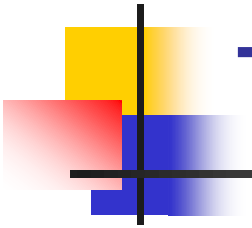
Problemi di ricerca e ottimizzazione

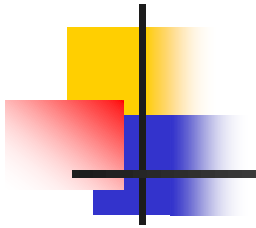
Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino



Tipologie di problemi

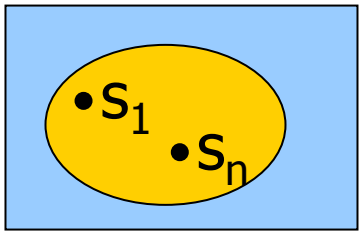
Problemi di calcolo:

- soluzione con procedimento matematico che porta, senza scelte e con un numero finite di passi, alla soluzione
- Esempi:
  - fattoriale
  - determinante
  - numeri di Fibonacci, di Catalan, di Bell etc.



Problemi di ricerca:

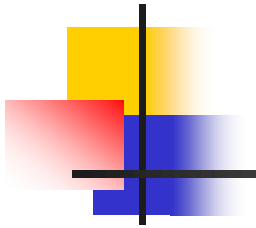
- dati:
  - S: spazio (insieme) delle soluzioni possibili
  - V: spazio delle soluzioni valide
  - in generale $V \subset S$
- appurare se $V = \varnothing$
- elencare gli elementi di V
  - almeno 1
  - tutti in caso di enumerazione.



S: spazio delle soluzioni

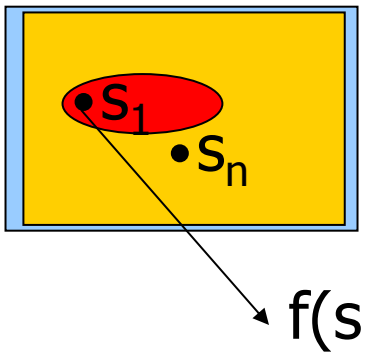
V: soluzioni valide

- Esempi:
  - insiemi delle parti che soddisfano condizione,
  - 8 regine,
  - Sudoku,
  - in grafo tutti i cammini semplici da un vertice



Problemi di ottimizzazione:

- $S \equiv V$
- 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

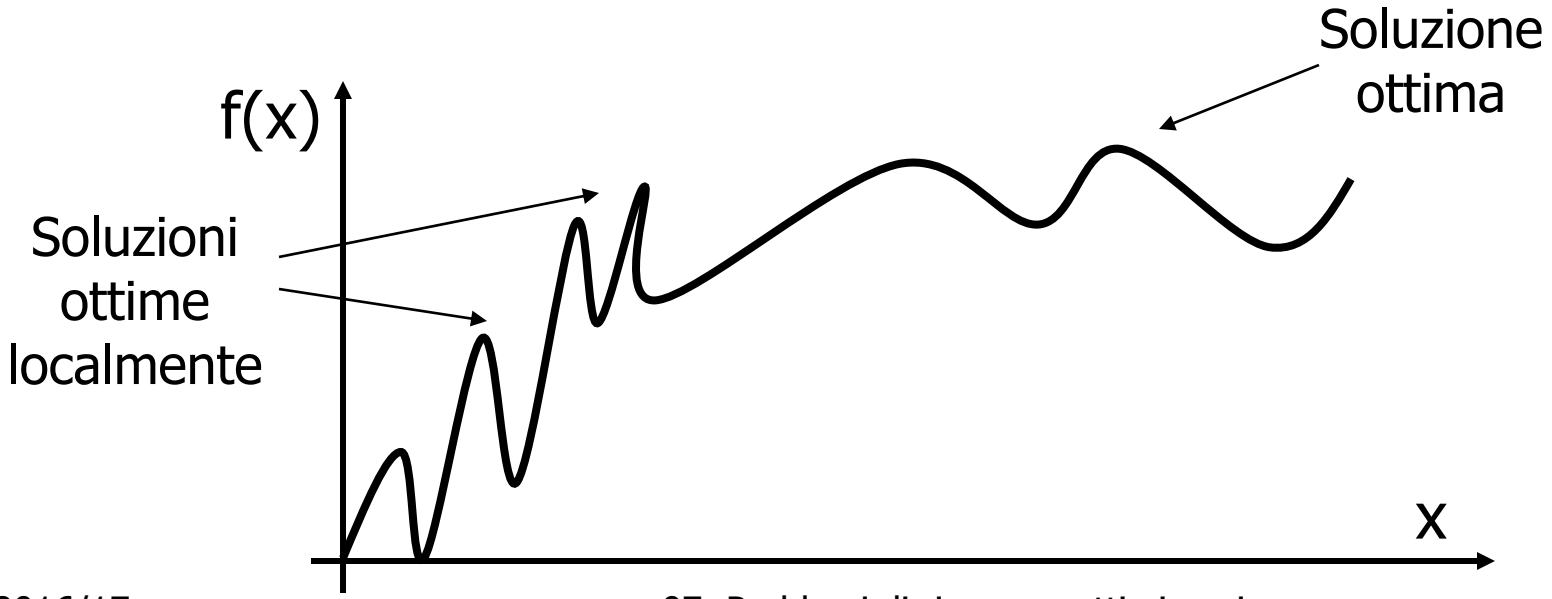
≡

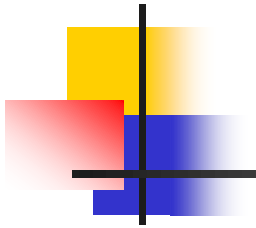
V: soluzioni valide

- 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

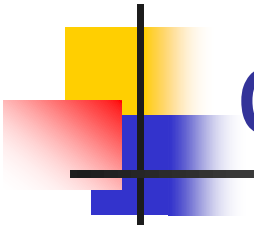




Risposte possibili:

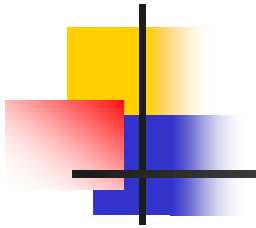
- contare il numero di soluzioni valide
- trovare almeno una soluzione valida
- trovare tutte le soluzioni valide
- trovare tra le soluzioni valide quella (o quelle) ottima(e) secondo un criterio di ottimalità.

Esplorazione dello spazio delle soluzioni



Approccio incrementale:

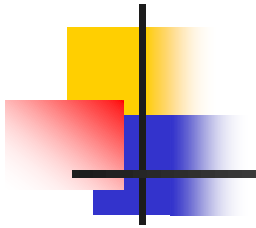
- soluzione iniziale vuota
- estensione della soluzione mediante applicazione di scelte
- terminazione al raggiungimento di soluzione



Algoritmo generico che usa una struttura dati SD:

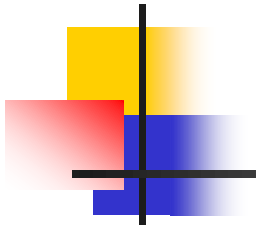
Ricerca():

- metti la soluzione iniziale in SD
- finché SD non diventa vuoto:
  - 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:

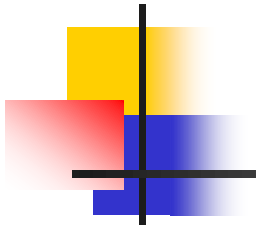
- una coda (FIFO), la ricerca è in ampiezza (breadth-first)
- una pila (LIFO), la ricerca è in profondità (depth-first)
- una coda a priorità, la ricerca è best-first.



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.



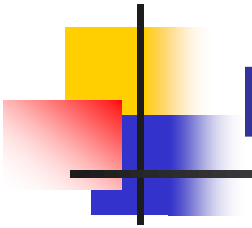
Approccio seguito: algoritmo di ricerca

- in profondità
- non informato
- completo
- ricorsivo.

Rappresentazione

Spazio delle soluzioni rappresentato come albero di ricerca:

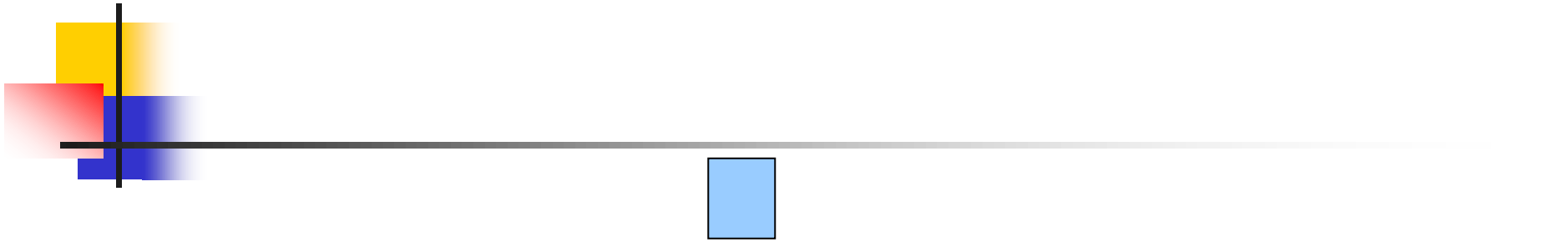
- di altezza n, dove n è la dimensione della soluzione
- di grado k, dove k è il massimo numero di scelte possibili
- la radice è la soluzione iniziale vuota
- i nodi intermedi sono etichettati con le soluzioni parziali
- le foglie sono le soluzioni. Una funzione determina se sono soluzioni valide

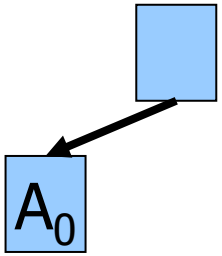


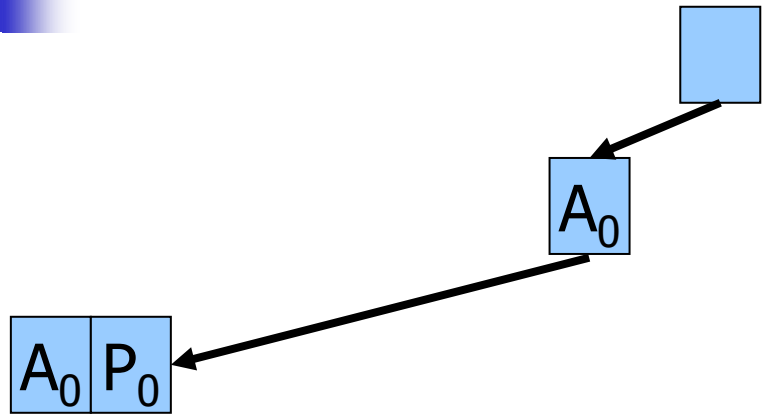
Esempio

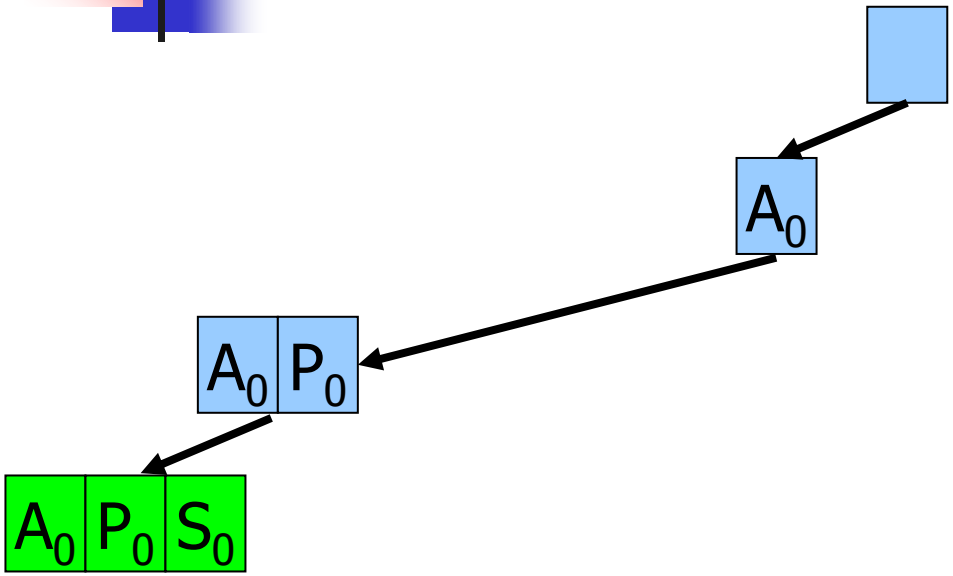
In un ristorante c'è un menu a prezzo fisso composto da antipasto, primo e secondo.
Il cliente può scegliere tra 2 antipasti A_0, A_1, 3 primi P_0, P_1 e P_2 e 2 secondi S_0, S_1.

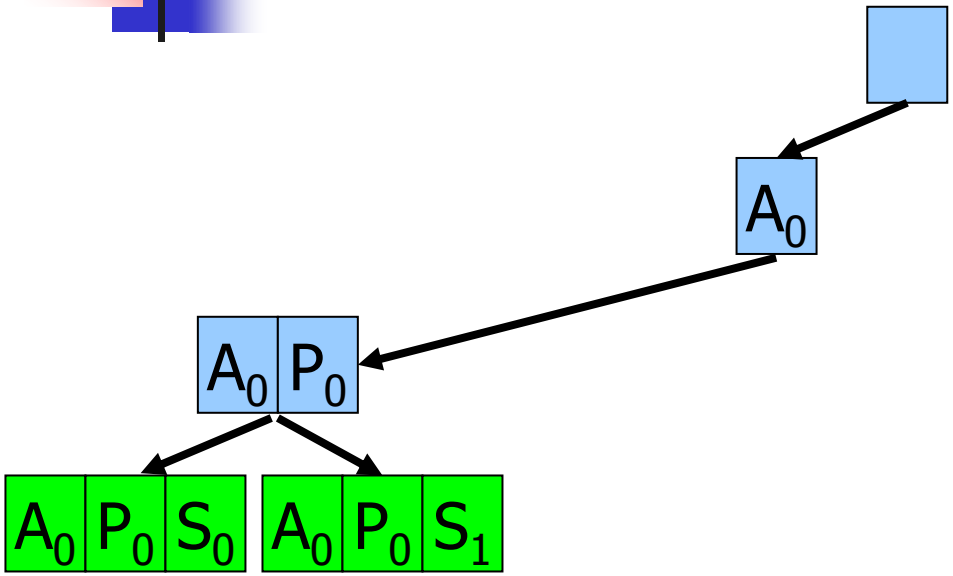
Quanti e quali pranzi diversi si possono scegliere con questo menu?

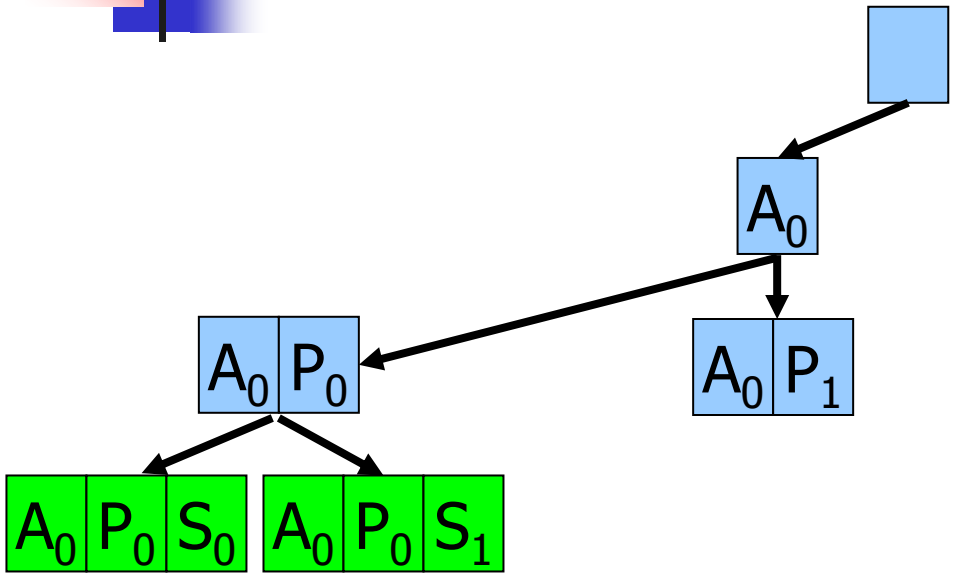


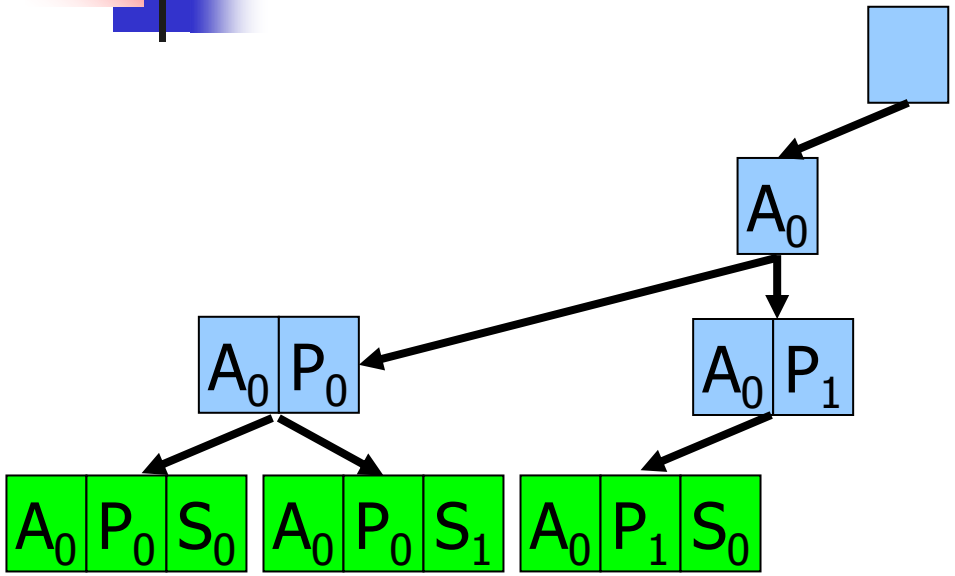

A_0

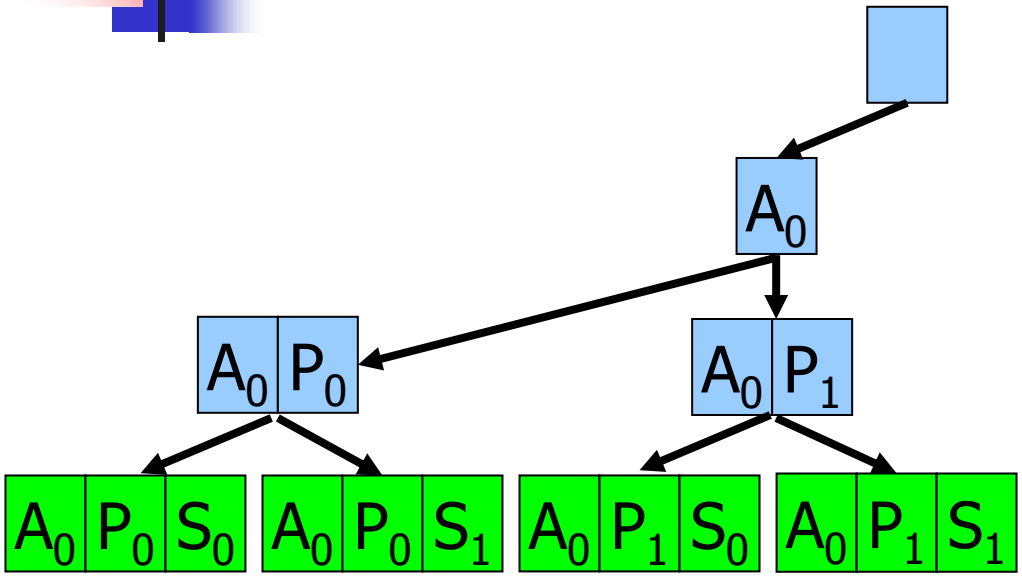

A_0
A_0 P_0

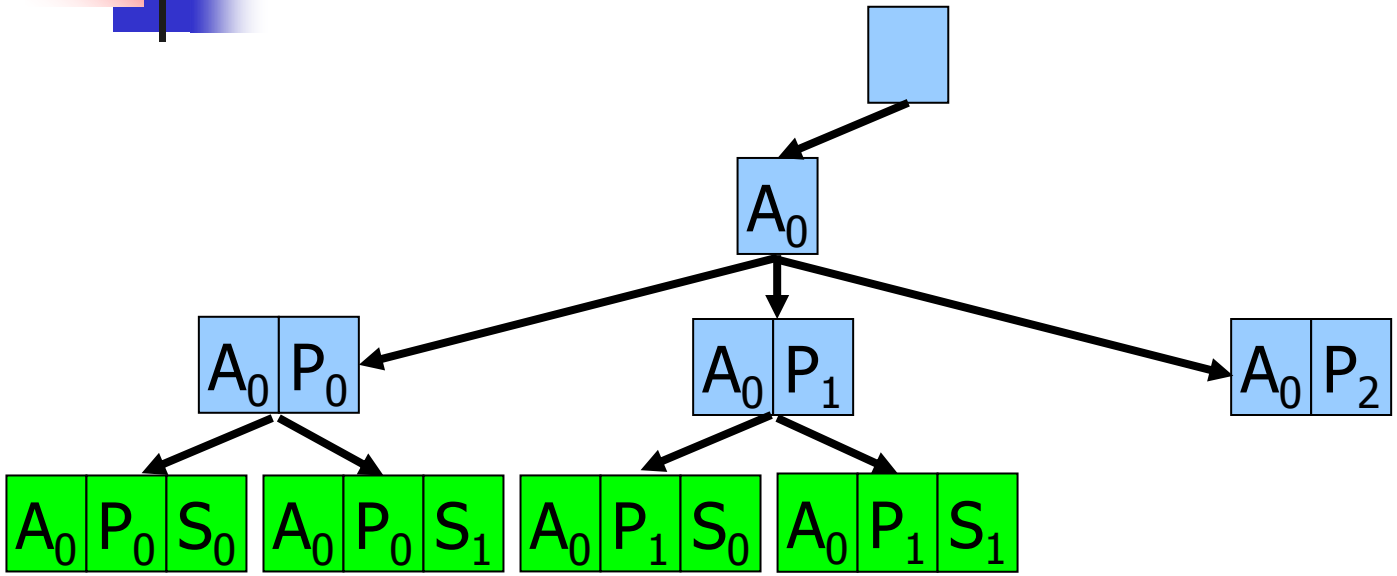

A_0
A_0 P_0
A_0 P_0 S_0

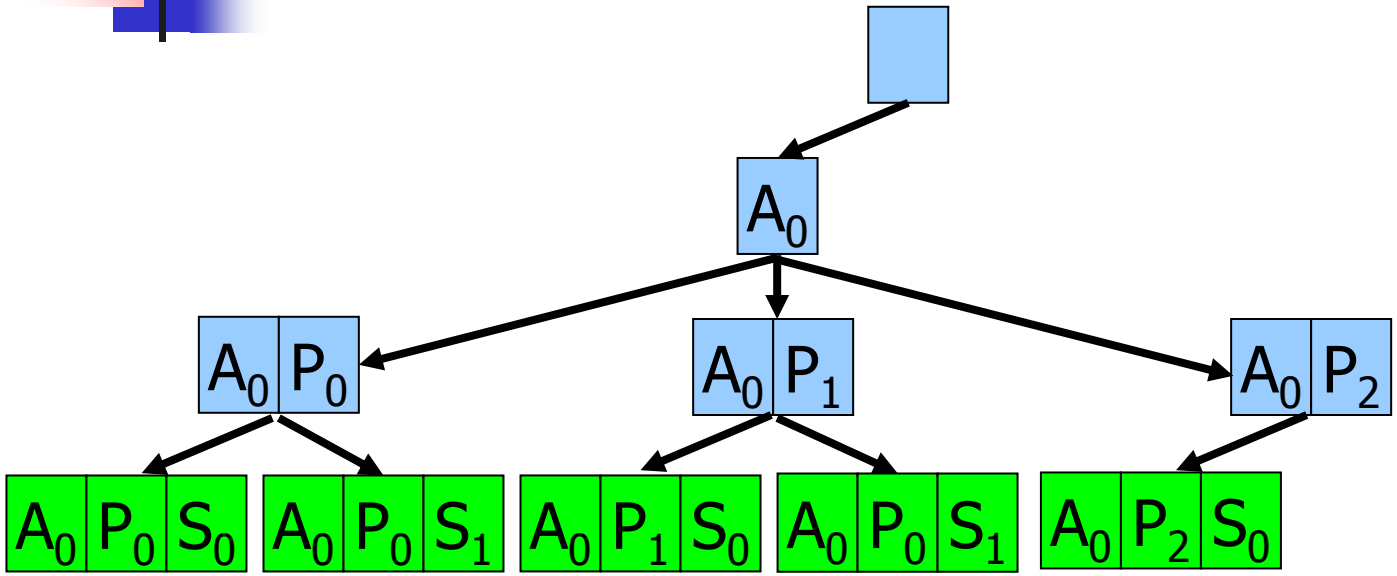

A_0
A_0 P_0
A_0 P_0 S_0
A_0 P_0 S_1

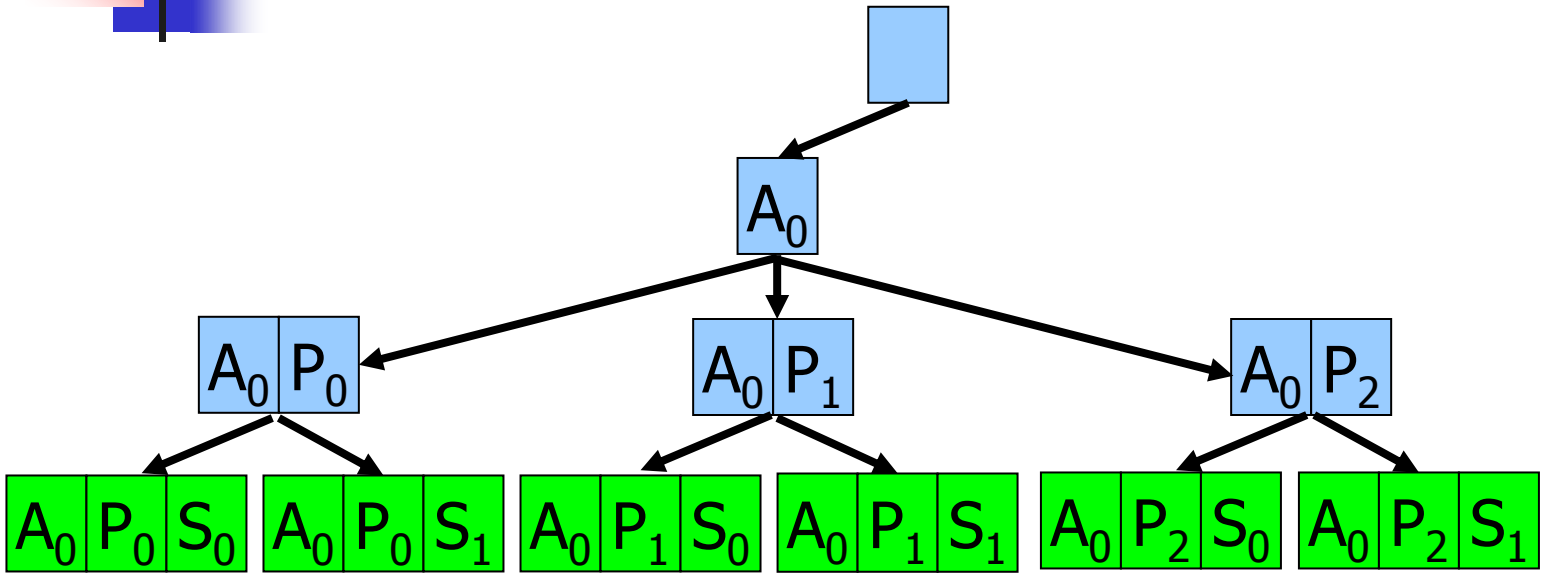

A0
A0 P0
A0 P1
A0 P0 S0
A0 P0 S1

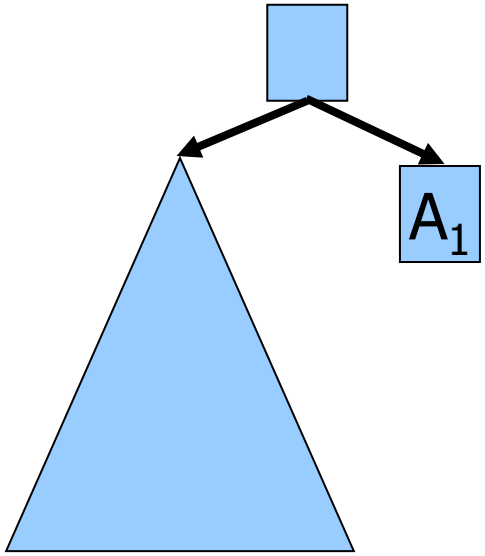

A_0
A_0 P_0
A_0 P_1
A_0 P_0 S_0
A_0 P_0 S_1
A_0 P_1 S_0


A_0
A_0 P_0
A_0 P_1
A_0 P_0 S_0
A_0 P_0 S_1
A_0 P_1 S_0
A_0 P_1 S_1


A0
A0 P0
A0 P1
A0 P2
A0 P0 S0
A0 P0 S1
A0 P1 S0
A0 P1 S1


A_0
A_0 P_0
A_0 P_1
A_0 P_2
A_0 P_0 S_0
A_0 P_0 S_1
A_0 P_1 S_0
A_0 P_0 S_1
A_0 P_2 S_0


A_0
A_0 P_0
A_0 P_1
A_0 P_2
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



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

Richiami di Calcolo Combinatorio

Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

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.

