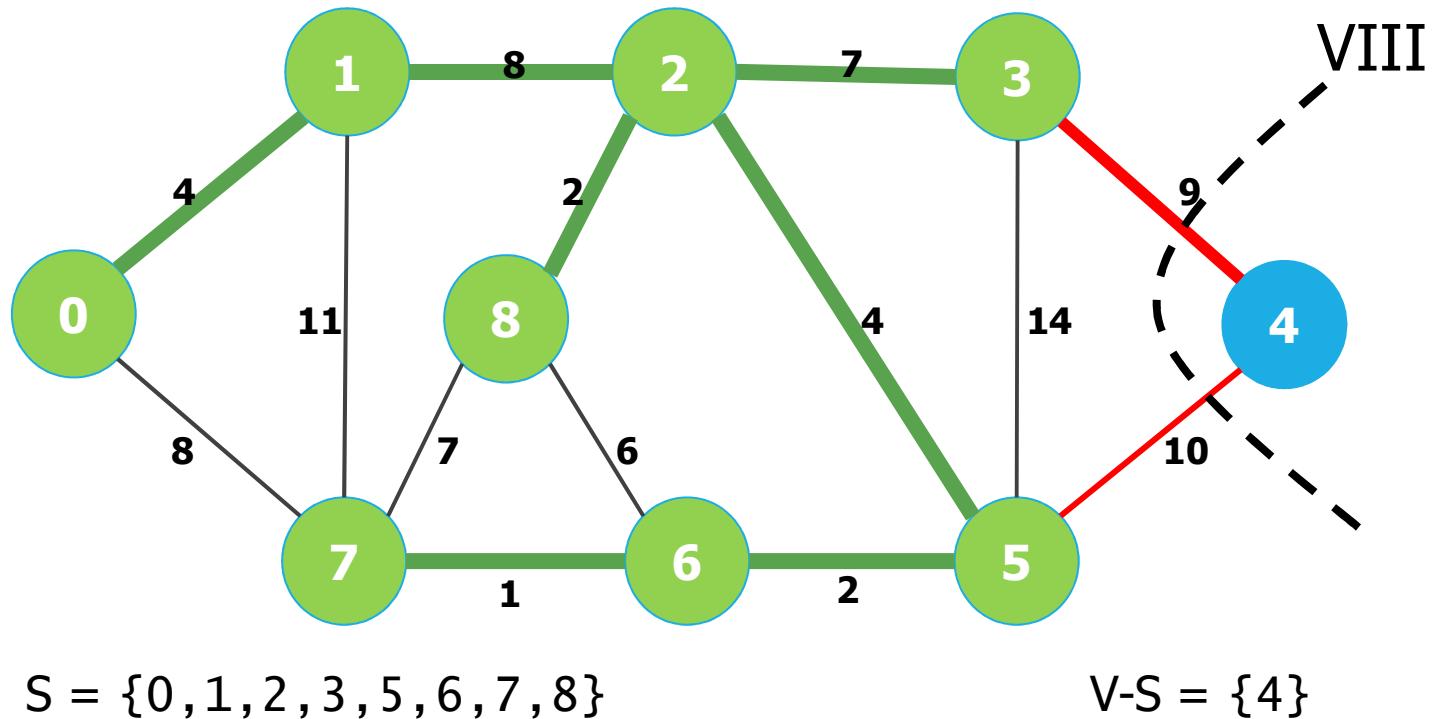




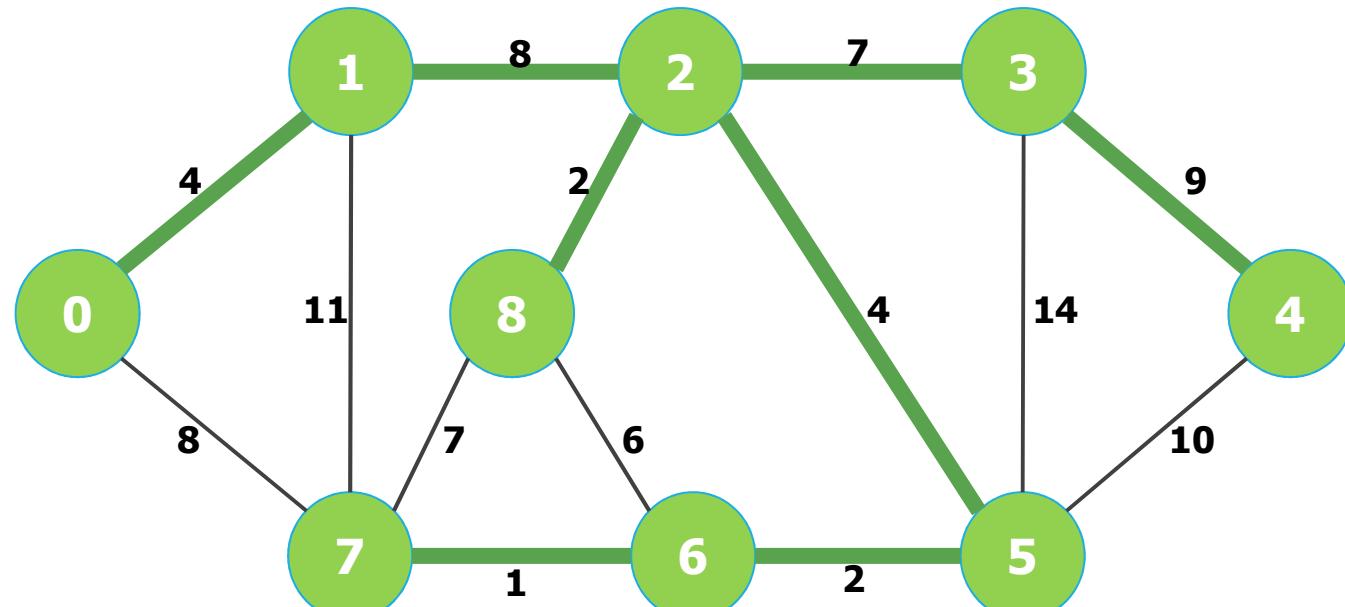








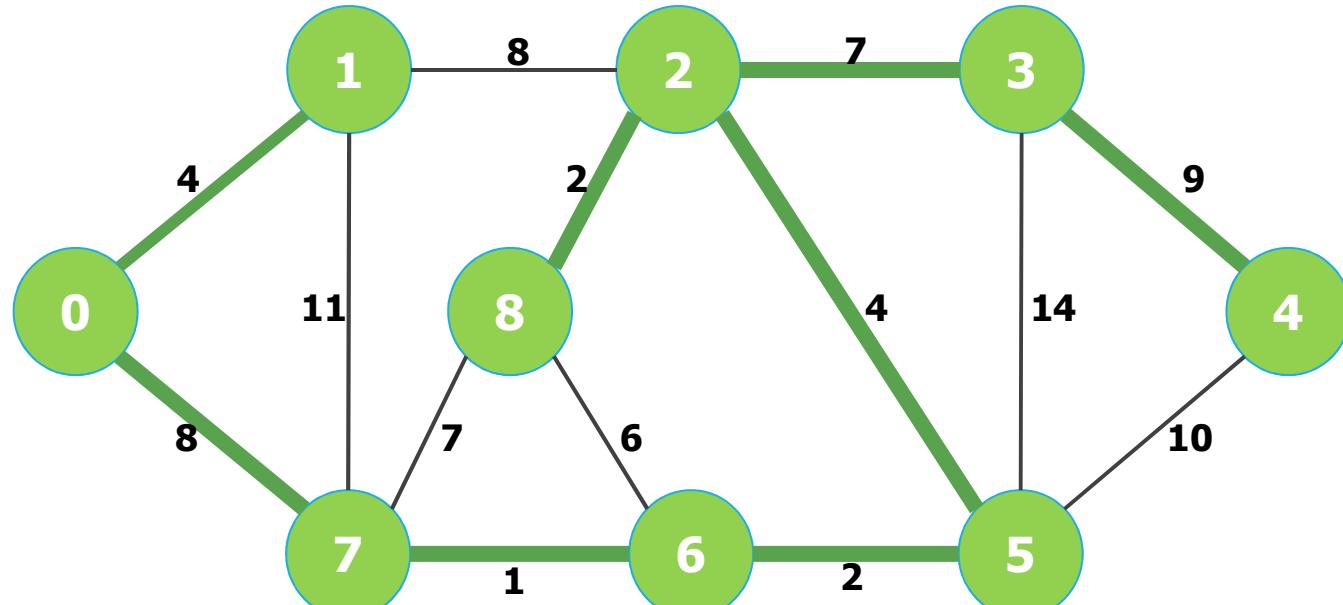
Somma minima dei pesi degli archi della soluzione: 37



$$S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$$

$$V-S = \emptyset$$

Soluzione equivalente



$$S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$$

$$V-S = \emptyset$$

Algoritmo migliorato

- L'approccio incrementale consiste nell'aggiungere ad ogni passo un vertice v a S
- ciò che interessa è la distanza minima da ogni vertice ancora in $V-S$ ai vertici già in S
- quando si aggiunge il vertice v a S , ogni vertice w in $V-S$ può avvicinarsi ai vertici già in S
- non serve memorizzare la distanza tra w e tutti i vertici in S , basta quella minima e verificare se l'aggiunta di v a S la diminuisce, nel qual caso la si aggiorna.

Strutture dati

- Grafo rappresentato come matrice delle adiacenze dove l'assenza di un arco si indica con `maxWT` anziché 0
- vettore `st` di $G \rightarrow V$ elementi per registrare per ogni vertice che appartiene ad S il padre
- vettore fringe (frangia) `fr` di $G \rightarrow V$ elementi per registrare per ogni vertice di $V - S$ quale è il vertice di S più vicino. È dichiarato static in `Graph.c`

- vettore wt di $G \rightarrow V+1$ elementi per registrare:
 - per vertici di S il peso dell'arco al padre
 - per vertici di $V-S$ il peso dell'arco verso il vertice di S più vicino
 - si considera un elemento fittizio con arco di peso maxWT
 - il vettore è inizializzato con maxWT
- variabile min per il vertice in $V-S$ più vicino a vertici di S .

Azioni:

- ciclo esterno sui vertici prendendo di volta in volta quello a minima distanza (identificato da \min) e aggiungendolo a S . Inizialmente \min è il vertice 0

```
for (min=0; min!=G->V; ) {  
    v=min; st[min]=fr[min];
```

Notare che nel ciclo `for` non si incrementa \min , \min viene assegnato opportunamente nel corpo del ciclo

- ciclo interno sui vertici w non ancora in S ($st[w]==-1$):
 - se l'arco (v,w) migliora la stima (`if (G->madj[v][w]<wt[w])`)
 - la si aggiorna ($wt[w] = G->madj[v][w]$)
 - e si indica che il vertice in S più vicino a w è v ($fr[w] = v$)
 - se w è diventato il vertice più vicino a S (`if (wt[w]<wt[min])`), si aggiorna \min ($\min=w$).

wrapper

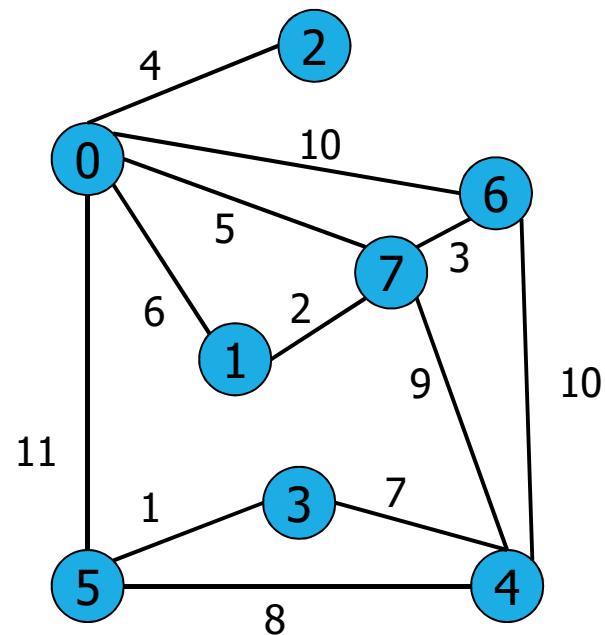
```
void GRAPHmstP(Graph G) {  
    int v, *st, *wt, weight = 0;  
    st = malloc(G->V*sizeof(int));  
    wt = malloc((G->V+1)*sizeof(int));  
  
    mstv(G, st, wt);  
  
    printf("\nEdges in the MST: \n");  
    for (v=0; v < G->V; v++) {  
        if (st[v] != v) {  
            printf("(%s-%s)\n", STsearchByIndex(G->tab, st[v]),  
                   STsearchByIndex(G->tab, v));  
            weight += wt[v];  
        }  
    }  
    printf("\nminimum weight: %d\n", weight);  
}
```

```

void mstv(Graph G, int *st, int *wt) {
    int v, w, min, *fr = malloc(G->V*sizeof(int));
    for (v=0; v < G->V; v++) {
        st[v] = -1; fr[v] = v; wt[v] = maxWT;
    }
    st[0] = 0; wt[0] = 0; wt[G->V] = maxWT;
    for (min = 0; min != G->V; ) {
        v = min; st[min] = fr[min];
        for (w = 0, min = G->V; w < G->V; w++)
            if (st[w] == -1) {
                if (G->madj[v][w] < wt[w]) {
                    wt[w] = G->madj[v][w]; fr[w] = v;
                }
                if (wt[w] < wt[min]) min = w;
            }
    }
}

```

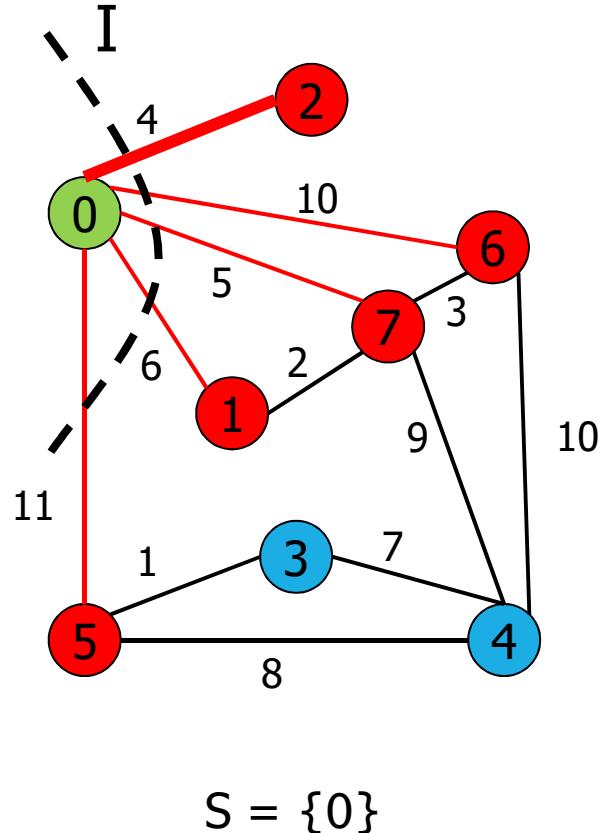
Esempio



$$S = \emptyset$$

	0	1	2	3	4	5	6	7	8
st	-1	-1	-1	-1	-1	-1	-1	-1	
wt	∞								
fr	0	1	2	3	4	5	6	7	

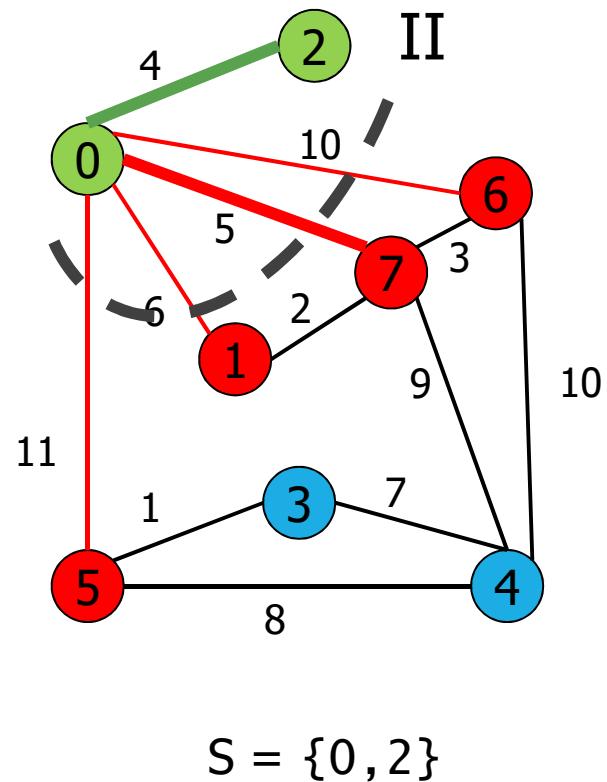
$$V-S = \{0, 1, 2, 3, 4, 5, 6, 7\}$$



	0	1	2	3	4	5	6	7	8
st	0	-1	-1	-1	-1	-1	-1	-1	
wt	0	6	4	∞	∞	11	10	5	∞
fr	0	0	0	3	4	0	0	0	

- $\min = 0$, aggiorno $st[0]$ e $wt[0]$
- fringe contiene 1, 2, 5, 6, 7
- aggiorno wt e fr in base agli archi che attraversano il taglio
- il vertice 2 è quello più vicino a S , in quanto 0-2 è l'arco a peso minimo che attraversa il taglio I
- $\min = 2$

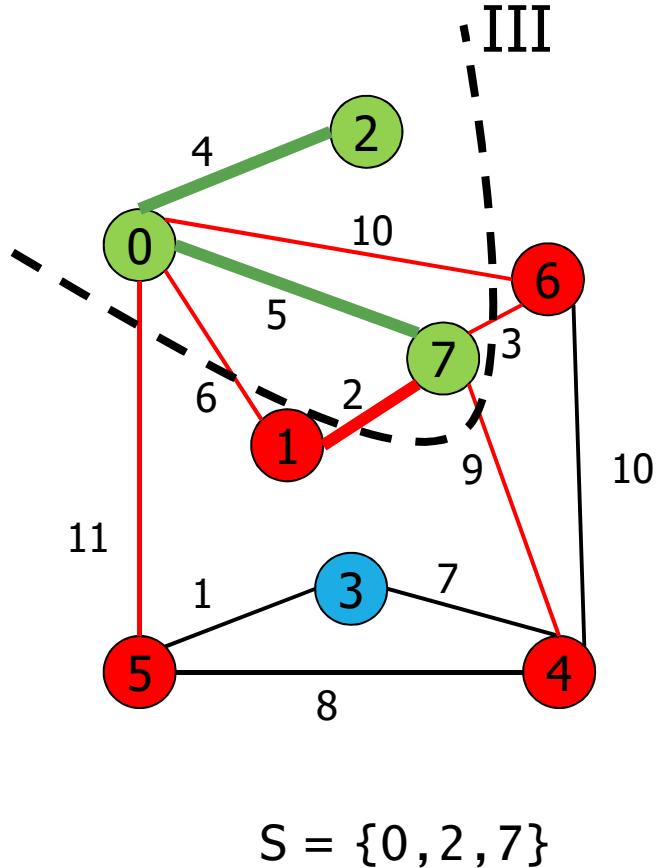
$$V-S = \{1, 2, 3, 4, 5, 6, 7\}$$



	0	1	2	3	4	5	6	7	8
st	0	-1	0	-1	-1	-1	-1	-1	
wt	0	6	4	∞	∞	11	10	5	∞
fr	0	0	0	3	4	0	0	0	

- aggiungo 2 alla soluzione e aggiorno $st[2]$
- fringe contiene 1, 5, 6, 7
- aggiorno wt e fr in base agli archi che attraversano il taglio
- il vertice 7 è quello più vicino a S , in quanto 0-7 è l'arco a peso minimo che attraversa il taglio II
- $\min = 7$

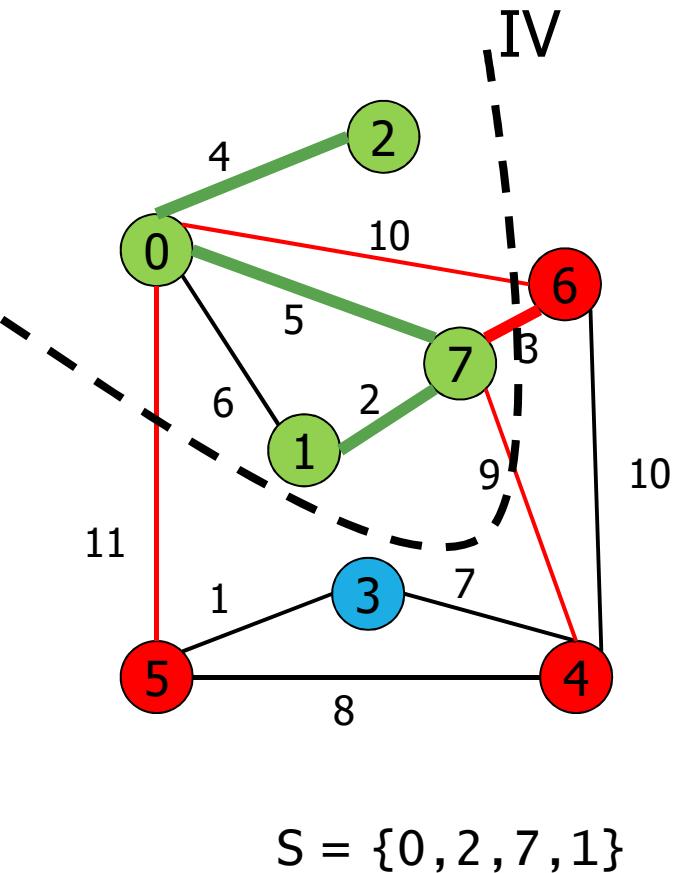
$$V-S = \{1, 3, 4, 5, 6, 7\}$$



	0	1	2	3	4	5	6	7	8
st	0	-1	0	-1	-1	-1	-1	0	
wt	0	2	4	∞	9	11	3	5	∞
fr	0	7	0	3	7	0	7	0	

- aggiungo 7 alla soluzione e aggiorno $st[7]$
- fringe contiene 1, 4, 5, 6
- aggiorno wt e fr in base agli archi che attraversano il taglio
- il vertice 1 è quello più vicino a S , in quanto 1-7 è l'arco a peso minimo che attraversa il taglio III
- $min = 1$

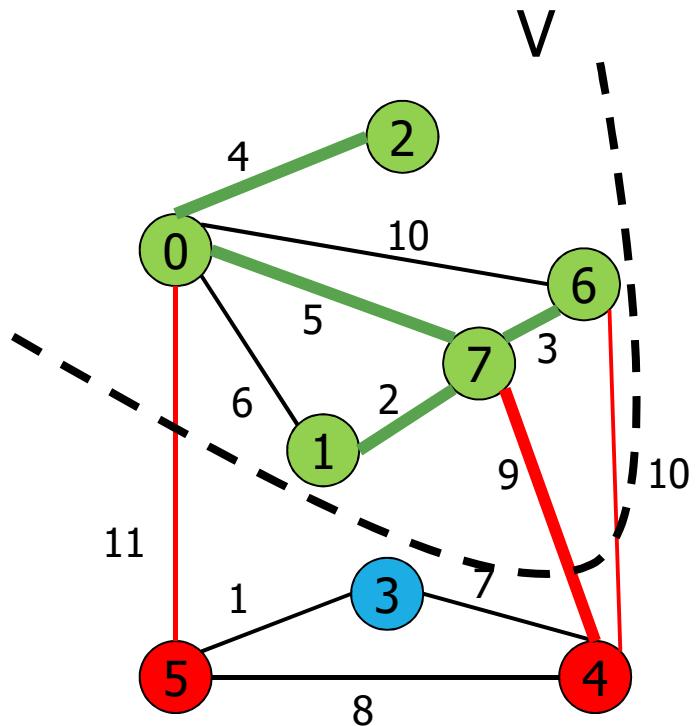
$$V-S = \{1, 3, 4, 5, 6\}$$



	0	1	2	3	4	5	6	7	8
st	0	7	0	-1	-1	-1	-1	0	
wt	0	2	4	∞	9	11	3	5	∞
fr	0	7	0	3	7	0	7	0	

- aggiungo 1 alla soluzione e aggiorno $st[1]$
- fringe contiene 4, 5, 6
- aggiorno wt e fr in base agli archi che attraversano il taglio
- il vertice 6 è quello più vicino a S , in quanto 6-7 è l'arco a peso minimo che attraversa il taglio IV
- $\min = 6$

$$V-S = \{3, 4, 5, 6\}$$

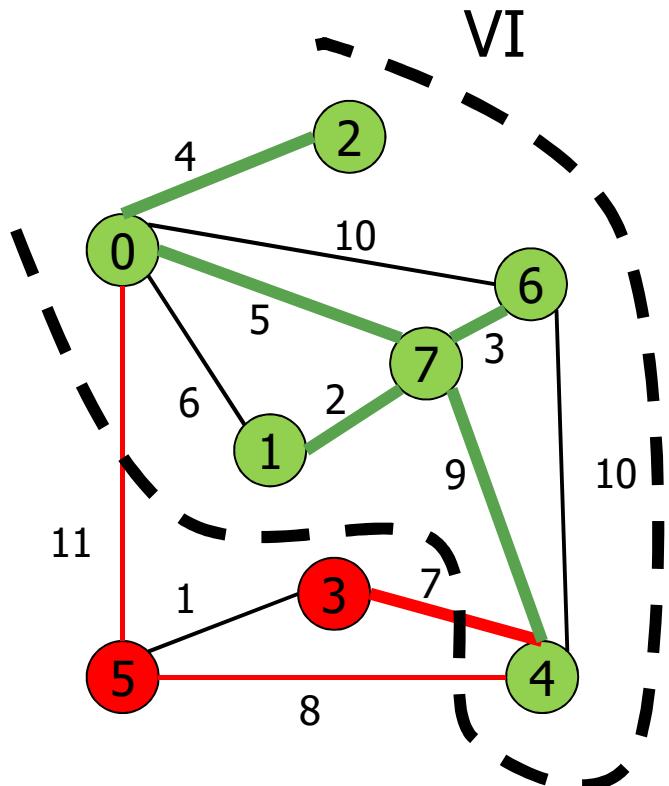


$$S = \{0, 2, 7, 1, 6\}$$

	0	1	2	3	4	5	6	7	8
st	0	7	0	-1	-1	-1	7	0	
wt	0	2	4	∞	9	11	3	5	∞
fr	0	7	0	3	7	0	7	0	

- aggiungo 6 alla soluzione e aggiorno st[6]
- fringe contiene 4, 5
- aggiorno wt e fr in base agli archi che attraversano il taglio
- il vertice 4 è quello più vicino a S, in quanto 4-7 è l'arco a peso minimo che attraversa il taglio V
- min = 4

$$V-S = \{3, 4, 5\}$$

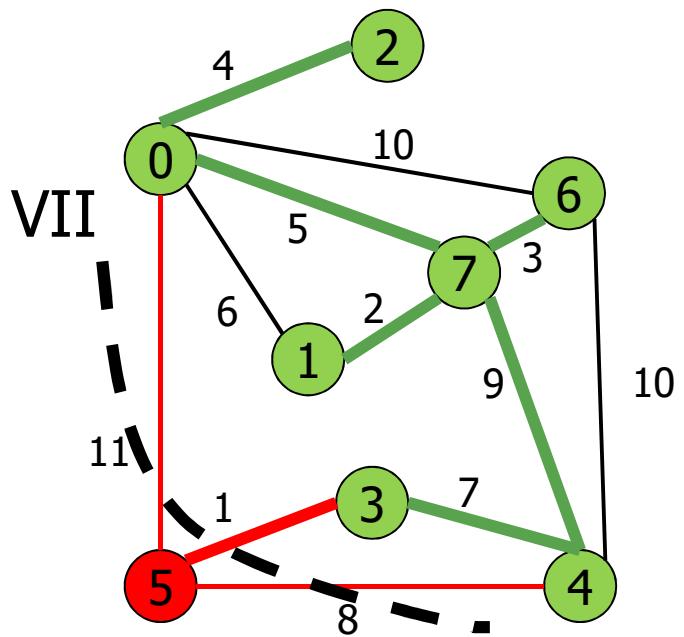


$$S = \{0, 2, 7, 1, 6, 4\}$$

	0	1	2	3	4	5	6	7	8
st	0	7	0	-1	7	-1	7	0	
wt	0	2	4	7	9	8	3	5	∞
fr	0	7	0	4	7	4	7	0	

- aggiungo 4 alla soluzione e aggiorno st[4]
- fringe contiene 3, 5
- aggiorno wt in base agli archi che attraversano il taglio
- il vertice 3 è quello più vicino a S, in quanto 3-4 è l'arco a peso minimo che attraversa il taglio VI
- min = 3

$$V-S = \{3, 5\}$$

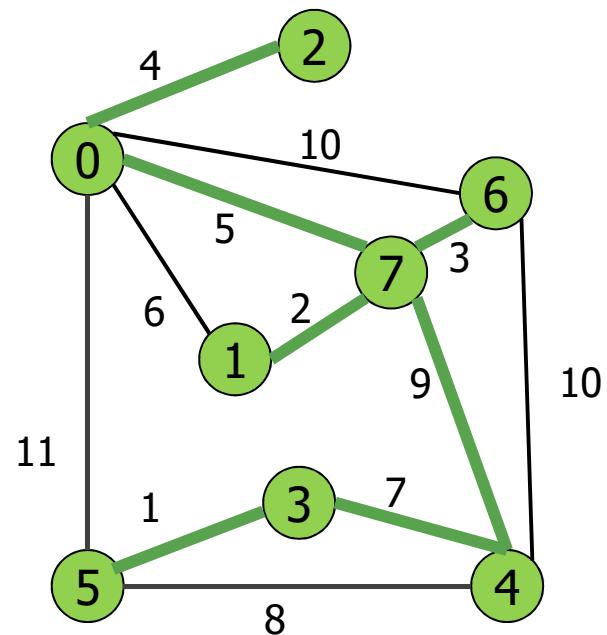


$$S = \{0, 2, 7, 1, 6, 4, 3\}$$

$$V-S = \{5\}$$

	0	1	2	3	4	5	6	7	8
st	0	7	0	4	7	-1	7	0	
wt	0	2	4	7	9	1	3	5	∞
fr	0	7	0	4	7	3	7	0	

- aggiungo 3 alla soluzione e aggiorno st[3]
- fringe contiene 5
- aggiorno wt in base agli archi che attraversano il taglio
- il vertice 5 è quello più vicino a S, in quanto 3-5 è l'arco a peso minimo che attraversa il taglio VII
- min = 5



	0	1	2	3	4	5	6	7	8
st	0	7	0	4	7	3	7	0	
wt	0	2	4	7	9	1	3	5	∞
fr	0	7	0	4	7	3	7	0	

- aggiungo 5 alla soluzione e aggiorno st[5]
- terminazione
- somma minima dei pesi 31.

$$S = \{0, 2, 7, 1, 6, 4, 3, 5\}$$

$$V-S = \emptyset$$

Complessità

Per grafi densi: $T(n) = O(|V|^2)$

Possibili miglioramenti per grafi sparsi: usare una coda a priorità per gestire la fringe.

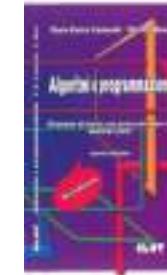
Con coda a priorità implementata con heap $T(n) = O(|E|\log|V|)$.

Riferimenti

- Rappresentazione:
 - Sedgewick Part 5 20.1
- Principi:
 - Sedgewick Part 5 20.2
 - Cormen 23.1
- Algoritmo di Kruskal
 - Sedgewick Part 5 20.4
 - Cormen 23.2
- Algoritmo di Prim
 - Sedgewick Part 5 20.3
 - Cormen 23.2

Esercizi di teoria

- 11. Alberi ricoprenti minimi
 - 11.2 Algoritmi di Kruskal e Prim



I cammini minimi

Paolo Camurati



I cammini minimi

$G = (V, E)$ grafo orientato, pesato ($w: E \rightarrow \mathbb{R}$).

Definizioni:

peso $w(p)$ di un cammino p :

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

peso $\delta(s, v)$ di un cammino minimo da s a v :

$$\delta(s, v) = \begin{cases} \min\{w(p): \text{se } \exists s \rightarrow_p v\} \\ \infty \text{ altrimenti} \end{cases}$$

Cammino minimo da s a v :

qualsiasi cammino p con $w(p) = \delta(s, v)$

Problemi classici

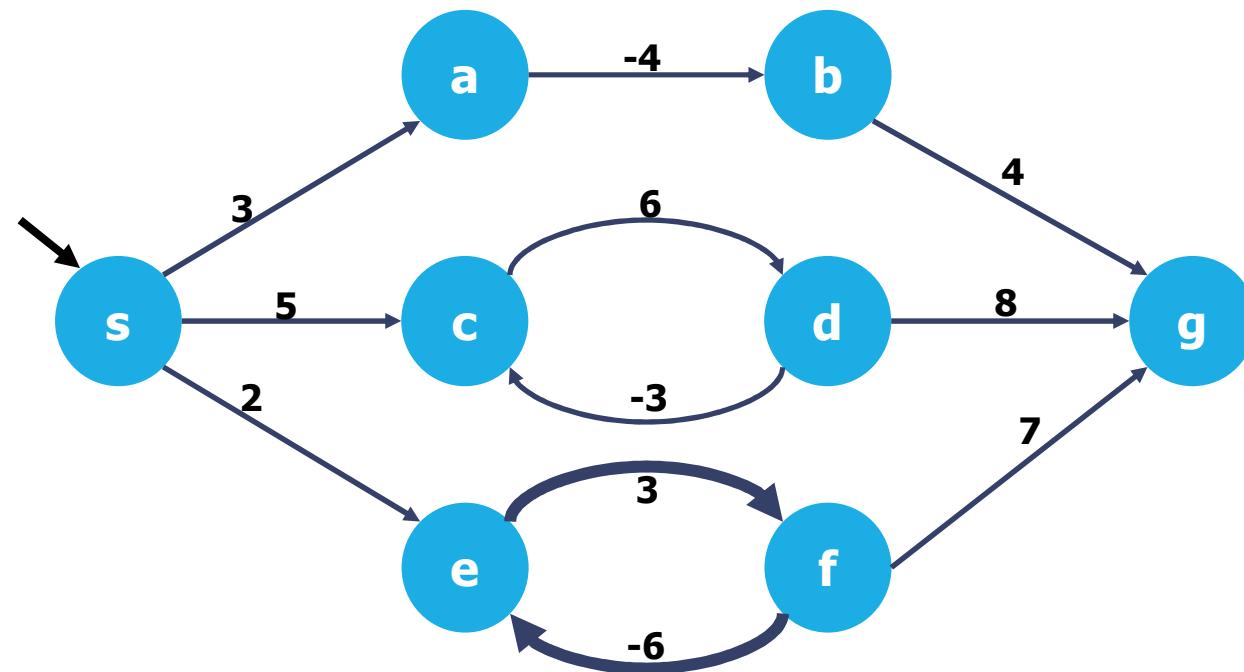
Cammini minimi:

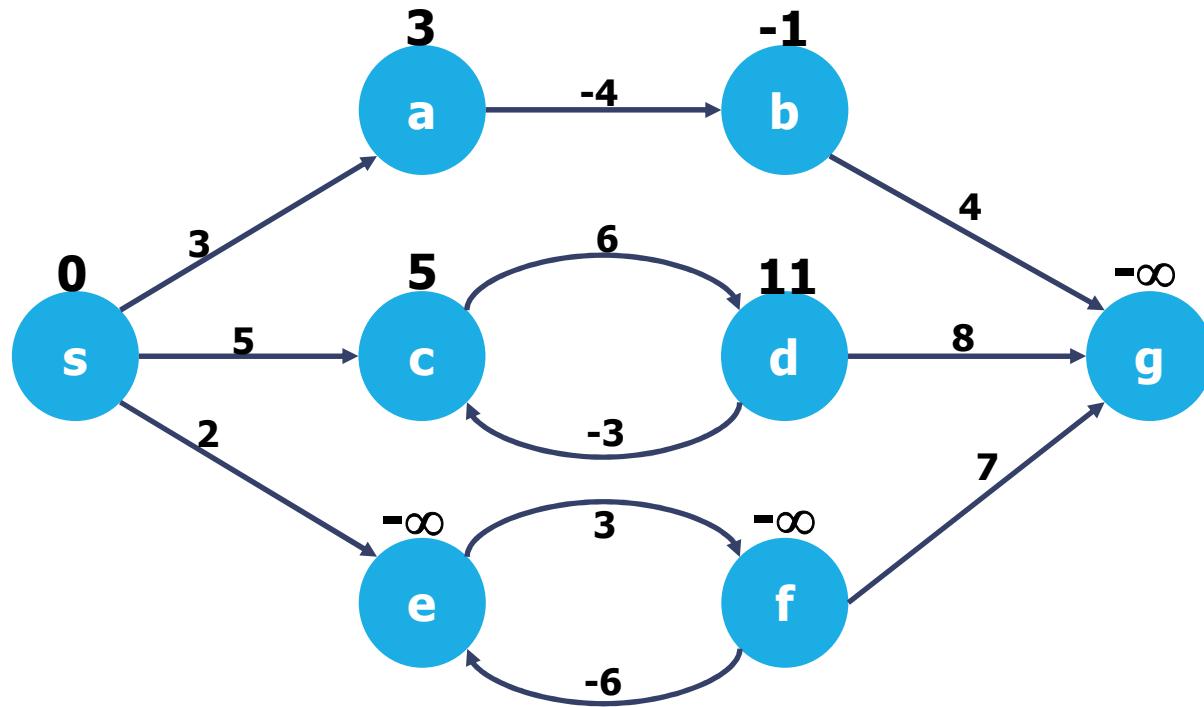
- da sorgente singola: cammino minimo e suo peso da s a ogni altro vertice v
 - algoritmo di Dijkstra
 - algoritmo di Bellman-Ford
- con destinazione singola
- tra una coppia di vertici
- tra tutte le coppie di vertici.

Archi con pesi negativi

- $\exists (u,v) \in E$ per cui $w(u,v) < 0$ ma \nexists ciclo a peso < 0:
 - algoritmo di Djikstra: soluzione ottima non garantita
 - algoritmo di Bellman-Ford: soluzione ottima garantita
- \exists ciclo a peso < 0: problema privo di significato, \nexists soluzione:
 - algoritmo di Djikstra: risultato senza significato
 - algoritmo di Bellman-Ford: rileva ciclo <0.

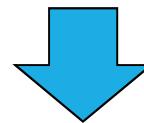
Esempio





Cammini minimi e cicli

- I cammini minimi non possono contenere cicli a peso negativo in quanto il problema perderebbe di significato
- I cammini minimi non possono contenere cicli a peso positivo, in quanto non sarebbero minimi (eliminando il ciclo si avrebbe peso inferiore)



- I cammini minimi sono semplici
- I cammini minimi constano di al più $|V|-1$ archi

Approccio brute-force

In assenza di cicli a peso negativo:

- enumerare i sottoinsiemi di archi di cardinalità tra 1 e $|V|-1$
- l'ordine conta, quindi powerset con disposizioni semplici
- accettabilità: il sottoinsieme di archi forma un cammino
- ottimalità: selezionare quello a peso minimo.

Costo esponenziale.

Sottostruttura ottima di un cammino minimo

Lemma: un sottocammino di un cammino minimo è un cammino minimo.

$G = (V, E)$: grafo orientato, pesato $w: E \rightarrow \mathbb{R}$.

$p = \langle v_1, v_2, \dots, v_k \rangle$: un cammino minimo da v_1 a v_k .

$\forall i, j \ 1 \leq i \leq j \leq k, p_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ (sottocammino di p da v_i a v_j)
 p_{ij} è un cammino minimo da v_i a v_j .

Dimostrazione (per assurdo):

Scomponiamo p in p_{1i} , p_{ij} e p_{jk} con $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$

Se p_{ij} non fosse minimo, esisterebbe un p'_{ij} minimo con peso $w'(p_{ij}) < w(p_{ij})$.

Sostituendo p'_{ij} a p_{ij} , $w(p)$ non sarebbe minimo, contraddicendo l'ipotesi.

Rappresentazione dei cammini minimi

1 - **Vettore** dei predecessori $st[v]$:

$$\forall v \in V \quad st[v] = \begin{cases} \text{parent}(v) \text{ se } \exists \\ -1 \text{ altrimenti} \end{cases}$$

2- **Sottografo** dei predecessori:

$G_\pi = (V_\pi, E_\pi)$, dove

- $V_\pi = \{v \in V : st[v] \neq -1\} \cup \{s\}$
- $E_\pi = \{(st[v], v) \in E : v \in V_\pi - \{s\}\}$

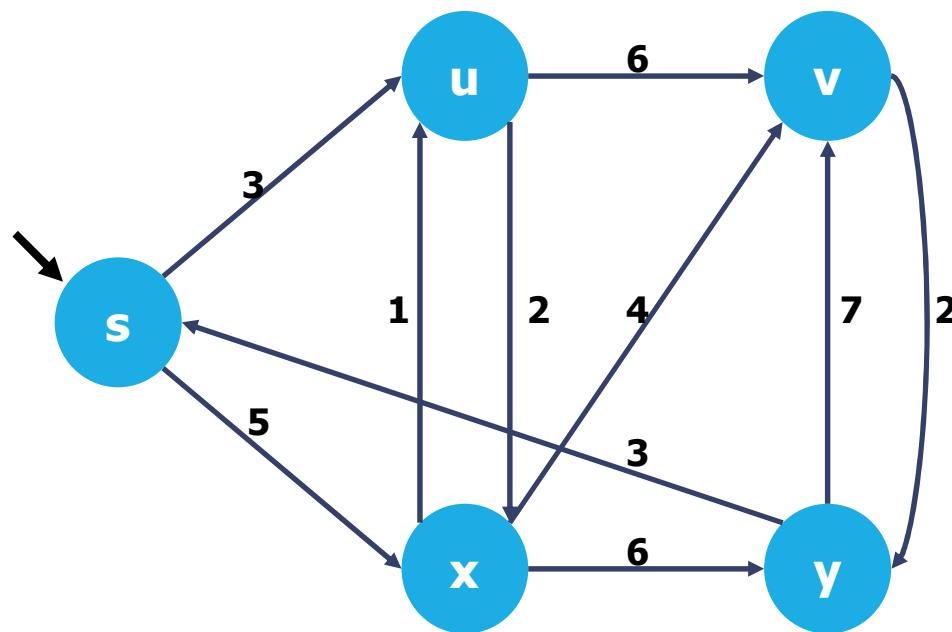
3- **Albero** dei cammini minimi:

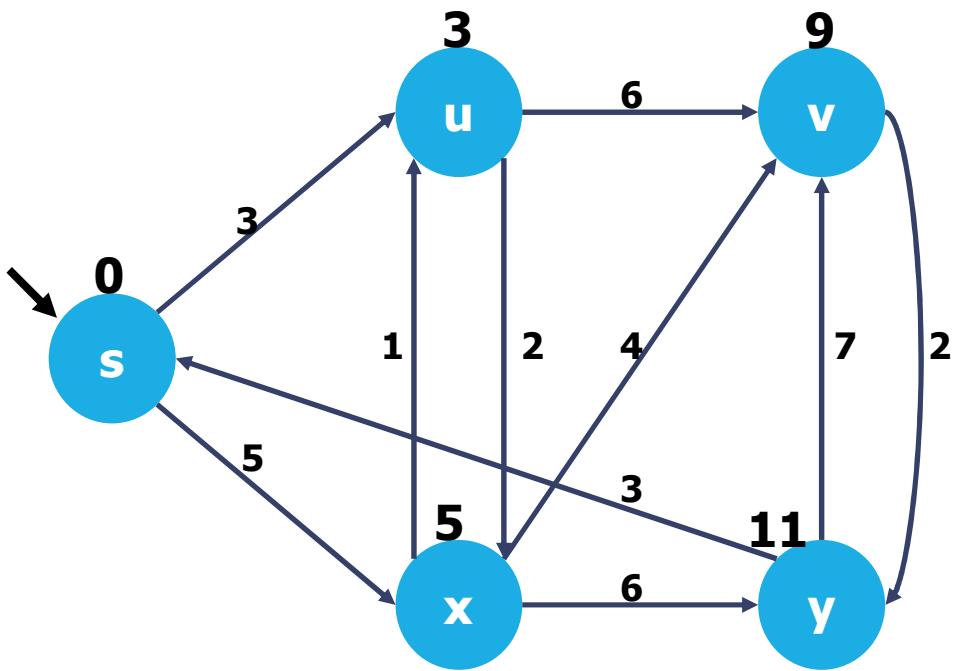
$G' = (V', E')$ dove $V' \subseteq V$ && $E' \subseteq E$

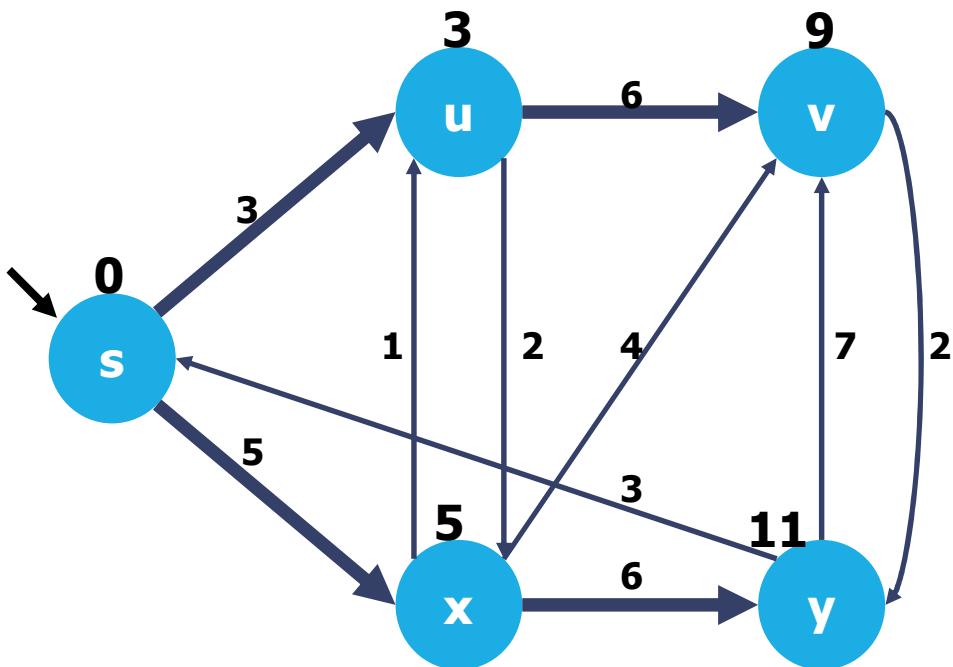
- V' : insieme dei vertici raggiungibili da s
- s radice dell'albero
- $\forall v \in V'$ l'unico cammino semplice da s a v in G' è un cammino minimo da s a v in G .

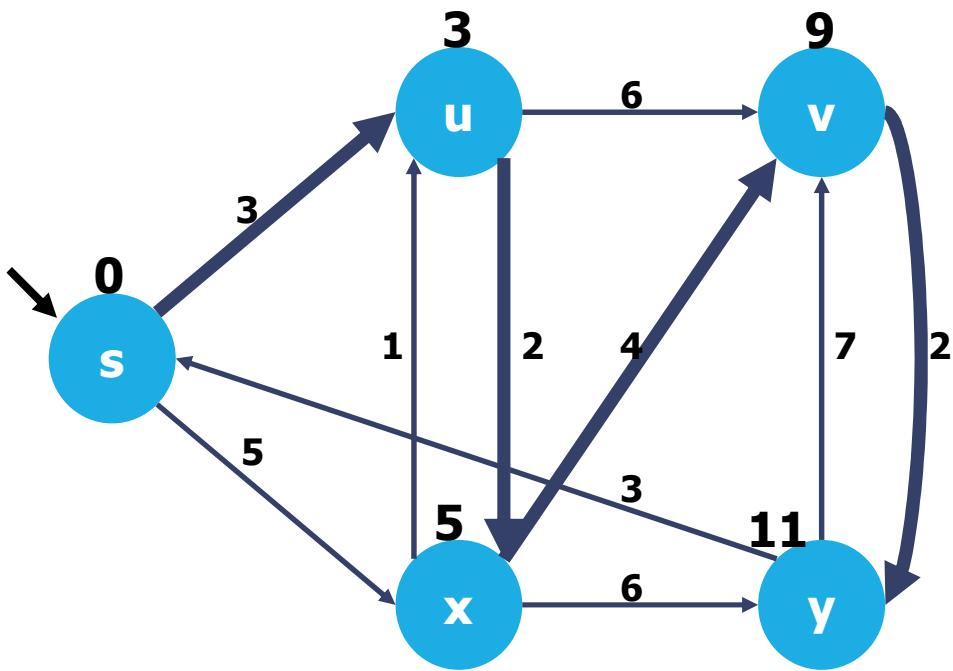
Nei grafi non pesati: si ottiene con una visita in ampiezza.

Esempio







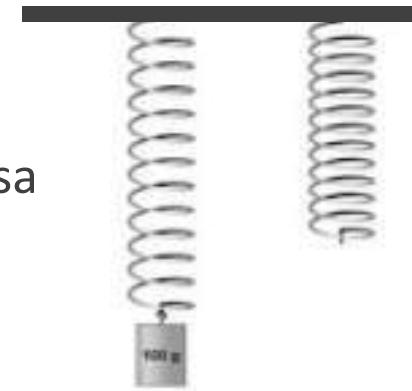


Rilassamento (relaxation)

In generale: lasciare che una soluzione violi un vincolo temporaneamente e poi eliminare la violazione.

Analogia: molla elicoidale di tensione

sovrafflotta del cammino minimo sta a molla tesa
come
cammino minimo sta a molla a riposo



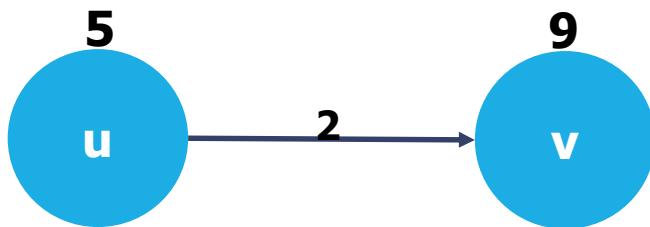
- $d[v]$: stima (limite superiore) del peso del cammino minimo da s a v
inizialmente:

$$\forall v \in V \quad d[v] = \maxWT, \text{st}[v] = -1; \\ d[s] = 0; \text{st}[s] = 0;$$

- rilassare: $d[v]$ e $\text{st}[v]$ verificando se conviene il cammino da s a u e l'arco $e = (u,v)$, dove $w(u,v)$ è il peso dell'arco:

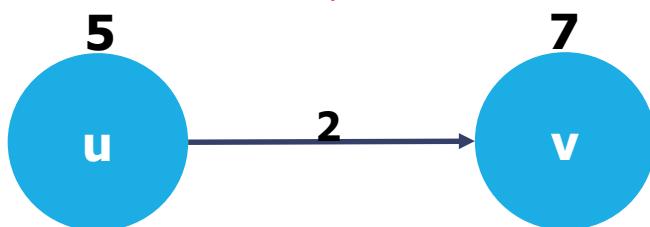
```
if (d[v]>d[u]+w(u,v)) {
    d[v] = d[u]+w(u,v);
    st[v] = u;
}
```

Esempio

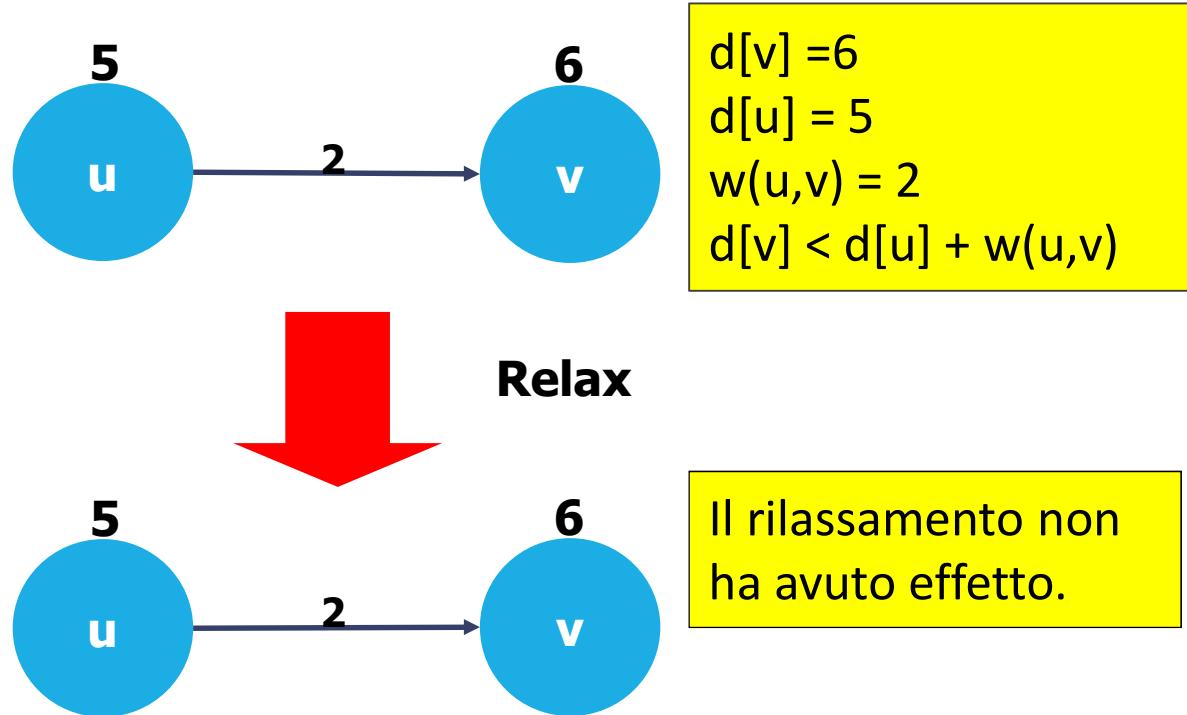


$d[v] = 9$
 $d[u] = 5$
 $w(u,v) = 2$
 $d[v] > d[u] + w(u,v)$

Relax



$d[v] = 7$
 $st[v] = u$
cammino minimo da s a v =
cammino minimo da s a u +
arco (u,v)



Proprietà

Lemma (diseguaglianza triangolare):

$G=(V,E)$: grafo orientato, pesato $w: E \rightarrow \mathbb{R}$ con sorgente s .

$$\forall (u,v) \in E$$

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

Un cammino minimo da s a v non può avere peso maggiore del cammino formato da un cammino minimo da s a u e da un arco (u, v) .