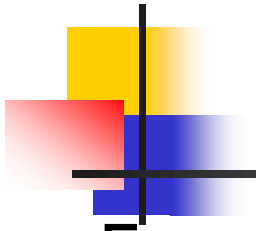
Principi base: addizione

Se un insieme S di oggetti è diviso in sottoinsiemi S_0 .. S_{n-1} a 2 a 2 disgiunti

$$S = S_0 \cup S_1 \cup .... S_{n-1} \;\; \&\& \; \forall \; i \neq j \; S_i \cap S_j \; = \varnothing$$

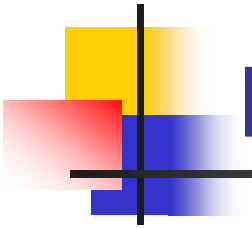
il numero degli oggetti in S può essere determinato sommando il numero degli oggetti in ciascuno degli insiemi S_0 .. S_{n-1}

$$|S| = \sum_{i=0}^{n-1} |Si|$$



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 dell'addizione

Numero di scelte = 4 + 5 = 9

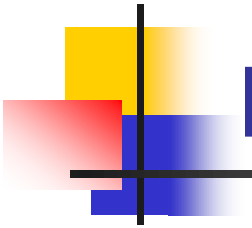
Principi base: moltiplicazione

Dati n insiemi S_i $(0 \leq i < n)$ ciascuno di cardinalità $|S_i|$, il numero di n-uple ordinate $(s_0 \ldots s_{n-1})$ con $s_0 \in S_0 \ \ldots \ s_{n-1} \in S_{n-1}$ 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 $(x_0 \ldots x_{n-1})$ può essere fatta in $p_0 \cdot p_1 \ldots \cdot p_{n-1}$ 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 della moltiplicazione

Numero di scelte = 2 x 3 x 2 x 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

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 ($0 \leq k \leq n$).

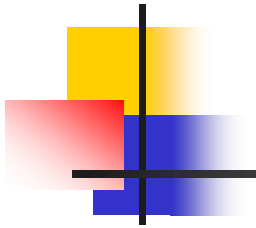
insieme

l'ordinamento conta

Vi sono

$$D_{n,k} = \frac{n!}{(n-k)!} = n \cdot (n-1) \cdot \ldots\ldots \cdot (n-k+1)$$

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?

no cifre ripetute

n = 4

k = 2

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

nessun limite superiore!

ripetizioni

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.

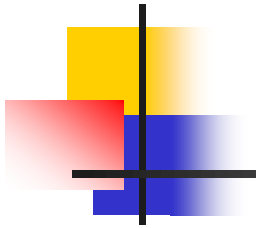
insieme

l'ordinamento conta

Vi sono

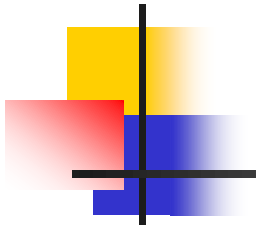
$$D'_{n,k} = n^k$$

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?

Ogni bit può assumere valore 0 o 1.

k = 4

n = 2

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

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.

Vi sono

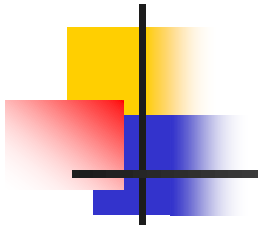
$$P_n = D_{n,n} = n!$$

permutazioni semplici di n oggetti.

no ripetizioni

insieme

l'ordinamento conta



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

rappresentazione posizionale: l'ordine conta!

Quanti e quali sono gli anagrammi di ORA (parola di 3 lettere distinte)?

n = 3

no ripetizioni

Modello: permutazioni semplici

$P_3 = 3! = 6$

Soluzione
{ ORA, OAR, ROA, RAO, AOR, ARO }

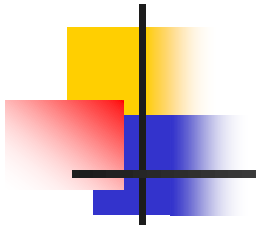
Permutazioni con ripetizione

elementi ripetuti

Dato un multiset di n oggetti di cui α uguali fra loro, β uguali fra loro, etc., il numero di permutazioni distinte con oggetti ripetuti è:

l'ordinamento conta

$$P_n^{(\alpha, \beta, ..)} = \frac{n!}{(\alpha! \cdot \beta! \ \ ...)}$$



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 ripetuti un numero diverso di volte oppure 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

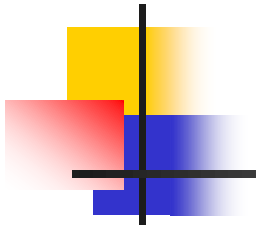
no ripetizioni

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 ($0 \leq k \leq n$)

insieme

l'ordinamento non conta

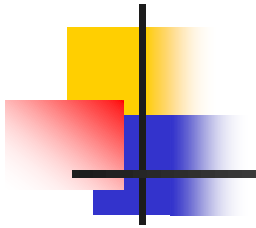
Il numero di combinazioni di n elementi a k a k è al numero di disposizioni di n elementi a k a k diviso per il numero di permutazioni di k elementi.



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.

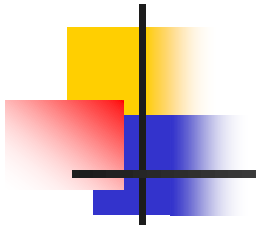


Vi sono:

coefficiente binomiale

$$C_{n,k} = \binom{n}{k} = \frac{D_{n,k}}{P_k} = \frac{n!}{k!(n-k)!}$$

combinazioni semplici di n oggetti a k a k.



Definizione ricorsiva del coefficiente binomiale:

$$\binom{n}{0} = \binom{n}{n} = 1$$

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Esempi

l'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

nessun limite superiore!

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.

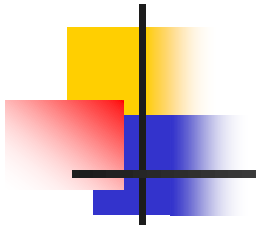
insieme

l'ordinamento non conta

Vi sono

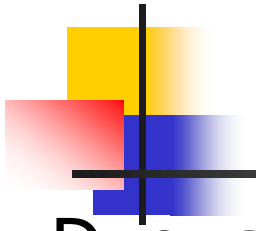
$$C'_{n,k} = \frac{(n+k-1)!}{k!(n-1)!}$$

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

l'ordine non conta!

Lanciando contemporaneamente 2 dadi, quante sono le composizioni con cui si possono presentare le facce?

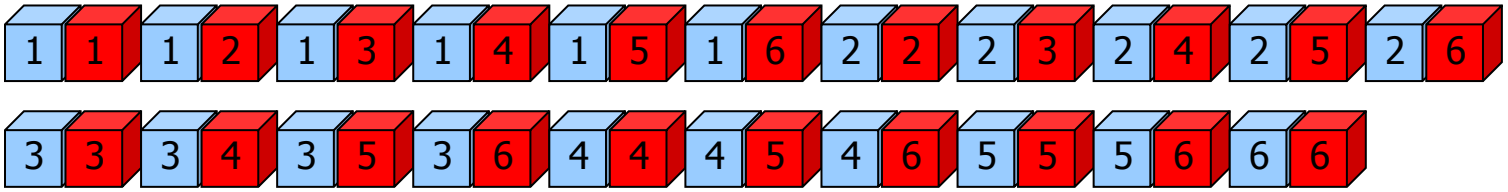
k = 2

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=card(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 non vuoti forma una partizione di I se e solo se valgono entrambe le seguenti condizioni:

- i blocchi sono a coppie disgiunti:

$$\forall\ S_i, S_j \in S \text{ con } i \neq j \quad S_i \cap S_j = \varnothing$$

- la loro unione è I:

$$I = \cup_i S_i$$

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}

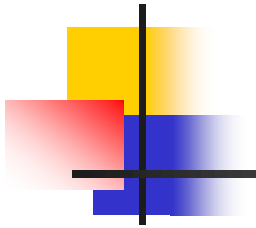
{1, 2}, {3, 4}

{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

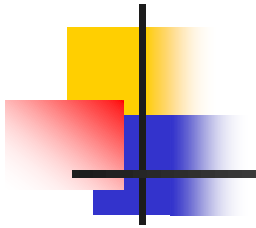
Il numero complessivo delle partizioni di un insieme I di k oggetti è dato dai numeri di Bell definiti dalla seguente ricorrenza:

$B_0 = 1$

$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} \cdot B_k$

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

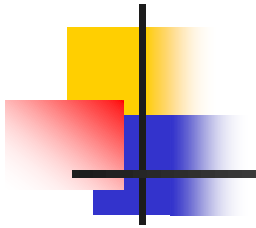
Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

Scomposizione in sottoproblemi

E’ 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.

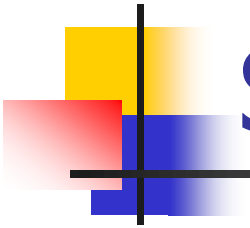
Si opera in maniera distribuita, senza visione unitaria della soluzione.



Approcci:

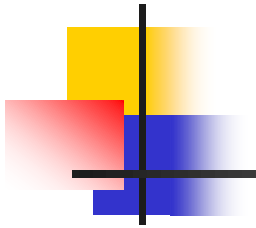
- ogni ricorsione sceglie un elemento della soluzione. Terminazione: la soluzione ha raggiunto la dimensione richiesta oppure non ci sono più scelte
- la ricorsione esamina uno degli elementi dell'insieme di partenza per decidere se e dove andrà aggiunto alla soluzione.

Si segue il primo approccio perché più intuitivo.



Strutture dati

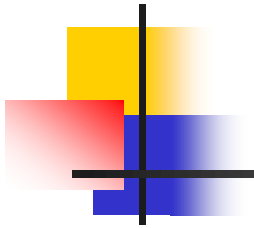
- Strutture dati
  - globali, cioè comuni a tutte le istanze della funzione ricorsiva
  - locali, cioè locali a ciascuna delle istanze
- Strutture dati globali:
  - dati del problema (matrice, mappa, grafo), vincoli, scelte disponibili, soluzione
- Strutture dati locali:
  - indici di livello di chiamata 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: tabelle 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 `count`) 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`).

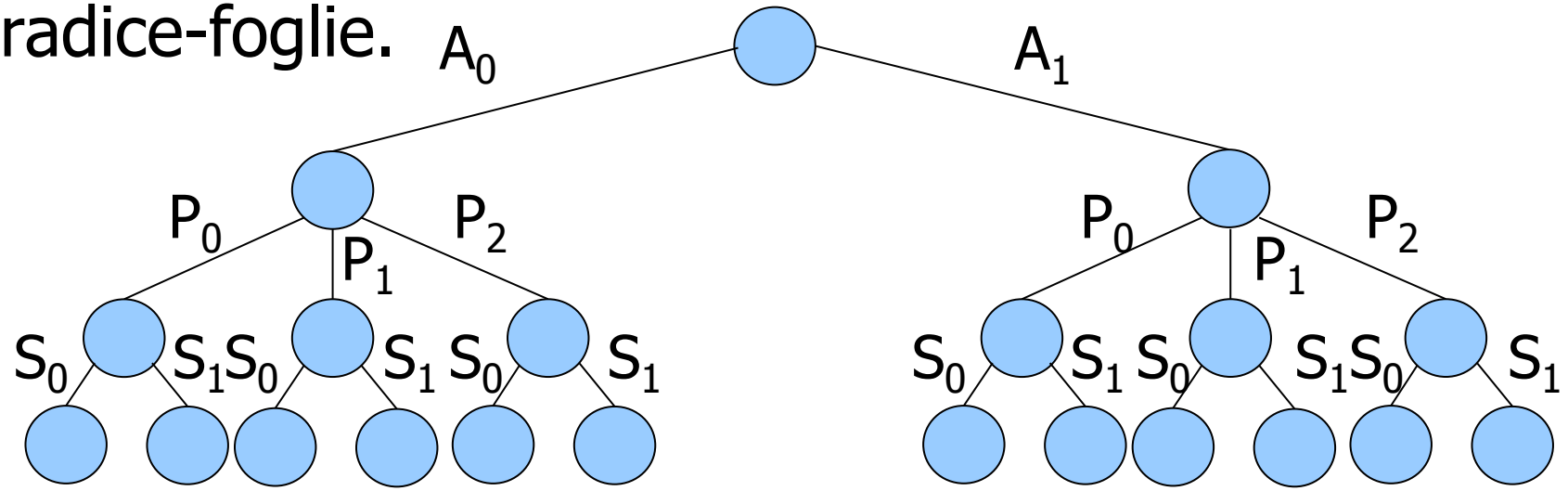
Principio di moltiplicazione

- 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.



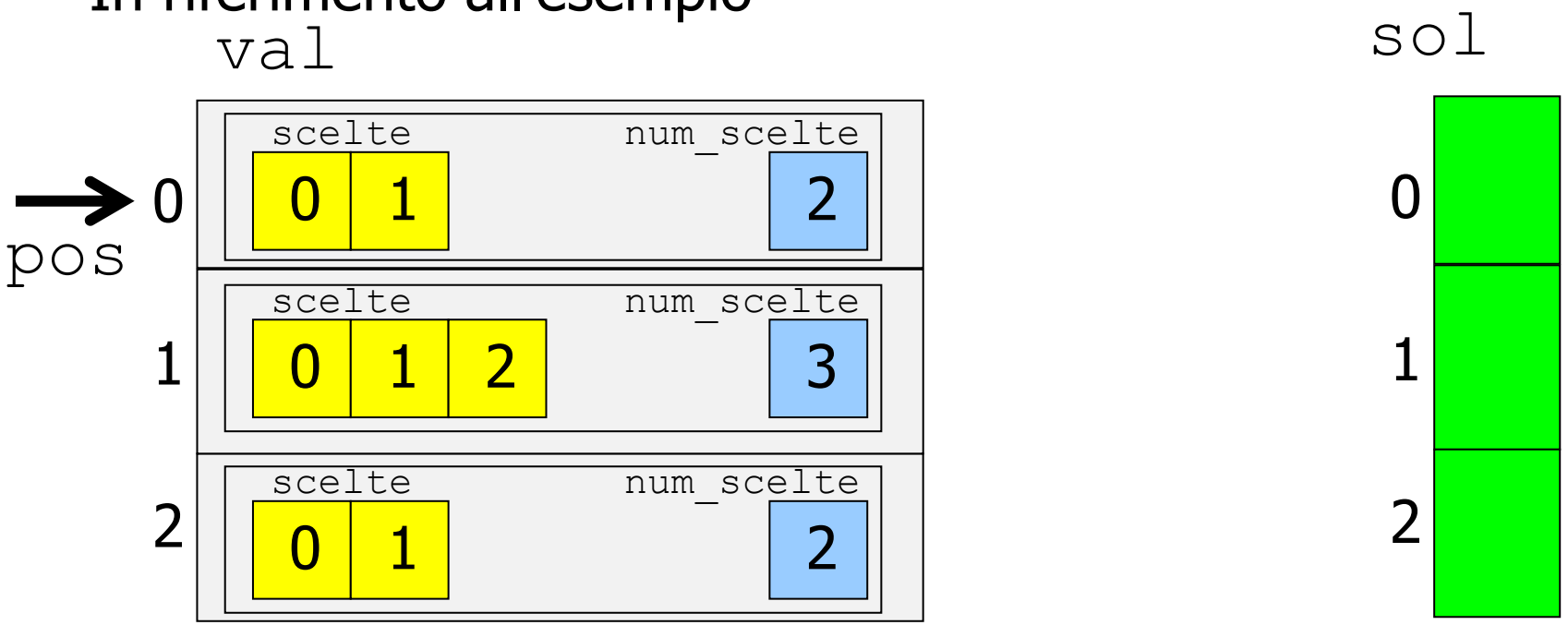
Soluzione:

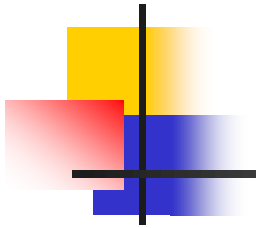
(A_0,P_0,S_0), (A_0,P_0,S_1), (A_0,P_1,S_0), (A_0,P_1,S_1), (A_0,P_2,S_0), (A_0,P_2,S_1),
(A_1,P_0,S_0), (A_1,P_0,S_1), (A_1,P_1,S_0), (A_1,P_1,S_1), (A_1,P_2,S_0), (A_1,P_2,S_1)

I principi-base dell'esplorazione

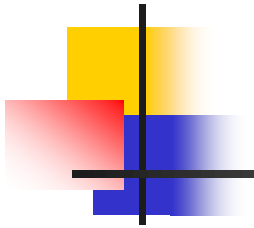
- Si prendono n *decisioni in sequenza,* ciascuna tra diverse *scelte* , il cui numero è fisso dato il livello di decisione, ma variabile di livello in livello
- le scelte sono in corrispondenza biunivoca con un sottoinsieme degli interi (non necessariamente contigui)
- le scelte possibili sono memorizzate 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 di interi di quella dimensione per le scelte `*scelte`.

In riferimento all'esempio

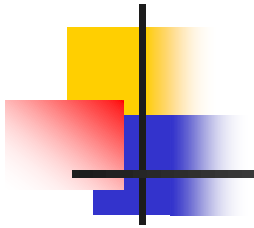




- si vogliono enumerare tutte le soluzioni, esplorandone l'intero spazio
- tutte le soluzioni sono valide
- le chiamate ricorsive sono associate alla soluzione, la cui dimensione ad ognuna di esse cresce di 1. Terminazione quando la dimensione della soluzione corrente è quella finale



- 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 `pos>=n` si è trovata una soluzione

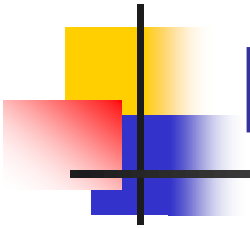


- il passo ricorsivo itera sulle scelte possibili per il valore corrente di `pos`, cioè il contenuto di `sol[pos]` è preso da `val[pos].scelte[i]` estendendo ogni volta la soluzione
- e ricorre sulla scelta `pos+1-esima`
- `count` è il valore di ritorno della ricorsione e conteggia il numero di soluzioni.

01calcolocombinatorio

```
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));
```

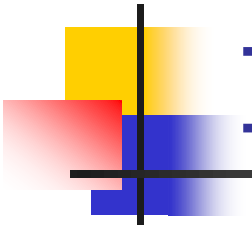
```
int princ_molt(int pos, Livello *val, int *sol,
               int n, int count) {
  int i;
  if (pos >= n) {
    for (i = 0; i < n; i++)
      printf("%d ", sol[i]);
    printf("\n");
    return count+1;
  }
  for (i = 0; i < val[pos].num_scelte; i++) {
    sol[pos] = val[pos].scelte[i];
    count = princ_molt(pos+1, val, sol, n, count);
  }
  return count;
}
```



Modelli

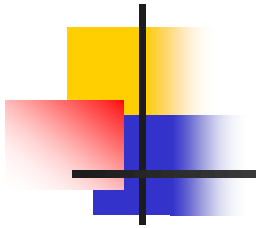
Lo spazio delle possibilità può essere modellato come quello delle:

- disposizioni semplici
- disposizioni con ripetizione (+ insieme delle parti)
- permutazioni semplici
- permutazioni con ripetizione
- combinazioni semplici
- combinazioni con ripetizione
- (partizioni).



I principi-base dell'esplorazione

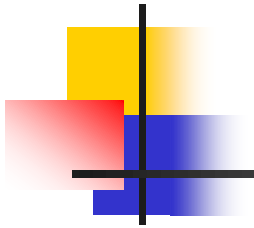
- Si immagini di dovere prendere delle *decisioni in sequenza,* ciascuna tra diverse *scelte* senza avere informazioni che guidano la decisione
- le scelte possibili sono memorizzate in un vettore `val` di interi di dimensione n
- si vogliono enumerare tutte le soluzioni, esplorandone l'intero spazio, decidendo, una volta raggiunta una soluzione, se è valida/ottima



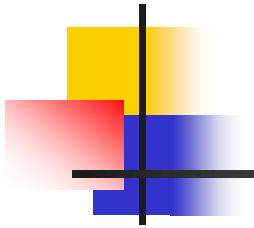
Alternative:

- 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.

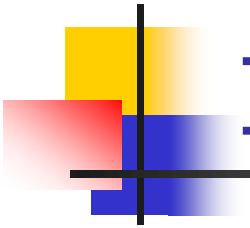
Nel seguito si adotta la prima alternativa.



- 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.

E’ piuttosto una tecnica algoritmica per esaminare ordinatamente le possibili istanze (soluzioni ammissibili o valide) di uno spazio di ricerca.

Disposizioni semplici

Per non generare elementi ripetuti:

- un vettore `mark` registra gli elementi già presi (`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 `mark[i]==0`, `mark[i]` viene assegnato con 1
- in fase di backtrack, `mark[i]` viene assegnato con 0
- `count` registra il numero di soluzioni.

```
val = malloc(n * sizeof(int));
sol = malloc(k * sizeof(int));
mark = malloc(n * sizeof(int));
```

01calcolocombinatorio

```
int disp(int pos,int *val, int *sol, int *mark,
         int n, int k,int count){
  int i;
  if (pos >= k){
    for (i=0; i<k; i++) printf("%d ", sol[i]);
    printf("\n");
    return count+1;
  }
  for (i=0; i<n; i++){
    if (mark[i] == 0) {
      mark[i] = 1;
      sol[pos] = val[i];
      count = disp(pos+1, val, sol, mark, n, k,count);
      mark[i] = 0;
    }
  }
  return count;
}
```

terminazione

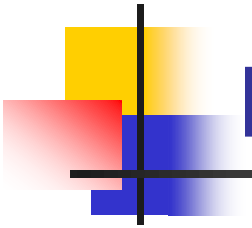
iterazione sulle n scelte

controllo ripetizione

marcamento e scelta

smarcamento

ricorsione



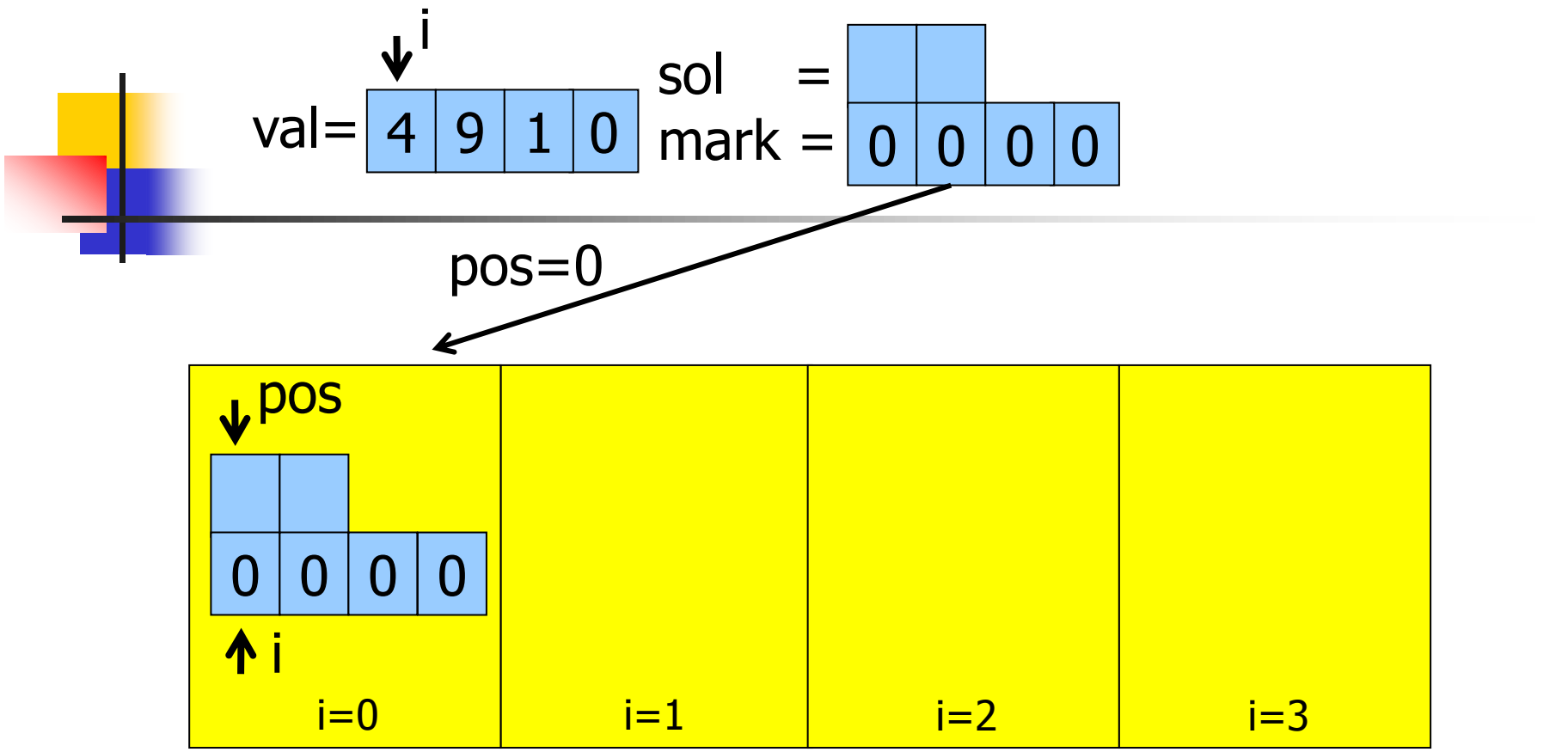
Esempio

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, val = {4, 9, 1, 0 }

$D_{4,2} = 4!/(4-2)! = 4 \cdot 3 = 12$

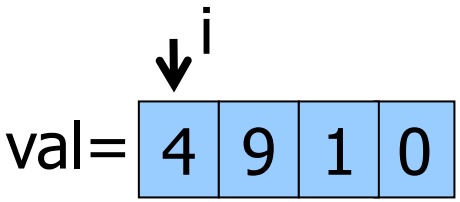
Soluzione:
{49, 41, 40, 94, 91, 90, 14, 19, 10, 04, 09, 01 }