Lemma (proprietà del limite superiore):

$G=(V,E)$: grafo orientato, pesato $w: E \rightarrow \mathbb{R}$ con sorgente s e stime per i vertici inizializzate.

$\forall v \in V$

$$d(v) \geq \delta(s,v)$$

Una volta che il limite superiore $d(v)$ assume il valore $\delta(s,v)$ esso non cambia più.

Corollario: (proprietà dell'assenza di cammino):

$G=(V,E)$: grafo orientato, pesato $w: E \rightarrow \mathbb{R}$ con stime per i vertici inizializzate.

Se non esiste un cammino tra s e v , allora si ha sempre

$$d(v) = \delta(s,v) = \infty$$

Lemma:

$G = (V, E)$: grafo orientato, pesato $w: E \rightarrow \mathbb{R}$.
 $e = (u, v) \in E$

Dopo il rilassamento di $e = (u, v)$ si ha che

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

A seguito del rilassamento $d[v]$ non può essere aumentato, ma

- o è rimato invariato (rilassamento senza effetto)
- o è diminuito per effetto del rilassamento.

Lemma (proprietà della convergenza):

$G=(V,E)$: grafo orientato, pesato $w: E \rightarrow \mathbb{R}$ con sorgente s e stime per i vertici inizializzate.

Sia il cammino minimo da s a v composto da

- cammino da s a u
- arco $e = (u,v)$

A seguito dell'applicazione del rilassamento su $e = (u,v)$

se prima del rilassamento $d[u] = \delta(s,u)$
dopo il rilassamento $d[v] = \delta(s,v)$.

Lemma (proprietà del rilassamento):

$G=(V,E)$: grafo orientato, pesato $w: E \rightarrow \mathbb{R}$ con sorgente s
e stime per i vertici inizializzate

Se $p = \langle v_1, v_2, \dots, v_k \rangle$ è un cammino minimo da v_1 a v_k

dopo tutti i passi di rilassamento sugli archi

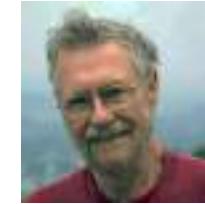
- $d(v_k) = \delta(s, v_k)$
- e $d(v_k)$ non cambia più.

Applicazione

Rilassamento:

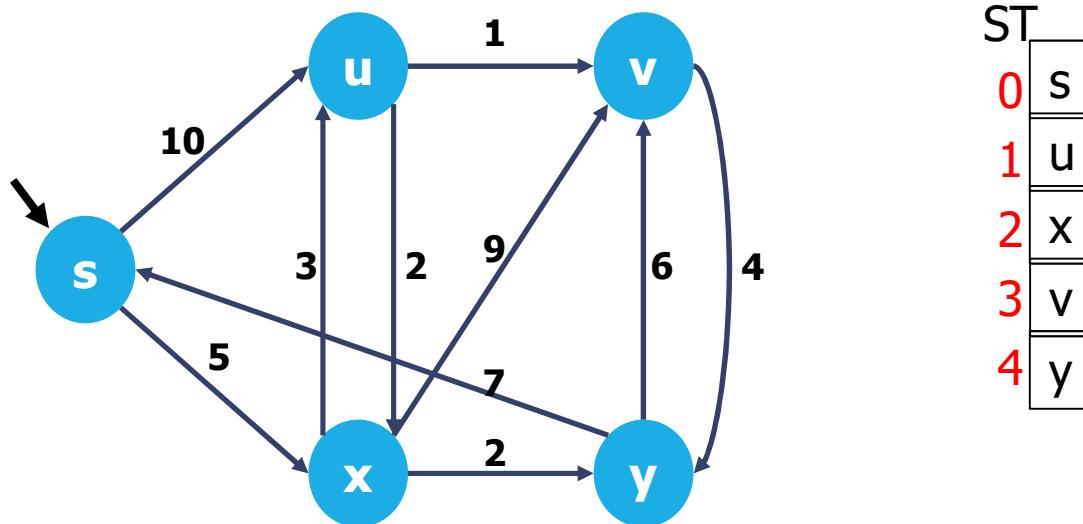
- applicato 1 volta ad ogni arco (Dijkstra) o più volte (Bellman-Ford)
- ordine con cui si rilassano gli archi.

Algoritmo di Dijkstra



- Ipotesi: \nexists archi a peso < 0
- Strategia: **greedy**
- S : insieme dei vertici il cui peso sul cammino minimo da s è già stato determinato
- $V-S$: coda a priorità PQ dei vertici ancora da stimare.
Termina per PQ vuota:
 - estraе u da $V-S$ ($d[u]$ minimo)
 - inserisce u in S
 - rilassa tutti gli archi uscenti da u.

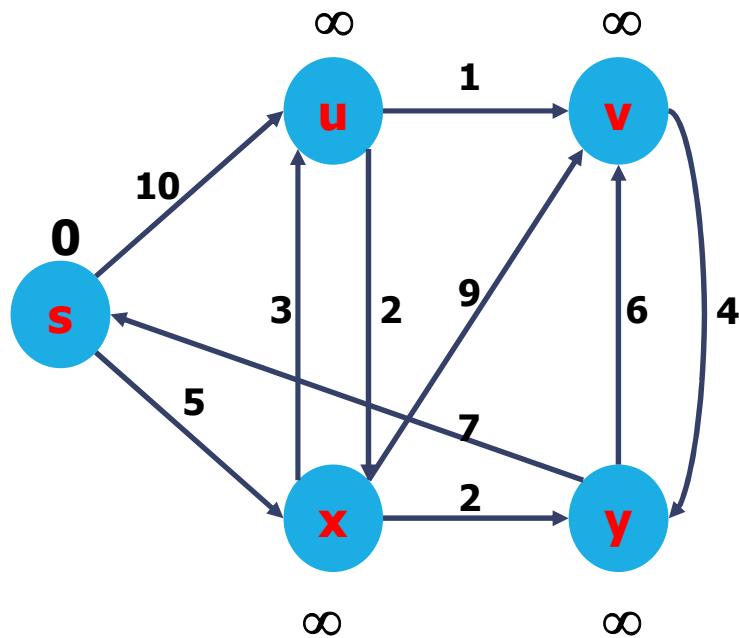
Esempio



Coda a priorità visualizzata per semplicità come vettore.
I nodi compaiono con il loro nome originale per leggibilità

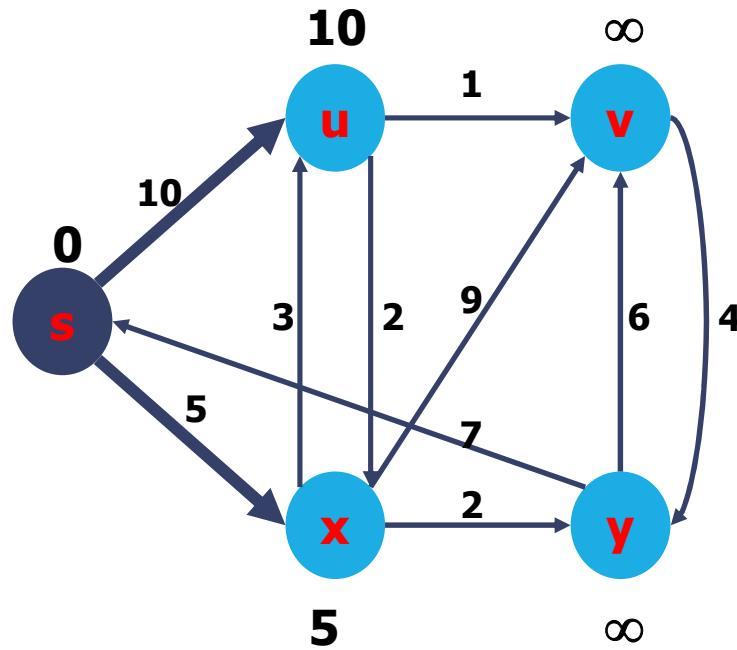
$$S = \{\}$$

$$PQ = \{s/\infty, u/\infty, v/\infty, x/\infty, y/\infty\}$$



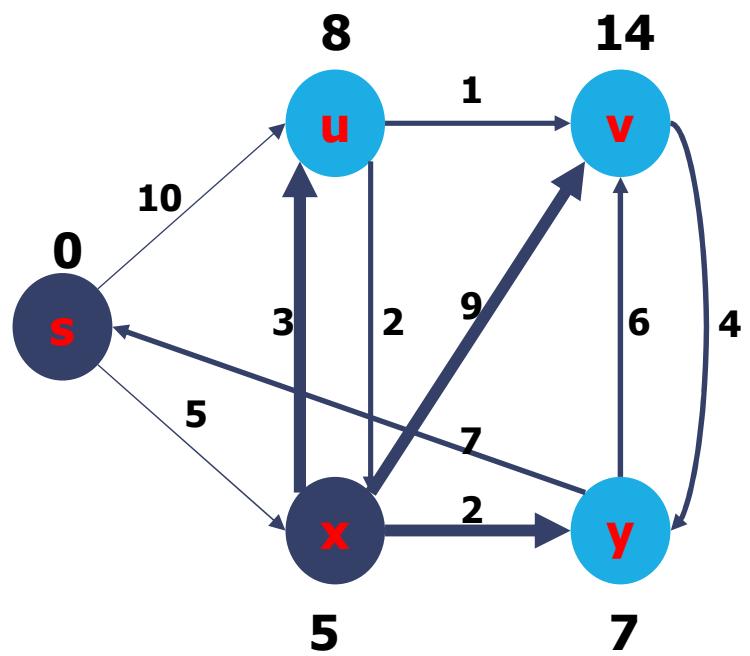
st	0	-1	-1	-1	-1
	0	1	2	3	4
d	0	∞	∞	∞	∞

0	1	2	3	4
0	1	2	3	4



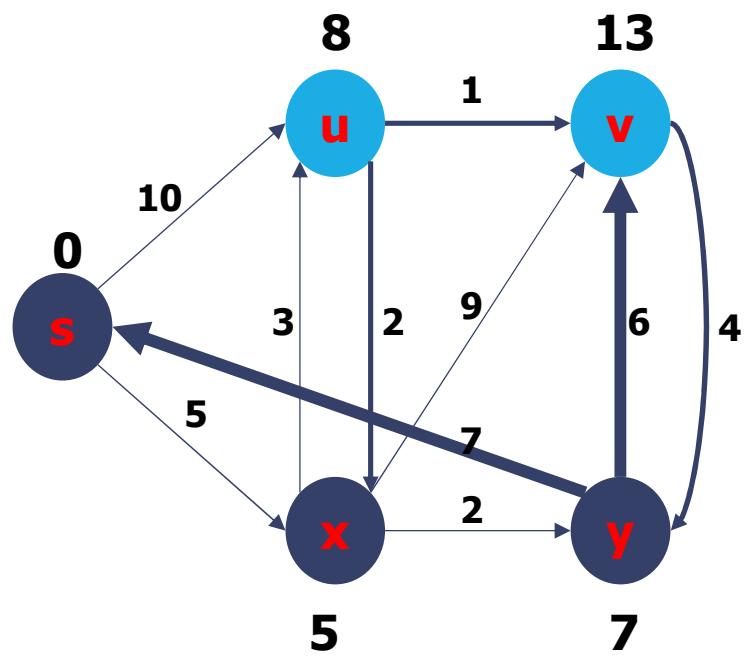
$S = \{s\}$
 relax $(s,u), (s,x)$
 $PQ = \{x/5, u/10, v/\infty, y/\infty\}$

st	0	0	0	-1	-1
	0	1	2	3	4
d	0	10	5	∞	∞
	0	1	2	3	4



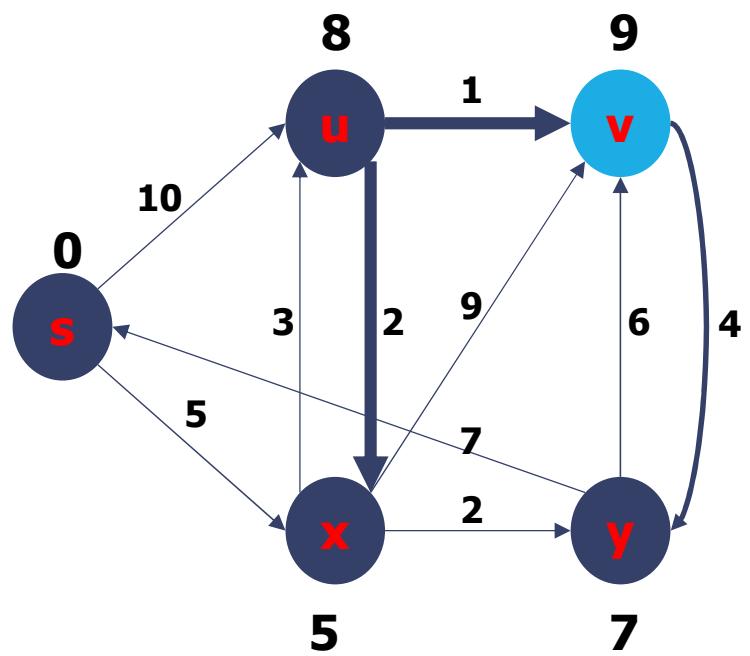
$S = \{s, x\}$
 relax $(x, u), (x, v), (x, y)$
 $PQ = \{y/7, u/8, v/14\}$

st	0	2	0	2	2
	0	1	2	3	4
d	0	8	5	14	7
	0	1	2	3	4



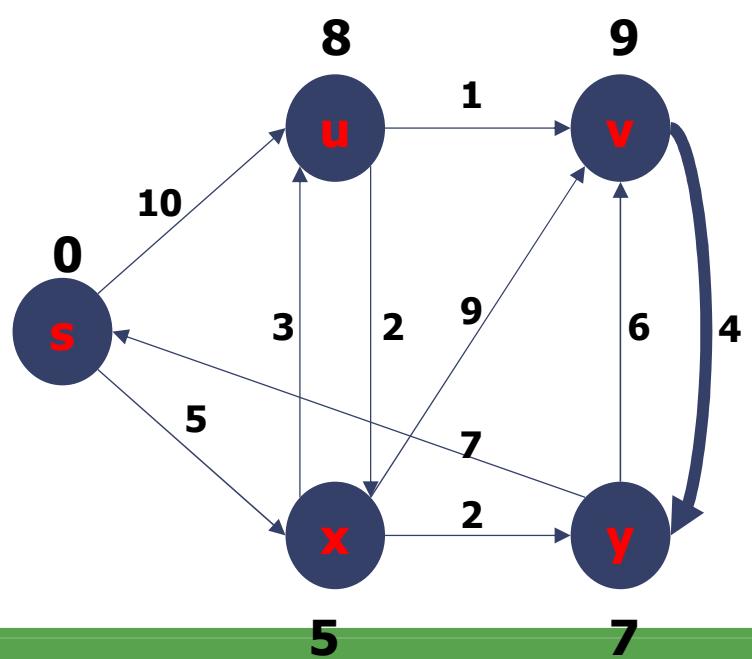
$S = \{s, x, y\}$
 relax $(y, s), (y, v)$
 $PQ = \{u/8, v/13\}$

st	0	2	0	4	2
	0	1	2	3	4
d	0	8	5	13	7
	0	1	2	3	4



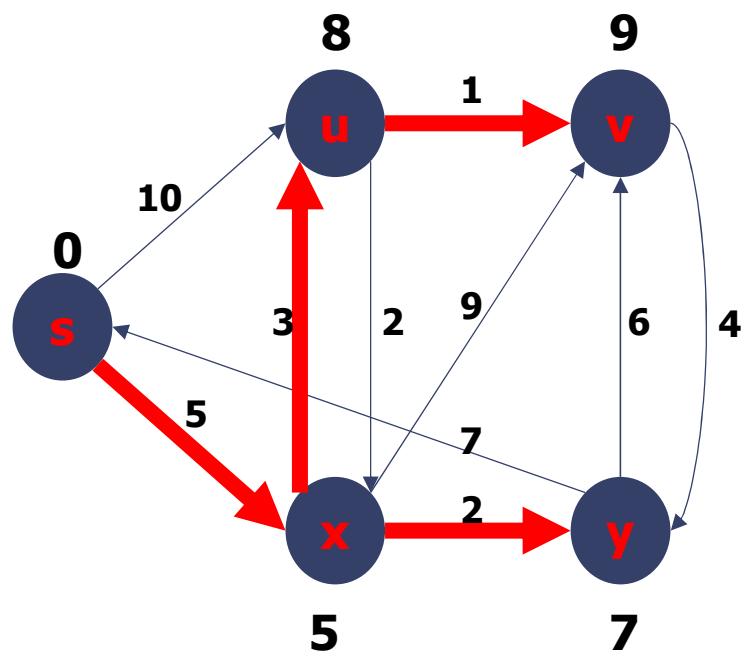
$S = \{s, x, y, u\}$
 relax $(u, v), (u, x)$
 $PQ = \{v/9\}$

st	0	2	0	1	2
	0	1	2	3	4
d	0	8	5	9	7
	0	1	2	3	4



$S = \{s, x, y, u, v\}$
 relax (v, y)
 $PQ = \{\}$

st	0	2	0	1	2
	0	1	2	3	4
d	0	8	5	9	7
	0	1	2	3	4



$S = \{s, x, y, u, v\}$
 $PQ = \{\}$

st	0	2	0	1	2
	0	1	2	3	4
d	0	8	5	9	7
	0	1	2	3	4

```
void GRAPHspD(Graph G, int id) {
    int v;
    link t;
    PQ pq = PQinit(G->V);
    int *st, *d;
    st = malloc(G->V*sizeof(int));
    d = malloc(G->V*sizeof(int));

    for (v = 0; v < G->V; v++){
        st[v] = -1;
        d[v] = maxWT;
        PQinsert(pq, d, v);
    }

    d[id] = 0;
    st[id] = id;
    PQchange(pq, d, id);
```

PQ con priorità in d

```

while (!PQempty(pq)) {
    if (d[v = PQextractMin(pq, d)] != maxWT)
        for (t=G->adj[v]; t!=G->z ; t=t->next)
            if (d[v] + t->wt < d[t->v]) {
                d[t->v] = d[v] + t->wt;
                PQchange(pq, d, t->v);
                st[t->v] = v;
            }
    }
    printf("\n Shortest path tree\n");
    for (v = 0; v < G->V; v++)
        printf("parent of %s is %s \n", STsearchByIndex(G->tab, v),
               STsearchByIndex (G->tab, st[v]));
    printf("\n Min.dist. from %s\n", STsearchByIndex(G->tab, s));
    for (v = 0; v < G->V; v++)
        printf("%s: %d\n", STsearchByIndex(G->tab, v), d[v]);
    PQfree(pq);
}

```

Complessità

$\Theta(|V|)$

V-S: coda a priorità pq dei vertici ancora da stimare. Termina per pq vuota. Implementando la pq con uno heap:

- estraie u da V-S (d[u] minimo)
- inserisce u in S
- rilassa tutti gli archi uscenti da u.

$O(\lg |V|)$

$O(\lg |V|)$

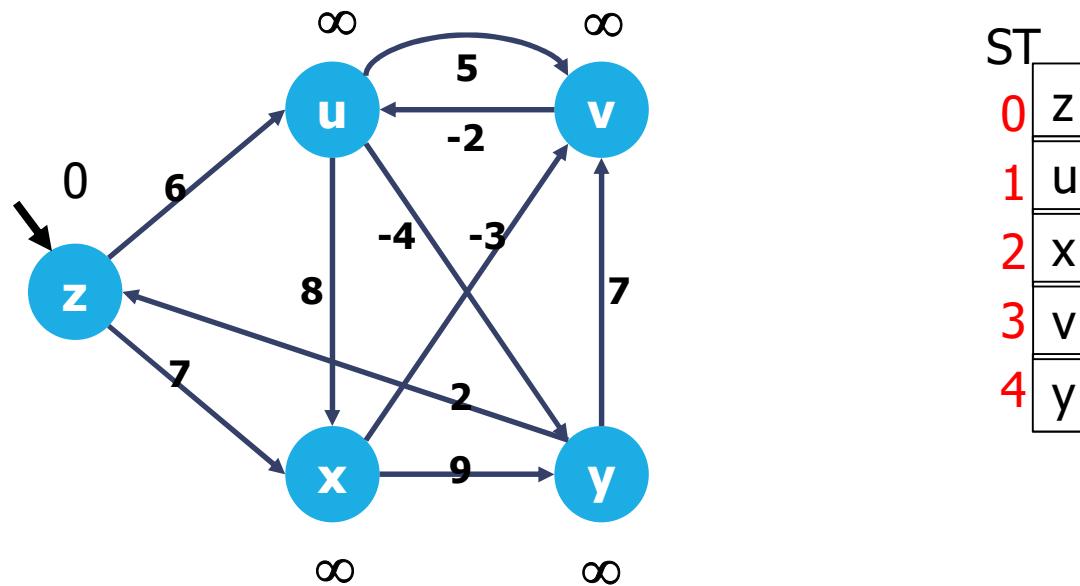
$T(n) = O((|V| + |E|) \lg |V|)$

$O(|E|)$

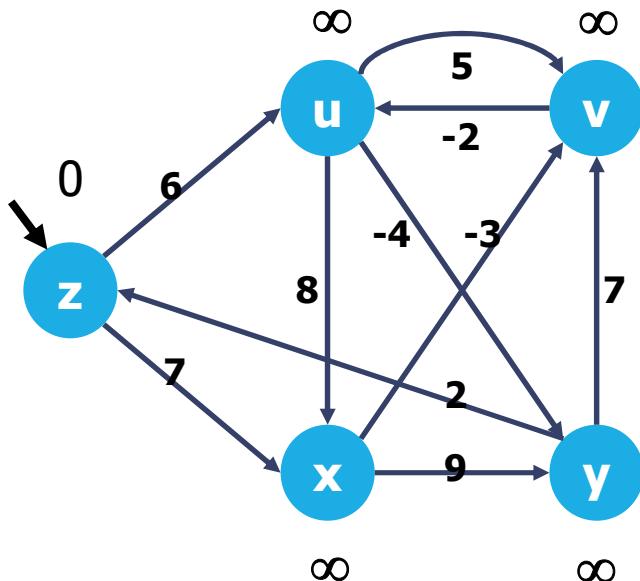
$T(n) = O(|E| \lg |V|)$ se tutti i vertici sono raggiungibili da s

Dijkstra e grafi con archi a peso negativo

- ✗ archi a peso negativo
- ✗ ciclo a peso negativo



ST
0
1
2
3
4



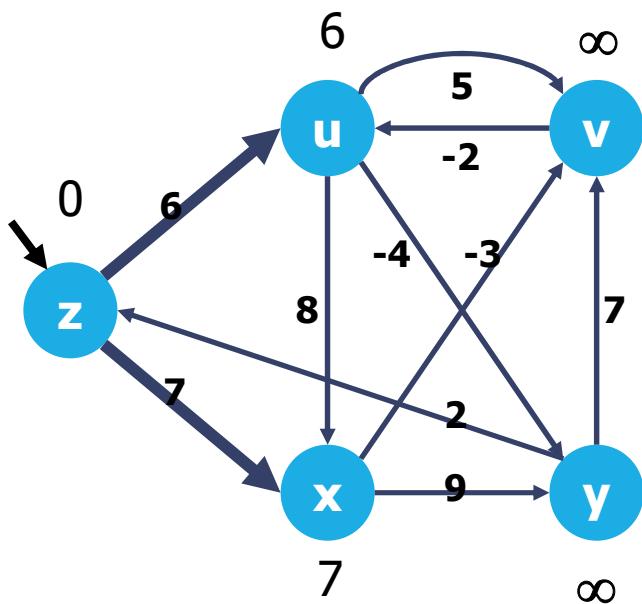
Coda a priorità visualizzata per semplicità come vettore. I nodi compaiono con il loro nome originale per leggibilità.

$$S = \{\}$$

$$PQ = \{z/0, u/\infty, v/\infty, x/\infty, y/\infty\}$$

$$\text{st} \quad \begin{array}{|c|c|c|c|c|c|} \hline 0 & -1 & -1 & -1 & -1 \\ \hline 0 & 1 & 2 & 3 & 4 \\ \hline \end{array}$$

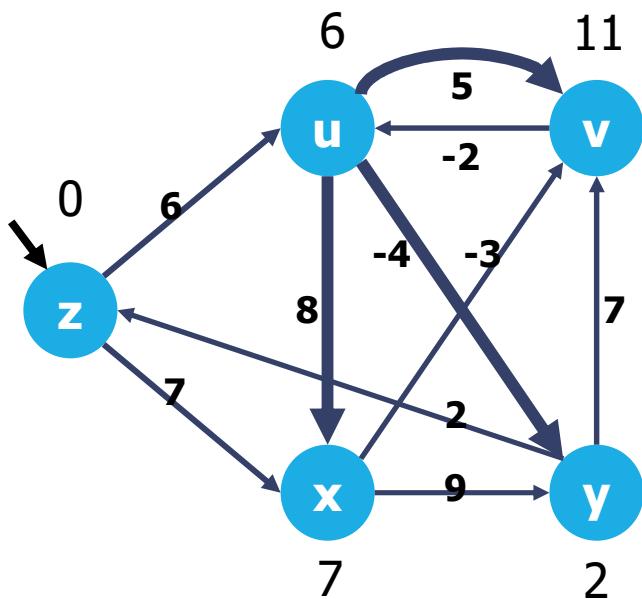
$$d \quad \begin{array}{|c|c|c|c|c|c|} \hline 0 & \infty & \infty & \infty & \infty \\ \hline 0 & 1 & 2 & 3 & 4 \\ \hline \end{array}$$



$S = \{z\}$
 relax $(z, u), (z, x)$
 $PQ = \{u/6, x/7, v/\infty, y/\infty\}$

st	0	0	0	-1	-1
	0	1	2	3	4
d	0	6	7	∞	∞

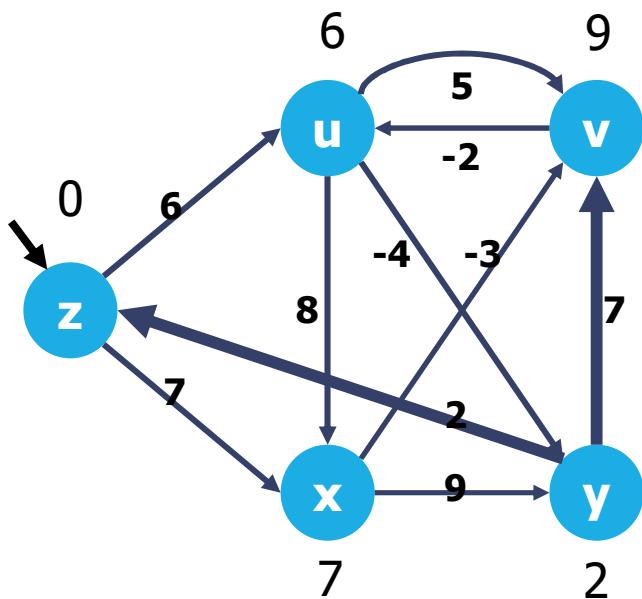
0	1	2	3	4
---	---	---	---	---



$S = \{z, u\}$
 relax (u, v) , (u, x) , (u, y)
 $PQ = \{y/2, x/7, v/11\}$

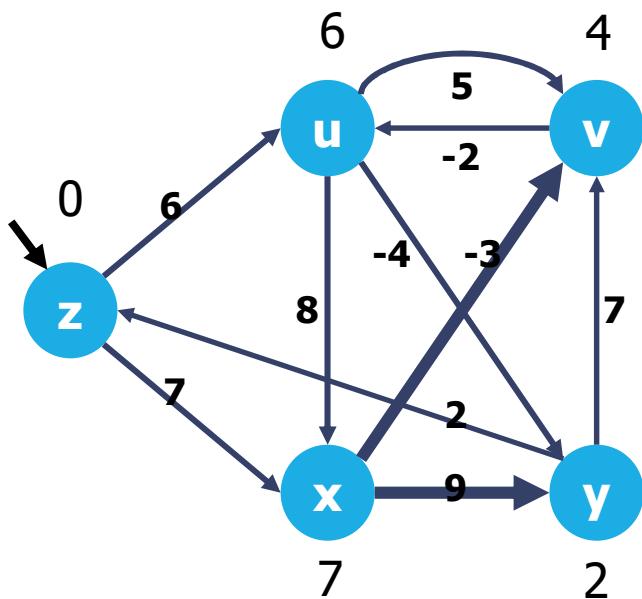
st	0	0	0	1	1
	0	1	2	3	4
d	0	6	7	11	2

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



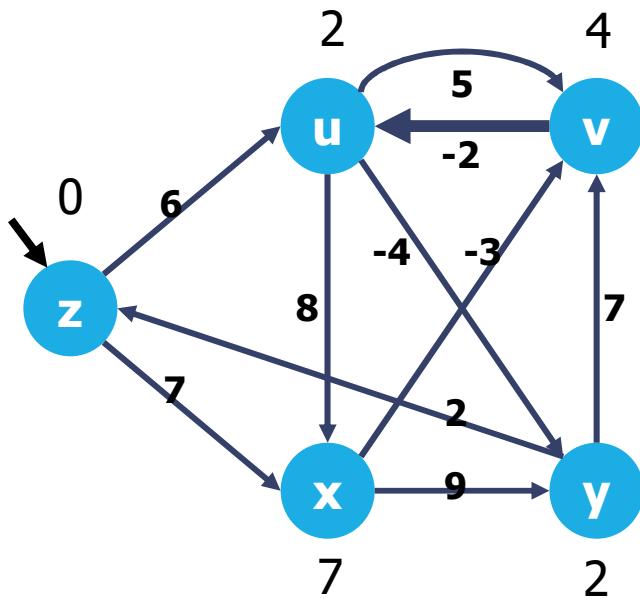
$S = \{z, u, y\}$
 relax $(y, z), (y, v)$
 $PQ = \{x/7, v/9\}$

st	0	0	0	4	1
	0	1	2	3	4
d	0	6	7	9	2
	0	1	2	3	4



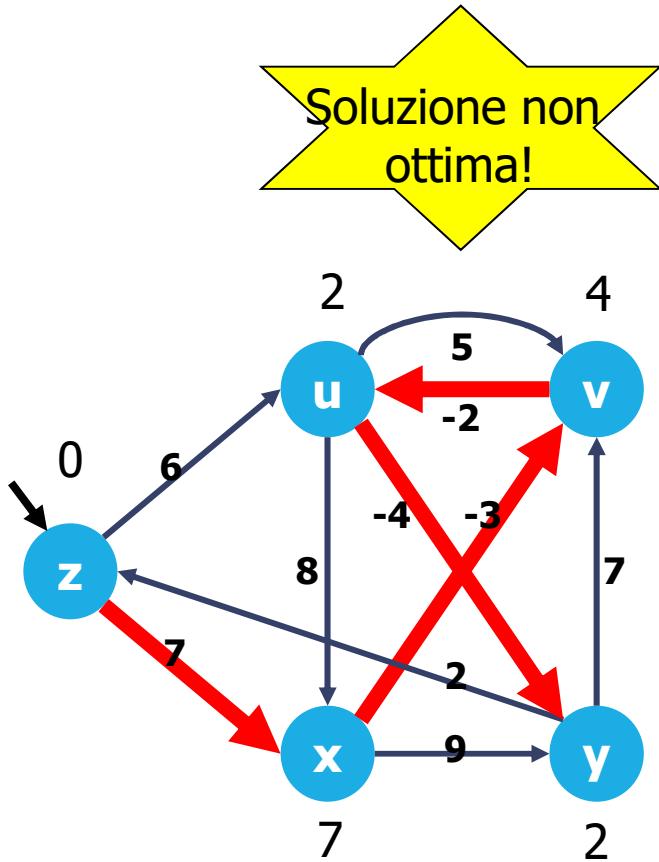
$S = \{z, u, y, x\}$
 relax $(x, v), (x, y)$
 $PQ = \{v/4\}$

st	0	0	0	2	1
	0	1	2	3	4
d	0	6	7	4	2
	0	1	2	3	4



$S = \{z, u, x, y, v\}$
 relax (v,u)
 $PQ = \{\}$

st	0	3	0	2	1
	0	1	2	3	4
d	0	2	7	4	2
	0	1	2	3	4



$S=\{z, u, x, y, v\}$
 relax (v,u)
 $PQ=\{\}$

st	0	3	0	2	1
	0	1	2	3	4
d	0	2	7	4	2

0	1	2	3	4
0	1	2	3	4

Se si riconsiderasse l'arco (u,y) la stima di y scenderebbe a -2 (**Soluzione ottima**).

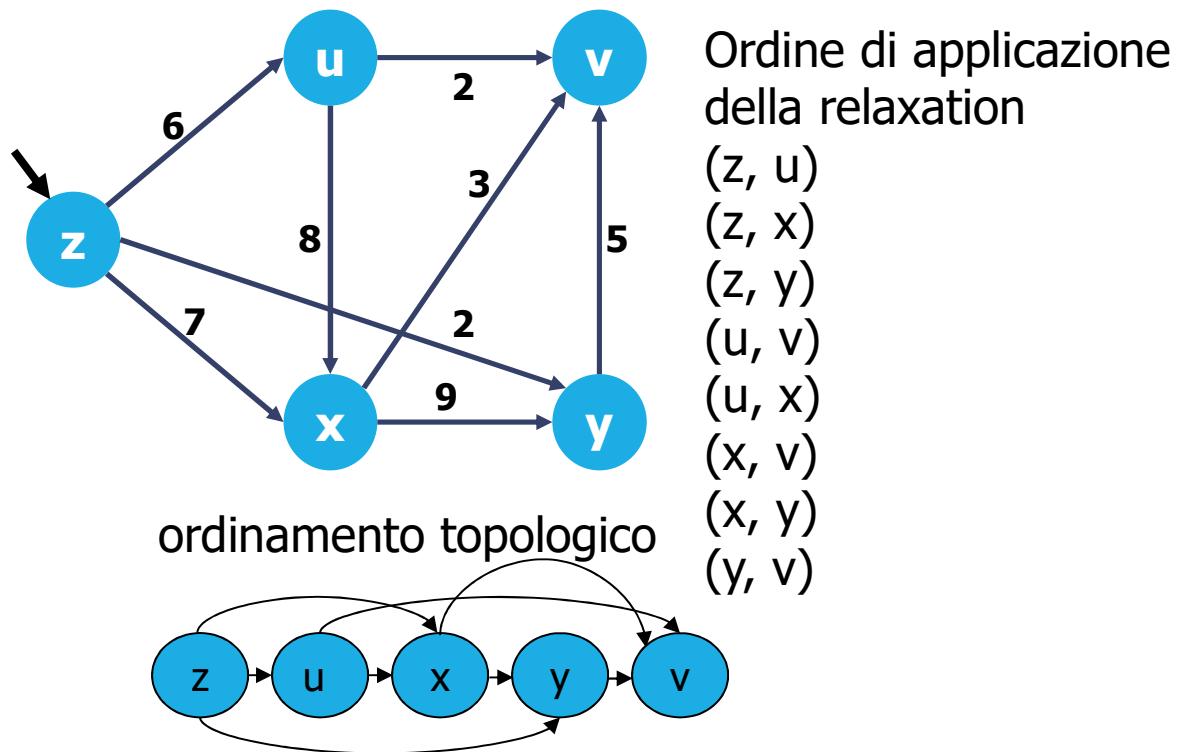
Cammini minimi su DAG pesati

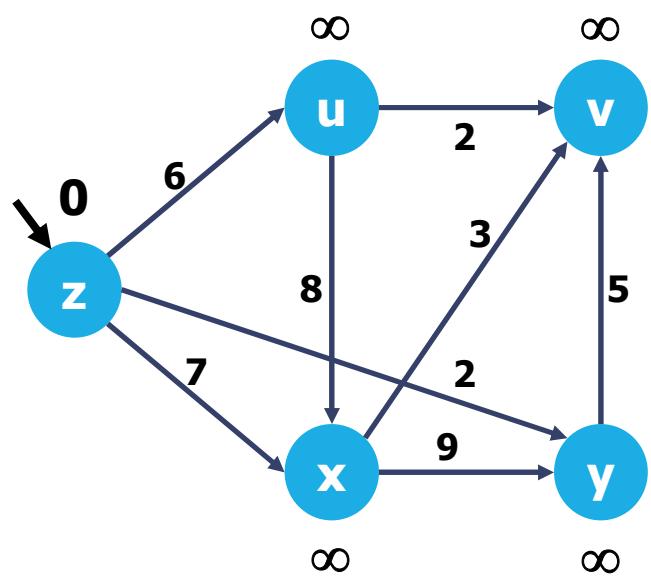
L'assenza di cicli semplifica l'algoritmo:

- ordinamento topologico del DAG
- per tutti i vertici ordinati:
- applica la relaxation da quel vertice.

Esempio

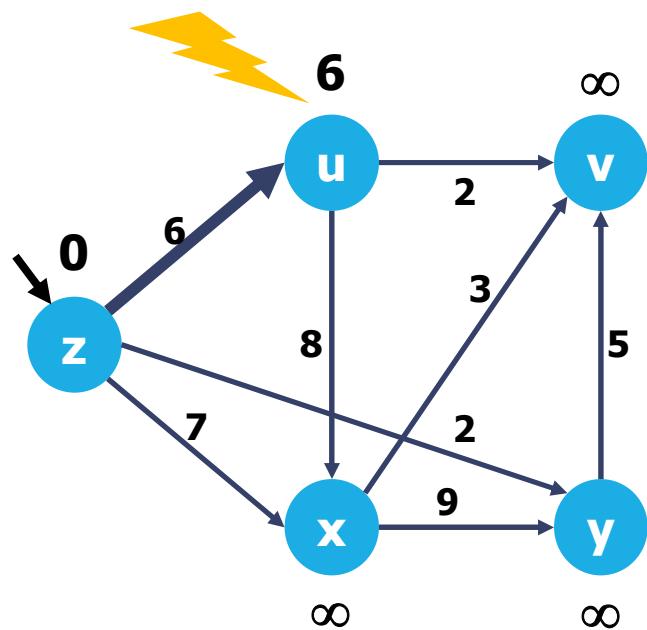
I nodi compaiono con il loro nome originale per leggibilità





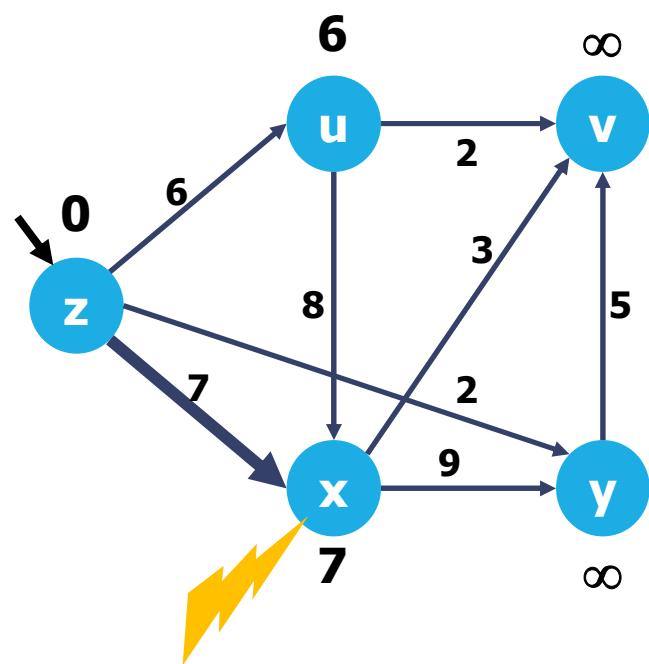
Ordine di applicazione
della relaxation

- (z, u)
- (z, x)
- (z, y)
- (u, v)
- (u, x)
- (x, v)
- (x, y)
- (y, v)



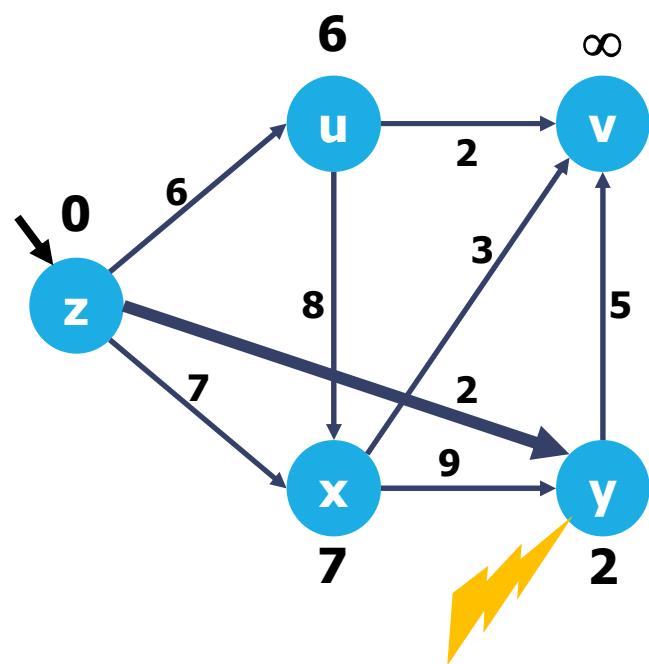
Ordine di applicazione
della relaxation

- (z, u) ←
- (z, x)
- (z, y)
- (u, v)
- (u, x)
- (x, v)
- (x, y)
- (y, v)



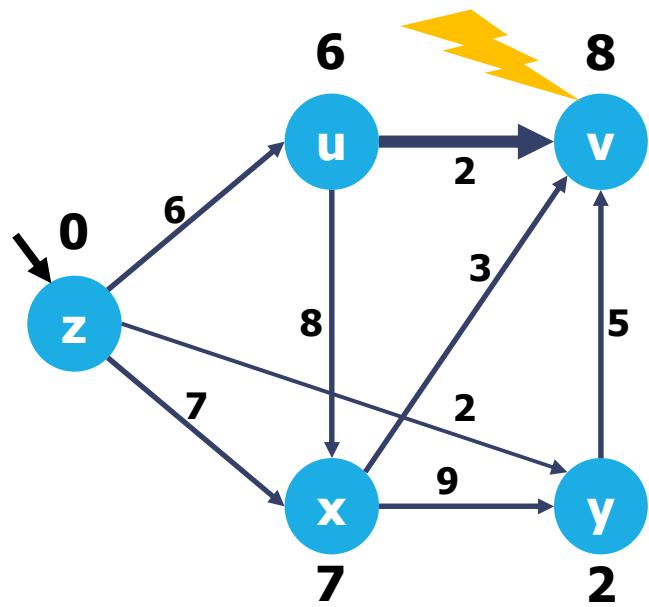
Ordine di applicazione
della relaxation

- (z, u)
- (z, x) ←
- (z, y)
- (u, v)
- (u, x)
- (x, v)
- (x, y)
- (y, v)



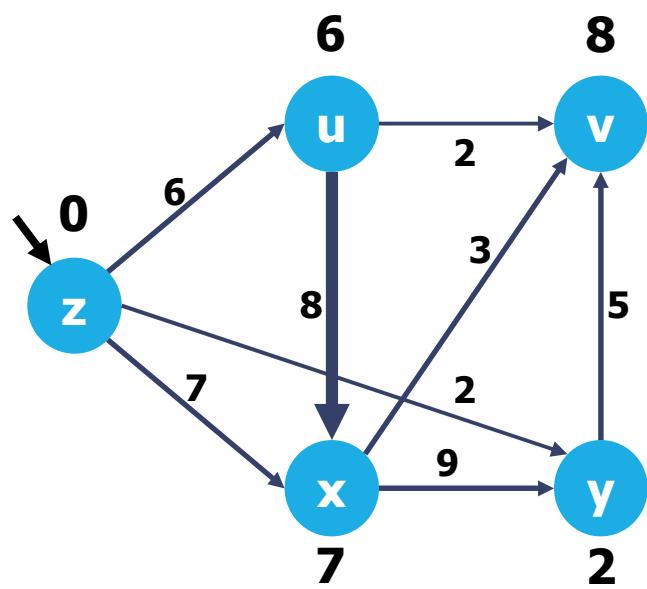
Ordine di applicazione
della relaxation

- (z, u)
- (z, x)
- (z, y) ←
- (u, v)
- (u, x)
- (x, v)
- (x, y)
- (y, v)



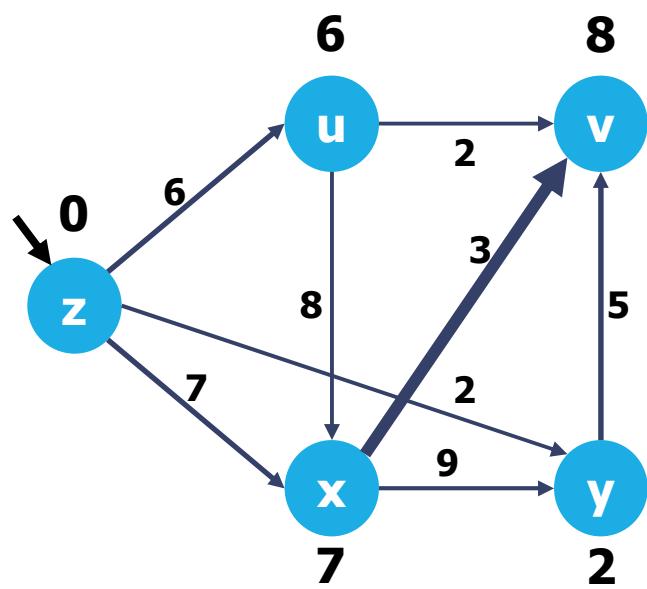
Ordine di applicazione
della relaxation

- (z, u)
- (z, x)
- (z, y)
- (u, v) ←
- (u, x)
- (x, v)
- (x, y)
- (y, v)



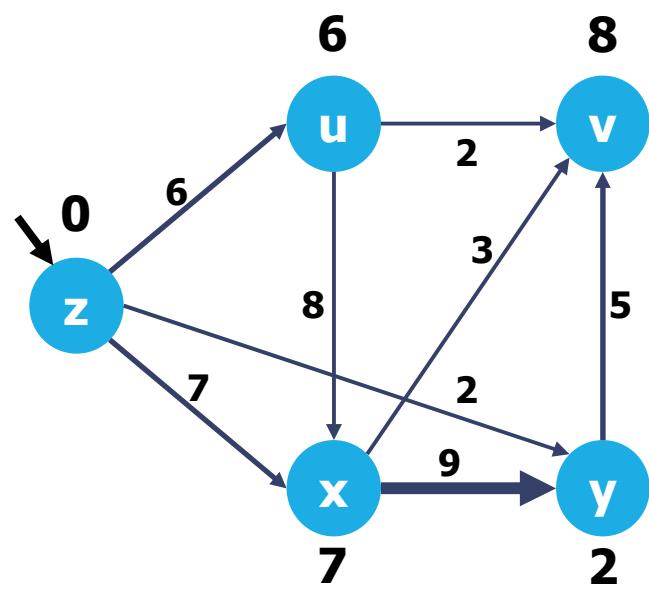
Ordine di applicazione
della relaxation

- (z, u)
- (z, x)
- (z, y)
- (u, v)
- (u, x) ←
- (x, v)
- (x, y)
- (y, v)



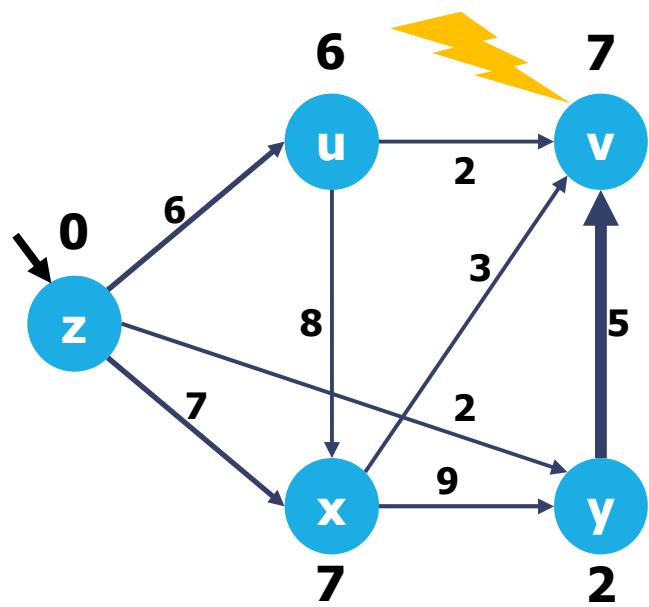
Ordine di applicazione
della relaxation

- (z, u)
- (z, x)
- (z, y)
- (u, v)
- (u, x)
- (x, v) ←
- (x, y)
- (y, v)



Ordine di applicazione
della relaxation

- (z, u)
- (z, x)
- (z, y)
- (u, v)
- (u, x)
- (x, v)
- (x, y) ←
- (y, v)



Ordine di applicazione
della relaxation

- (z, u)
- (z, x)
- (z, y)
- (u, v)
- (u, x)
- (x, v)
- (x, y)
- (y, v) ←

Complessità

- Applicabile a DAG anche con archi negativi
- $T(n) = O(|V|+|E|)$.

Applicazione: Seam Carving

Algoritmo di **image resizing** per minimizzare la distorsione (Avidan, Shamir).

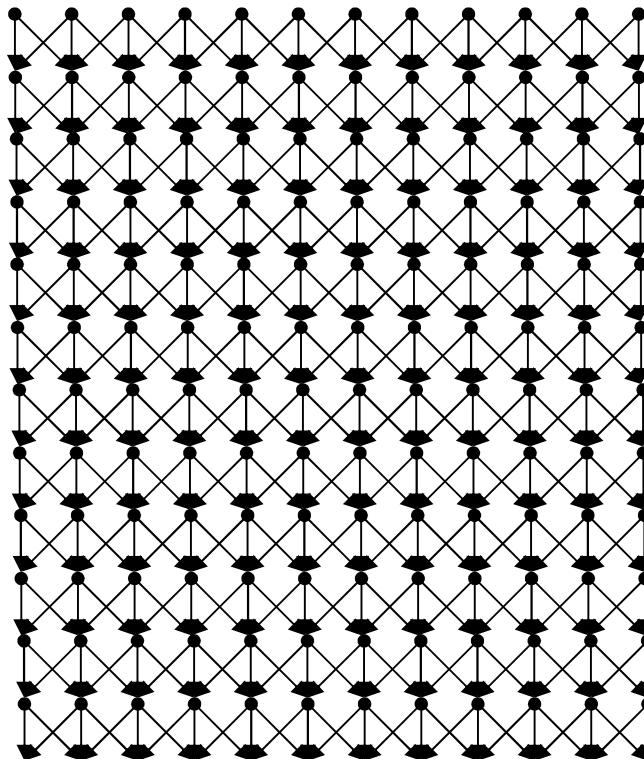
Modello: immagine come DAG pesato di pixel.

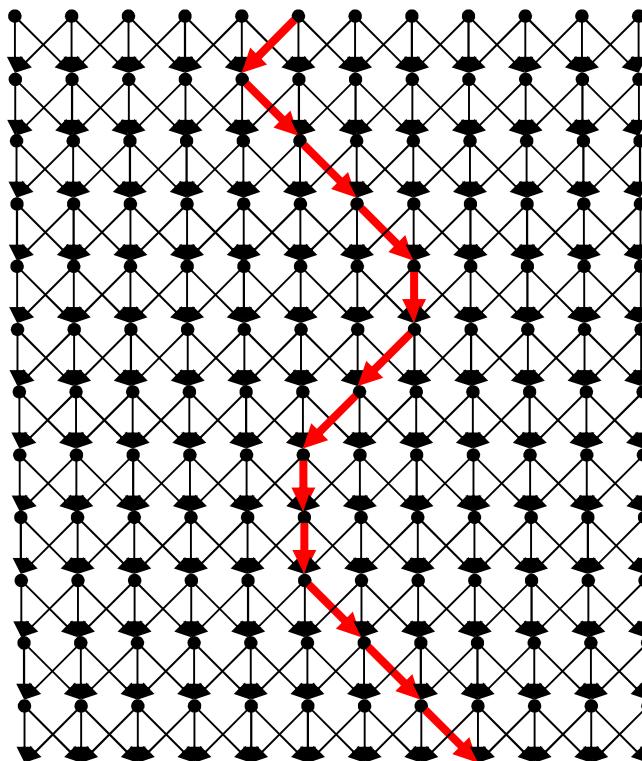
Peso dell'arco: misura del contrasto tra 2 pixel.

Algoritmo: determinazione di un cammino minimo da una sorgente (seam), eliminazione dei pixel su di esso.