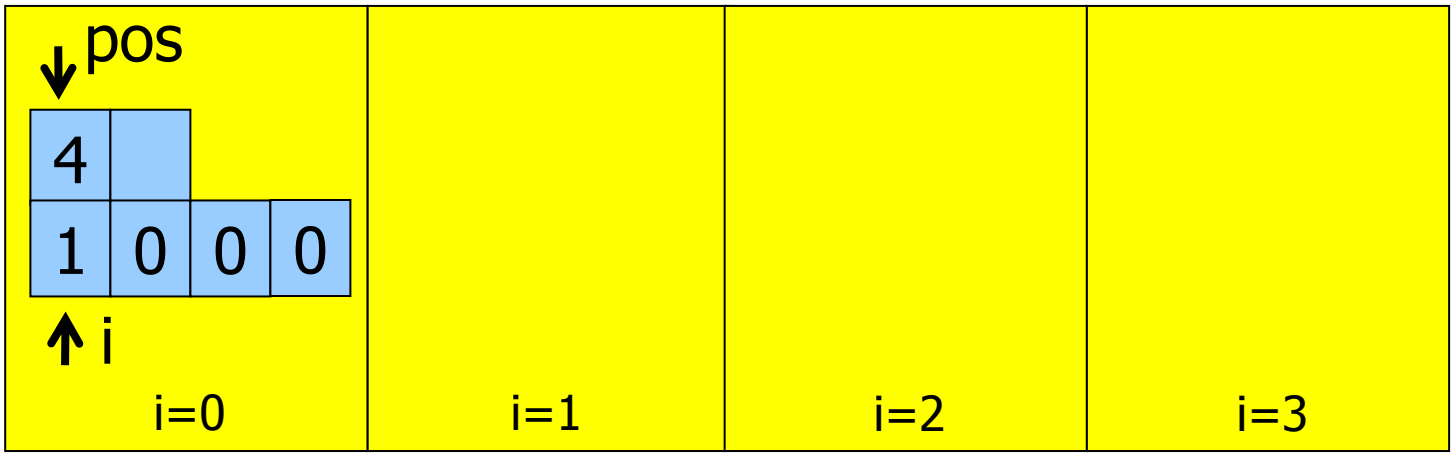


mark[i] = 0

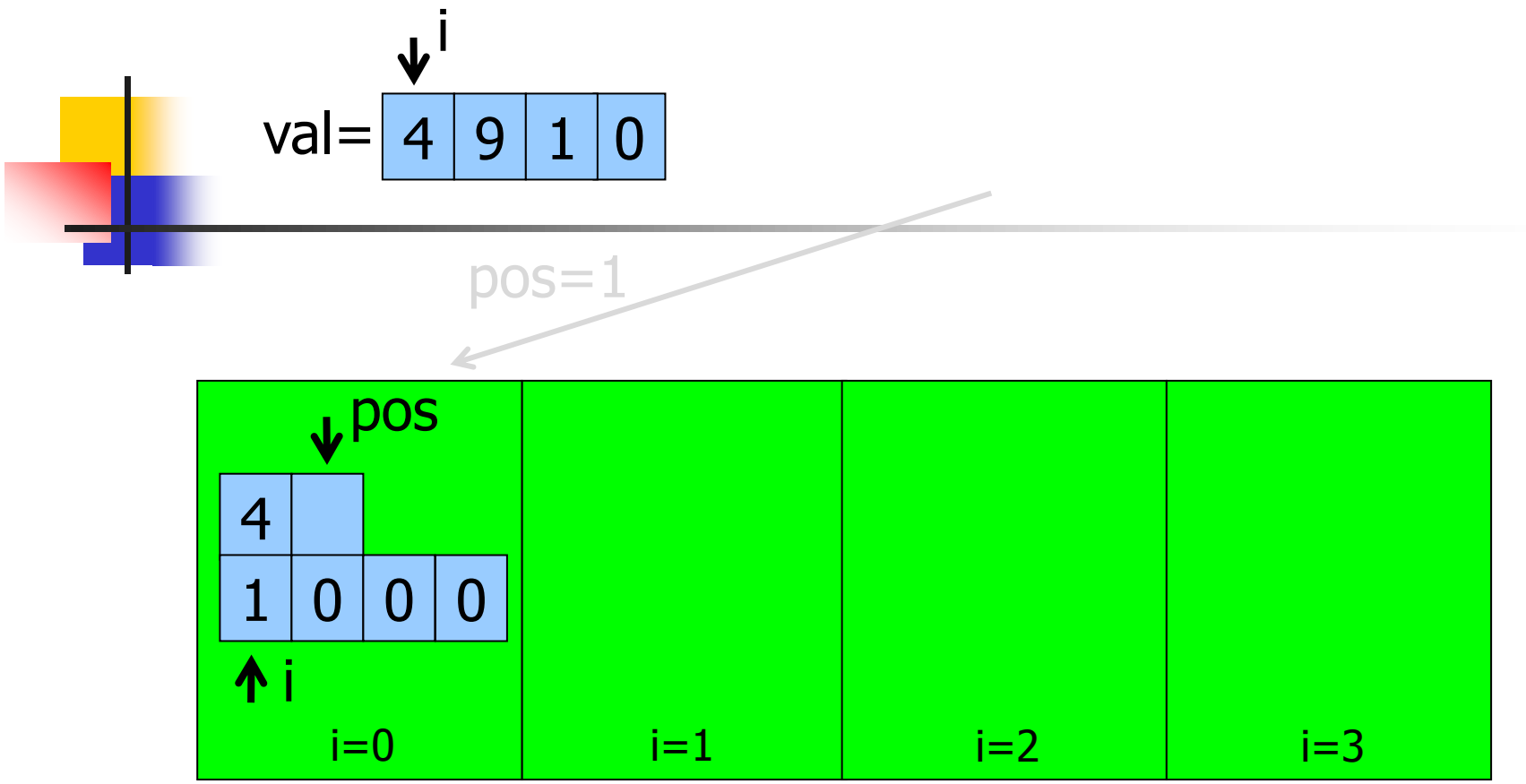
sol[pos] = val[i]

mark[i] = 1

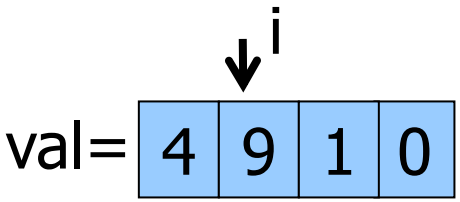


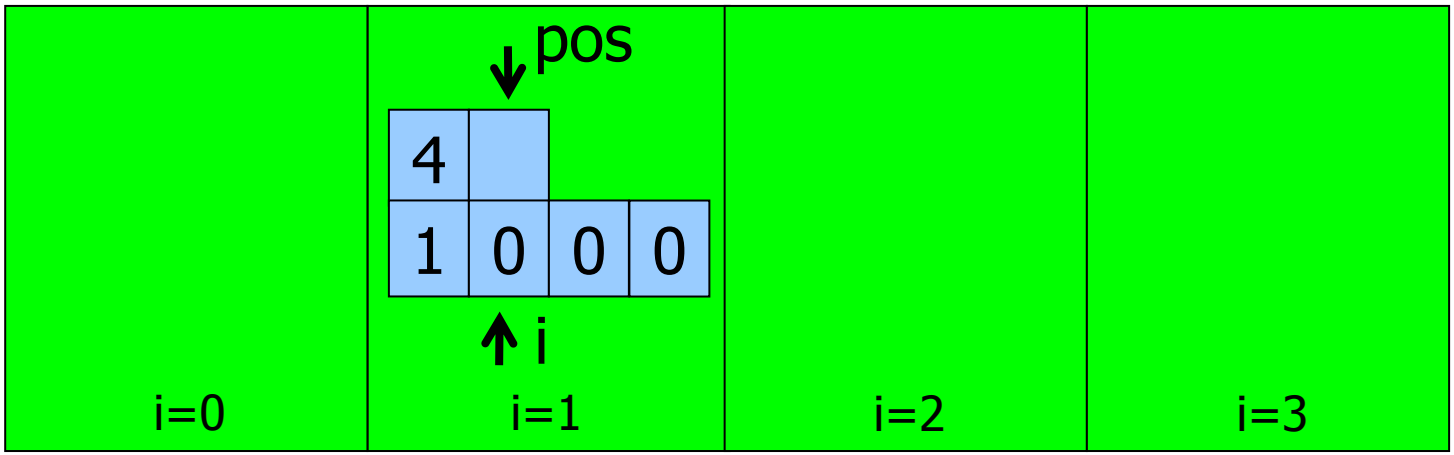


ricorsione
con pos=1



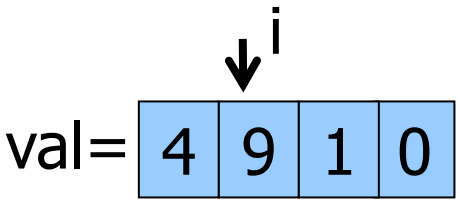
mark[i] = 1

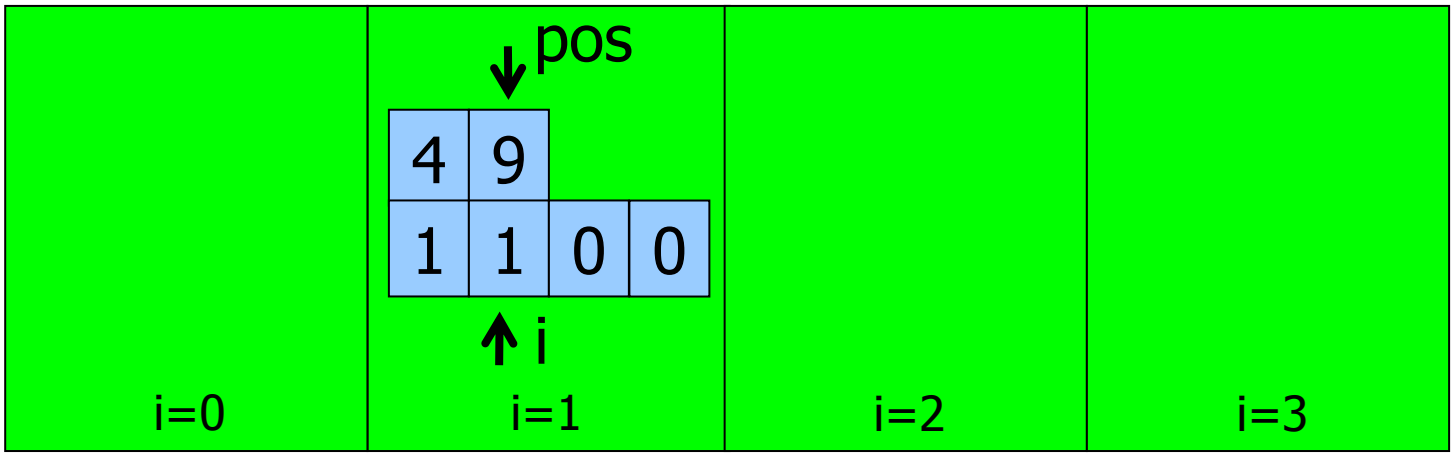




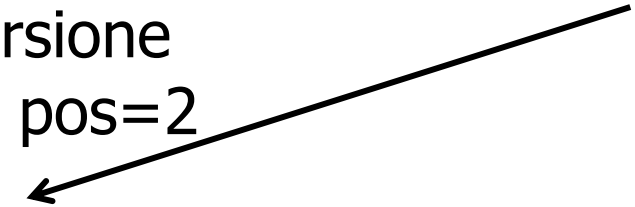
mark[i] = 0

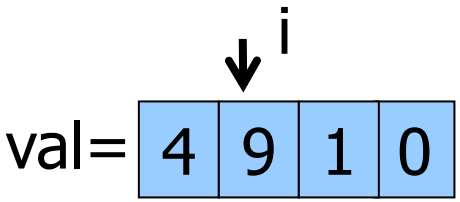
sol[pos] = val[i]
mark[i] = 1

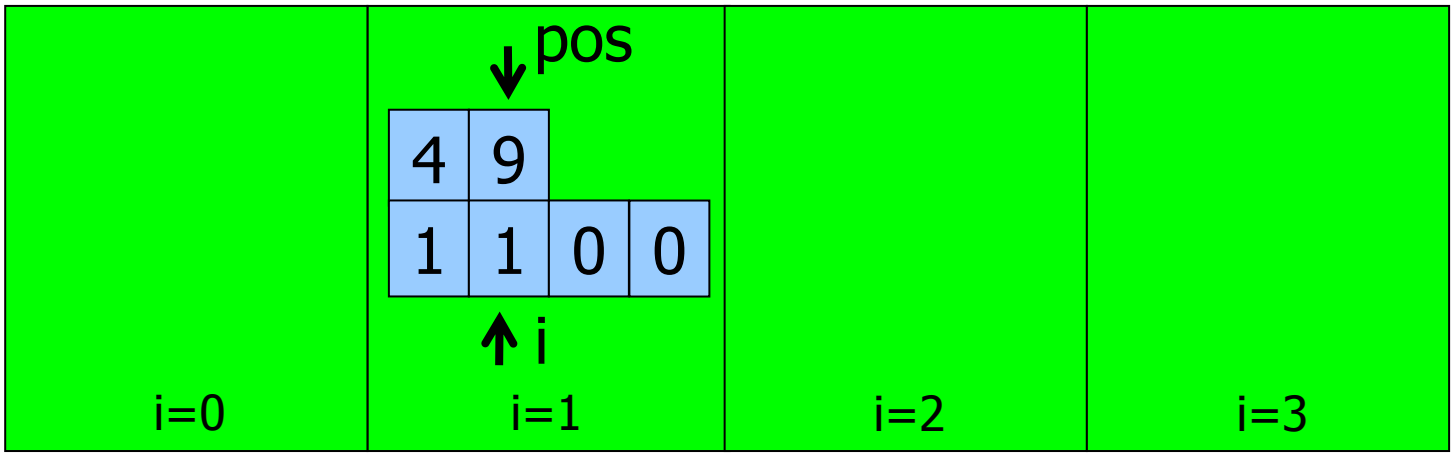




ricorsione
con pos=2



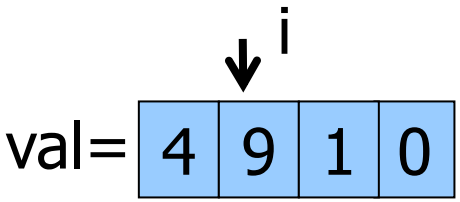


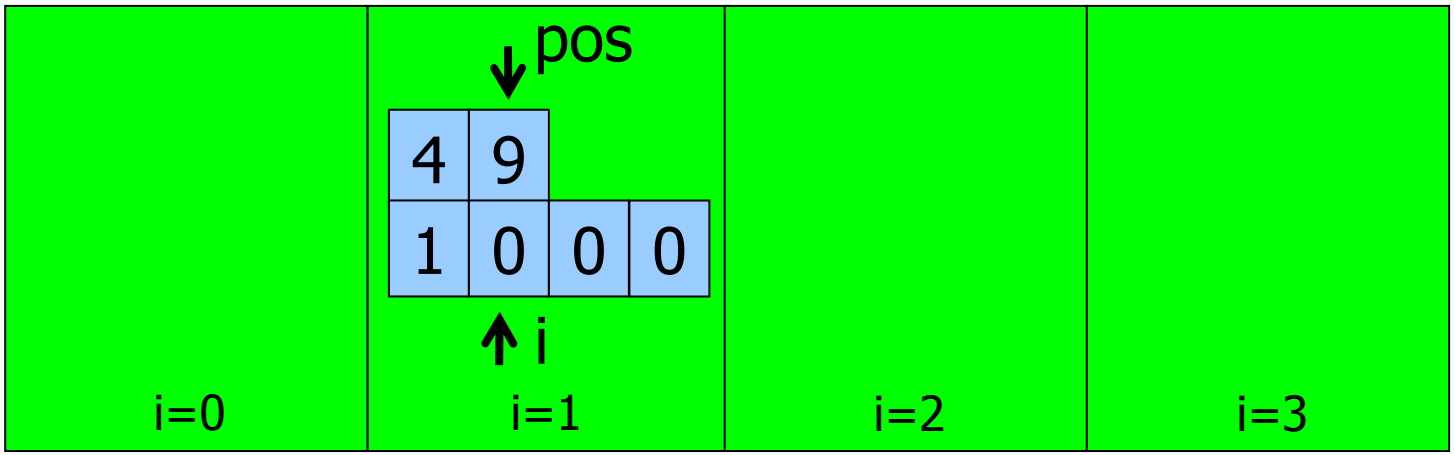


terminazione: visualizza, aggiorna count

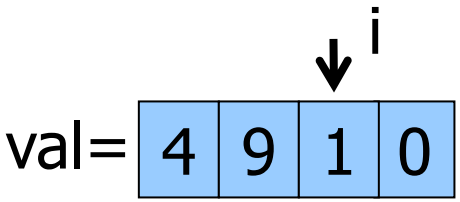
ritorna

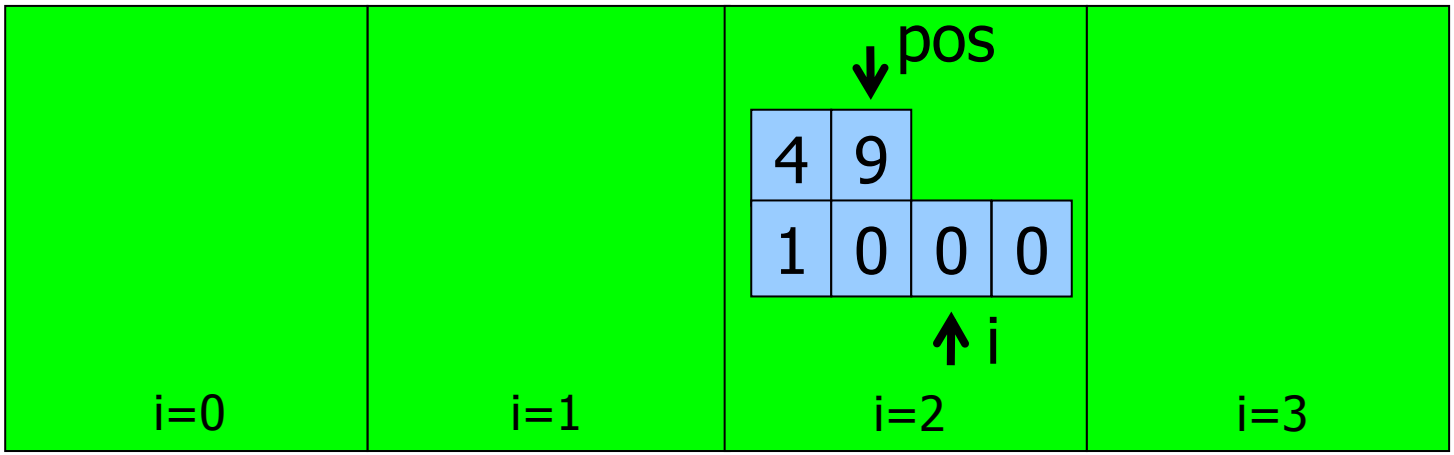
sol= 4 9





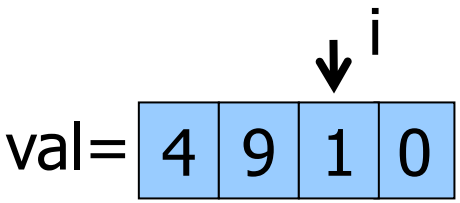
smarca mark[i]

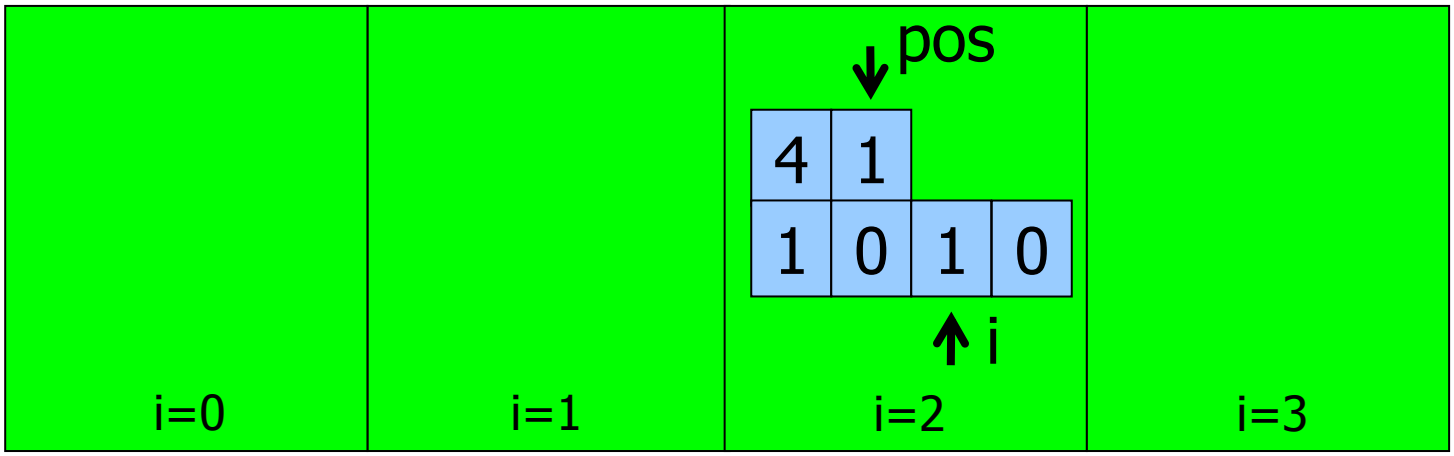




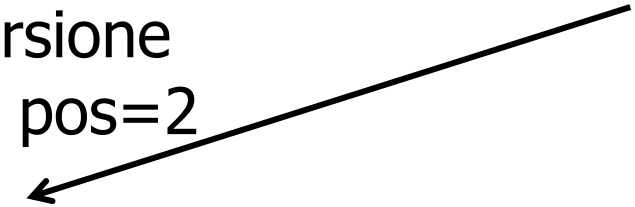
mark[i] = 0

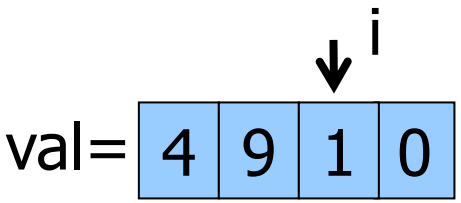
sol[pos] = val[i]
mark[i] = 1

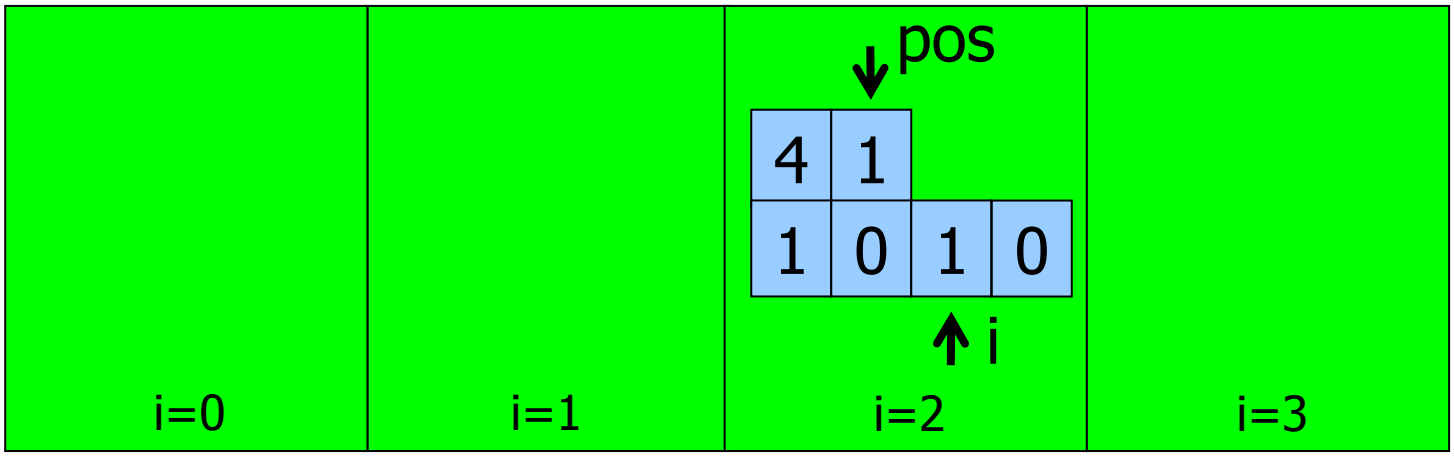




ricorsione
con pos=2



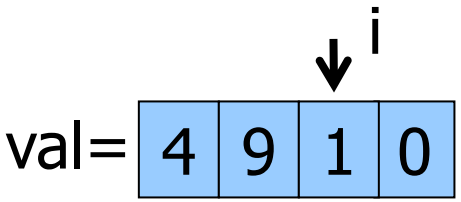


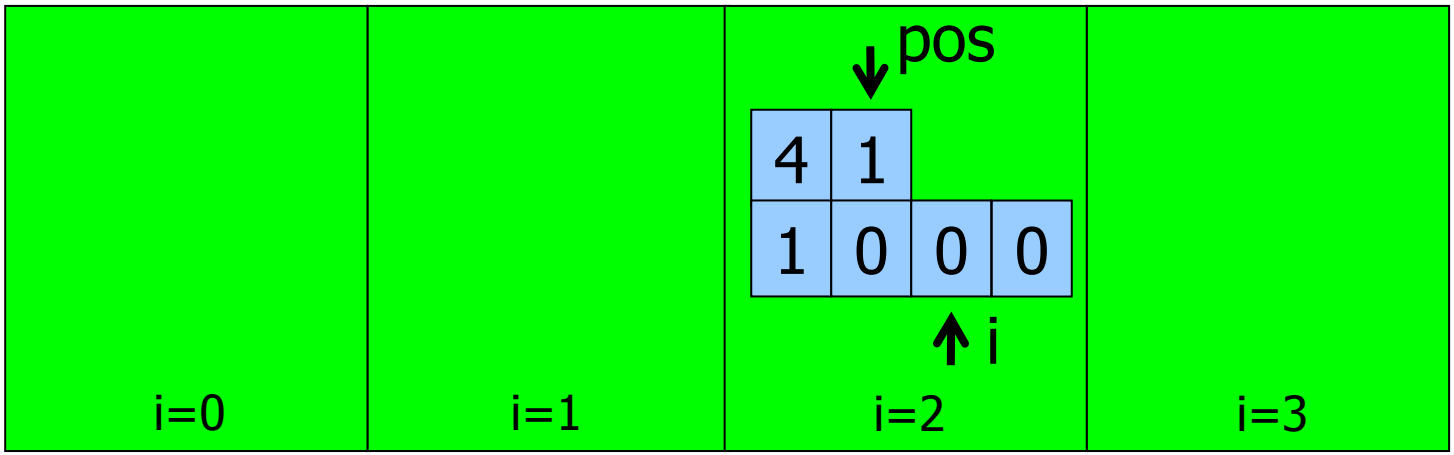


terminazione: visualizza, aggiorna count

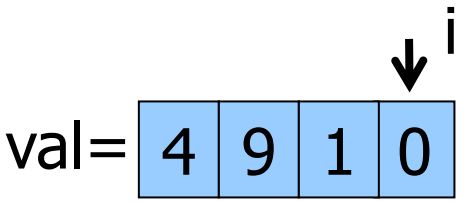
ritorna

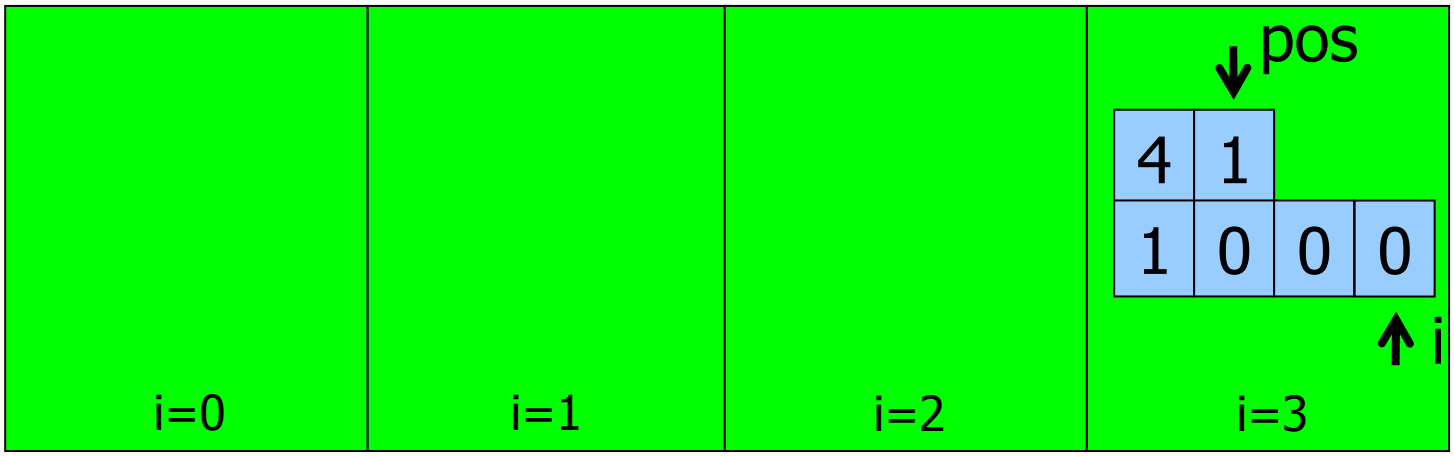
sol= 4 1





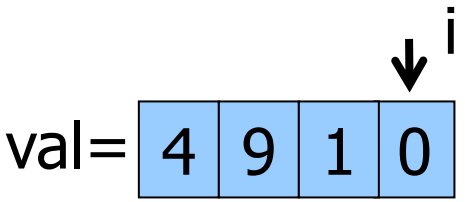
smarca mark[i]

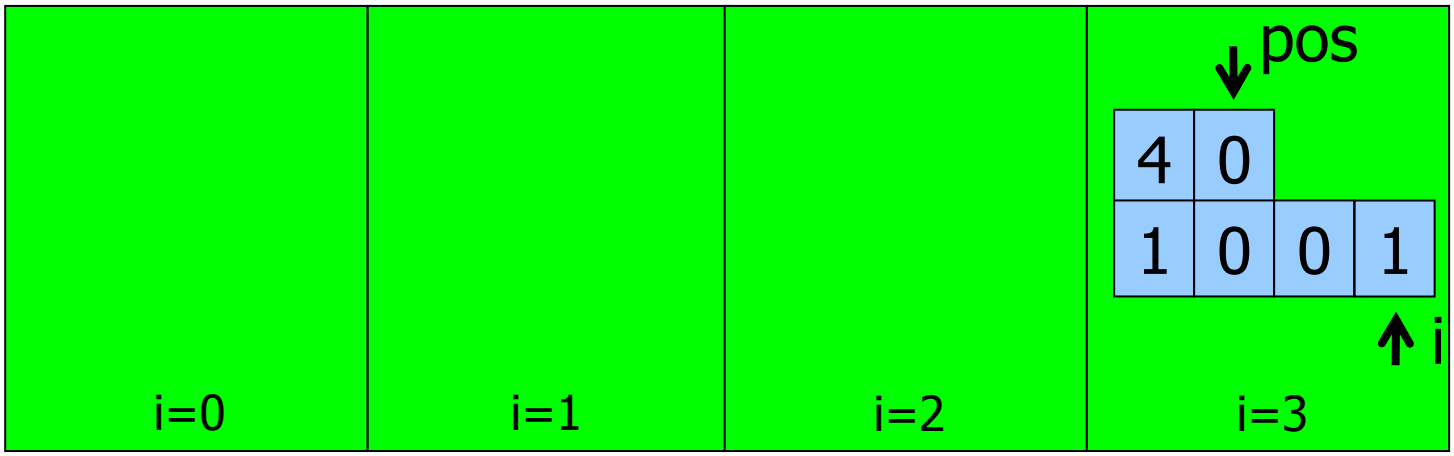




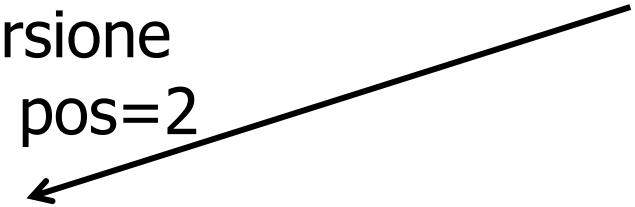
mark[i] = 0

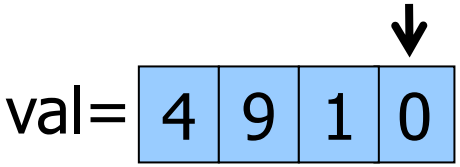
sol[pos] = val[i]
mark[i] = 1

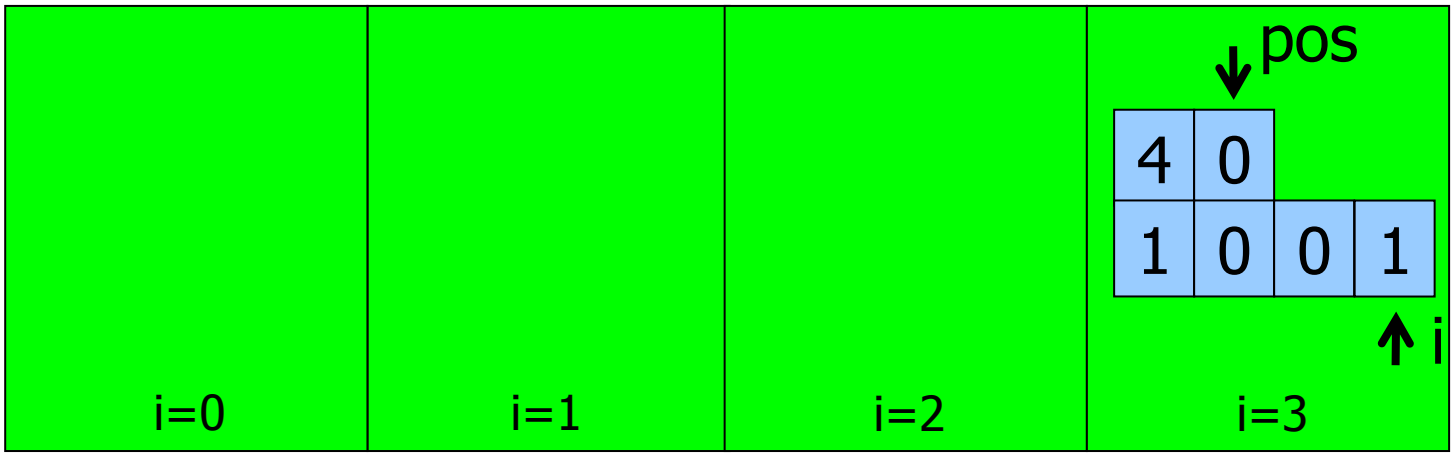




ricorsione
con pos=2

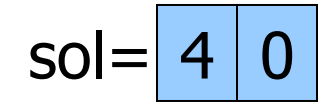


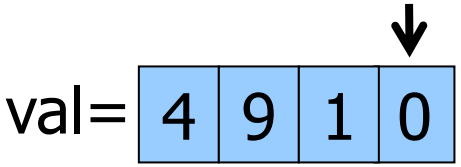


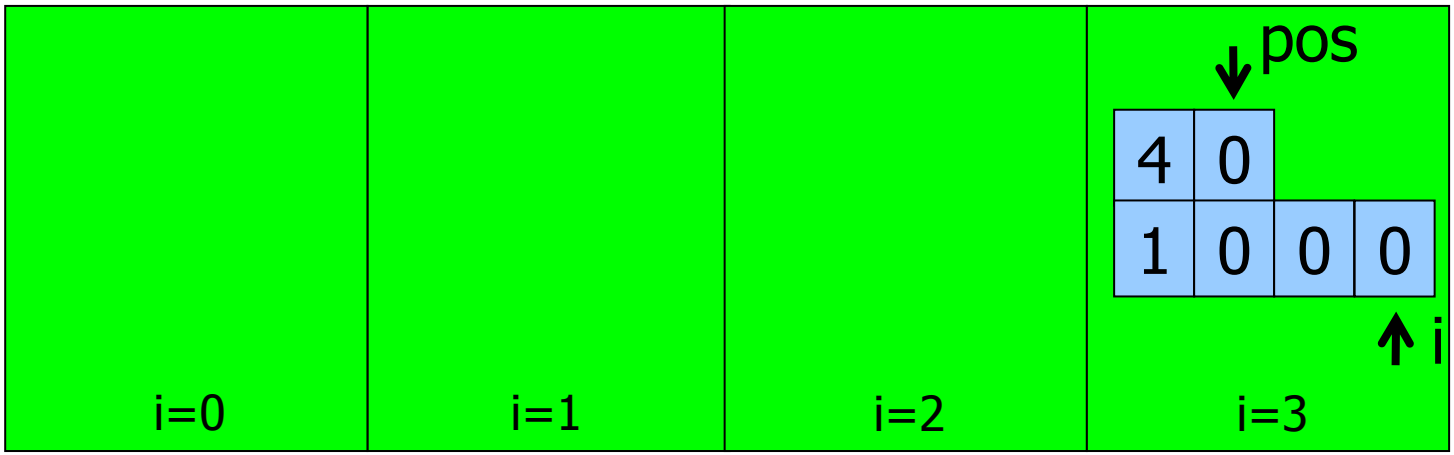


terminazione: visualizza, aggiorna count

ritorna

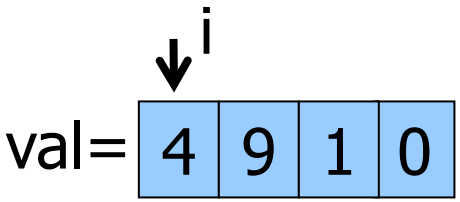


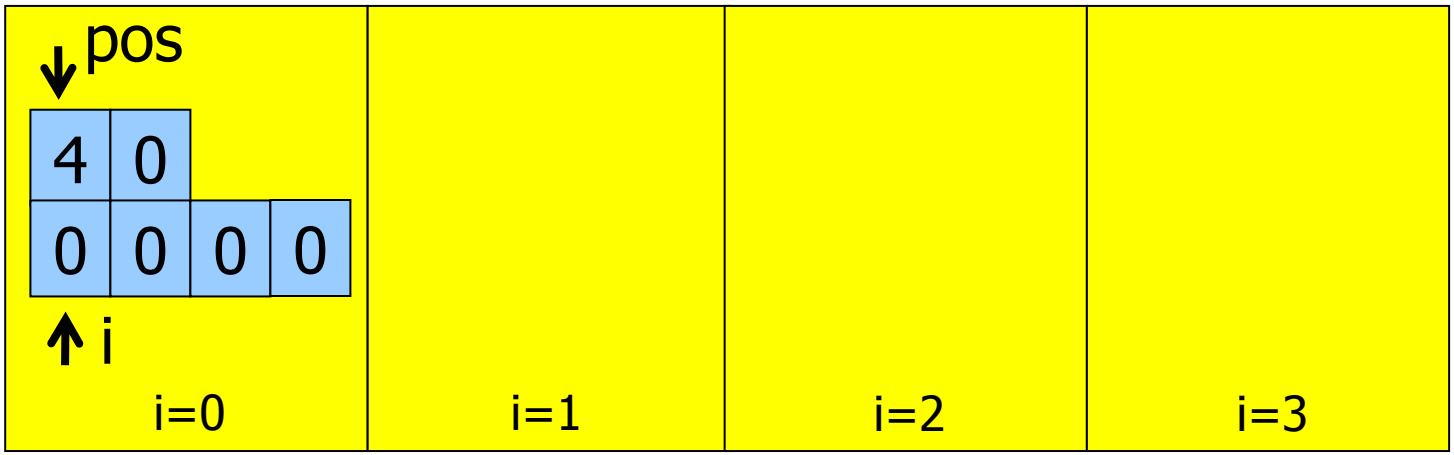




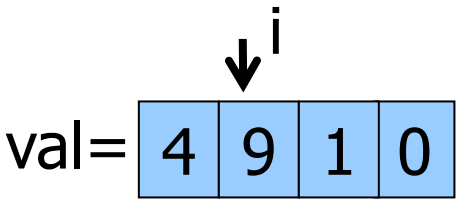
smarca mark[i]

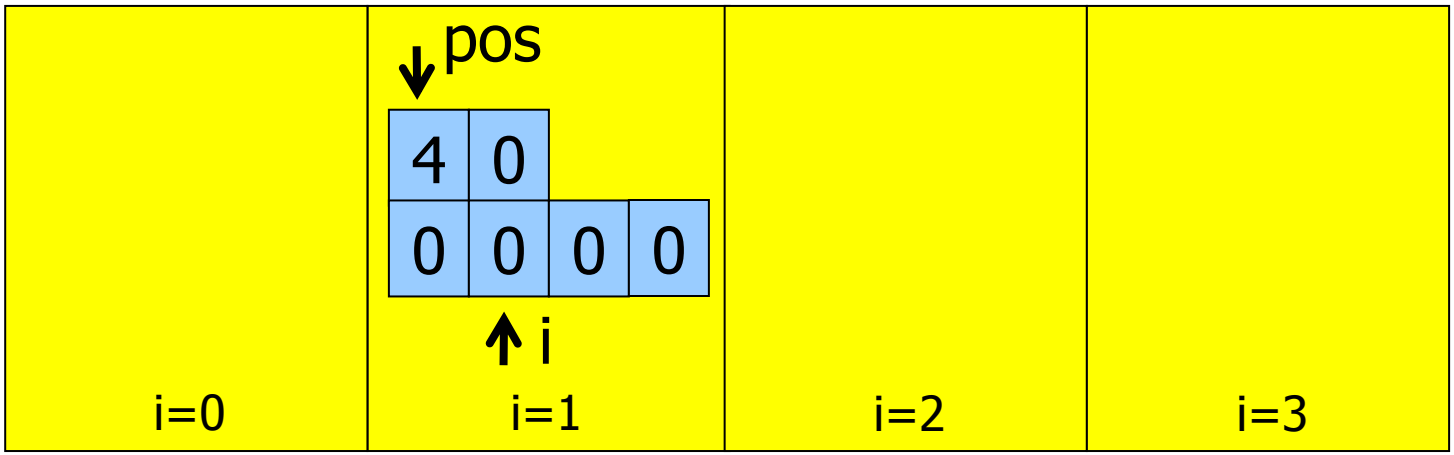
ciclo for terminato, ritorna





smarca mark[i]

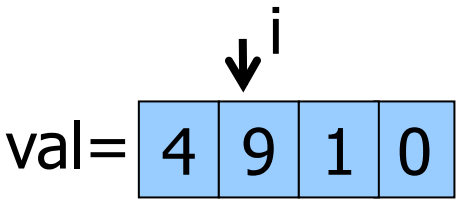


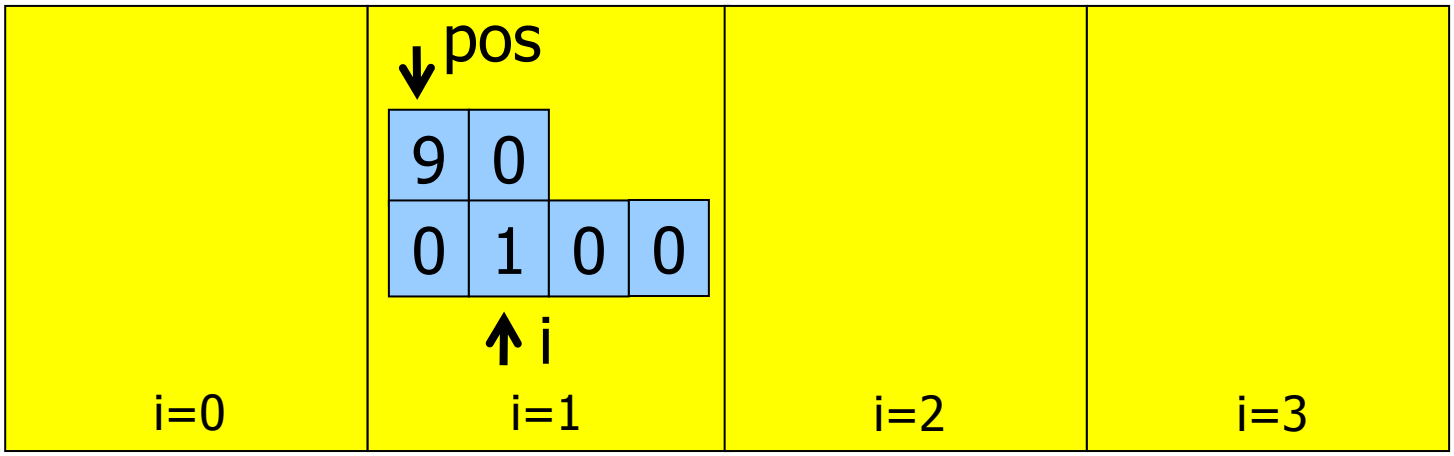


mark[i] = 0

sol[pos] = val[i]

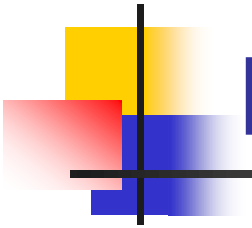
mark[i] = 1





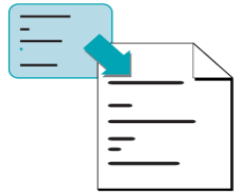
ricorsione
con pos=1

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
- `count` registra il numero di soluzioni.




```
int disp_rip(int pos,int *val,i                n,int k,int count){
  int i;
  if (pos >= k) {
    for (i=0; i<k; i++)
      printf("%d ", sol[i]);
    printf("\n");
    return count+1;
  }
  for (i = 0; i < n; i++) {
    sol[pos] = val[i];
    count = disp_rip(pos+1, val, sol, n, k, count);
  }
  return count;
}
```

terminazione

iterazione sulle n scelte

scelta

ricorsione

Permutazioni semplici

Per non generare elementi ripetuti:

- un vettore `mark` registra gli elementi già presi (`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 `mark[i]==0`, `mark[i]` viene assegnato con 1
- in fase di backtrack, `mark[i]` viene assegnato con 0
- `count` registra il numero di soluzioni.

```
val = malloc(n * sizeof(int));
sol = malloc(n * sizeof(int));
mark = malloc(n * sizeof(int));
```

01calcolocombinatorio

```
int perm(int pos,int *val,int *sol,int *mark,
         int n, int count){
  int i;
  if (pos >= n){
    for (i=0; i<n; i++) printf("%d ", sol[i]);
    printf("\n");
    return count+1;
  }
  for (i=0; i<n; i++)
    if (mark[i] == 0) {
      mark[i] = 1;
      sol[pos] = val[i];
      count = perm(pos+1, val, sol, mark, n, count);
      mark[i] = 0;
  }
  return count;
}
```

terminazione

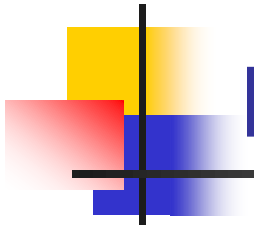
iterazione sulle n scelte

controllo ripetizione

marcamento e scelta

ricorsione

smarcamento



Esempio

Dato un insieme val di n interi, generare tutte le permutazioni di questi valori.

Il numero di permutazioni è n!.

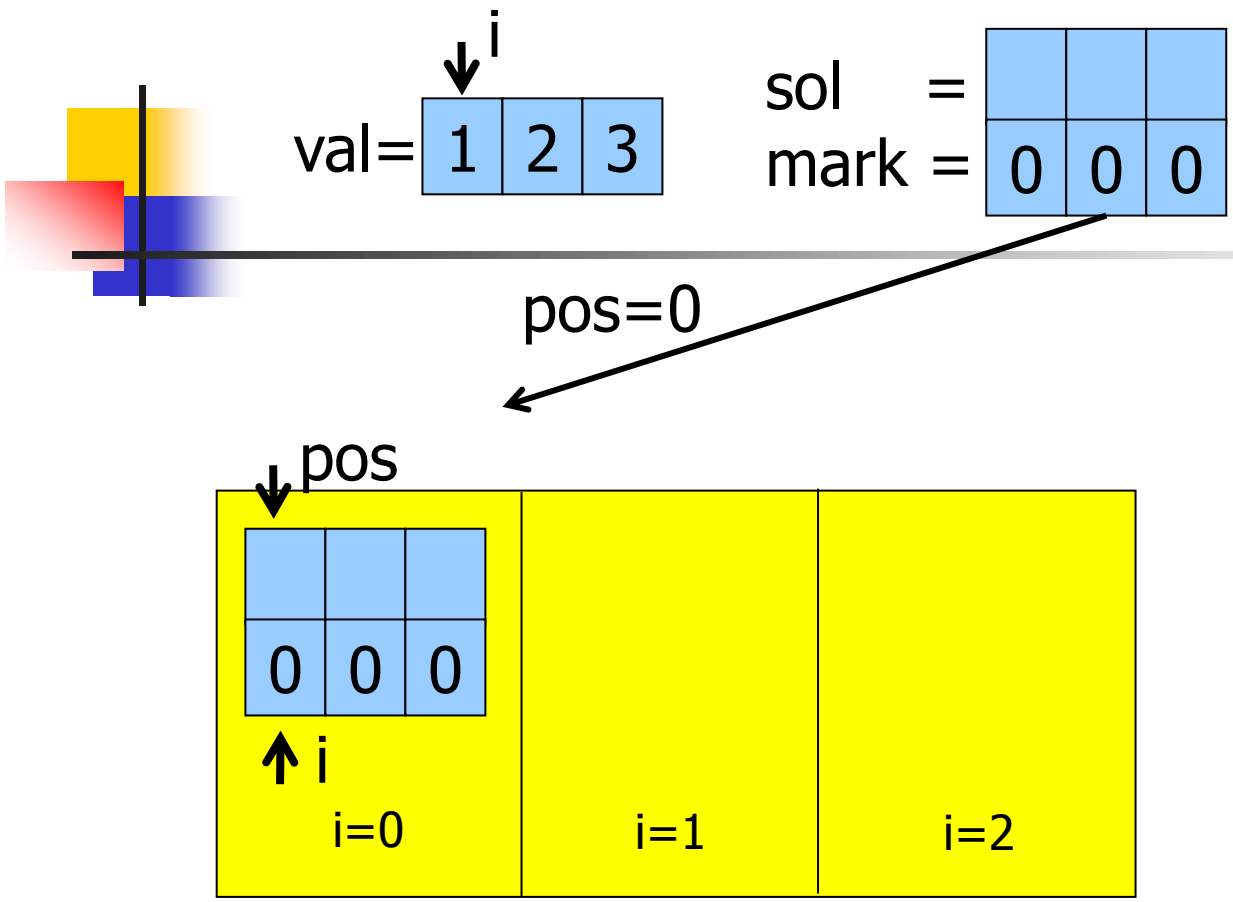
Esempio

val = {1, 2, 3} n = 3

n! = 6.

Le 6 permutazioni sono:

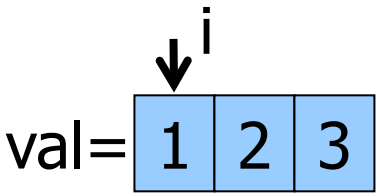
{1,2,3} {1,3,2} {2,1,3} {2,3,1} {3,1,2} {3,2,1}

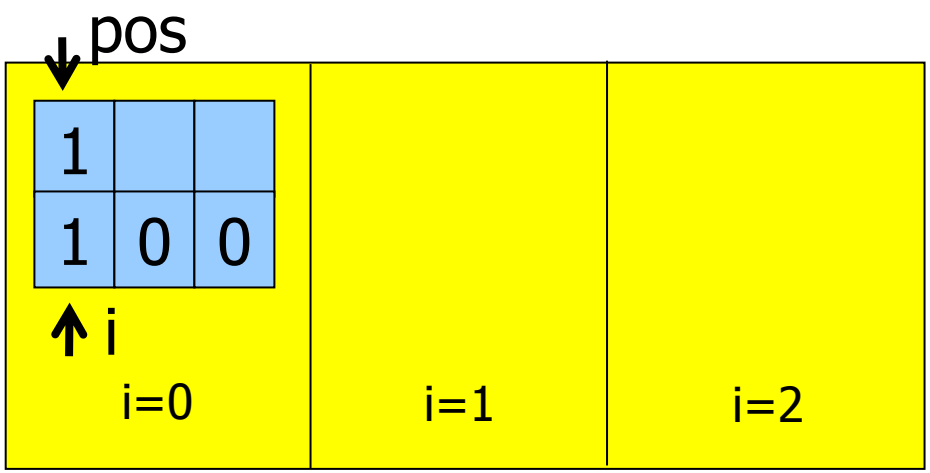


mark[i] = 0

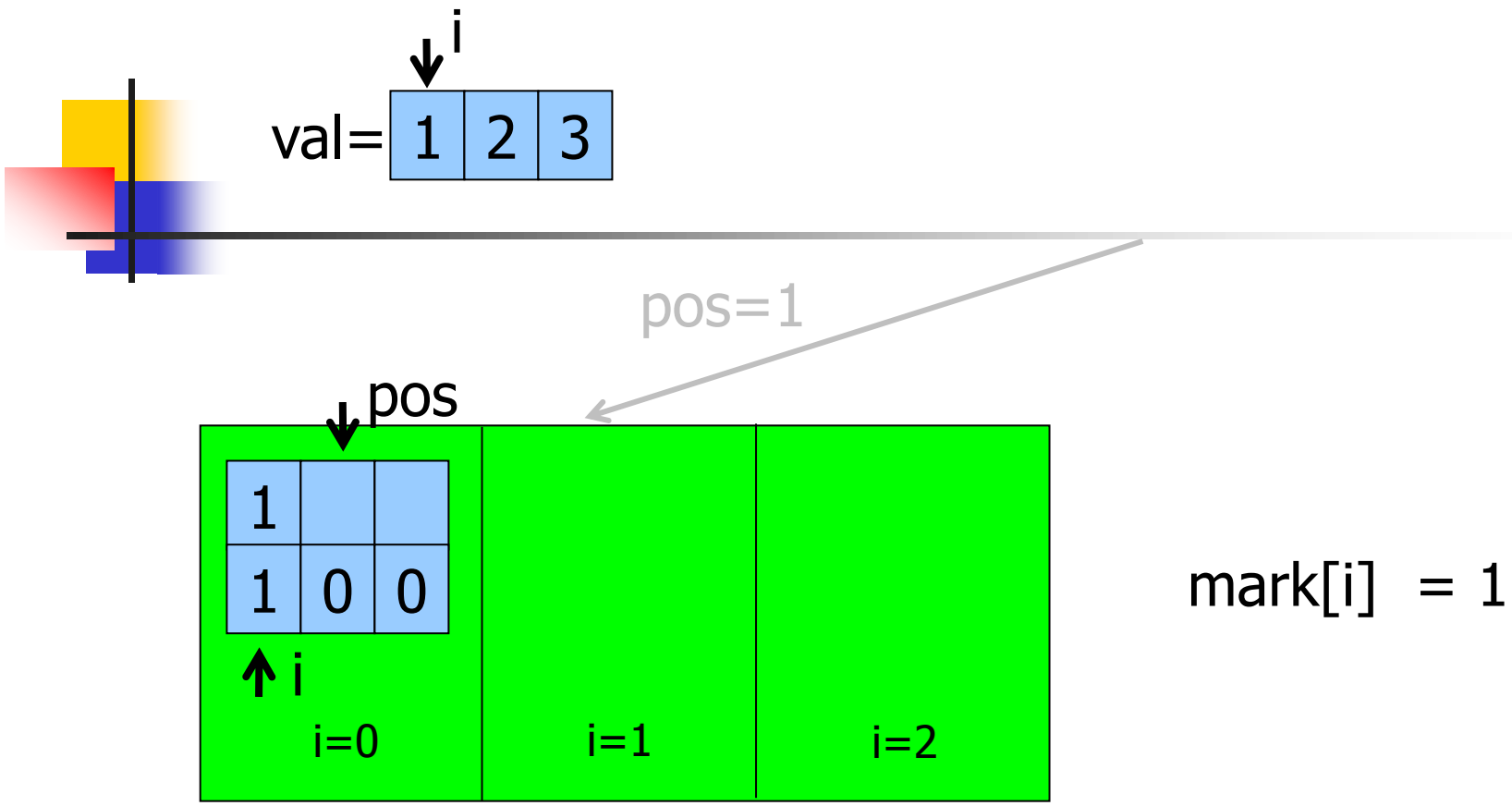
sol[pos] = val[i]