<http://en.wikipedia.org>

Sedgewick, Wayne, Algorithms Part I & II, www.coursera.org





seam



Cammini massimi su DAG pesati

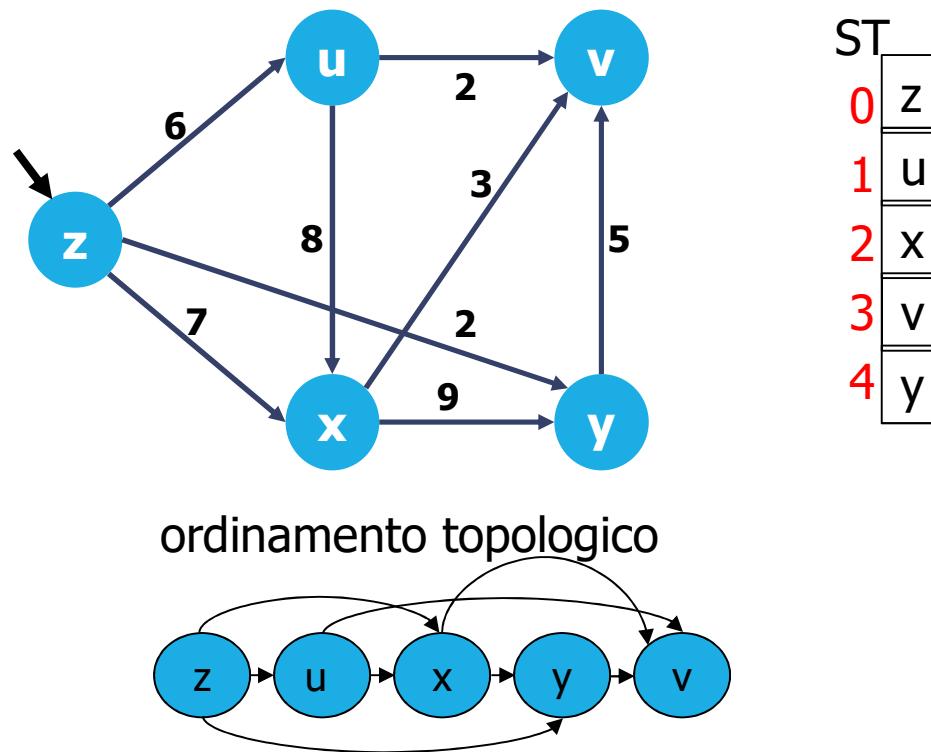
Problema non trattabile su grafi pesati qualsiasi.

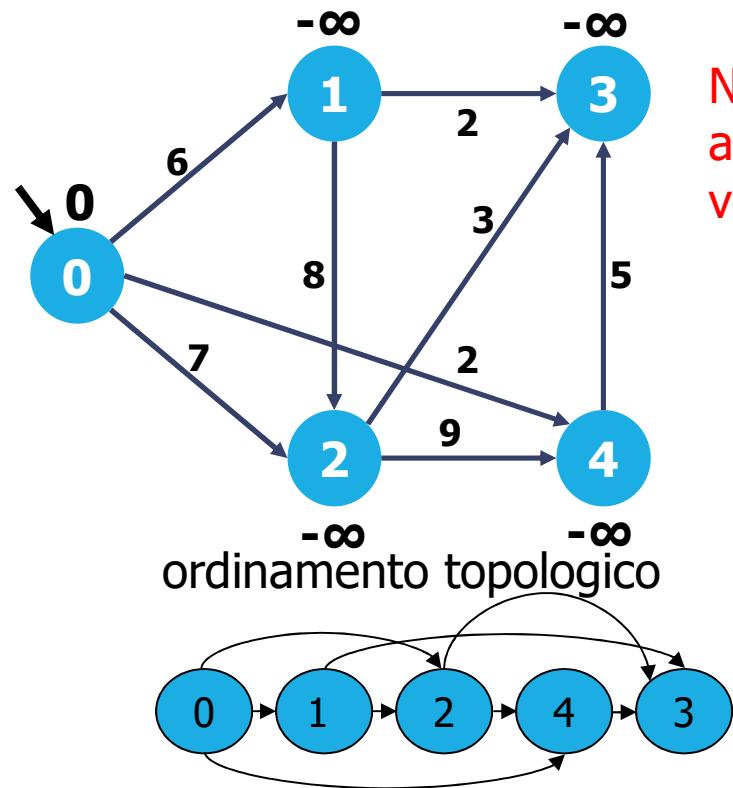
L'assenza di cicli tipica dei DAG rende facile il problema:

- ordinamento topologico del DAG
- per tutti i vertici ordinati:
- applica la relaxation «invertita» da quel vertice:

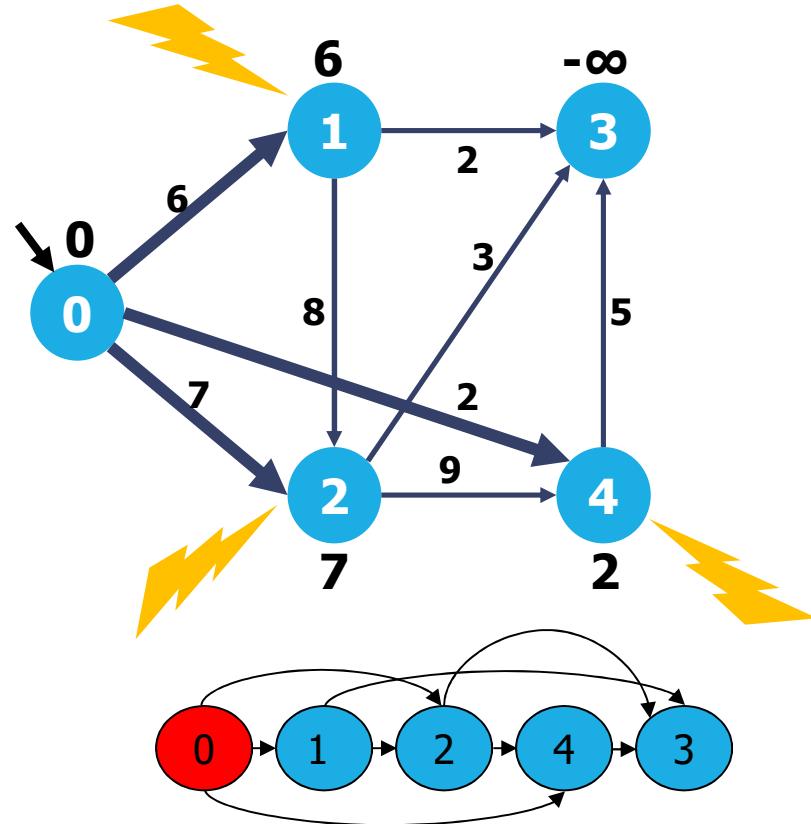
```
if (d[v] < d[u] + w(u,v)) {  
    d[v] = d[u] + w(u,v);  
    st[v] = u;  
}
```

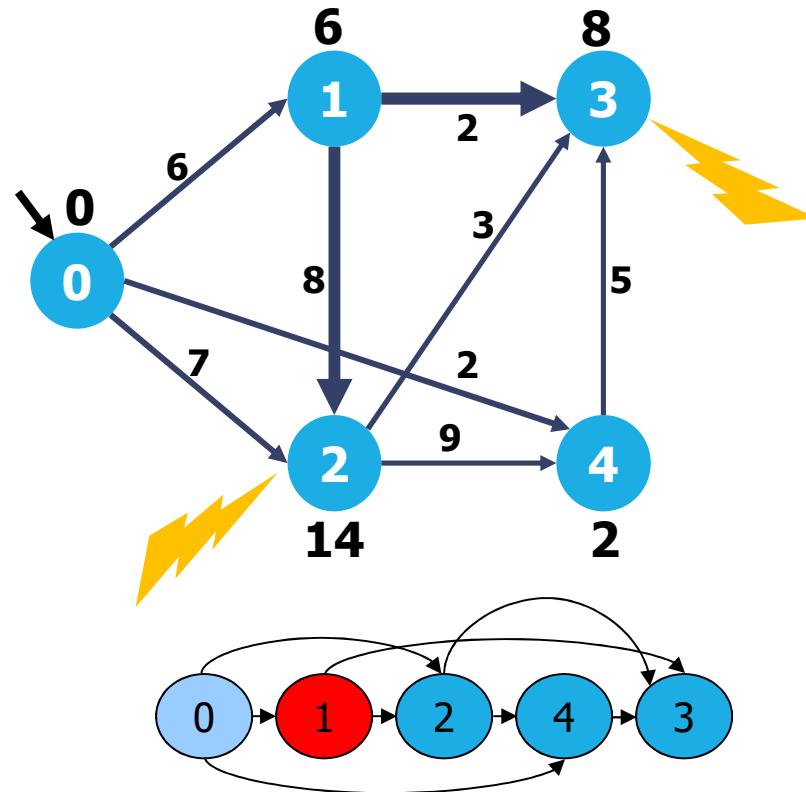
Esempio

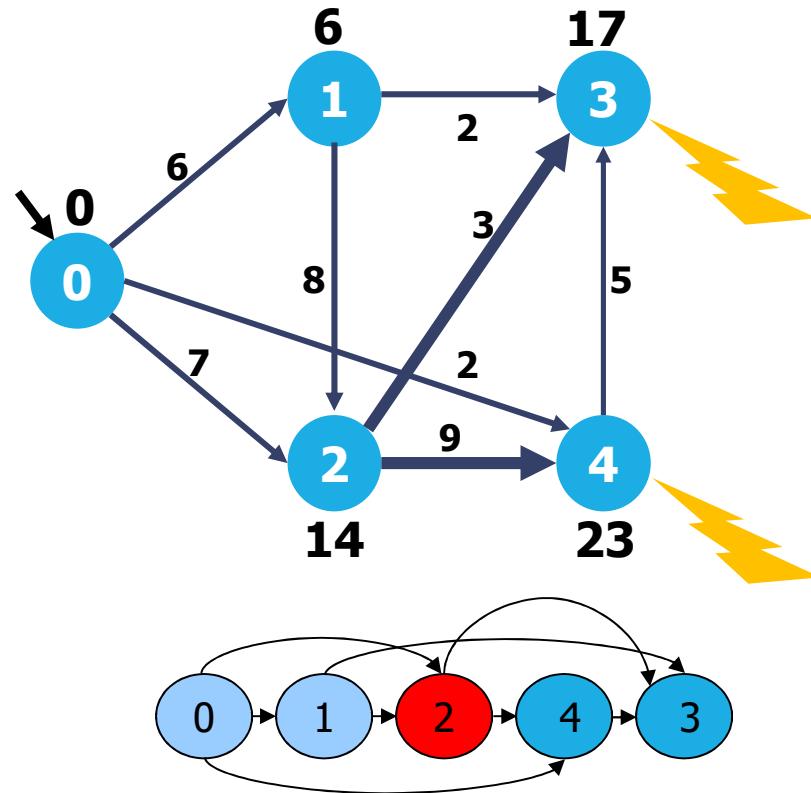


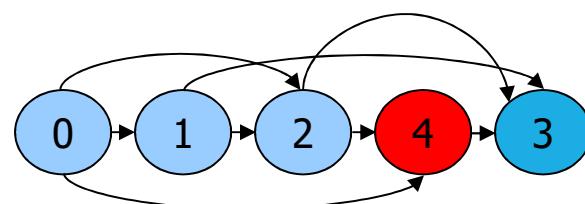
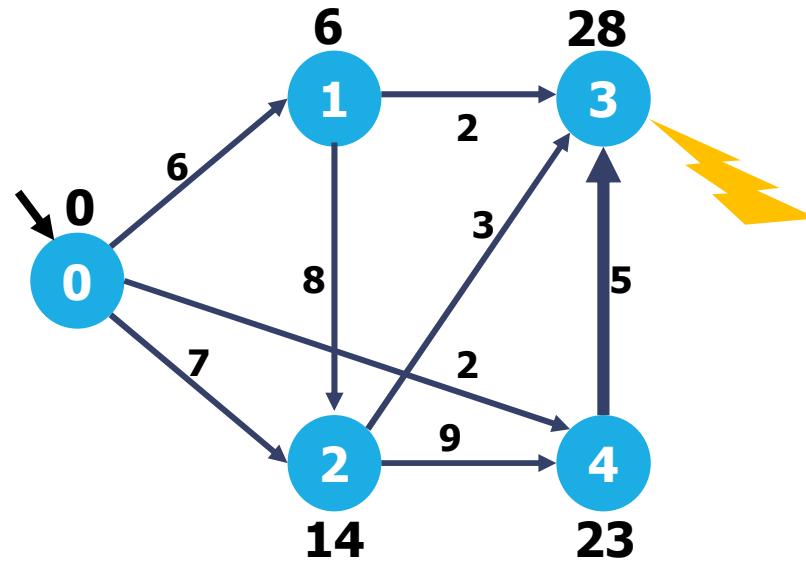


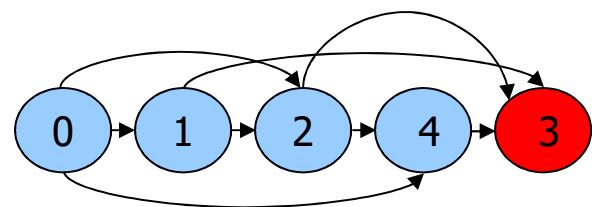
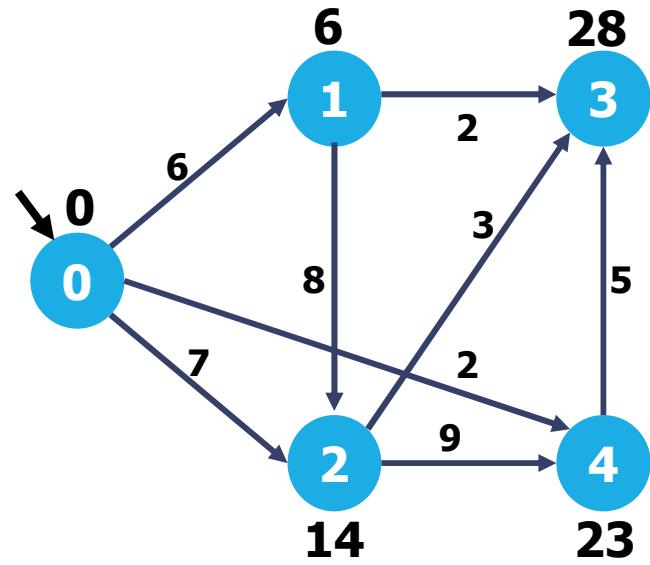
Notare le stime iniziali
a $-\infty$ tranne che del
vertice di partenza











Algoritmo di Bellman-Ford

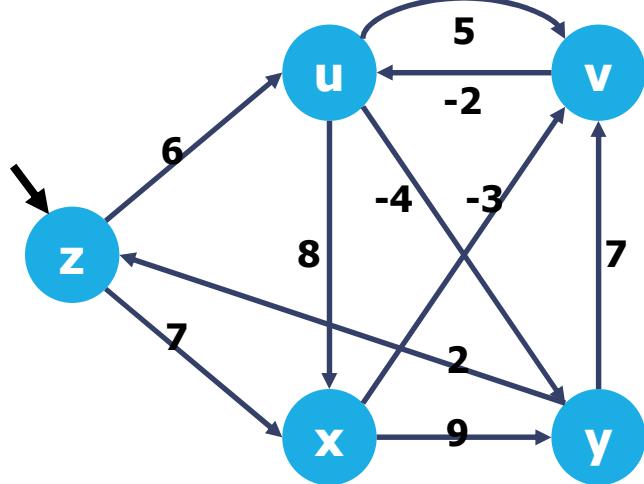
- Ipotesi: possono \exists archi a peso < 0
- Rileva cicli < 0
- Strategia: **programmazione dinamica** (applicabilità già dimostrata)
- soluzione ricorsiva: $d_w[v]$ è la lunghezza del cammino minimo da s a v con al più w archi
 - $d_0[v]$ vale 0 se v coincide con s , ∞ altrimenti
 - $d_w[v] = \min_{u \in \text{Inc}(v)} (d_{w-1}[u] + w(u,v))$
 - $\text{Inc}(v)$: insieme dei vertici da cui escono archi incidenti in v

Calcolo bottom-up:

- vettori d e st delle distanze minime e dei predecessori opportunamente inizializzati
- $|V|-1$ passi
- al passo i -esimo:
 - ciclo di rilassamento sugli archi «in avanti» (lista di adiacenza e non di incidenza)
- al $|V|$ -esimo passo:
 - diminuisce almeno una stima: \exists ciclo <0
 - altrimenti soluzione ottima.

Il numero di passi è al massimo $|V|-1$: se ci si accorge di aver raggiunto un punto fisso (non cambiano le stime da un passo a quello successivo), si può interrompere anticipatamente il ciclo.

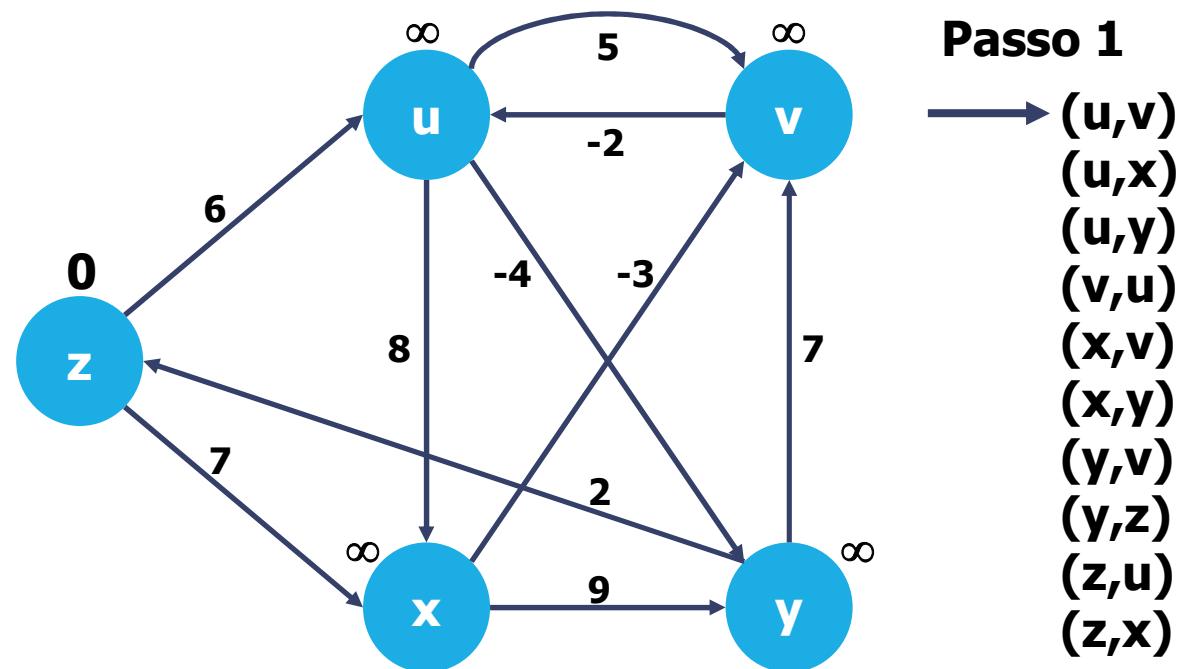
Esempio

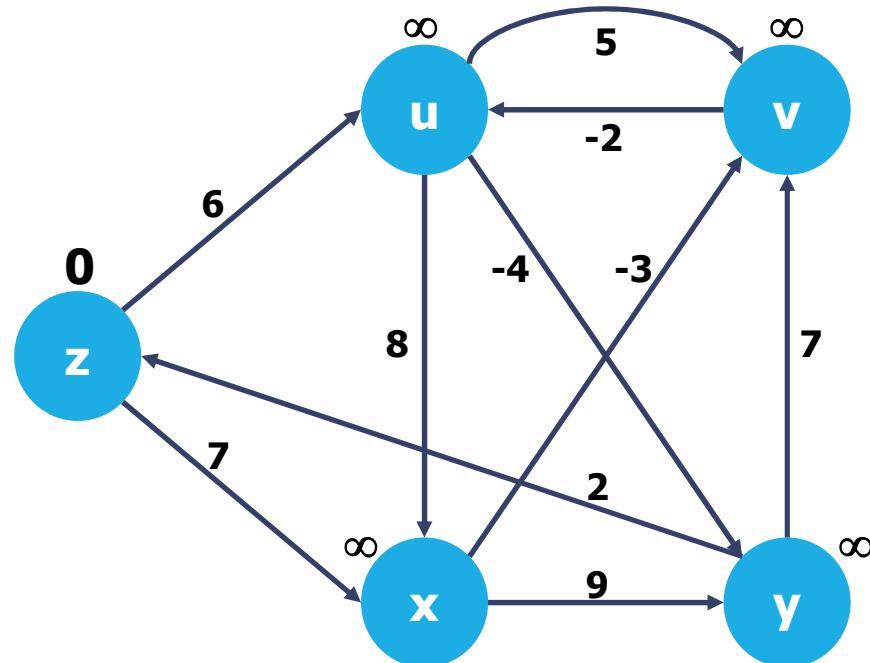


ST
0
1
2
3
4

Archi in ordine lessicografico:
(u,v)
(u,x)
(u,y)
(v,u)
(x,v)
(x,y)
(y,v)
(y,z)
(z,u)
(z,x)

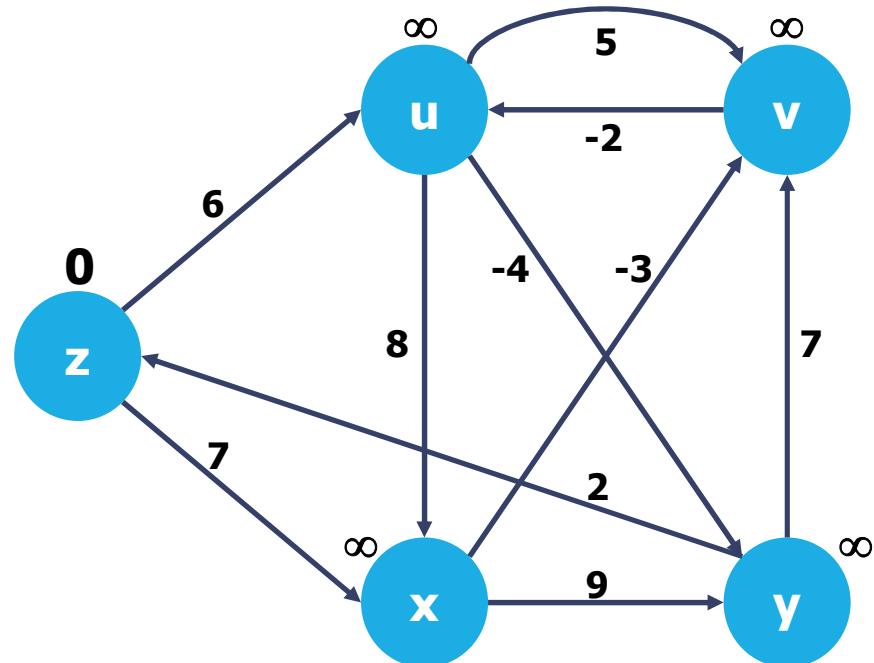
I nodi compaiono con il loro nome originale per leggibilità





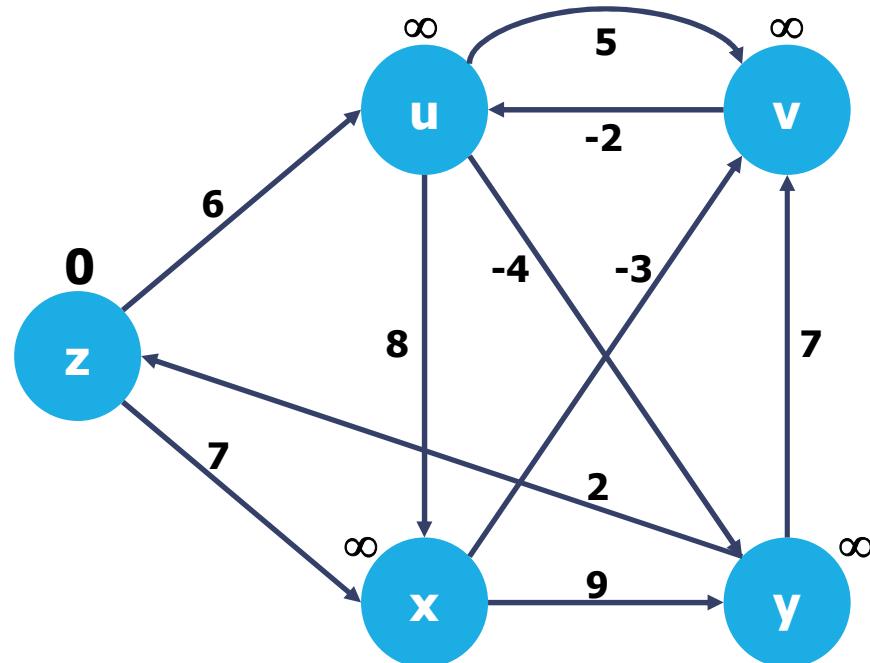
Passo 1

$\rightarrow (u,v)$
 $\rightarrow (u,x)$
 $\rightarrow (u,y)$
 $\rightarrow (v,u)$
 $\rightarrow (x,v)$
 $\rightarrow (x,y)$
 $\rightarrow (y,v)$
 $\rightarrow (y,z)$
 $\rightarrow (z,u)$
 $\rightarrow (z,x)$



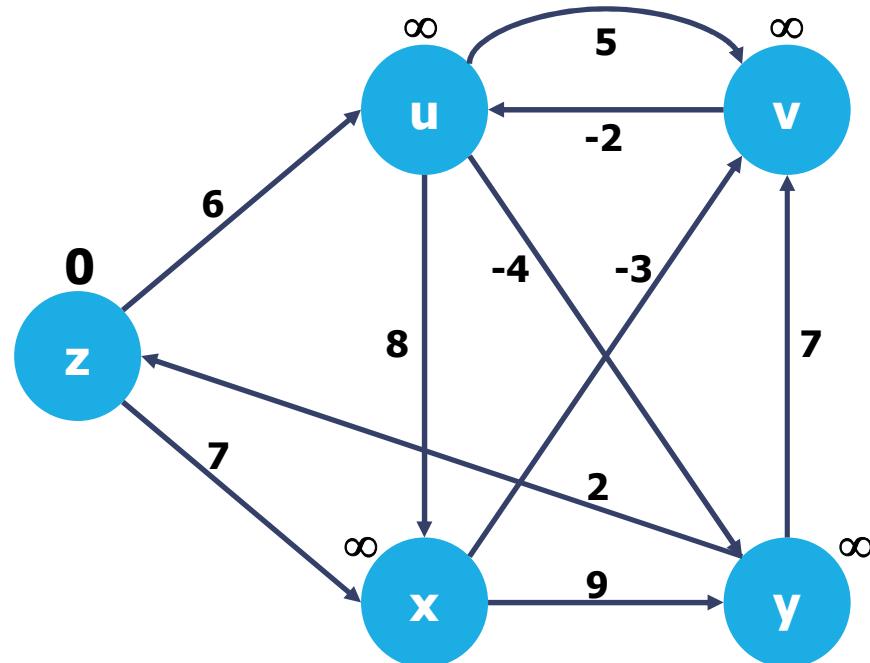
Passo 1

(u,v)
 (u,x)
 $\rightarrow (u,y)$
 (v,u)
 (x,v)
 (x,y)
 (y,v)
 (y,z)
 (z,u)
 (z,x)