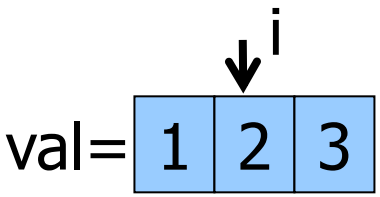
mark[i] = 1

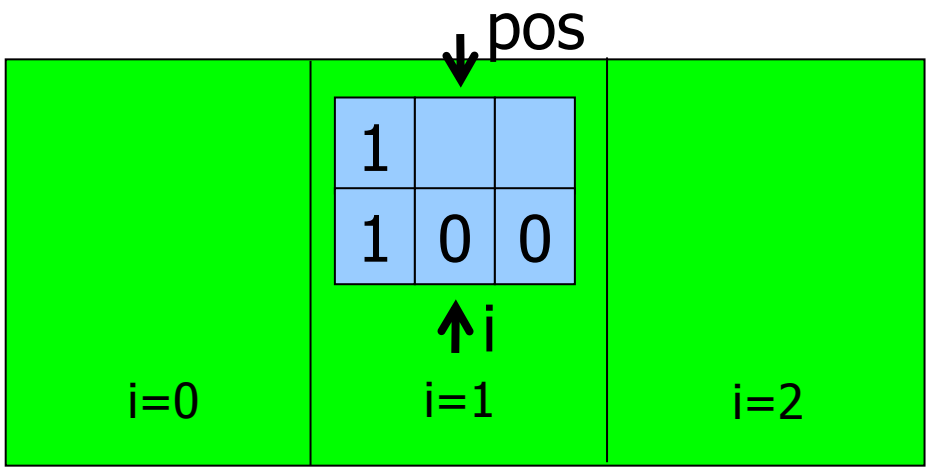




ricorsione
con pos=1


i
val=
1
2
3
pos=1
pos
1
1
0
0
i
i=0
i=1
i=2
mark[i] = 1

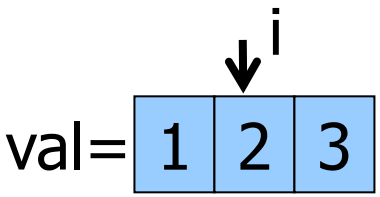


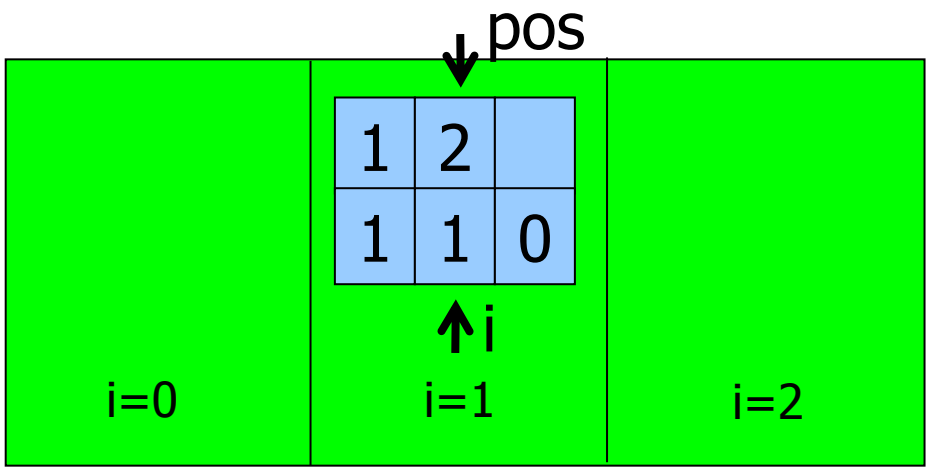


mark[i] = 0

sol[pos] = val[i]

mark[i] = 1

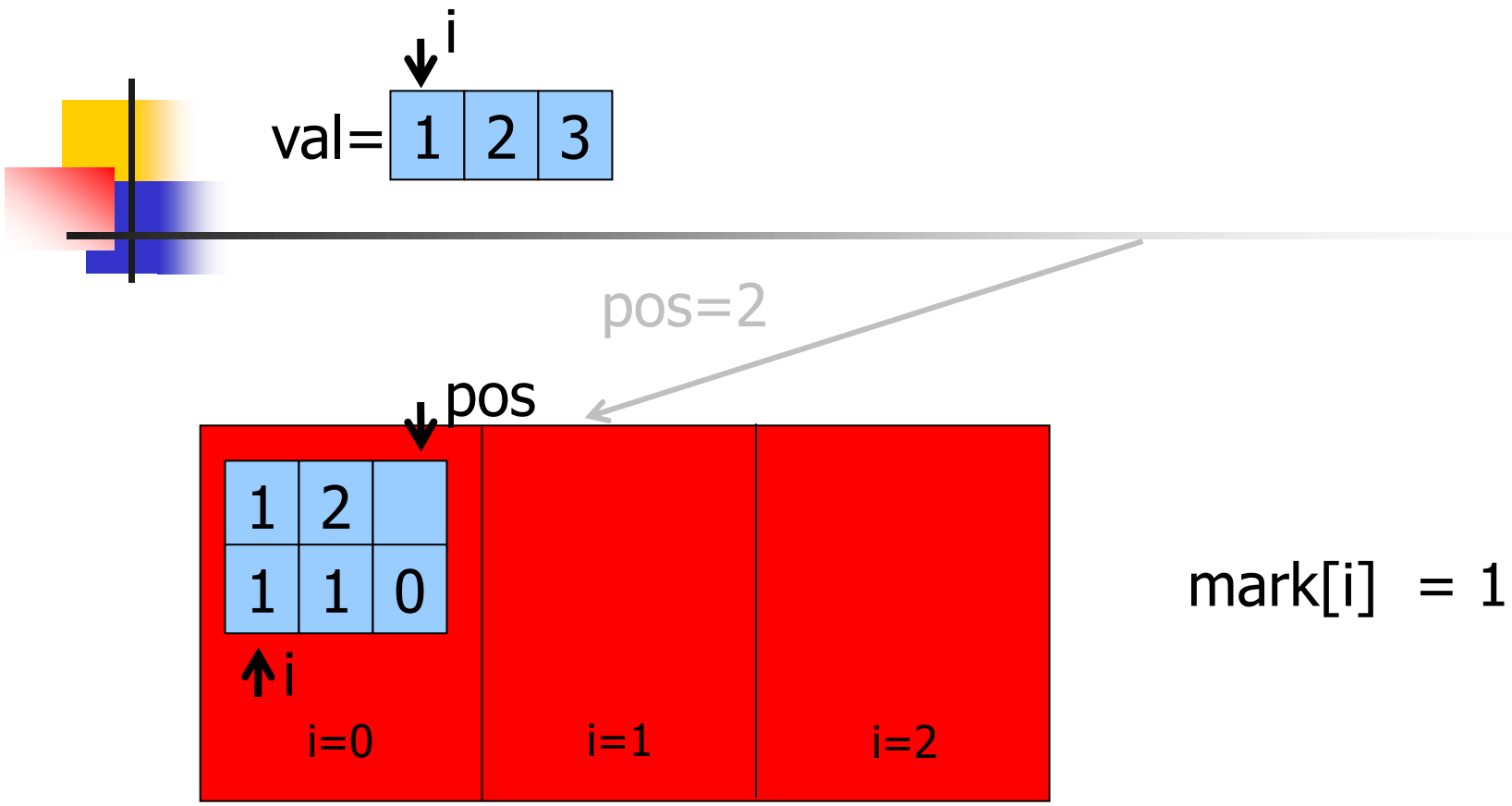


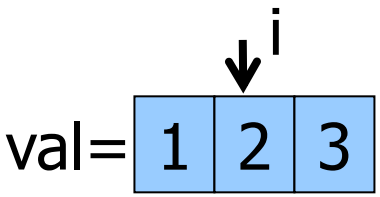


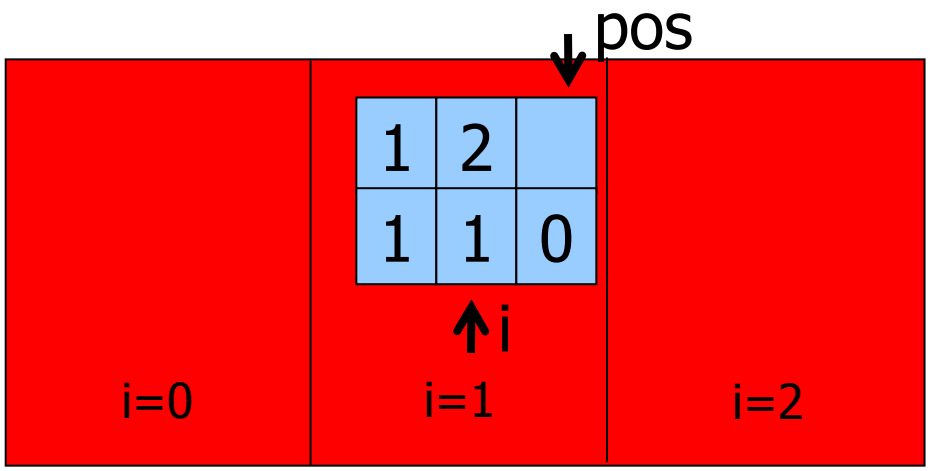
mark[i] = 0

sol[pos] = val[i]
mark[i] = 1

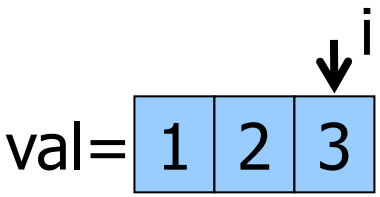
ricorsione
con pos=2

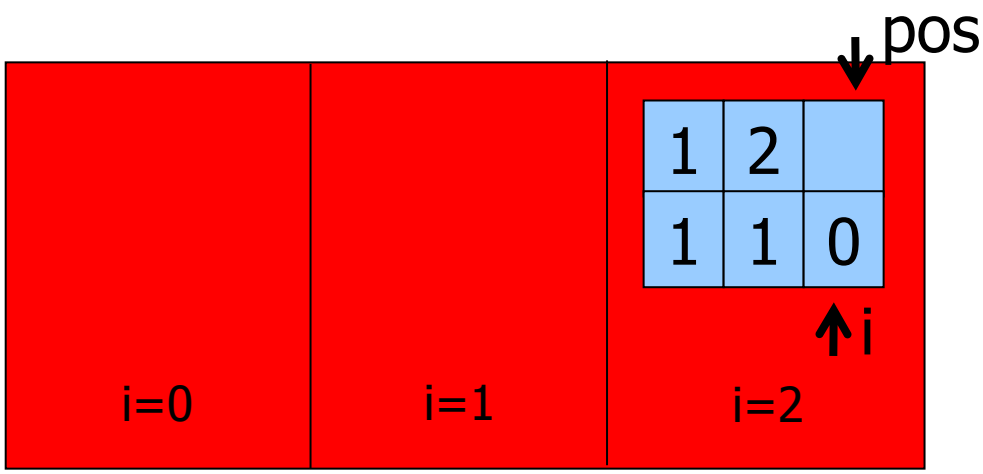

i
val=
1
2
3
pos=2
pos
1
2
1
1
0
i
i=0
i=1
i=2
mark[i] = 1





mark[i] = 1

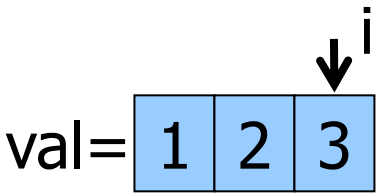


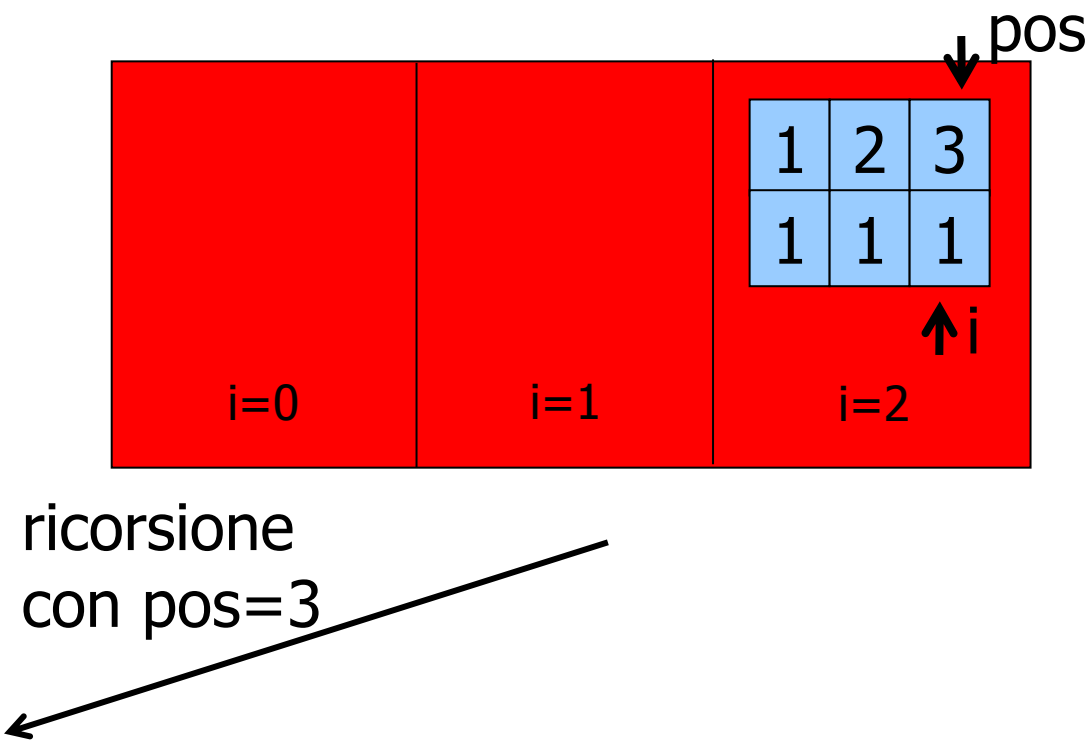


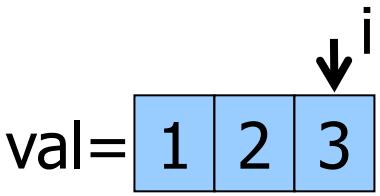
mark[i] = 0

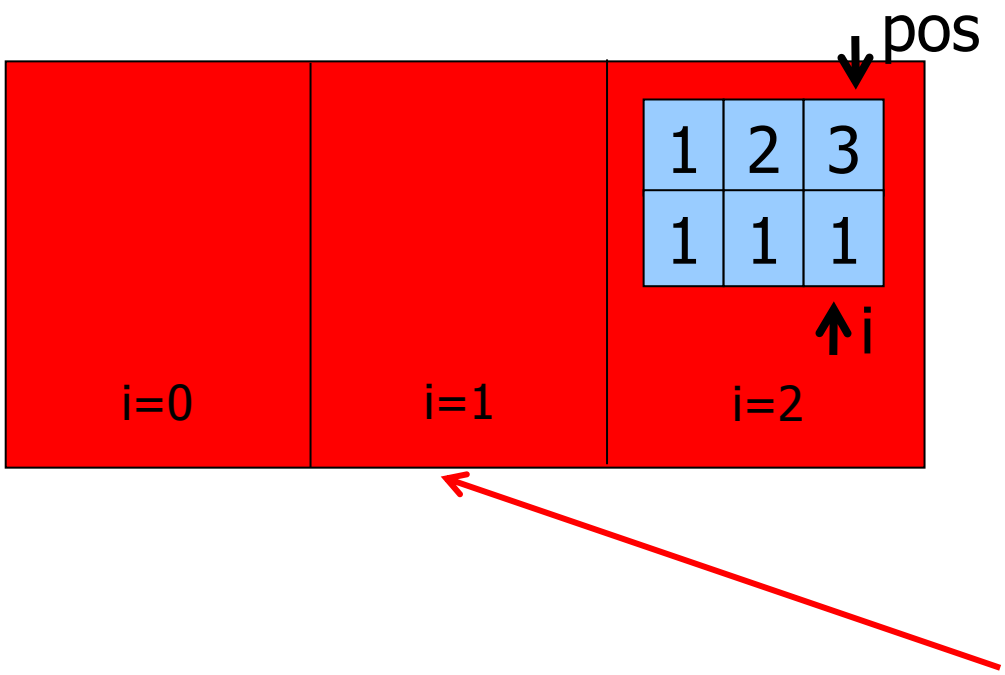
sol[pos] = val[i]

mark[i] = 1

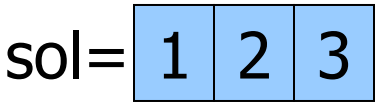

i
val=
1
2
3

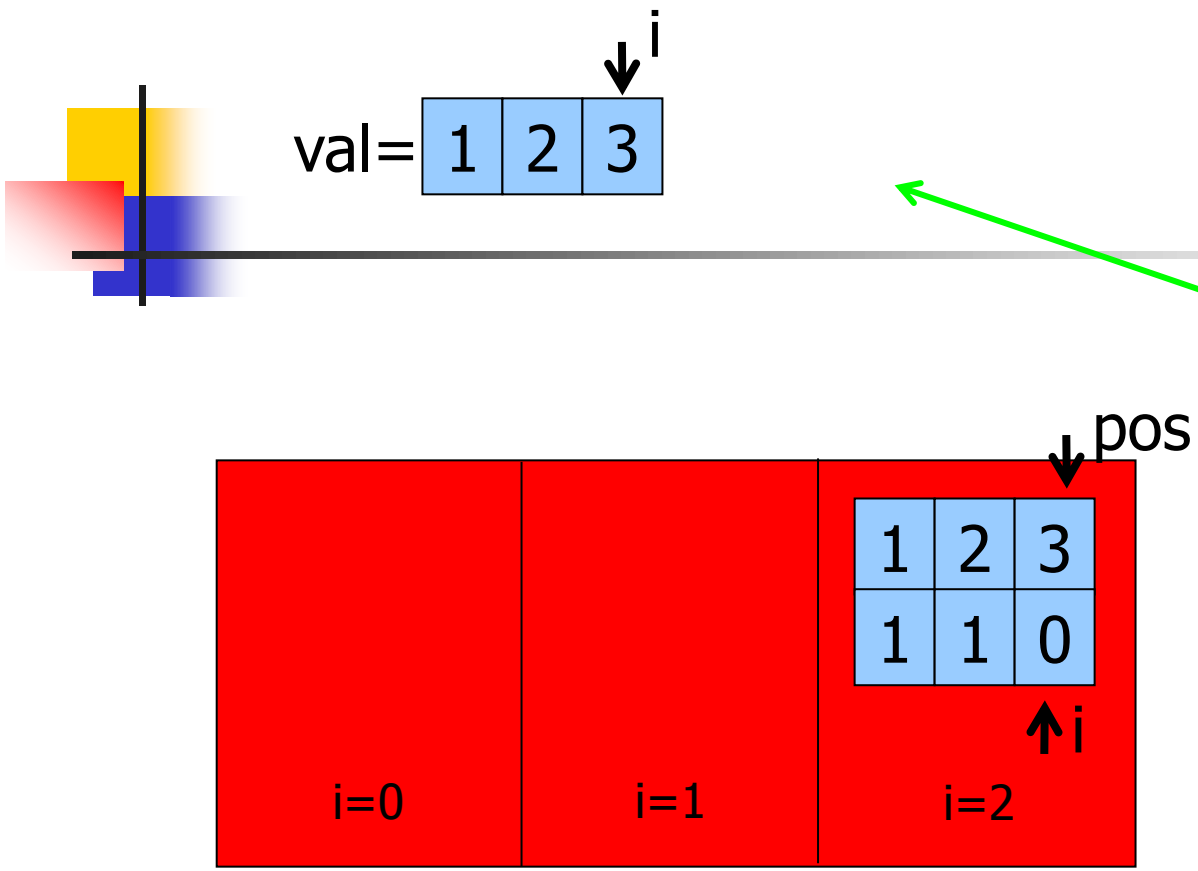

pos
1
2
3
1
1
1
i
i=0
i=1
i=2
ricorsione
con pos=3





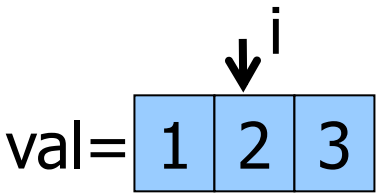
terminazione: visualizza, aggiorna count
ritorna

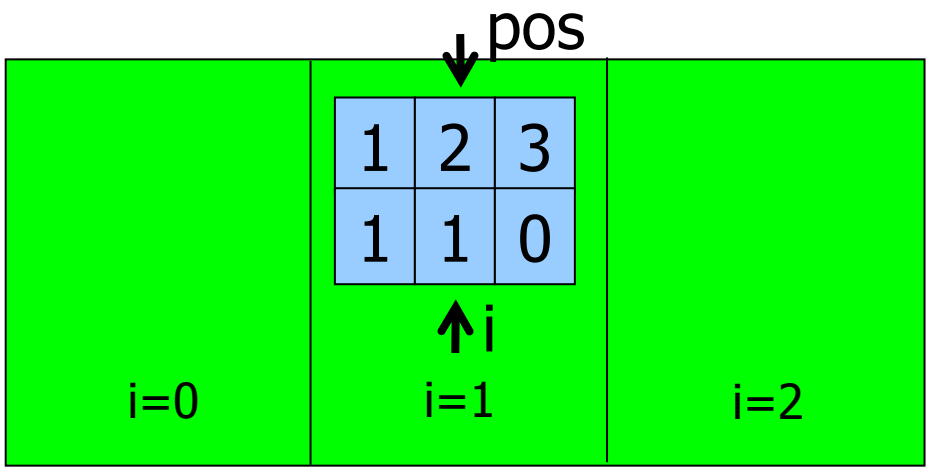


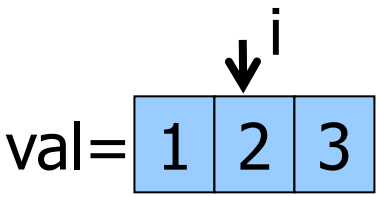


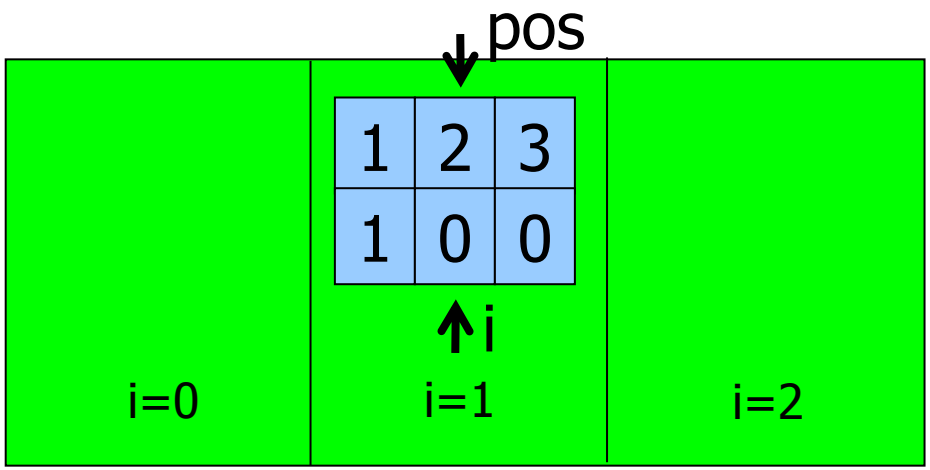
smarca mark[i]

ciclo for terminato, ritorna

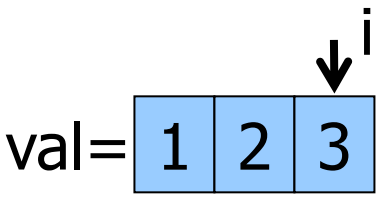

i
val=
1
2
3

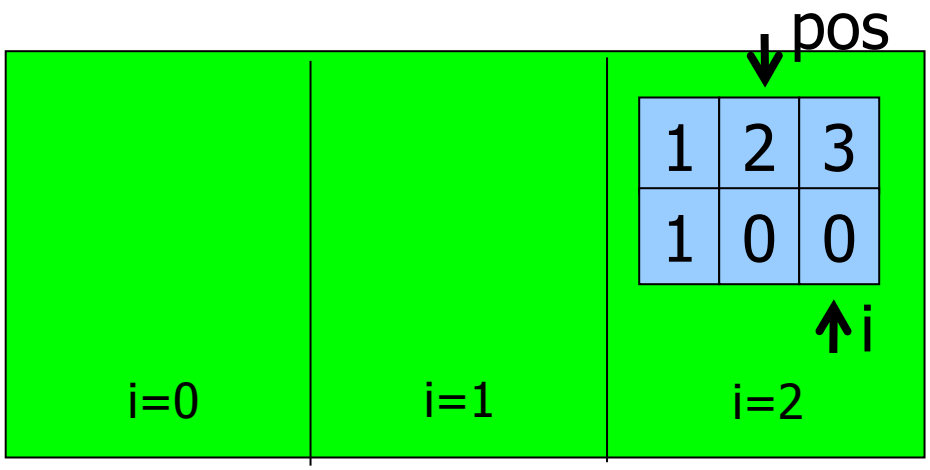

pos
1
2
3
1
1
0
i
i=0
i=1
i=2





smarca mark[i]

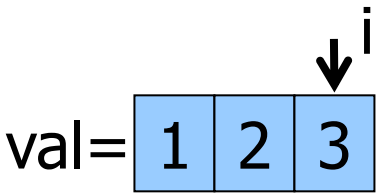


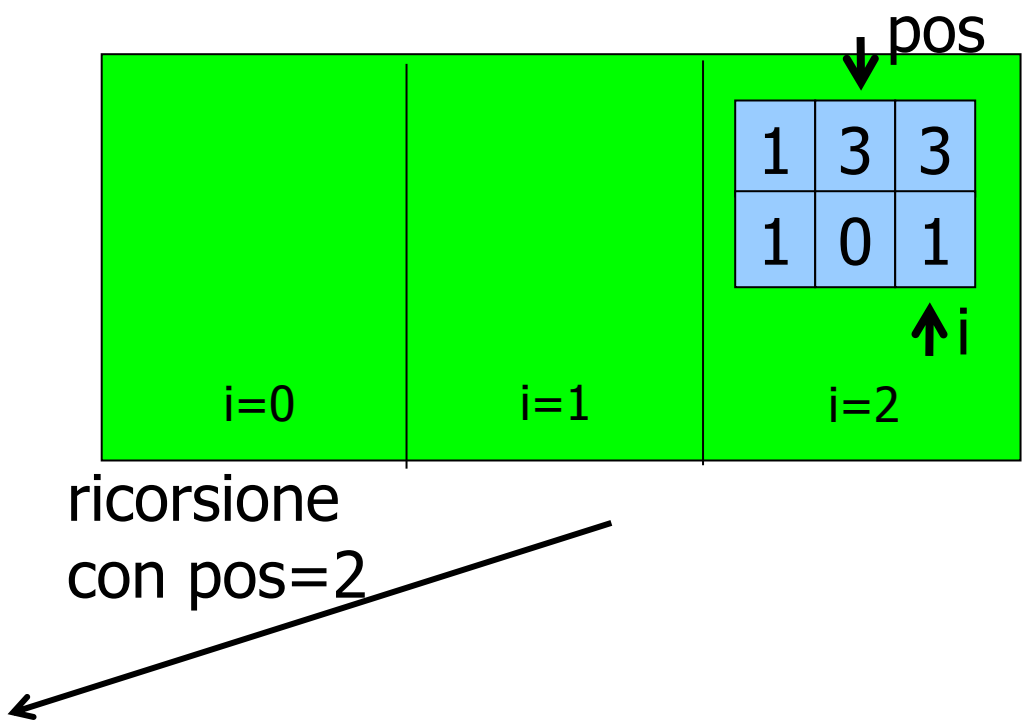


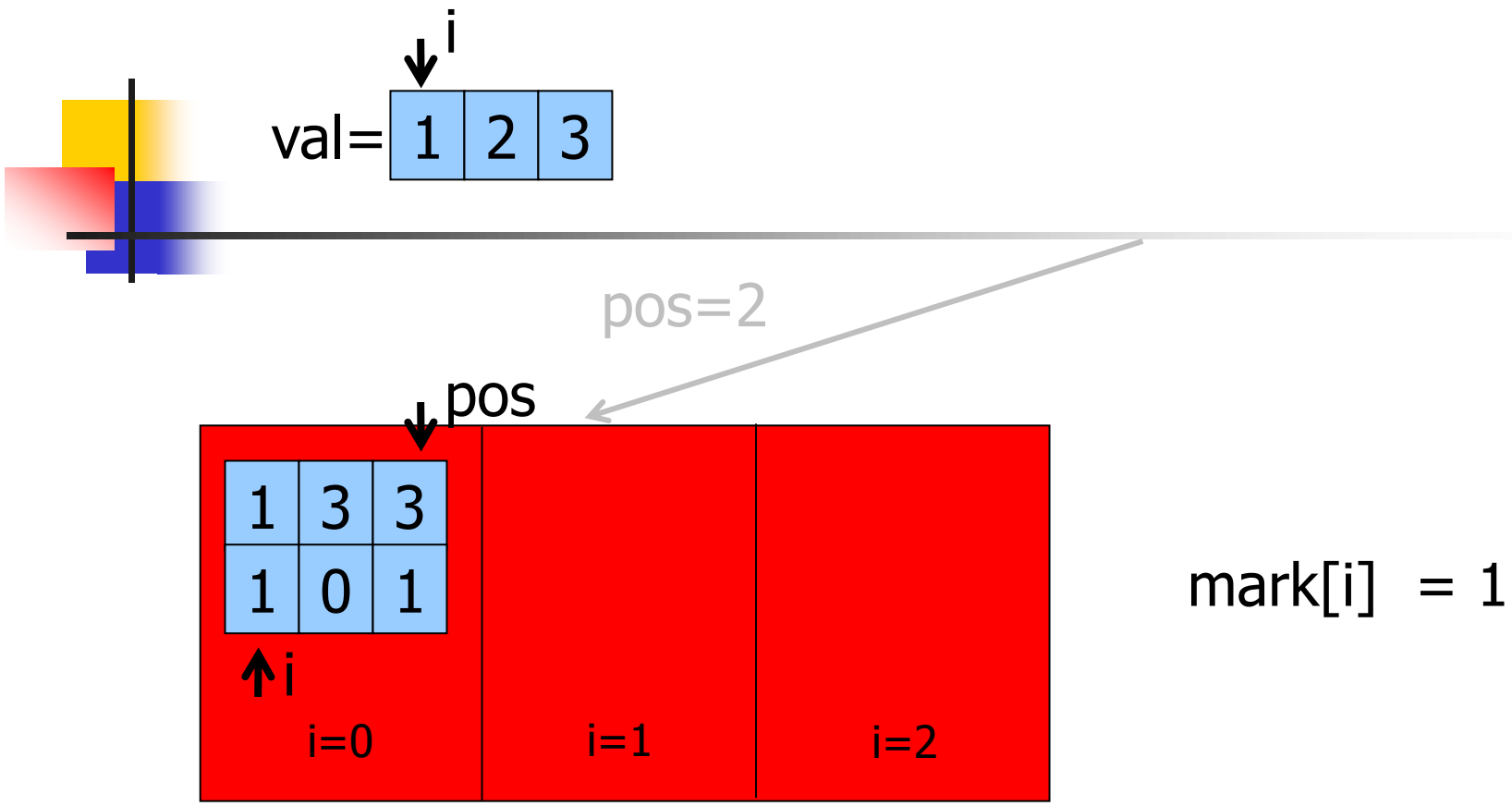
mark[i] = 0

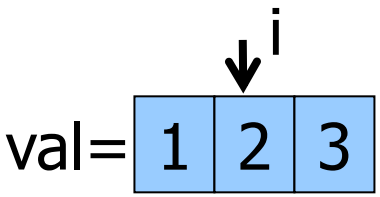
sol[pos] = val[i]

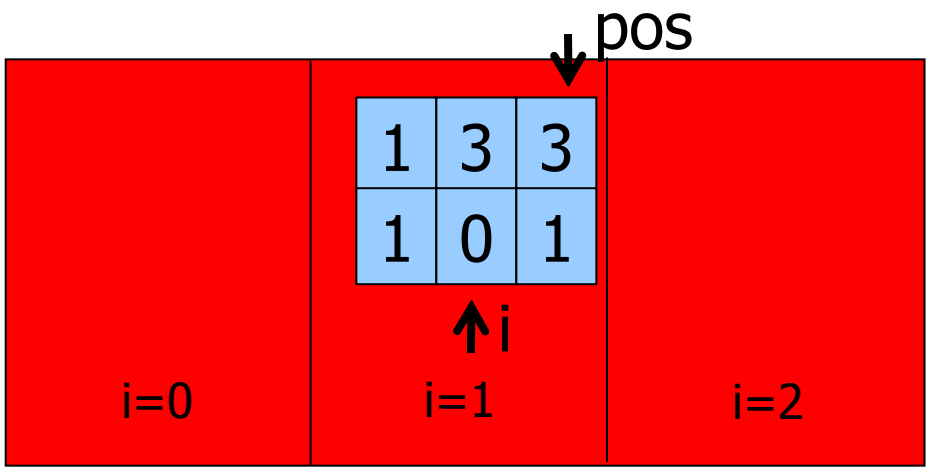
mark[i] = 1


i
val=
1
2
3


pos
1
3
3
1
0
1
i
i=0
i=1
i=2
ricorsione
con pos=2


i
val=
1
2
3
pos=2
pos
1
3
3
1
0
1
i
i=0
i=1
i=2
mark[i] = 1

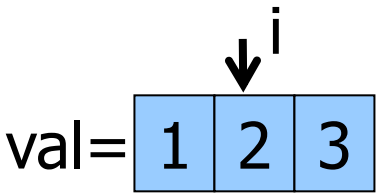


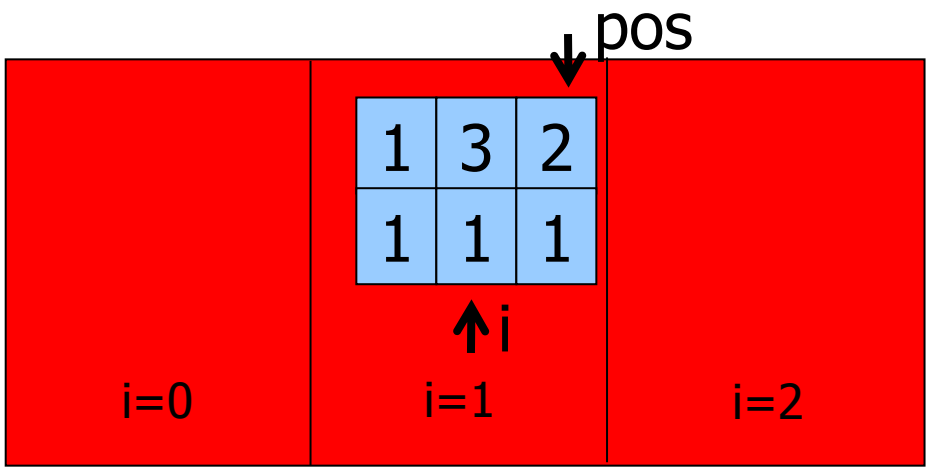


mark[i] = 0

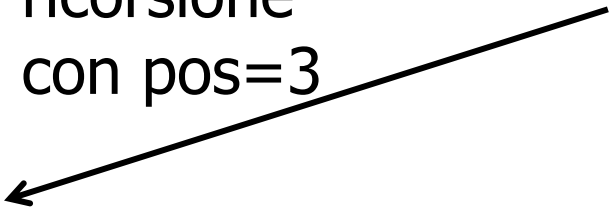
sol[pos] = val[i]

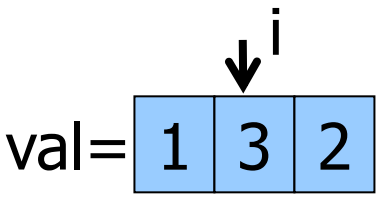
mark[i] = 1

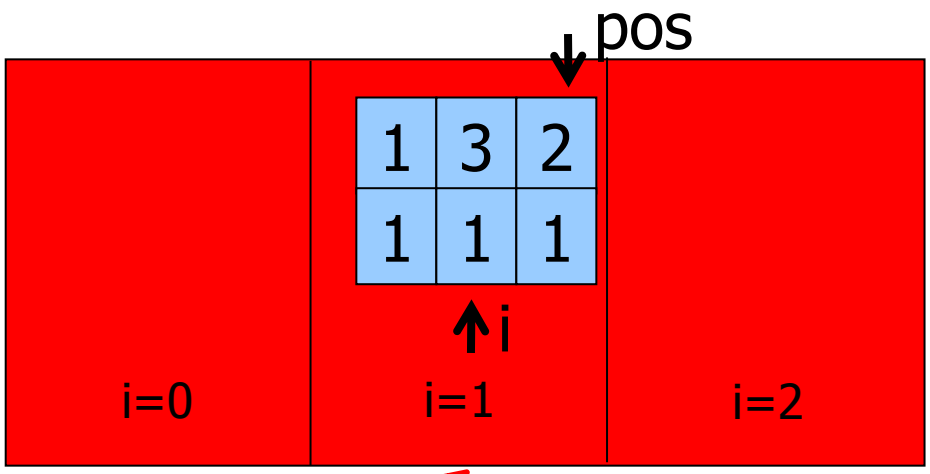




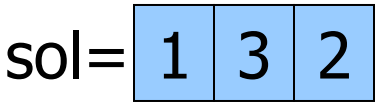
ricorsione
con pos=3

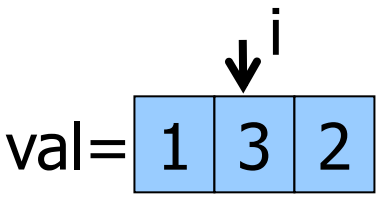


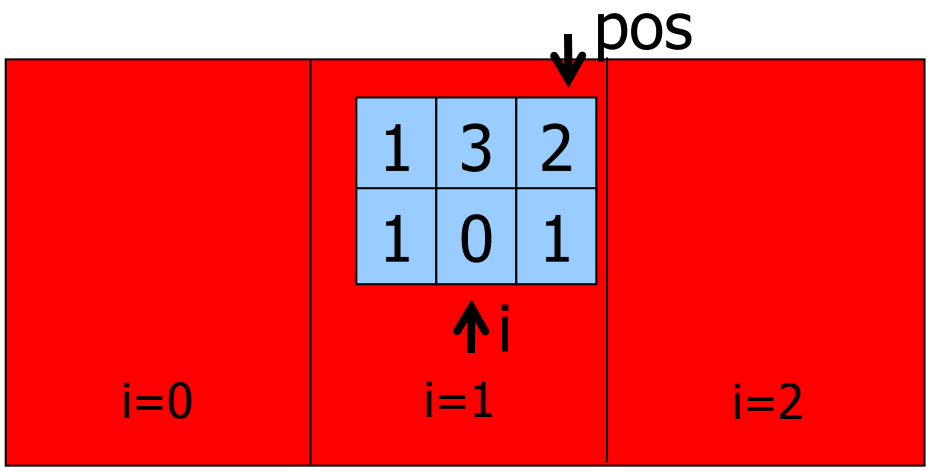




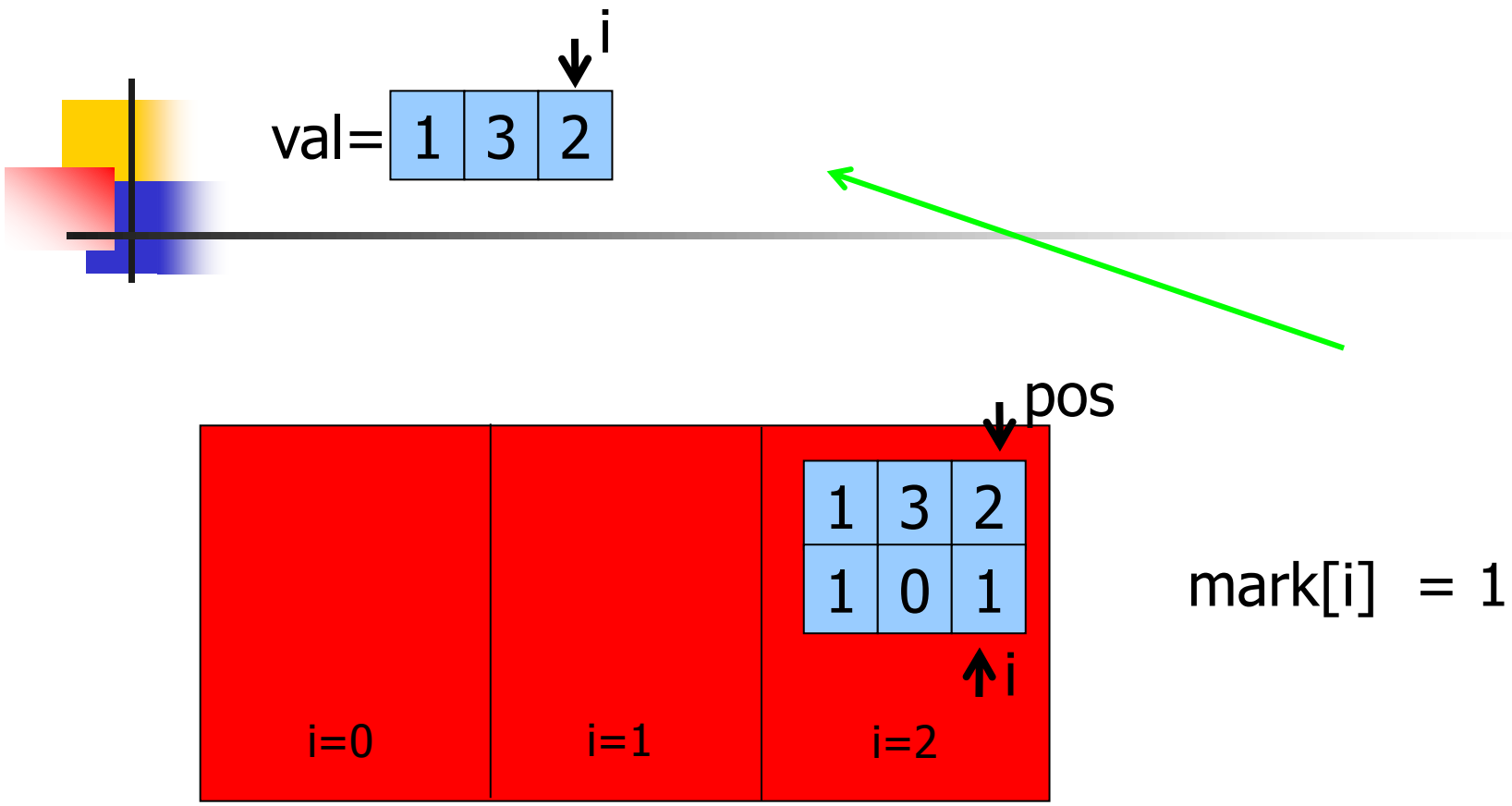
terminazione: visualizza, aggiorna count
ritorna



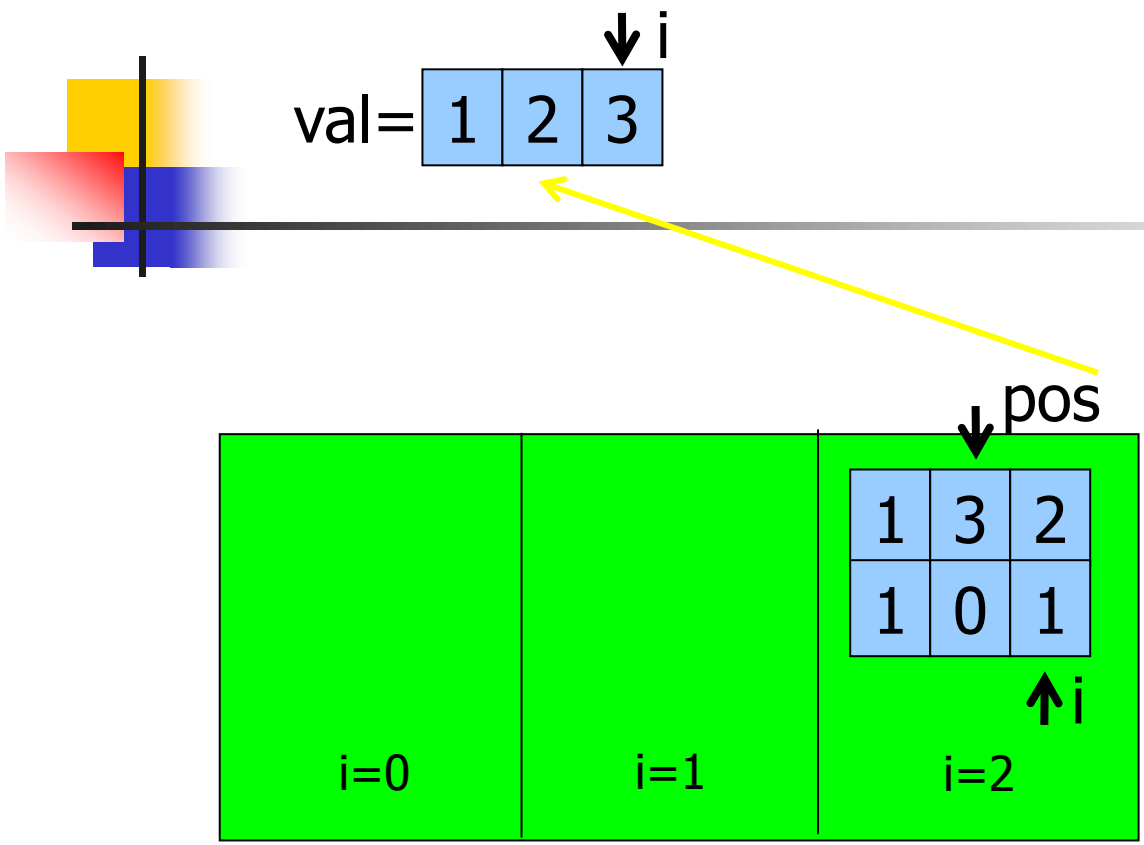




smarca mark[i]



ciclo for terminato, ritorna



ciclo for terminato, ritorna

etc. etc.

Esempio: anagrammi con ripetizioni

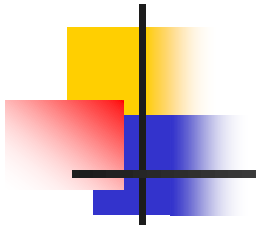
Si legga una stringa, se ne generino tutti gli anagrammi.

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

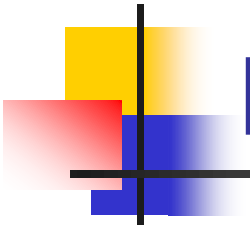
n! = 6.

I 6 anagrammi con ripetizione sono:

{ ORO, OOR, ROO, ROO, OOR, ORO }

02anagrammi_con_ripetizioni

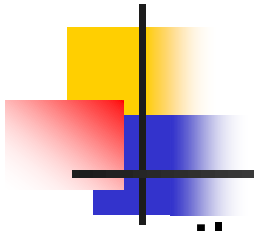
```
int anagrammi(int pos, char *sol, char *val
              int *mark, int n,  int count) {
  int i;
  if (pos >= n) {
     sol[pos] = '\0';
     printf("%s\n", sol);
     return count+1;
  }
  for (i=0; i<n; i++)
    if (mark[i] == 0) {
      mark[i] = 1;
      sol[pos] = val[i];
      count = anagrammi(pos+1,sol,val,mark,n,count);
      mark[i] = 0;
    }
  return count;
}
```



Permutazioni ripetute

Si procede in maniera analoga alle permutazioni semplici, con le seguenti variazioni:

- `n` è la cardinalità del multiset
- si memorizzano nel vettore `dist_val` di `n_dist` celle gli elementi distinti del multiset
  - si ordina il vettore `val` con un algoritmo O(nlogn)
  - si «compatta» `val` eliminando gli elementi duplicati e lo si memorizza in dist_`val`



- 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
- `count` registra il numero di soluzioni.

```
val = malloc(n*sizeof(int));
dist_val = malloc(n*sizeof(int));
sol = malloc(k*sizeof(int));
```


```
int perm_r(int pos, int *dist_val, int *sol,
           int *mark, int n, int n_dist, 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<n_dist; i++) {
    if (mark[i] > 0) {
      mark[i]--;
      sol[pos] = dist_val[i];
      cnt=perm_r(pos+1,dist_val,sol,mark,n, n_dist,count);
      mark[i]++;
    }
  }
  return count;
}
```

terminazione

iterazione sulle n_dist scelte

controllo occorrenze

marcamento e scelta

ricorsione

smarcamento

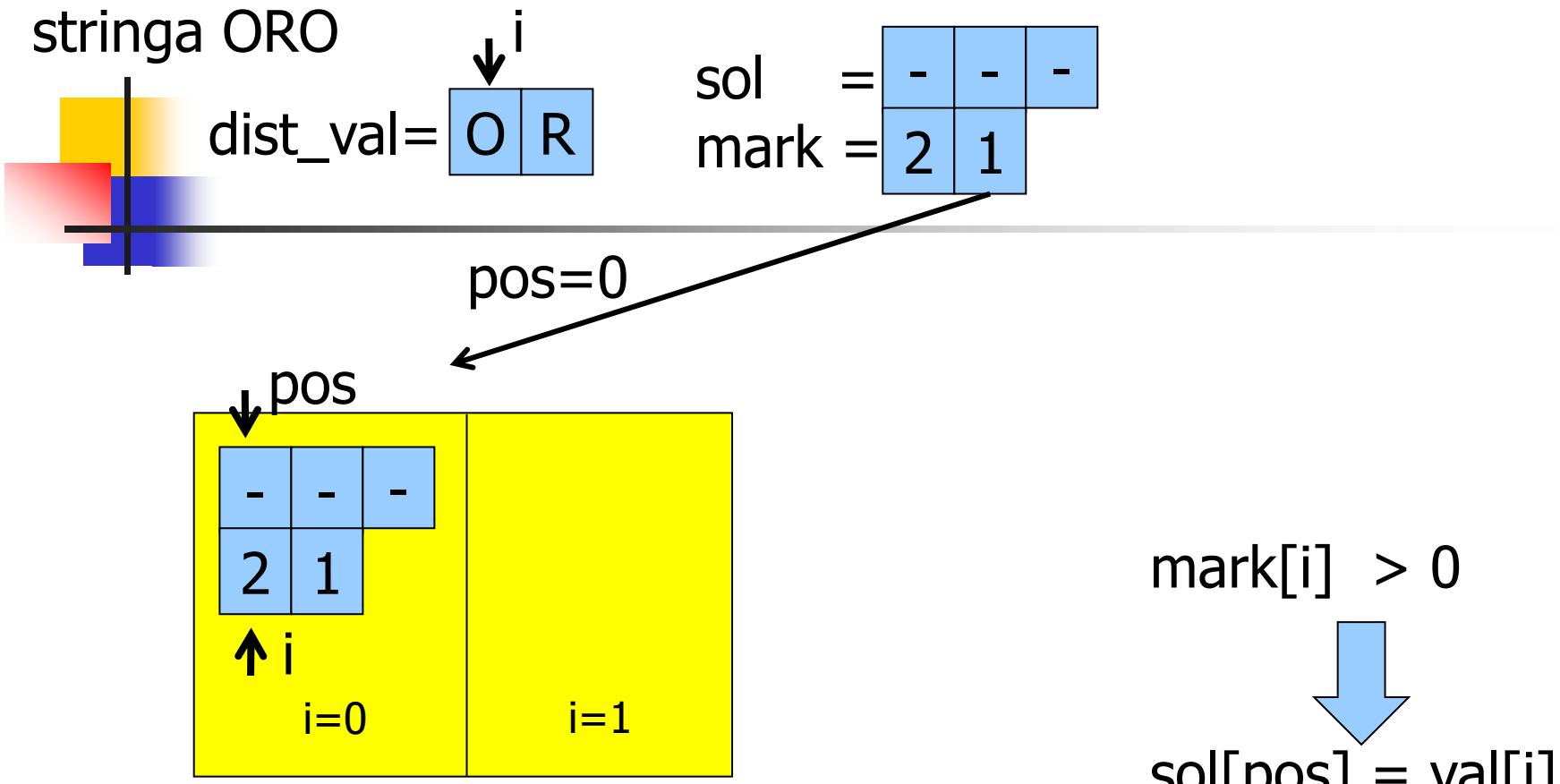
Esempio: anagrammi distinti

Data una stringa con lettere eventualmente ripetute, generare tutti i suoi anagrammi distinti.