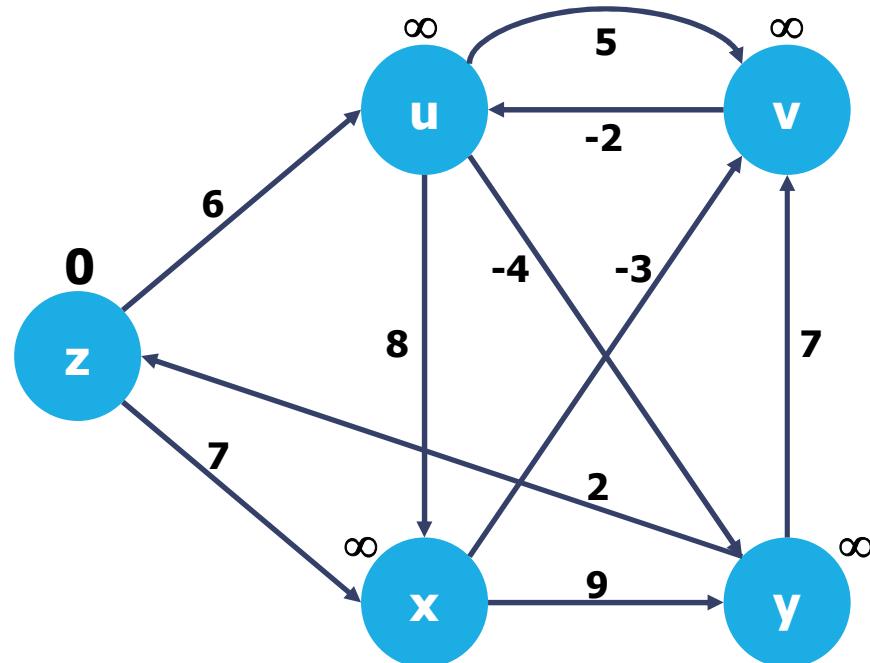
Passo 1

(u,v)
 (u,x)
 (u,y)
 $\rightarrow (v,u)$
 (x,v)
 (x,y)
 (y,v)
 (y,z)
 (z,u)
 (z,x)



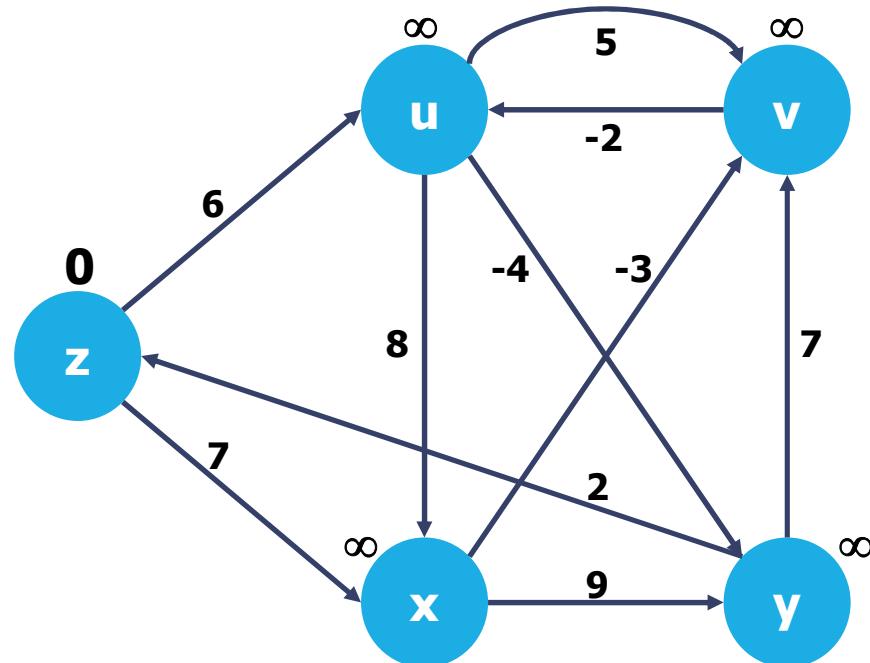
Passo 1

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



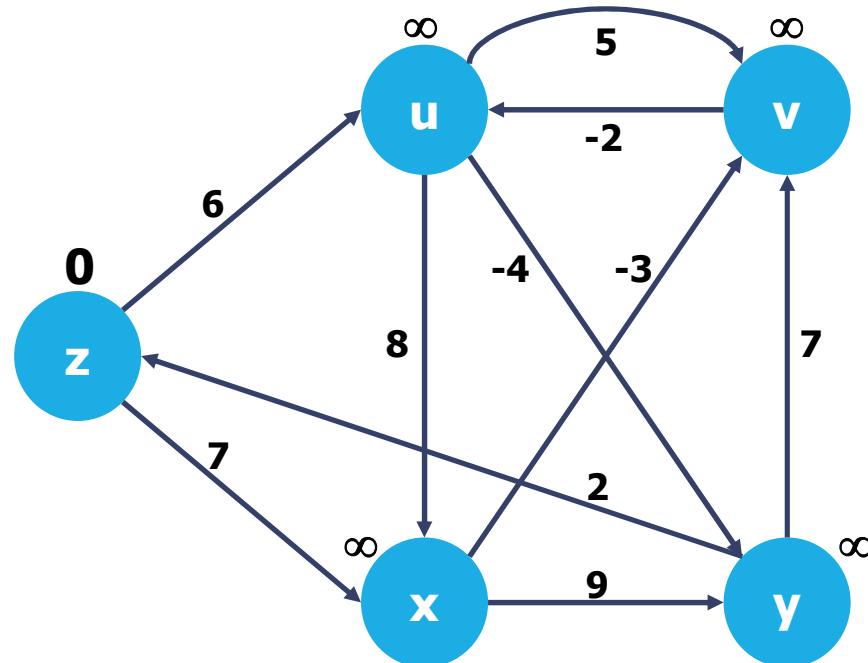
Passo 1

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



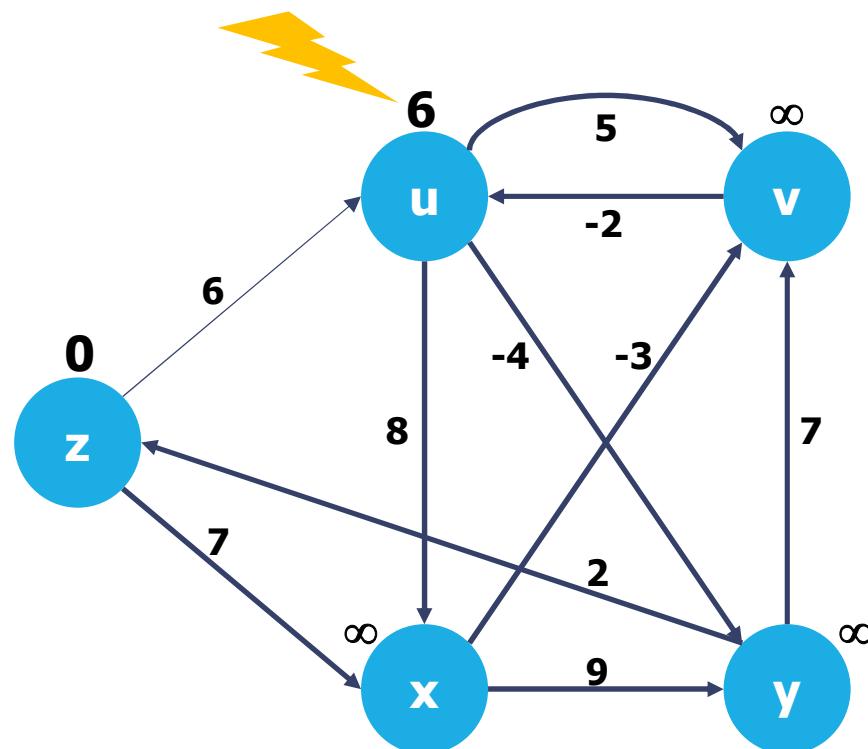
Passo 1

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



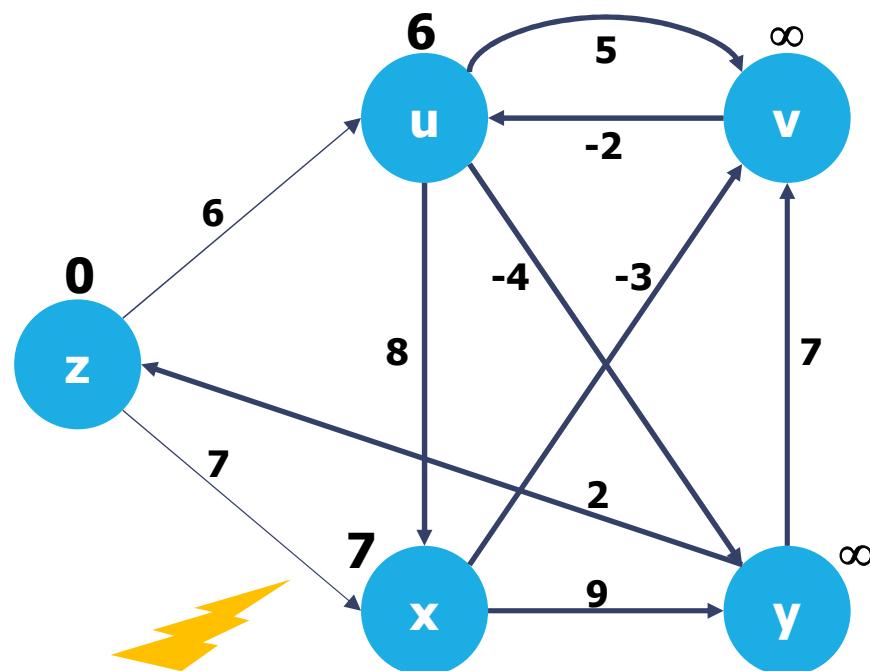
Passo 1

(u,v)
 (u,x)
 (u,y)
 (v,u)
 (x,v)
 (x,y)
 (y,v)
 $\rightarrow (y,z)$
 (z,u)
 (z,x)



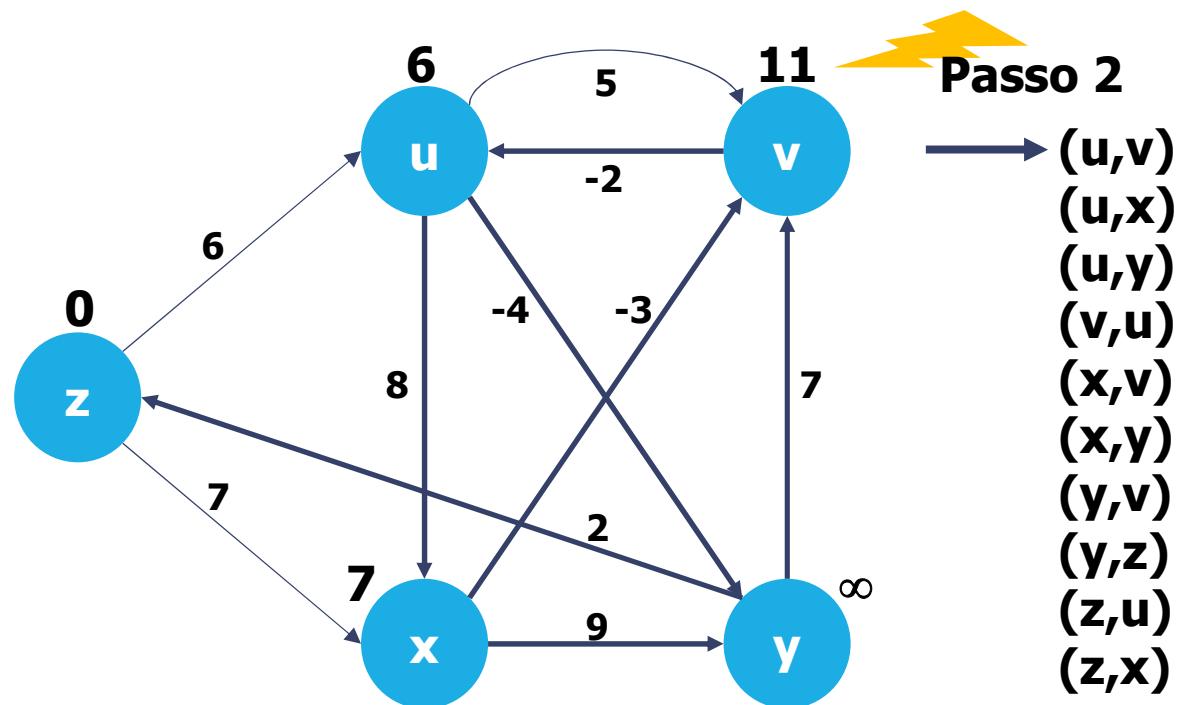
Passo 1

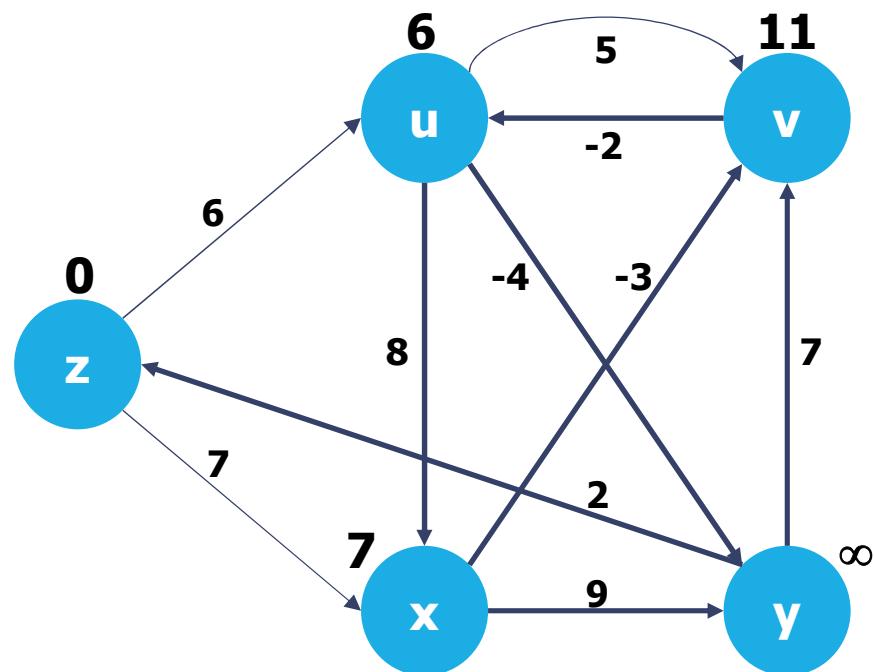
- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



Passo 1

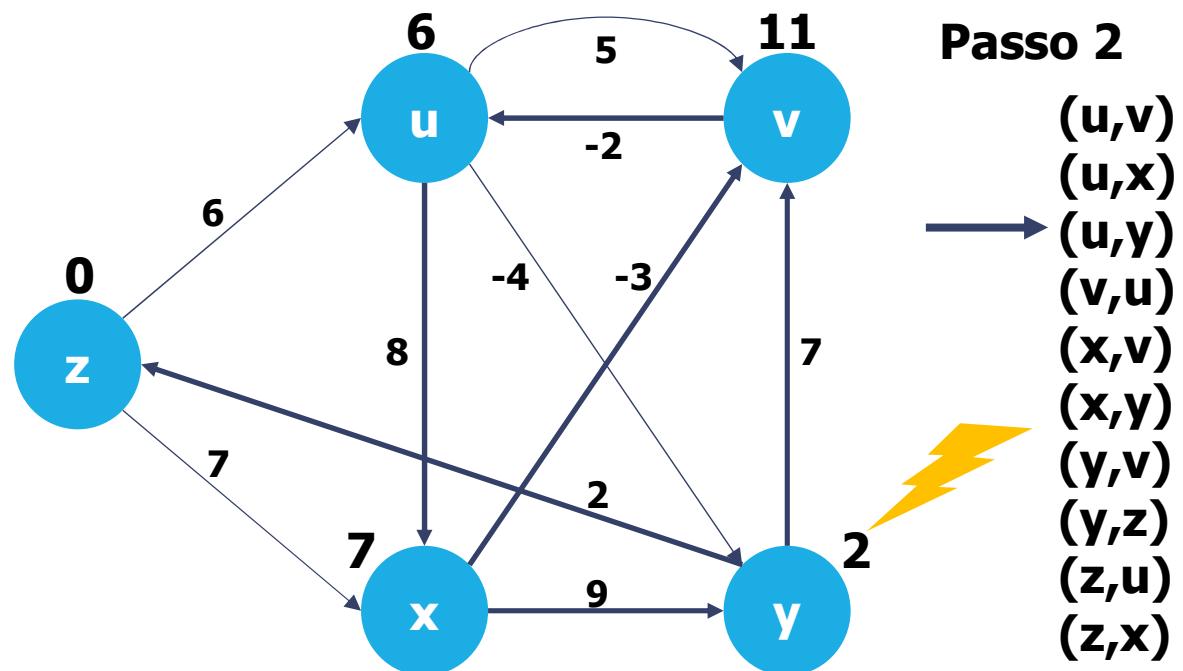
- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- \rightarrow (z,x)

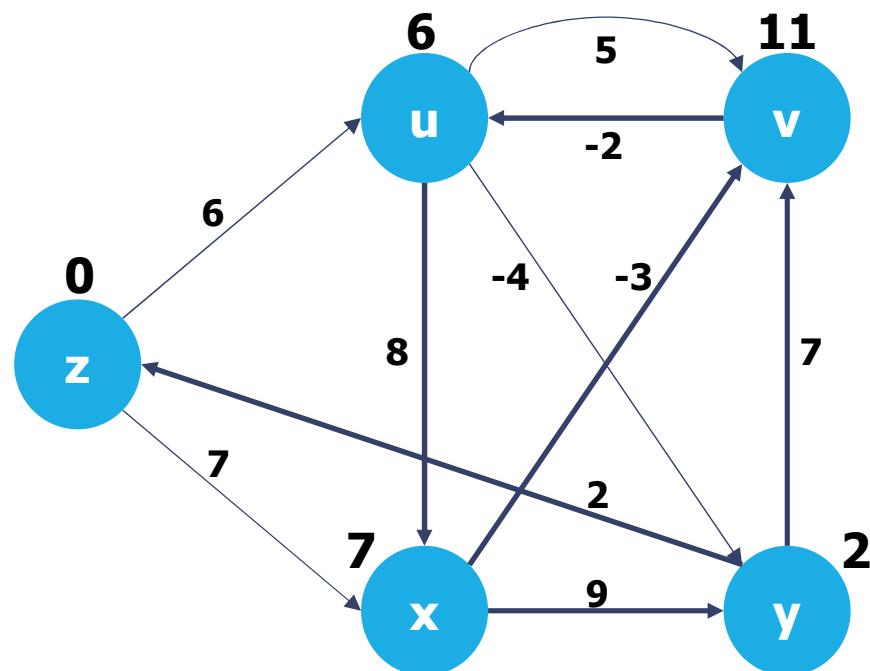




Passo 2

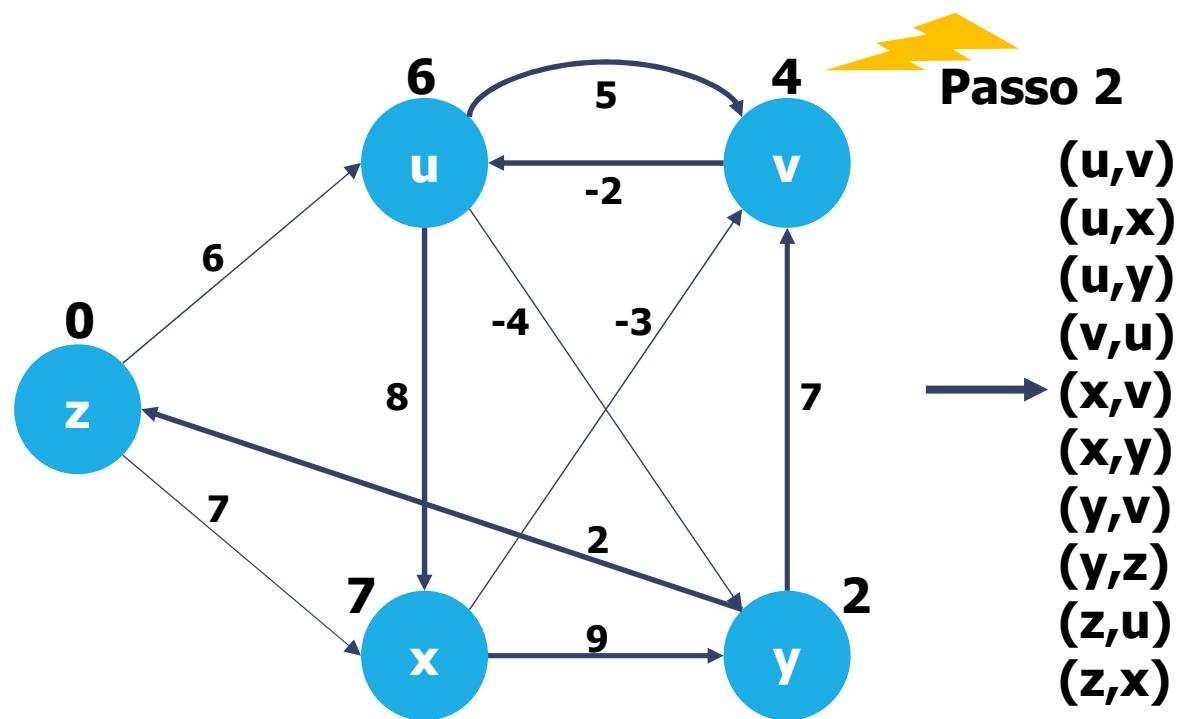
- (u,v)
 → (u,x)
 → (u,y)
 → (v,u)
 → (x,v)
 → (x,y)
 → (y,v)
 → (y,z)
 → (z,u)
 → (z,x)

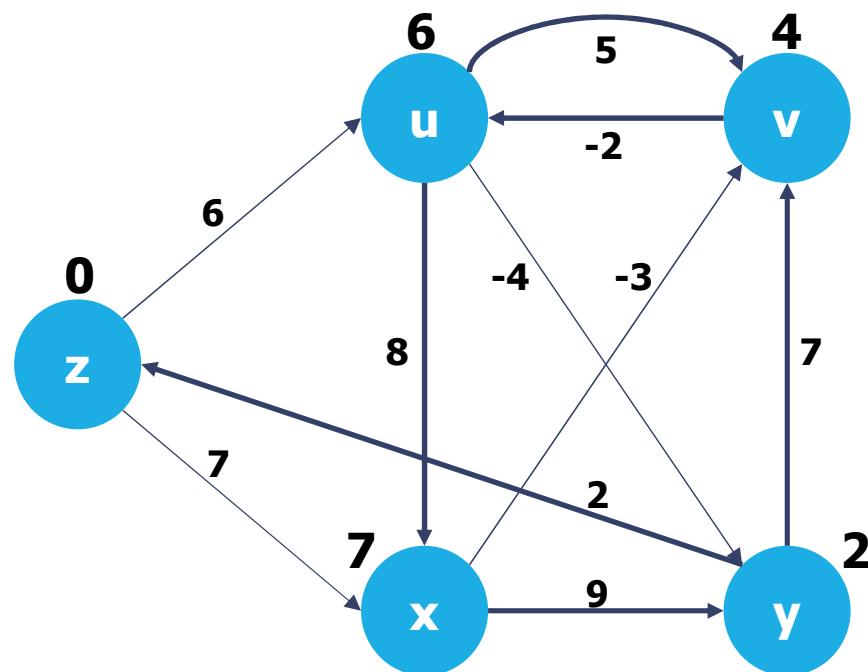




Passo 2

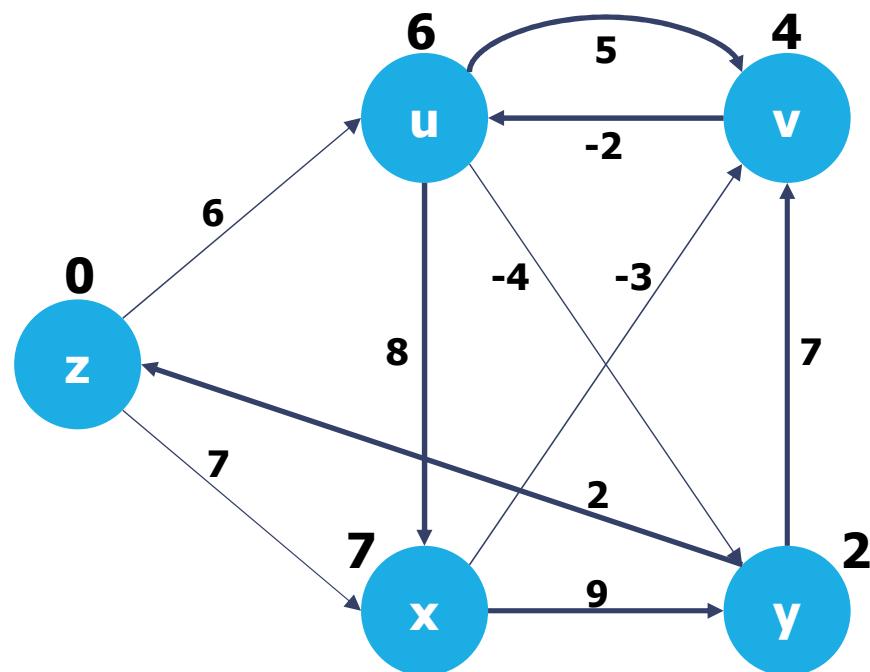
- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)





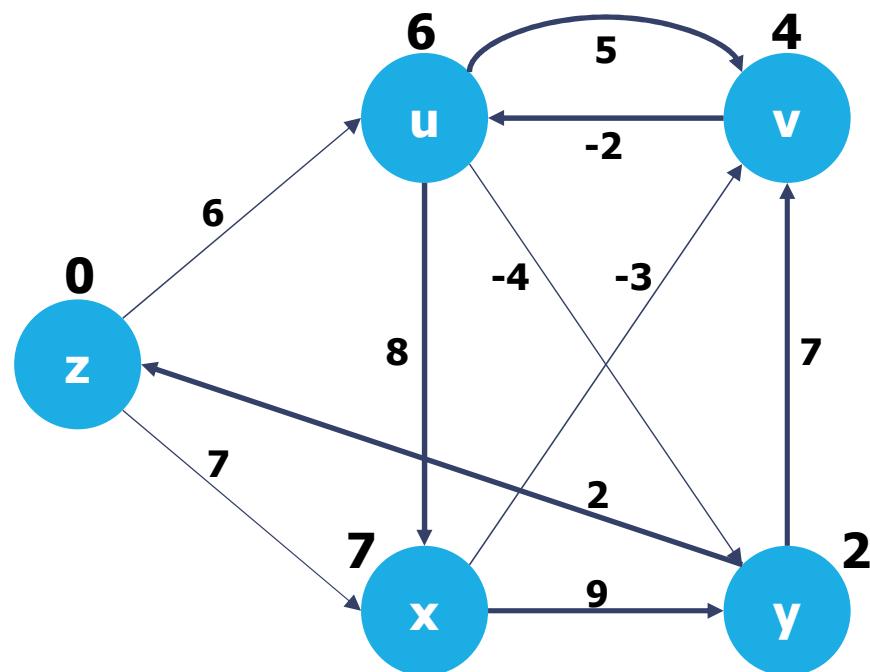
Passo 2

(u,v)
 (u,x)
 (u,y)
 (v,u)
 (x,v)
 $\rightarrow (x,y)$
 (y,v)
 (y,z)
 (z,u)
 (z,x)