Esempio:
stringa ORO n = 3
`dist_val = { O, R }, n_dist = 2`

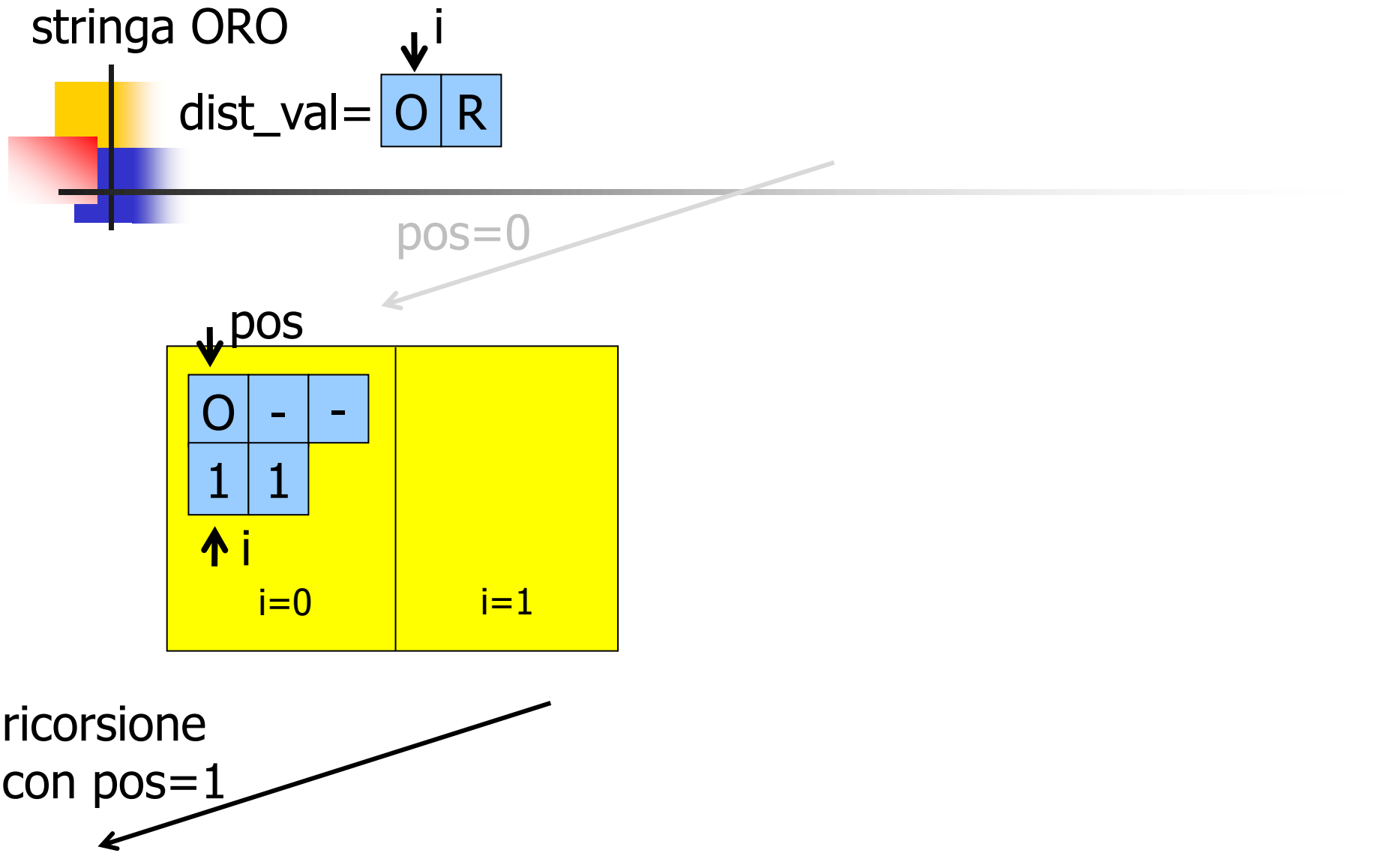
$P^{(2)}_3 = 3!/2! = 3$

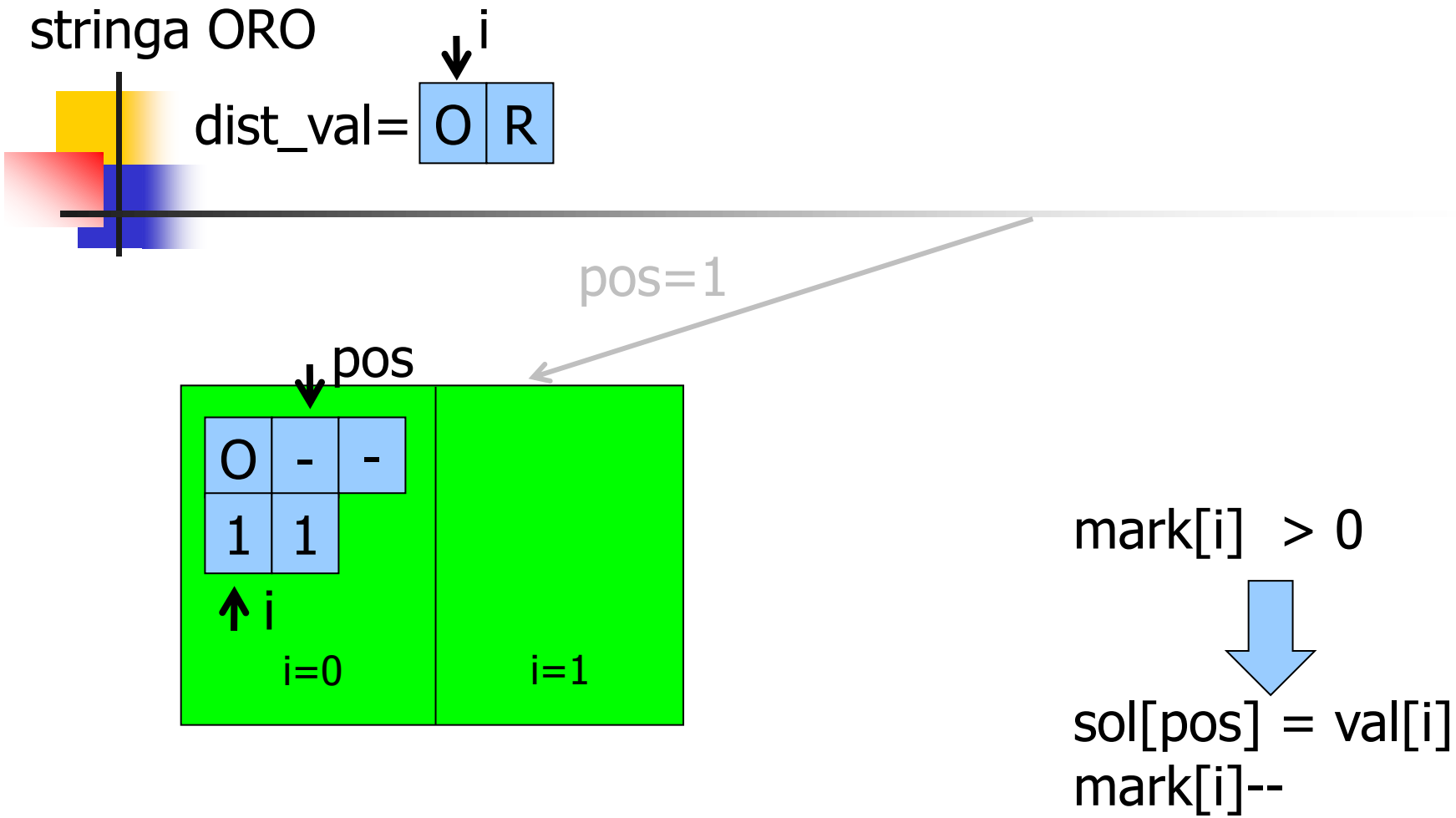
Soluzione
{ OOR, ORO, ROO }

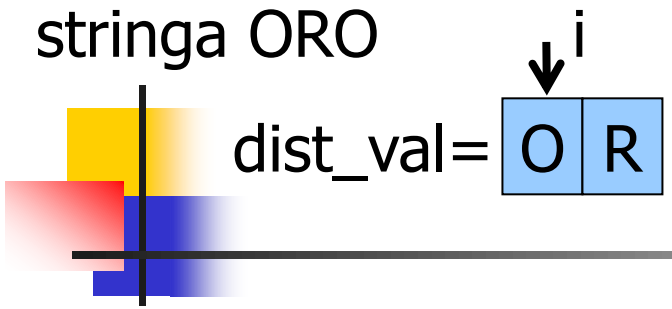


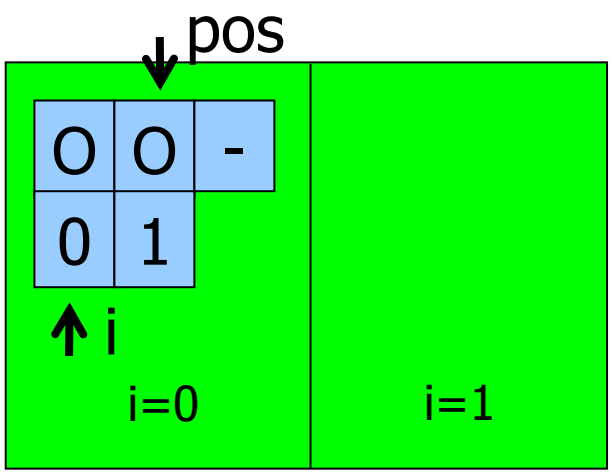
mark[i] > 0

sol[pos] = val[i]
mark[i]--

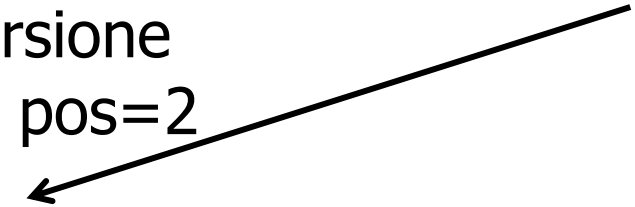

stringa ORO
i
dist_val=
O
R
pos=0
pos
O
-
-
1
1
i
i=0
i=1
ricorsione
con pos=1

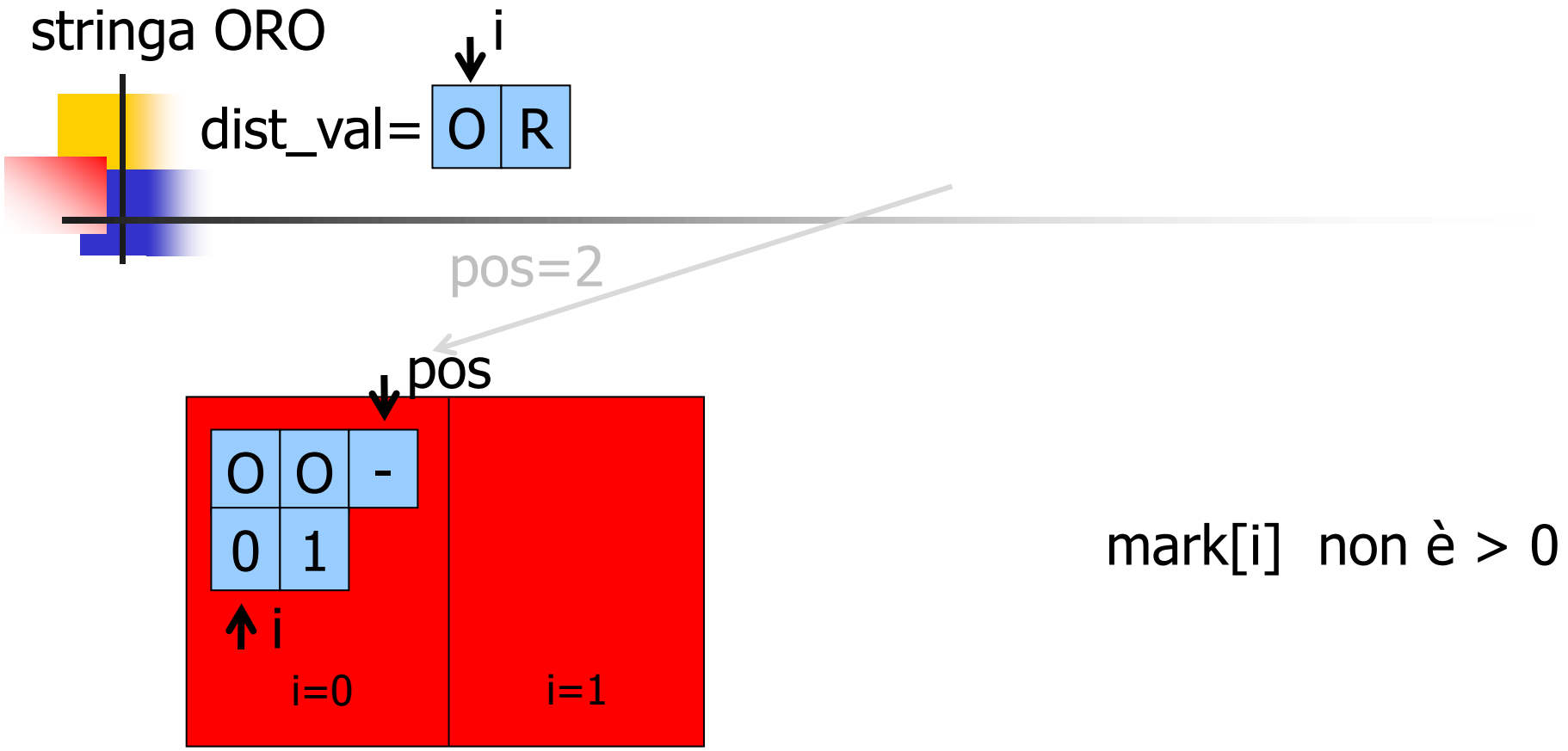

stringa ORO
i
dist_val=
O
R
pos=1
pos
O
-
-
1
1
i
i=0
i=1
mark[i] > 0
sol[pos] = val[i]
mark[i]--





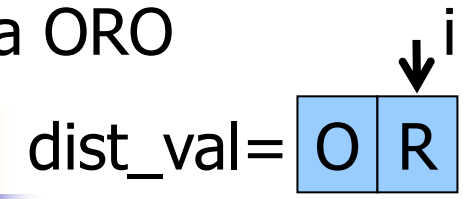
ricorsione
con pos=2

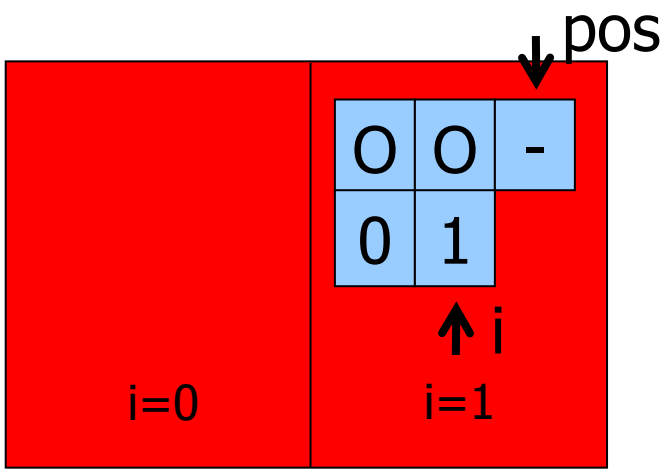




mark[i] non è > 0

stringa ORO

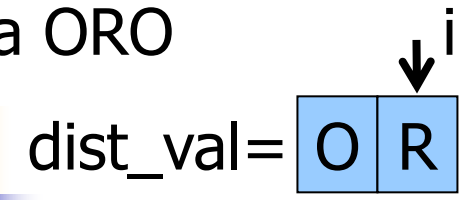


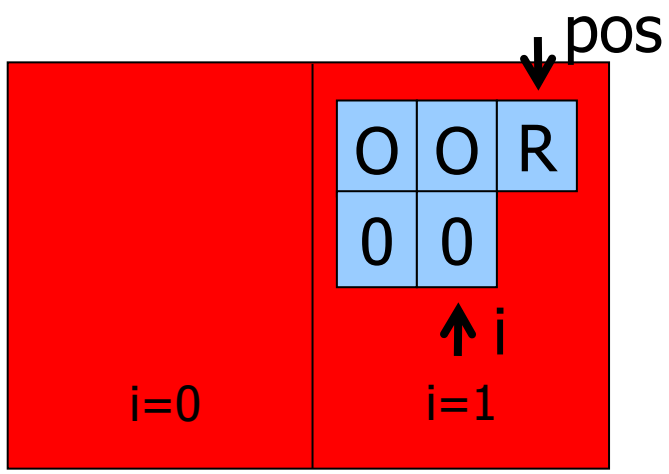


mark[i] > 0

sol[pos] = val[i]
mark[i]--

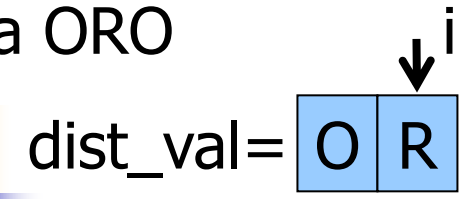
stringa ORO

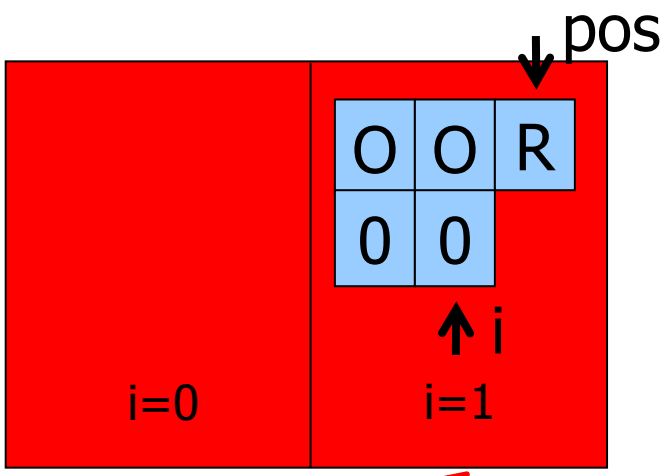




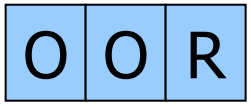
ricorsione
con pos=3

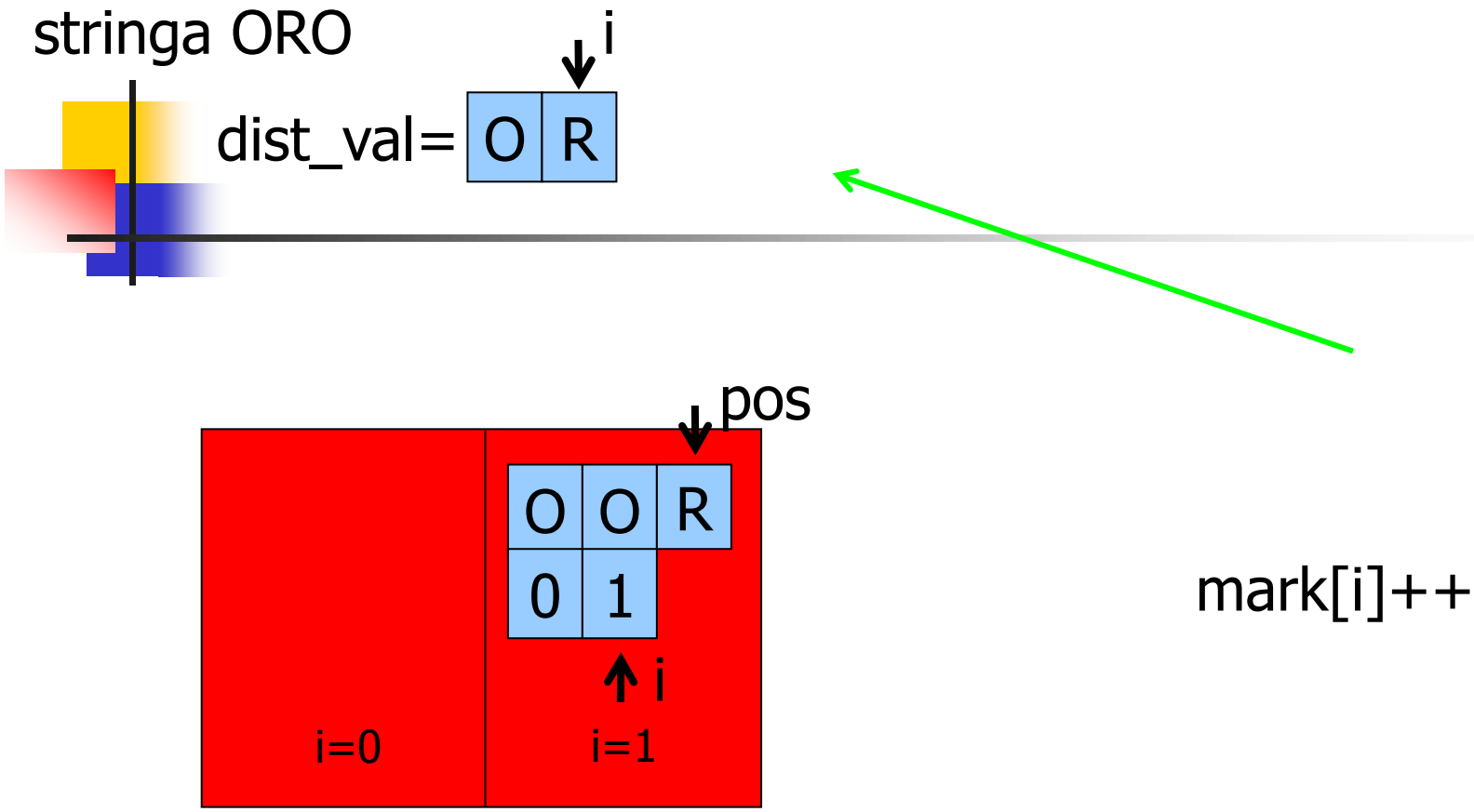
stringa ORO





terminazione: visualizza, aggiorna count
ritorna

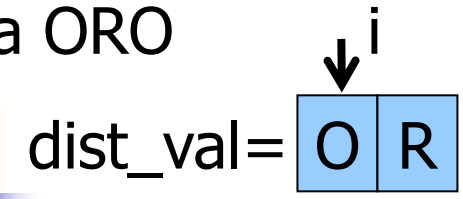


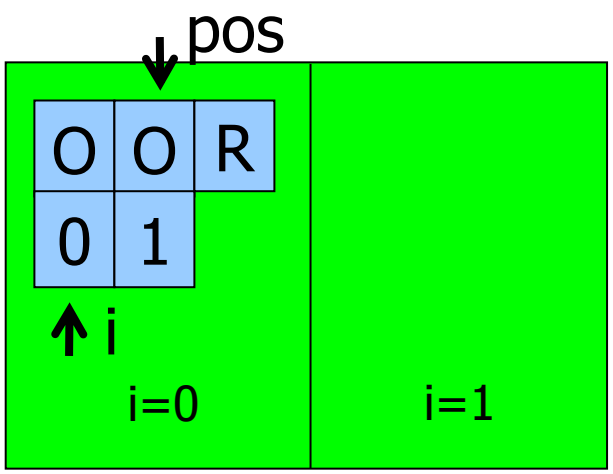


backtrack

ciclo for terminato, ritorna

stringa ORO

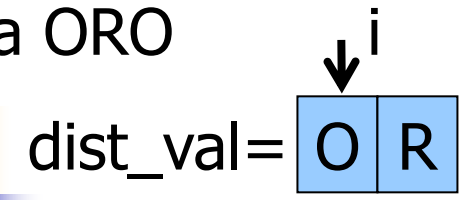


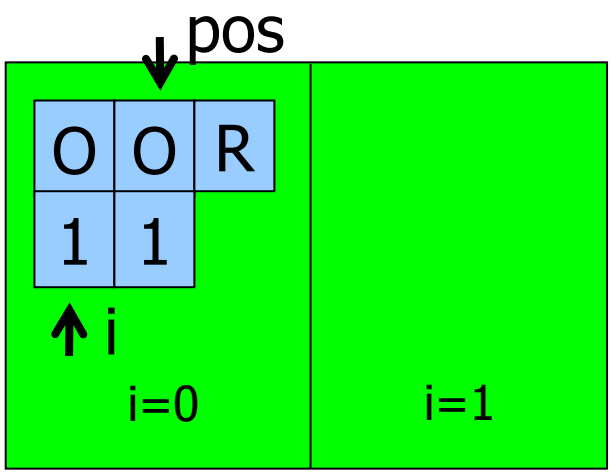


mark[i]++

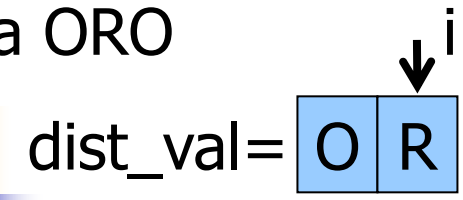
backtrack

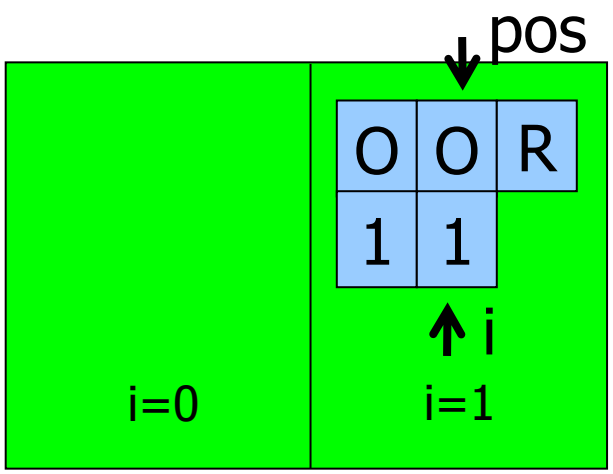
stringa ORO





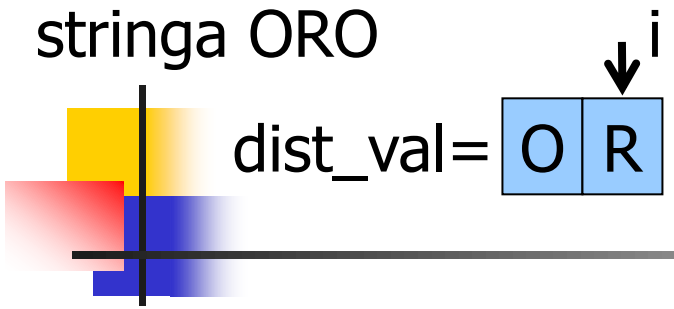
stringa ORO

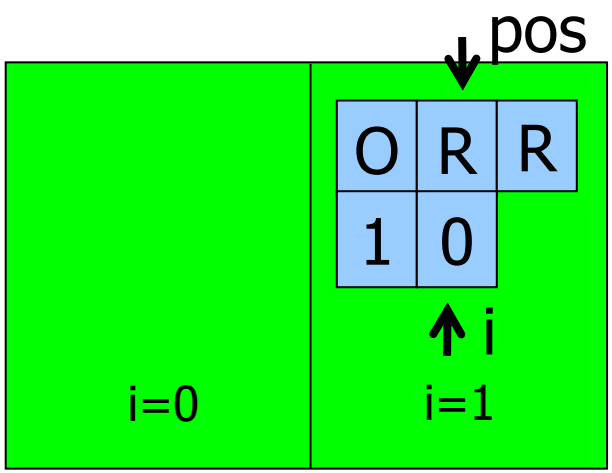




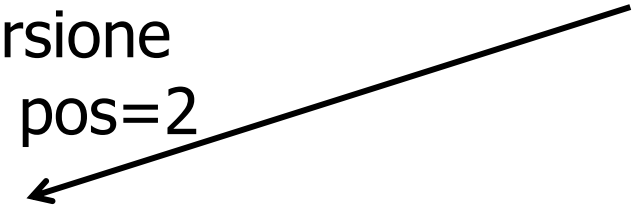
mark[i] > 0

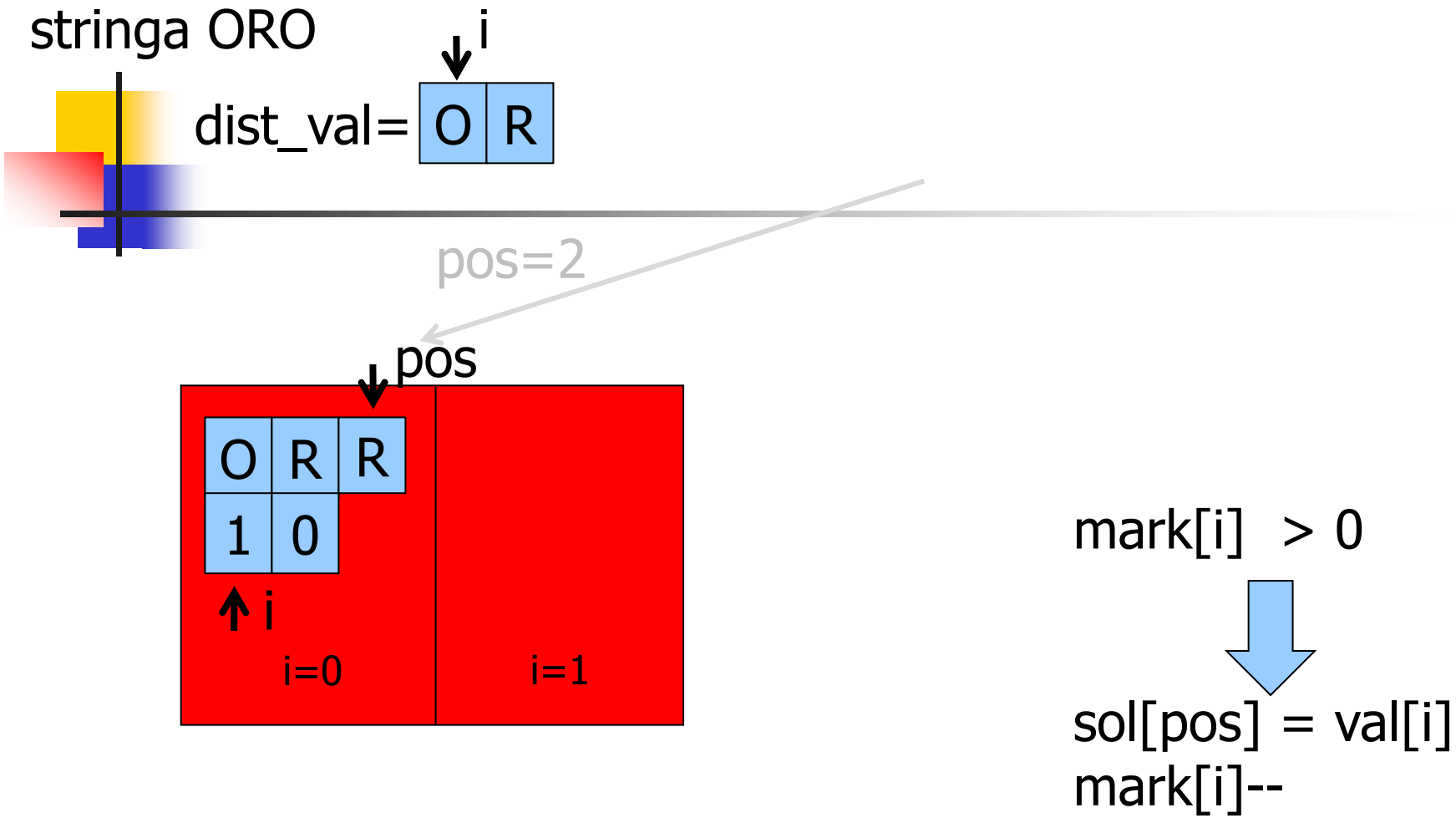
sol[pos] = val[i]
mark[i]--

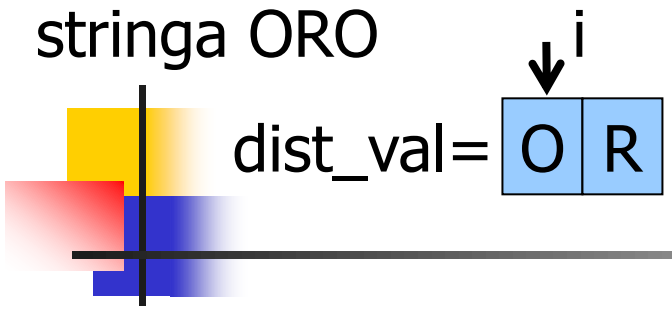


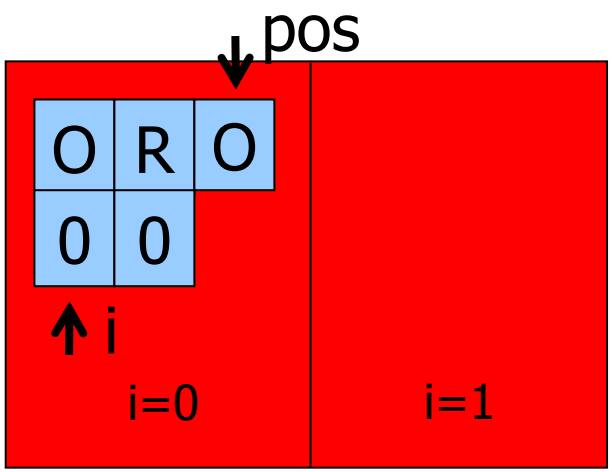


ricorsione
con pos=2

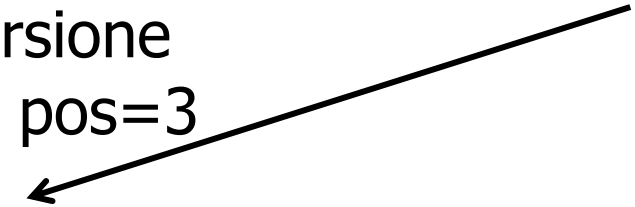



stringa ORO
i
dist_val=
O
R
pos=2
pos
O
R
R
1
0
i
i=0
i=1
mark[i] > 0
sol[pos] = val[i]
mark[i]--