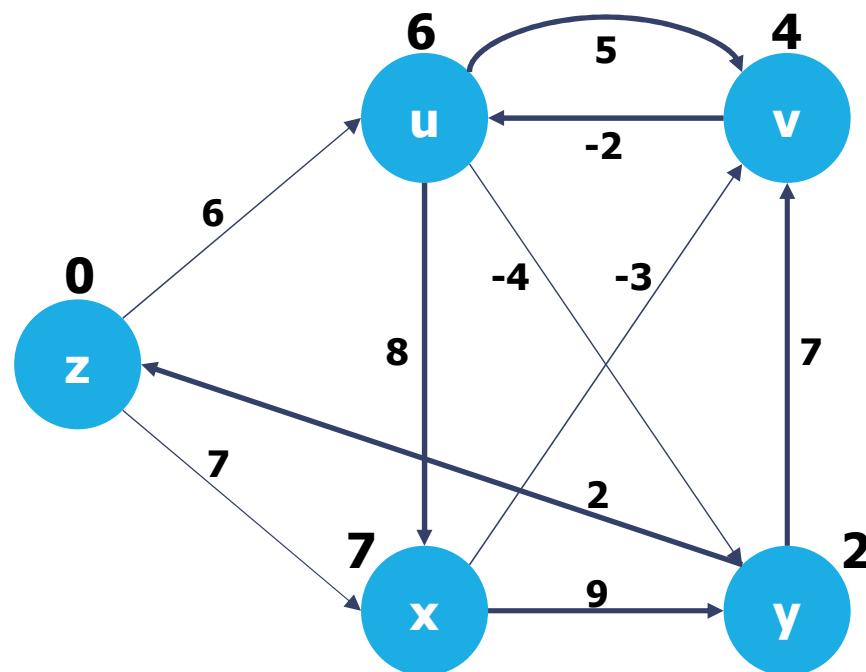
Passo 2

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



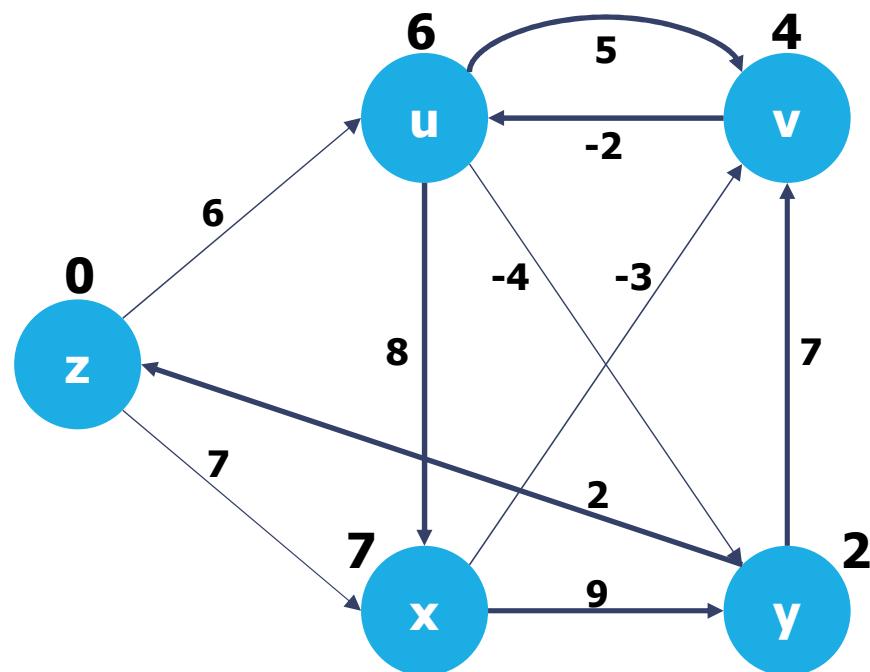
Passo 2

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



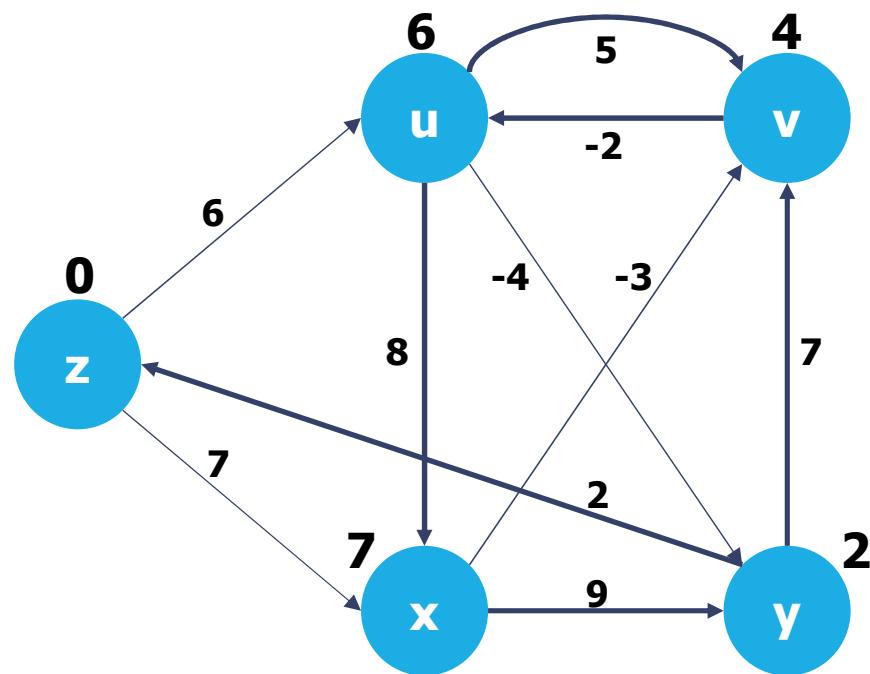
Passo 2

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- \rightarrow (z,u)
- (z,x)



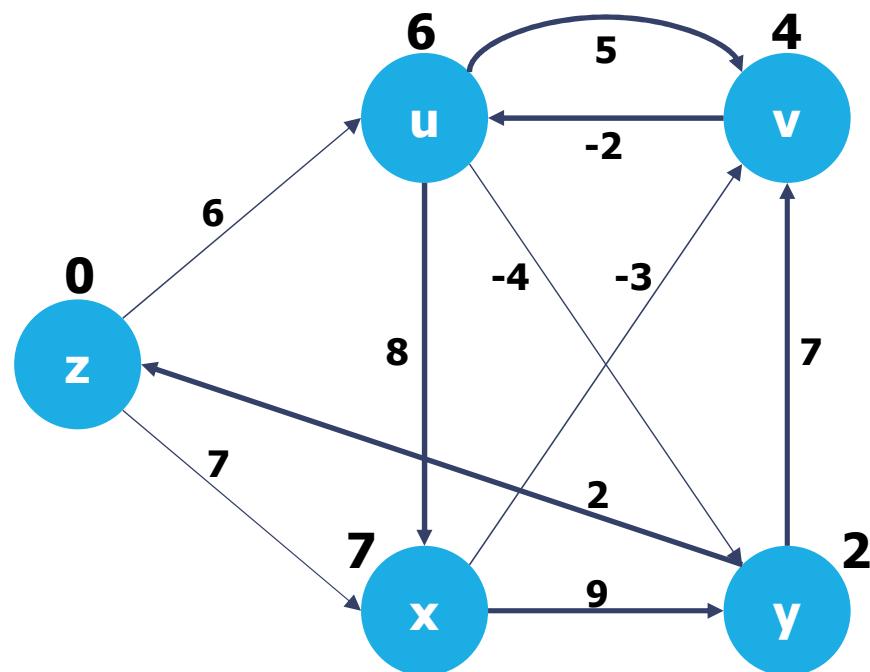
Passo 2

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



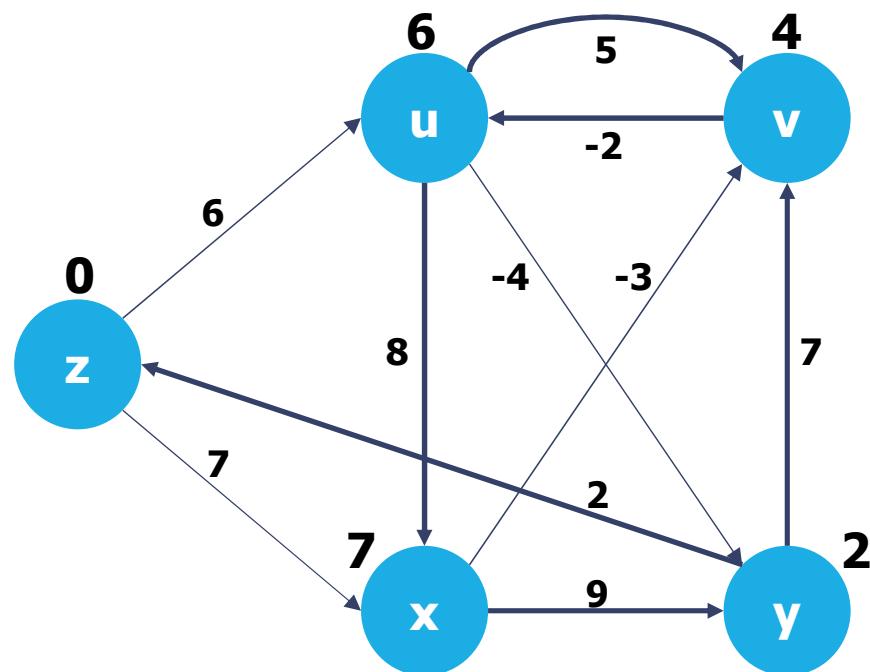
Passo 3

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



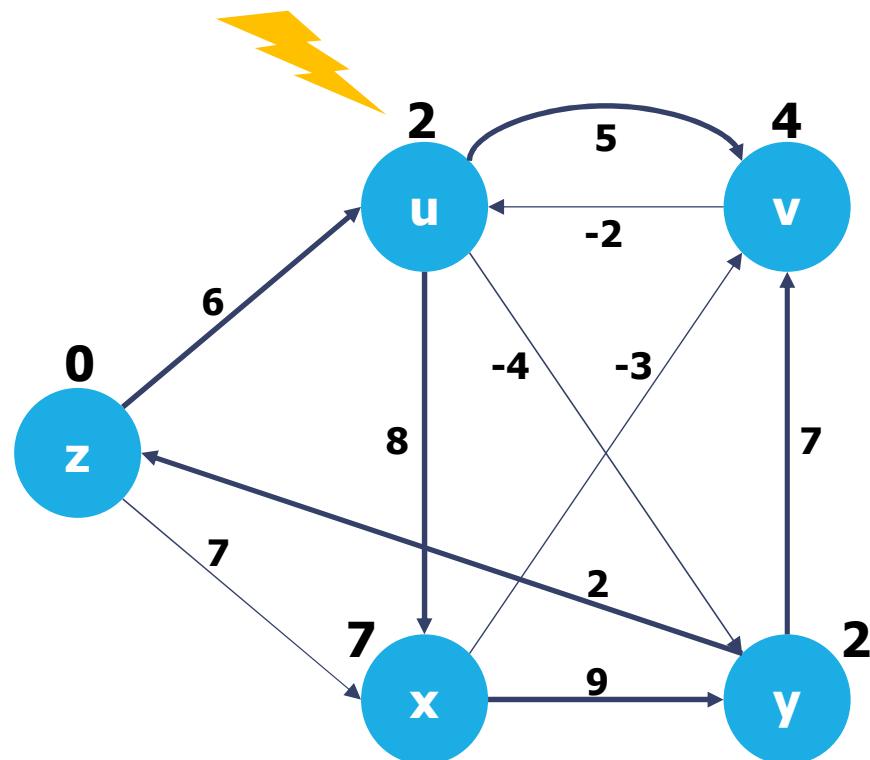
Passo 3

$\xrightarrow{\hspace{1cm}}$
 (u,v)
 (u,x)
 (u,y)
 (v,u)
 (x,v)
 (x,y)
 (y,v)
 (y,z)
 (z,u)
 (z,x)



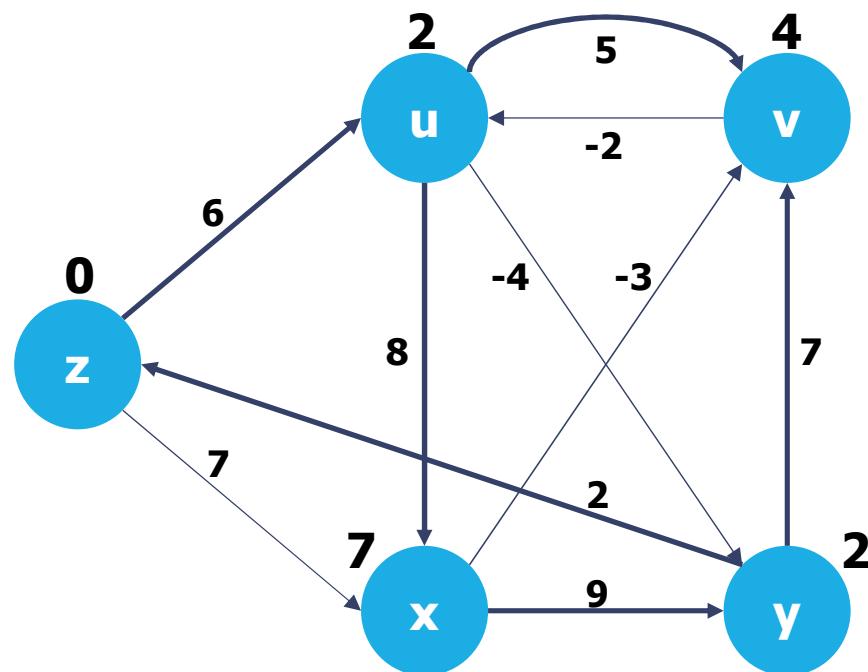
Passo 3

- \rightarrow (u,v)
 (u,x)
 (u,y)
 (v,u)
 (x,v)
 (x,y)
 (y,v)
 (y,z)
 (z,u)
 (z,x)



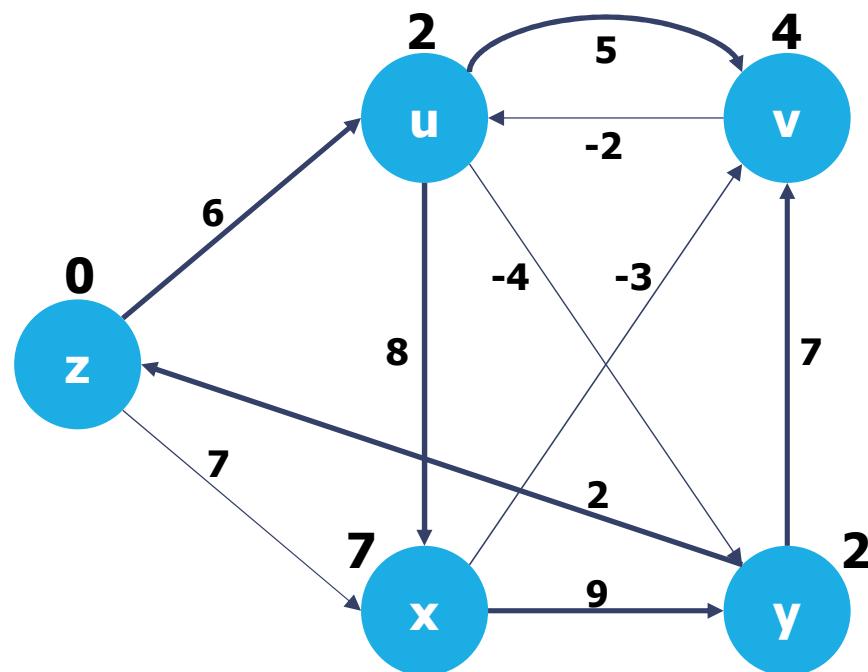
Passo 3

(u,v)
 (u,x)
 (u,y)
 $\rightarrow (v,u)$
 (x,v)
 (x,y)
 (y,v)
 (y,z)
 (z,u)
 (z,x)



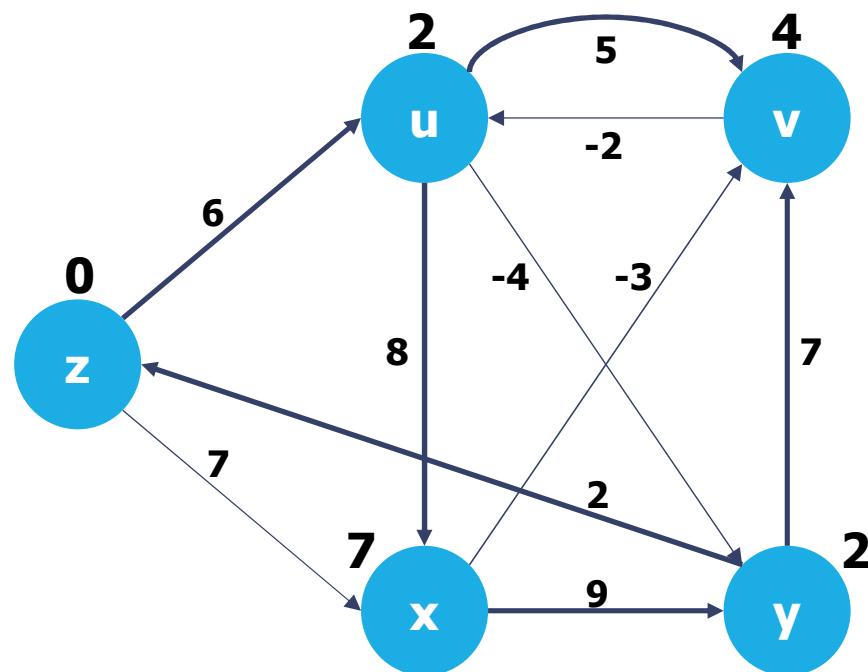
Passo 3

(u,v)
 (u,x)
 (u,y)
 (v,u)
 $\rightarrow (x,v)$
 (x,y)
 (y,v)
 (y,z)
 (z,u)
 (z,x)



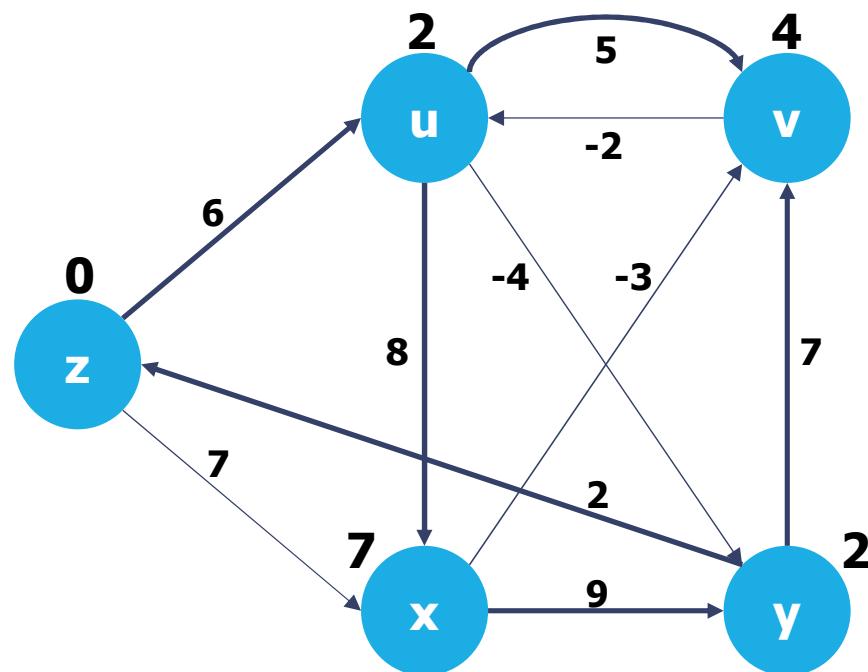
Passo 3

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



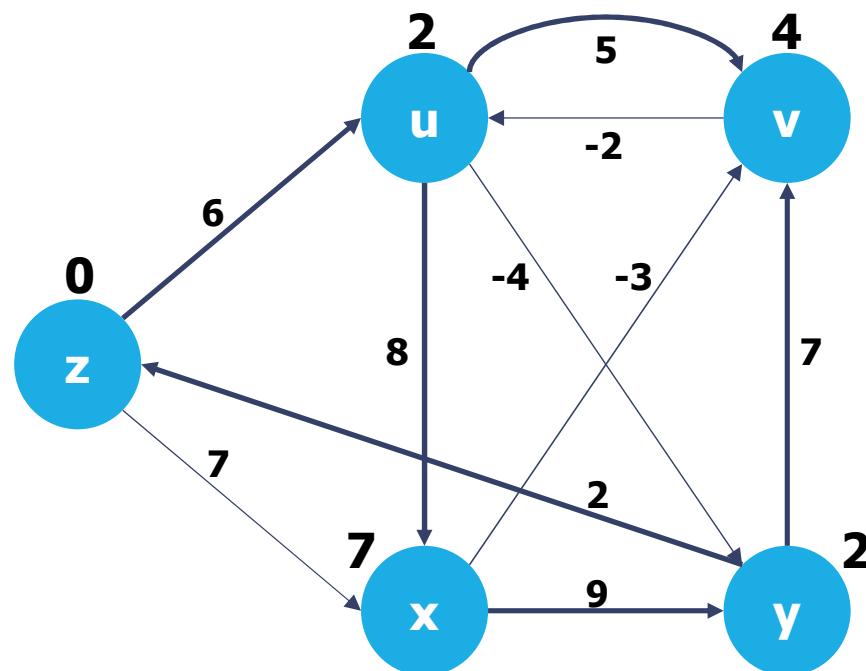
Passo 3

(u,v)
 (u,x)
 (u,y)
 (v,u)
 (x,v)
 (x,y)
 $\rightarrow (y,v)$
 (y,z)
 (z,u)
 (z,x)



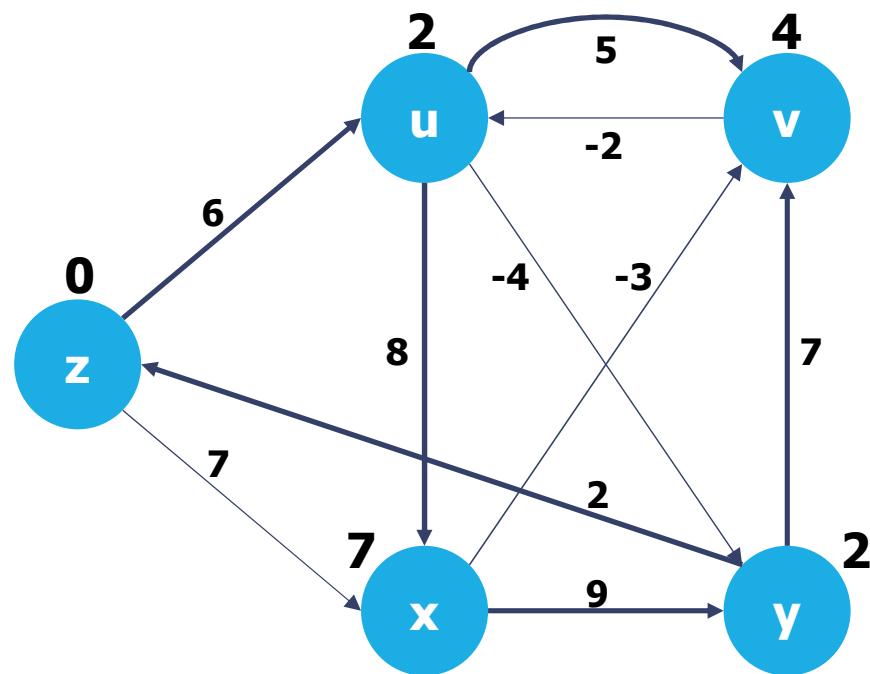
Passo 3

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



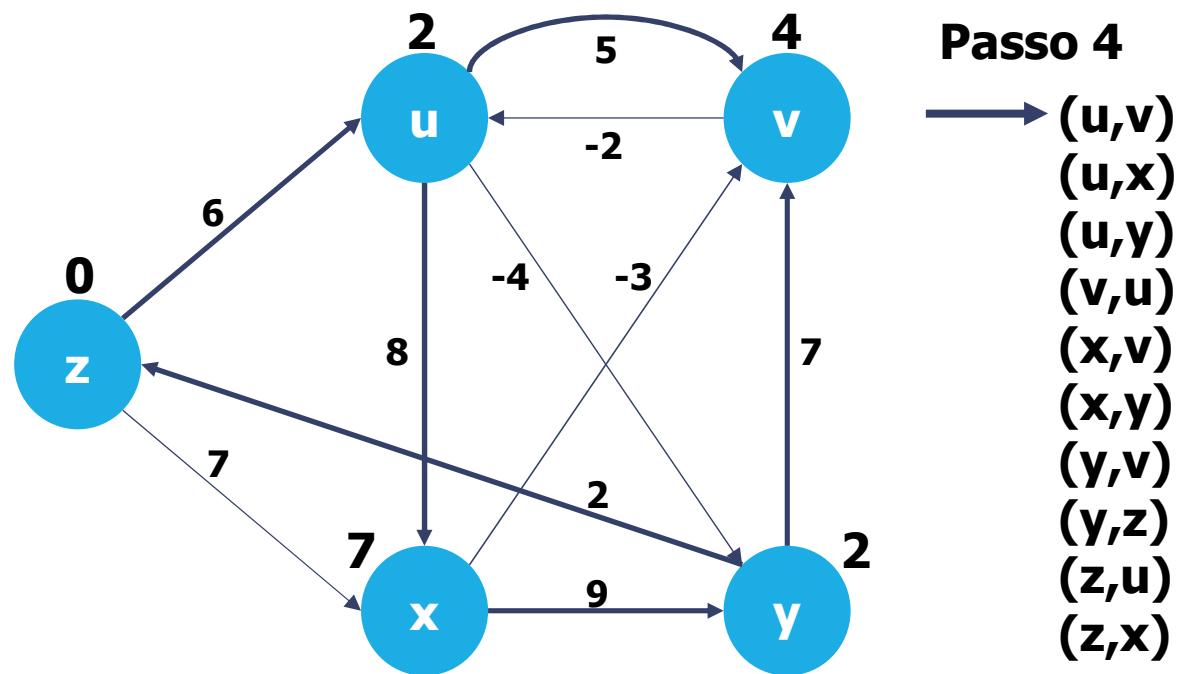
Passo 3

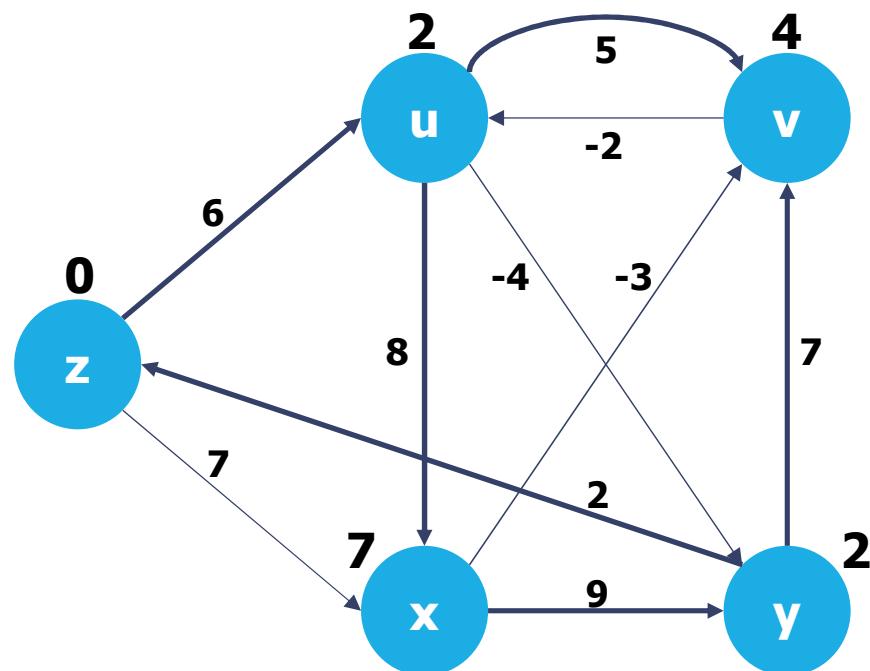
- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)**
- (z,x)**



Passo 3

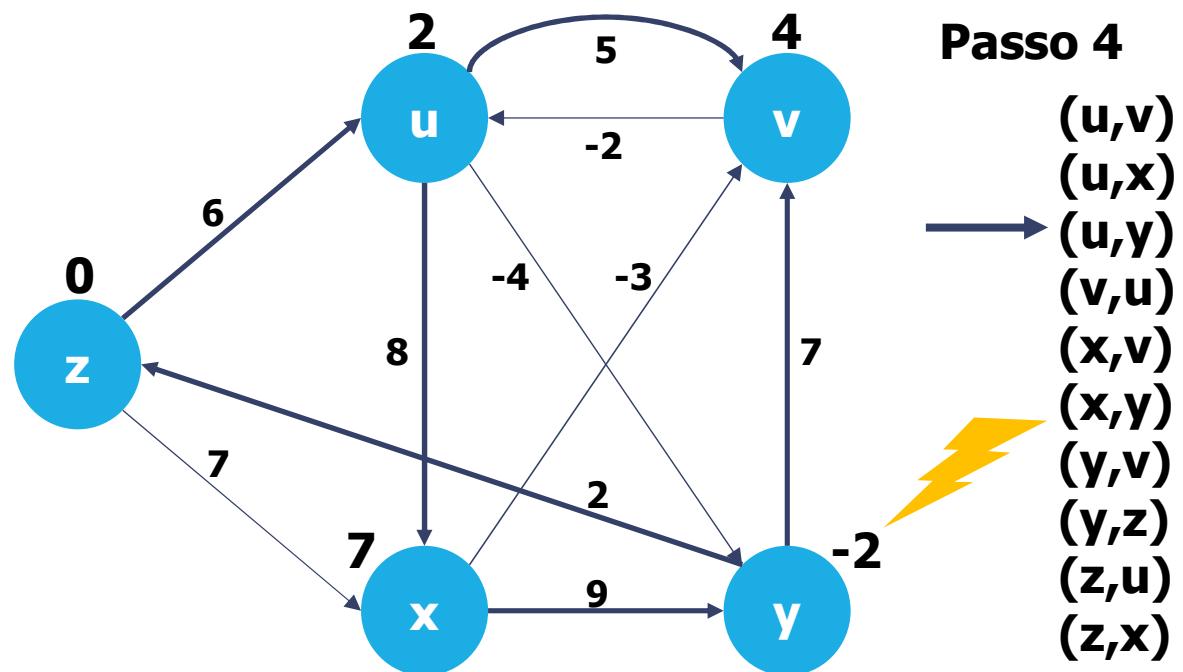
- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)

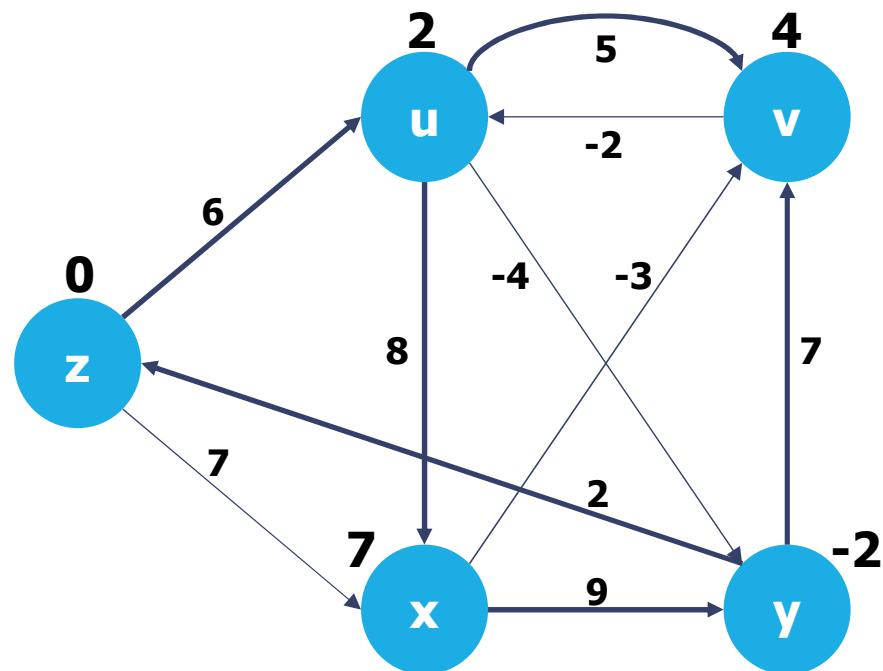




Passo 4

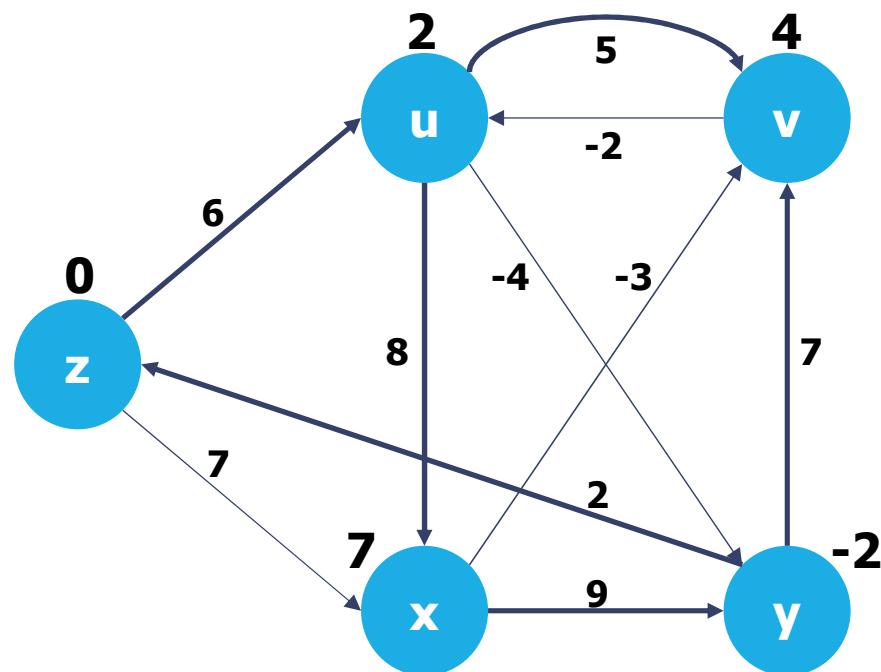
$\xrightarrow{\hspace{1cm}}$
 (u,v)
 (u,x)
 (u,y)
 (v,u)
 (x,v)
 (x,y)
 (y,v)
 (y,z)
 (z,u)
 (z,x)





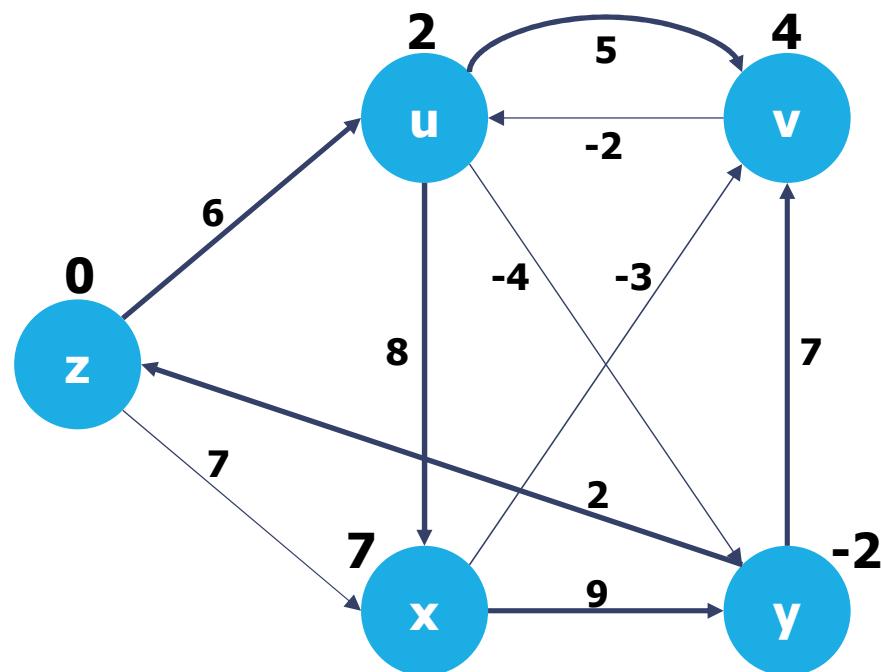
Passo 4

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



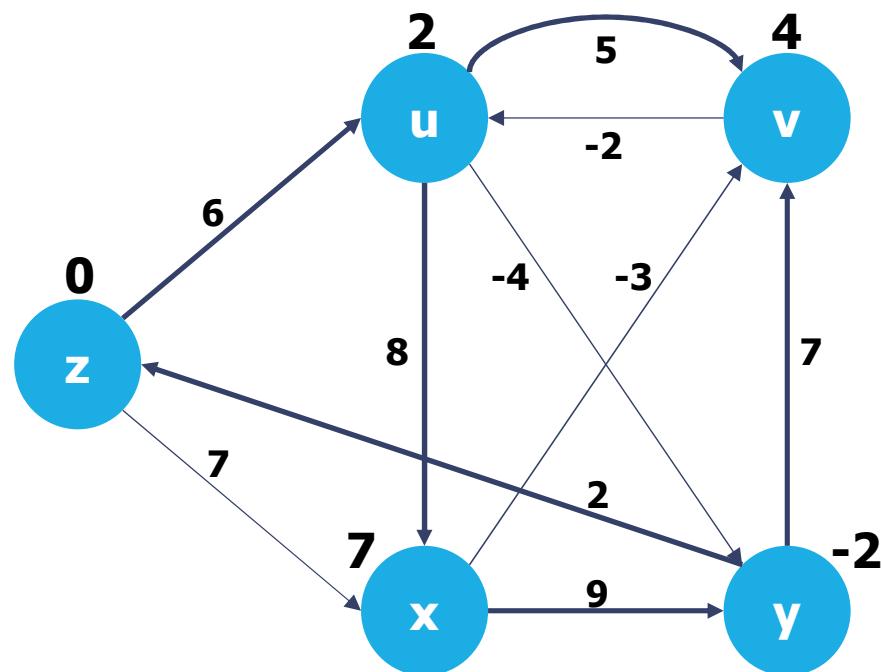
Passo 4

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



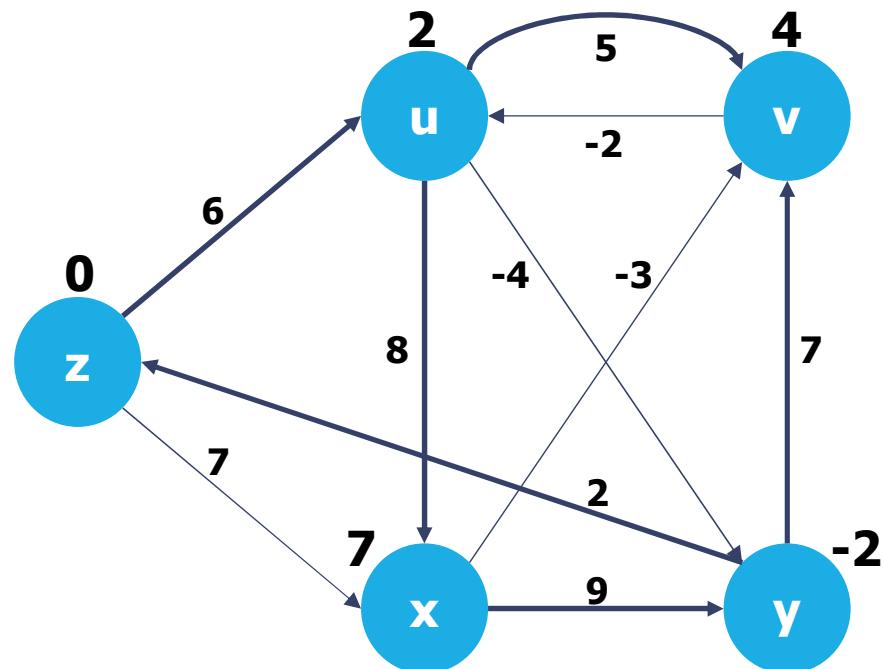
Passo 4

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



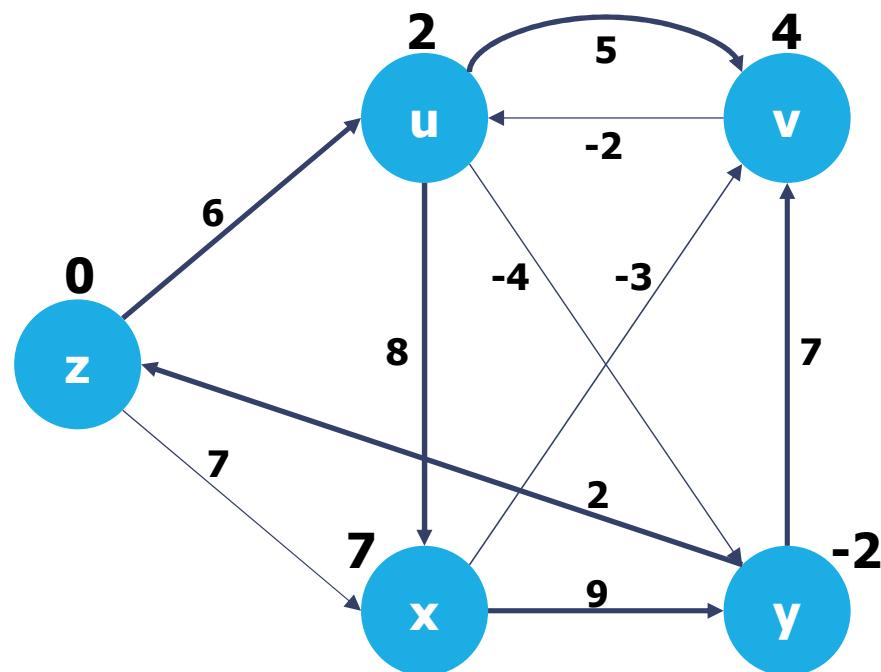
Passo 4

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



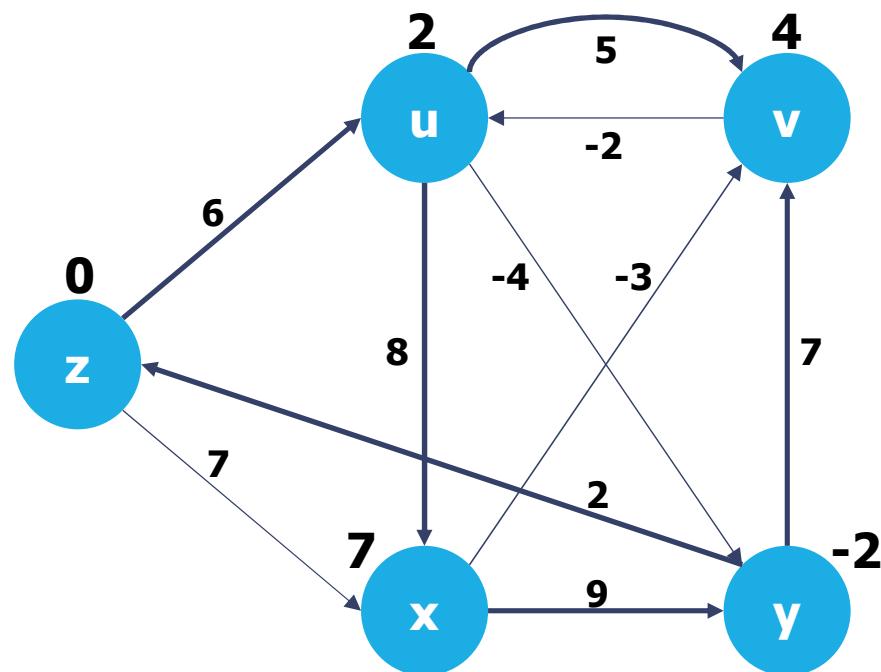
Passo 4

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)**
- (z,u)
- (z,x)



Passo 4

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)



Passo 4

- (u,v)
- (u,x)
- (u,y)
- (v,u)
- (x,v)
- (x,y)
- (y,v)
- (y,z)
- (z,u)
- (z,x)

Al $|V|$ -esimo passo di rilassamento non diminuisce alcuna stima:
terminazione con soluzione ottima.

Ricordare: la terminazione può anche avvenire prima.

Complessità

$O(|V|)$

- Inizializzazione
- $|V|-1$ passi in ciascuno dei quali si rilassano tutti gli archi
- $|V|$ -esimo passo di controllo in cui si rilassano tutti gli archi

$O(|V||E|)$

$$T(n) = O(|V| |E|).$$

$O(|E|)$

```
void GRAPHspBF(Graph G, int id){  
    int v, i, negcycfound;  
    link t;  
    int *st, *d;  
  
    st = malloc(G->V*sizeof(int));  
    d = malloc(G->V*sizeof(int));  
  
    for (v = 0; v < G->V; v++) {  
        st[v] = -1;  
        d[v] = maxWT;  
    }  
  
    d[id] = 0;  
    st[id] = id;
```

```

for (i=0; i<G->V-1; i++)
    for (v=0; v<G->V; v++)
        if (d[v] < maxWT)
            for (t=G->adj[v]; t!=G->z ; t=t->next)
                if (d[t->v] > d[v] + t->wt) {
                    d[t->v] = d[v] + t->wt;
                    st[t->v] = v;
                }
negcycfound = 0;
for (v=0; v<G->V; v++)
    if (d[v] < maxWT)
        for (t=G->adj[v]; t!=G->z ; t=t->next)
            if (d[t->v] > d[v] + t->wt)
                negcycfound = 1;

```

```

if (negcycfound == 0) {
    printf("\n Shortest path tree\n");
    for (v = 0; v < G->V; v++)
        printf("Parent of %s is %s \n",
               STsearchByIndex(G->tab, v),
               STsearchByIndex (G->tab, st[v]));
    printf("\n Min.dist. from %s\n",
           STsearchByIndex (G->tab, s));
    for (v = 0; v < G->V; v++)
        printf("%s: %d\n", STsearchByIndex (G->tab, v), d[v]);
}
else
    printf("\n Negative cycle found!\n");
}

```

Applicazione: arbitrage

In Economia e Finanza si definisce «**arbitrage**» l'acquisto e la vendita in simultanea di beni (asset) su mercati diversi per sfruttare piccole differenze nei prezzi.

Si applica ad asset quali: azioni (stocks), materie prime (commodities), valute (currencies).

L'arbitrageur ha la possibilità di guadagno a costo zero e senza rischi.

L'arbitrage offre un meccanismo che garantisce che i prezzi non devono sostanzialmente dal valore corretto per periodi lunghi.

Arbitrage trading program (ATP):

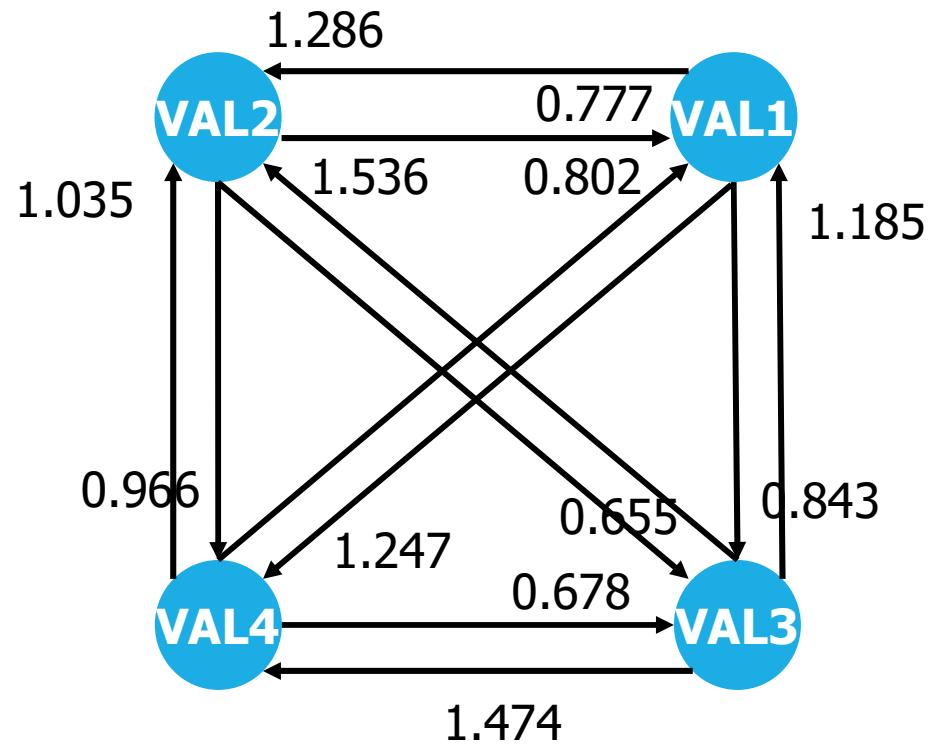
- osserva i prezzi sui mercati
- rileva anomalie di prezzo in millisecondi ("electronic eye")
- applica Bellman-Ford al grafo per rilevare cicli a peso negativo
- effettua le transazioni.

Esempio: mercato delle valute

	VAL1	VAL2	VAL3	VAL4
VAL1	1.000	1.286	0.843	1.247
VAL2	0.777	1.000	0.655	0.966
VAL3	1.185	1.526	1.000	1.474
VAL4	0.802	1.035	0.678	1.000

Tassi di cambio tra 4 valute

$$\forall i \neq j \text{ } VAL_i/VAL_j \neq 1/(VAL_j/VAL_i)$$



$$1000 \text{ VAL2} = 777 \text{ VAL1} = 655,11 \text{ VAL3} = 1006,10 \text{ VAL2}$$

Guadagno di 6,10 VAL2 \Rightarrow arbitrage

Sul ciclo

$$\text{VAL2} - \text{VAL1} - \text{VAL3} - \text{VAL2}$$

il prodotto dei tassi di cambio è

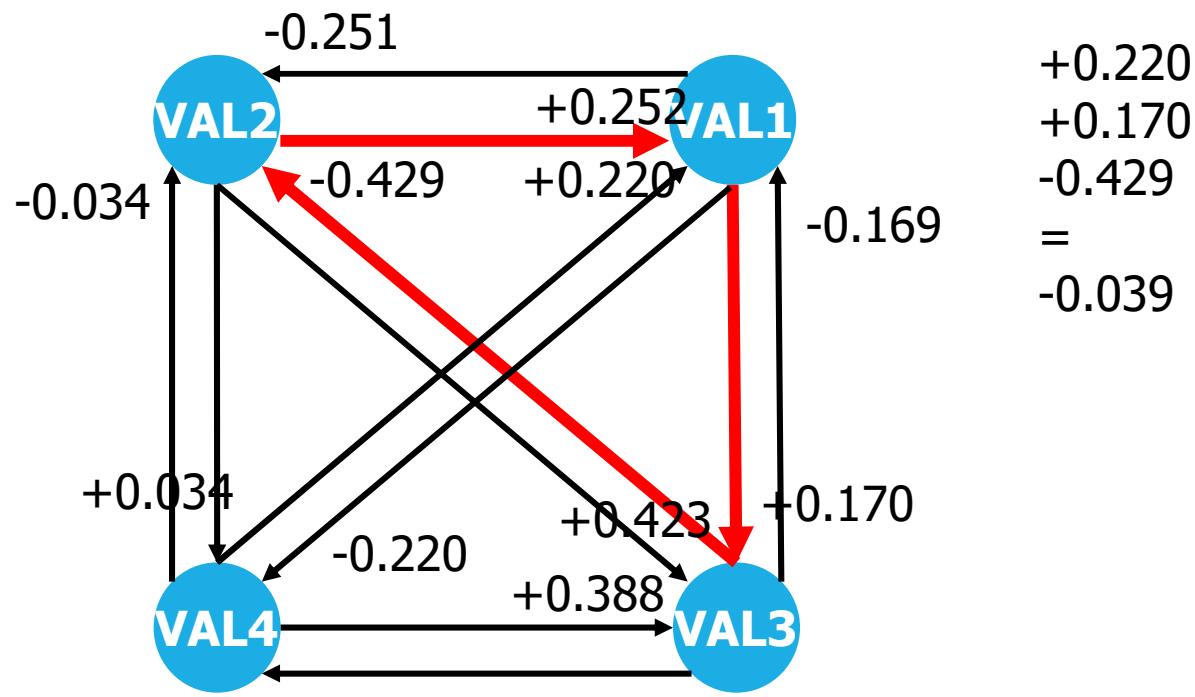
$$0.777 * 0.843 * 1.536 = 1,0061 > 1.0$$



c'è arbitrage quando \exists ciclo a peso > 1

Modello:

- grafo orientato pesato completo
- peso degli archi = $-\ln(\text{tasso di cambio})$
- il ciclo con prodotto dei cambi > 1 diventa un ciclo in cui la somma dei logaritmi ha peso negativo
- algoritmo di Bellman-Ford per rilevare il ciclo a peso negativo



$$\begin{aligned}
 &+0.220 \\
 &+0.170 \\
 &-0.429 \\
 &= \\
 &-0.039
 \end{aligned}$$

-0.388

14 I CAMMINI MINIMI

Riferimenti

- Principi:
 - Sedgewick Part 5 21.1
 - Cormen 24.1
- Algoritmo di Dijkstra:
 - Sedgewick Part 5 21.2
 - Cormen 24.4
- Cammini minimi e massimi in DAG:
 - Sedgewick Part 5 21.4
 - Cormen 24.3
- Algoritmo di Bellman-Ford:
 - Sedgewick Part 5 21.7
 - Cormen 24.2

Esercizi di teoria

- 12. Visite dei grafi e applicazioni
 - 12.1 Algoritmo di Dijkstra
 - 12.2 Algoritmo per i DAG pesati
 - 12.3 Algoritmo di Bellman-Ford