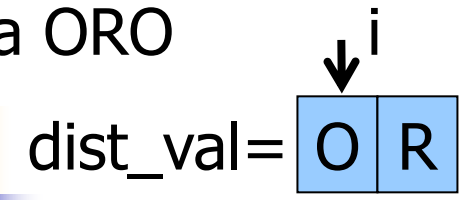


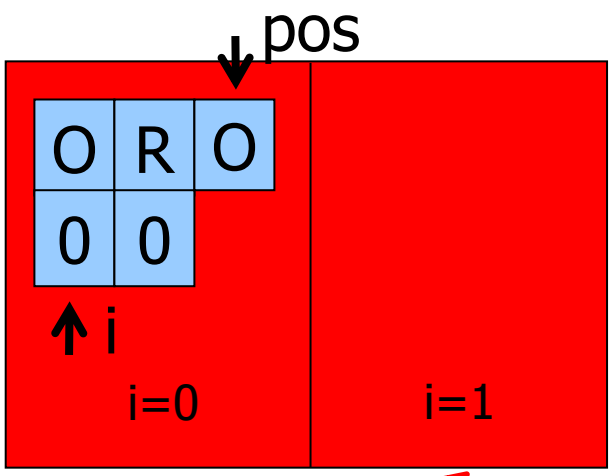


ricorsione
con pos=3

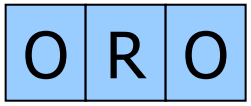


stringa ORO

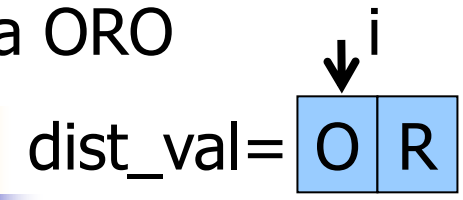


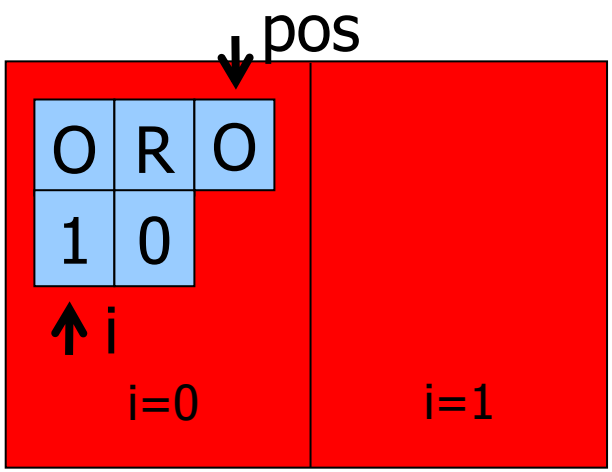


terminazione: visualizza, aggiorna count ritorna



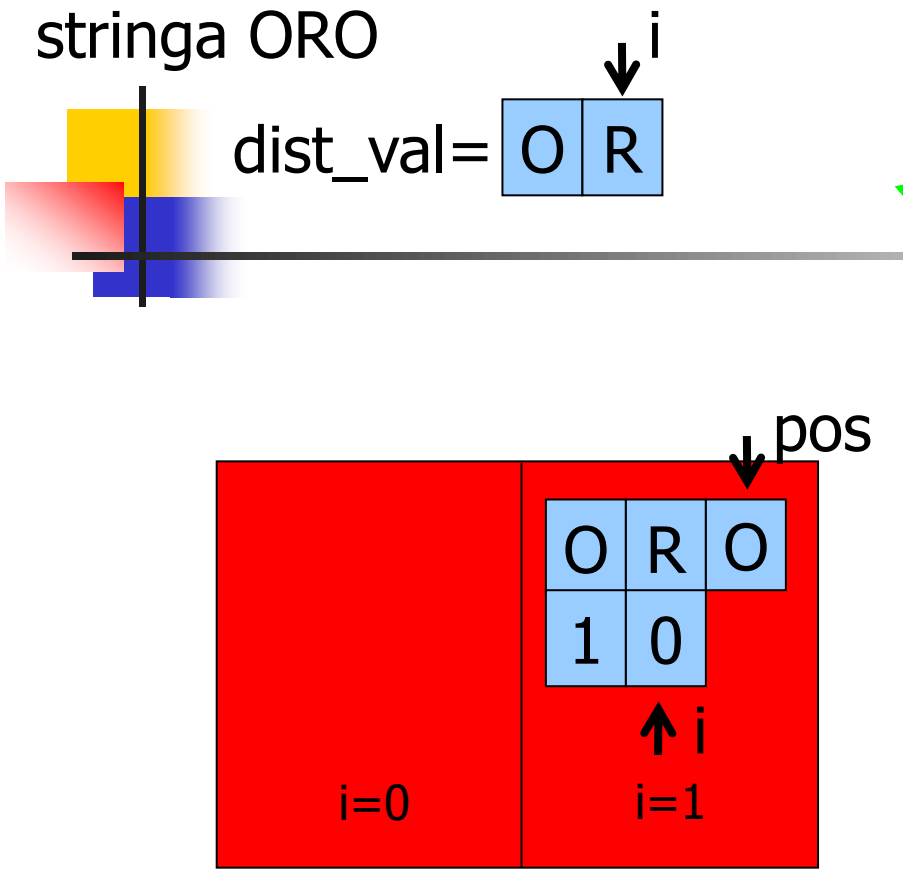
stringa ORO





mark[i]++

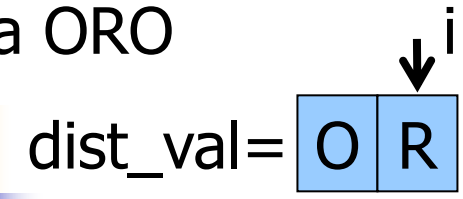
backtrack

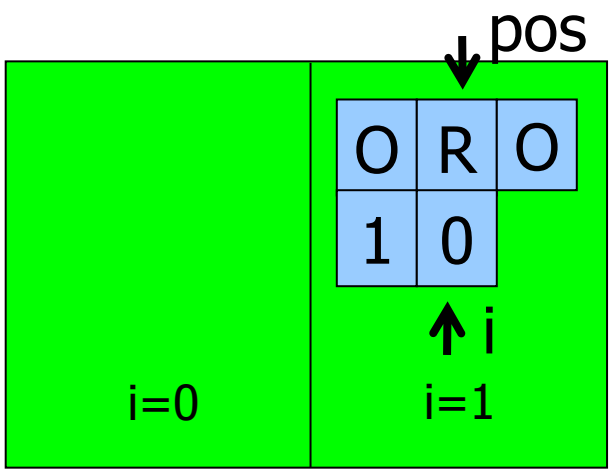


mark[i] non è > 0

ciclo for terminato, ritorna

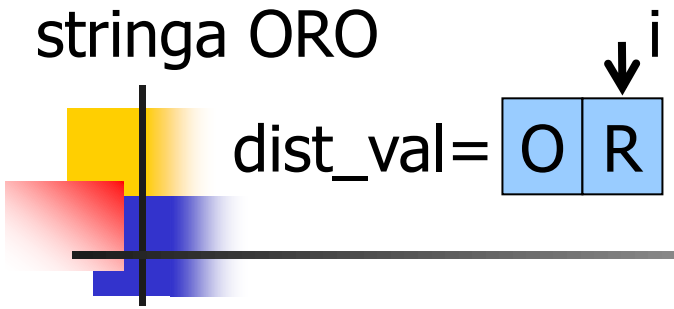
stringa ORO

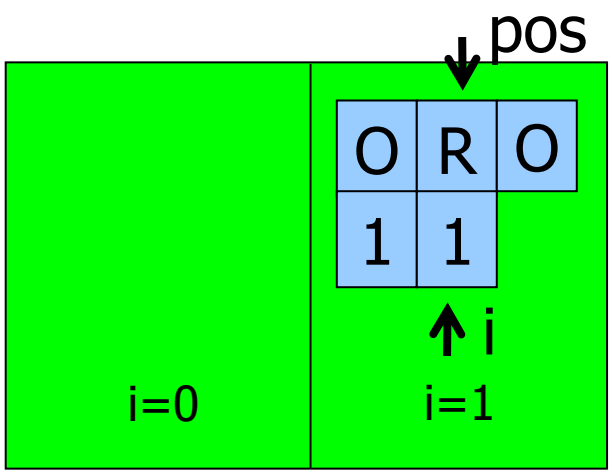




mark[i]++

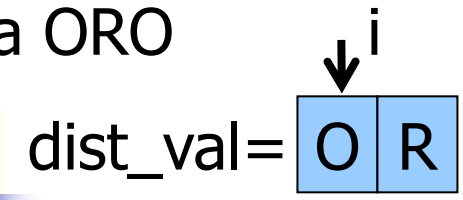
backtrack

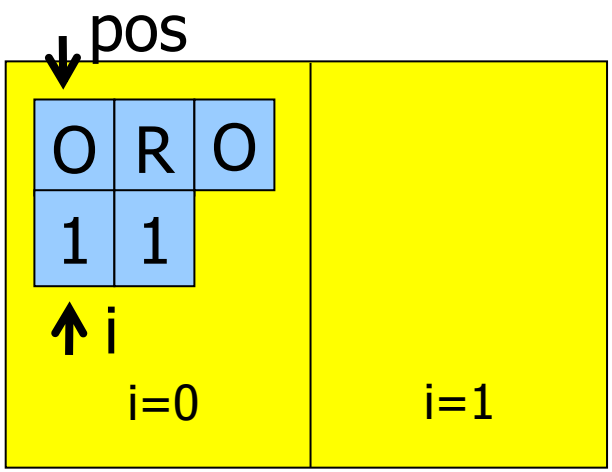




ciclo for terminato, ritorna

stringa ORO

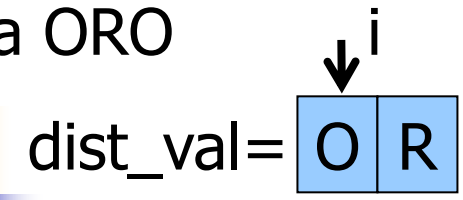


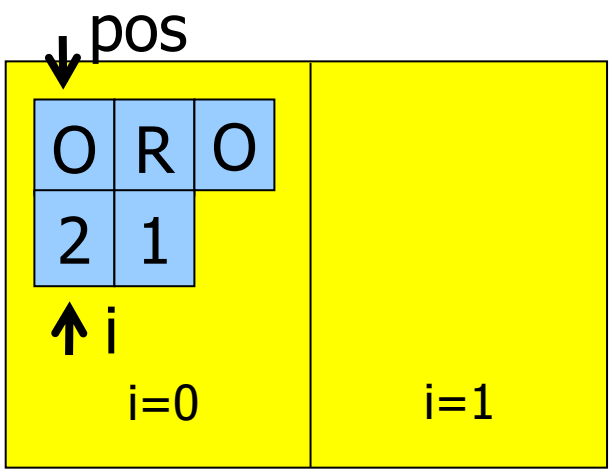


mark[i]++

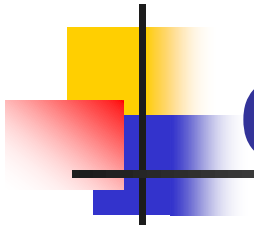
backtrack

stringa ORO





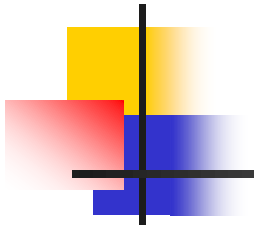
etc. etc.



Combinazioni semplici

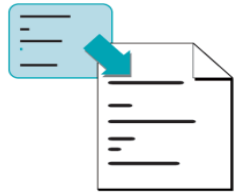
Rispetto alle disposizioni semplici si tratta di «forzare» uno dei possibili ordinamenti:

- 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`
- `count` registra il numero di soluzioni.

```
val = malloc(n * sizeof(int));
sol = malloc(k * sizeof(int));
```




```
int comb(int pos, int *val, int *sol, int n, int k,
         int start, int
  int i, j;
  if (pos >= k) {
    for (i=0; i<k; i++)
      printf("%d ", sol[i]);
    printf("\n");
    return count+1;
  }
  for (i=start; i<n; i++) {
    sol[pos] = val[i];
    count = comb(pos+1, val, sol, n, k, i+1, count);
  }
  return count;
}
```

terminazione

iterazione sulle scelte

scelta: sol[pos] riempito con i valori possibili di `val` da `start` in poi

ricorsione su prossima posizione e prossima scelta

Esempio: combinazioni 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!) .

Esempi

per val = {7, 2, 0, 4, 1} :

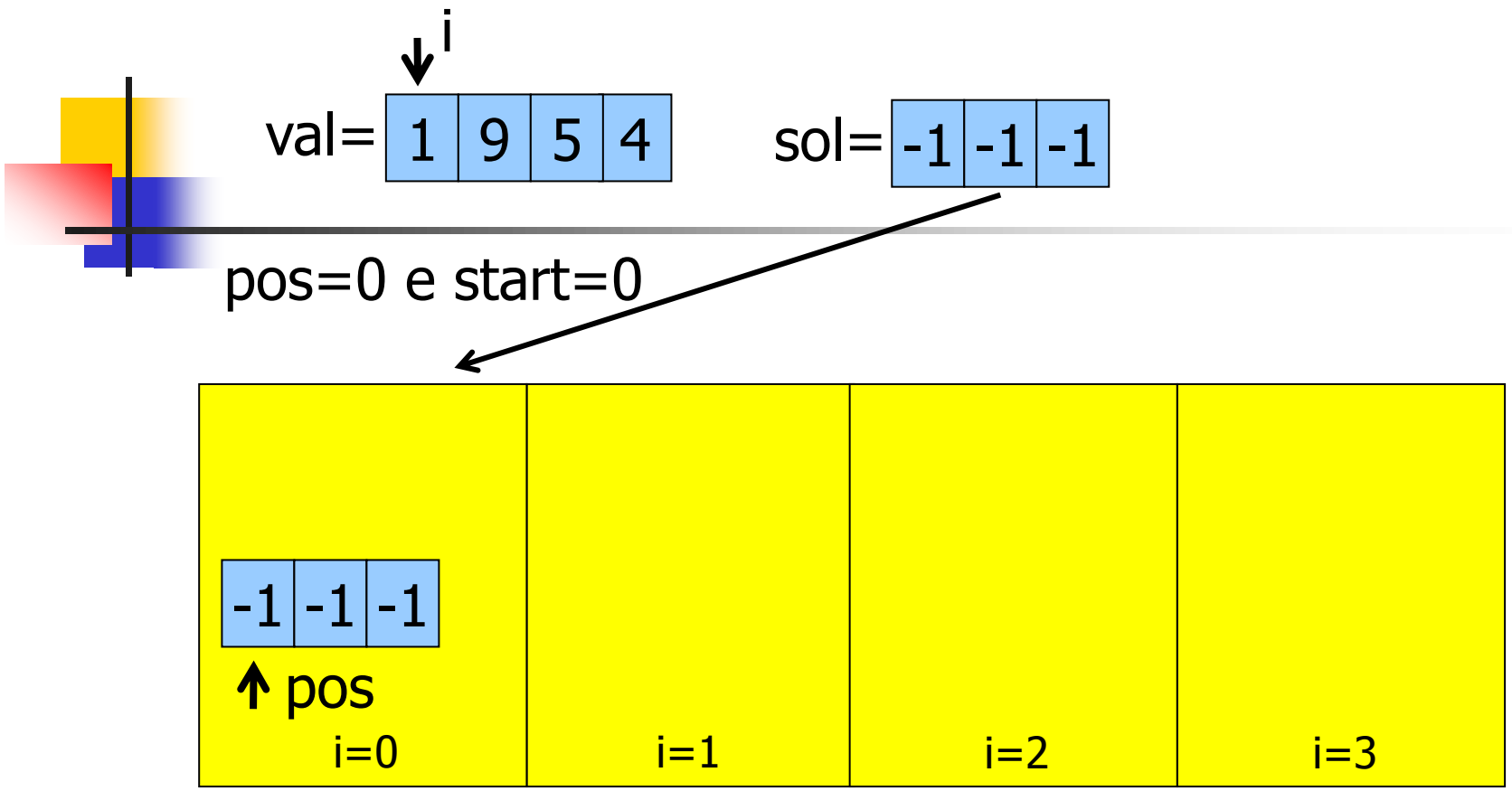
- n = 5 e k = 4: 5 combinazioni

{7,2,0,4} {7,2,0,1} {7,2,4,1} {7,0,4,1} {2,0,4,1}

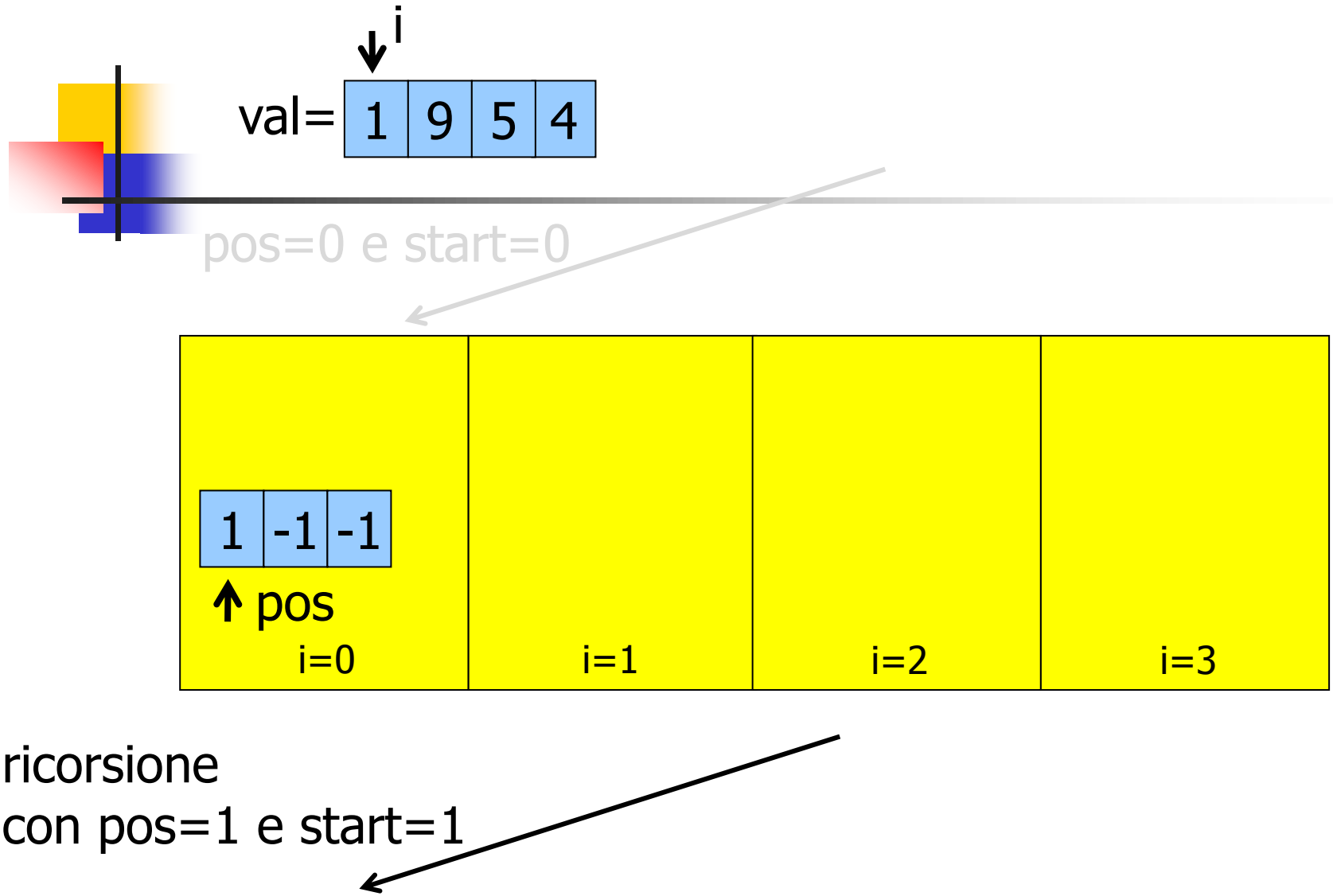
per val = {1, 9, 5, 4} :

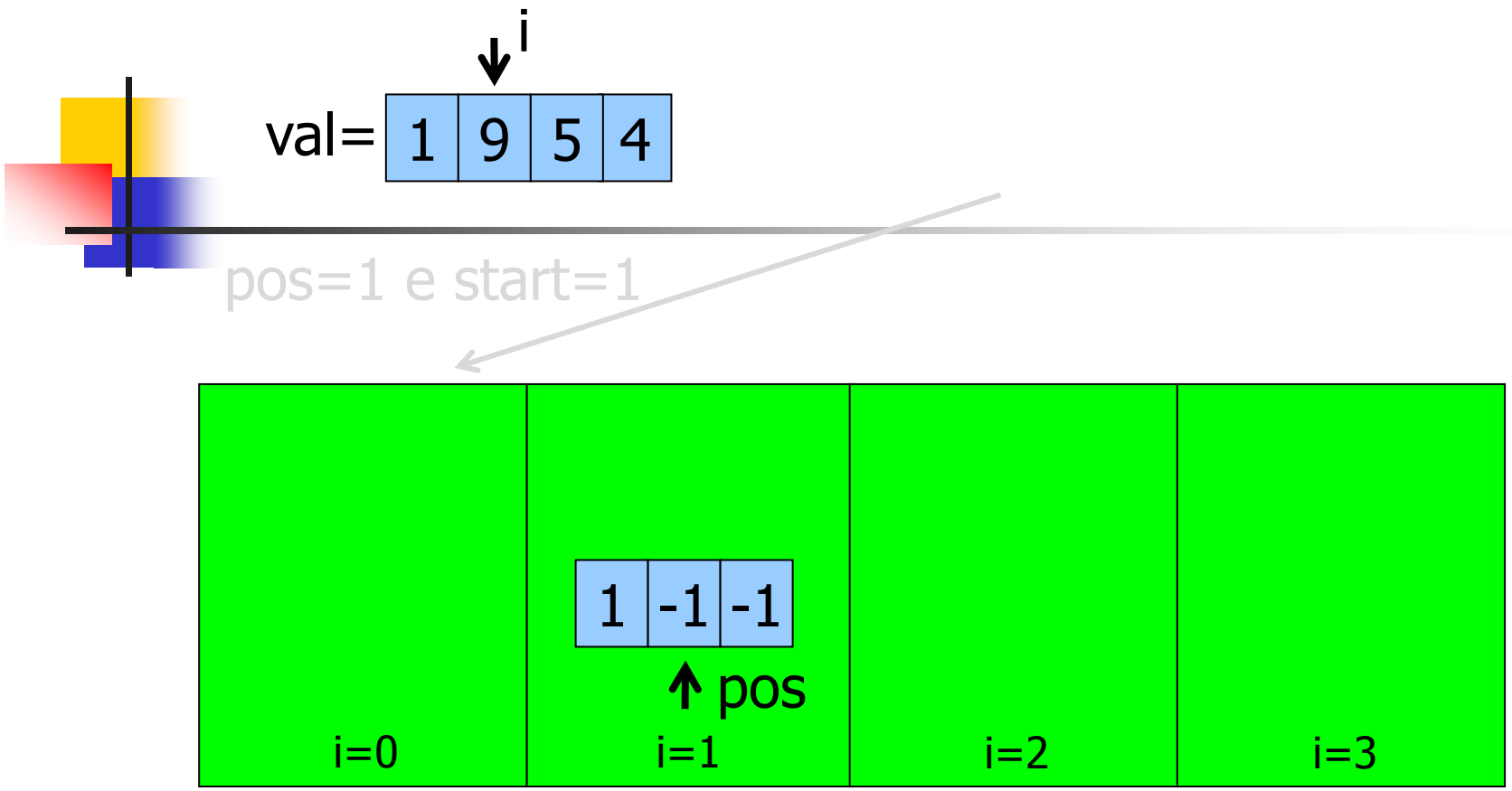
- n = 4 e k = 3: 4 combinazioni

{1,9,5} {1,9,4} {1,5,4} {9,5,4}

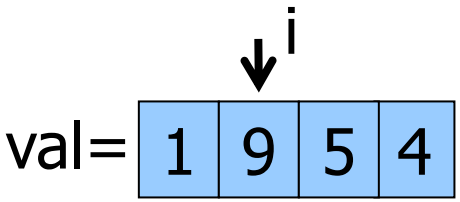


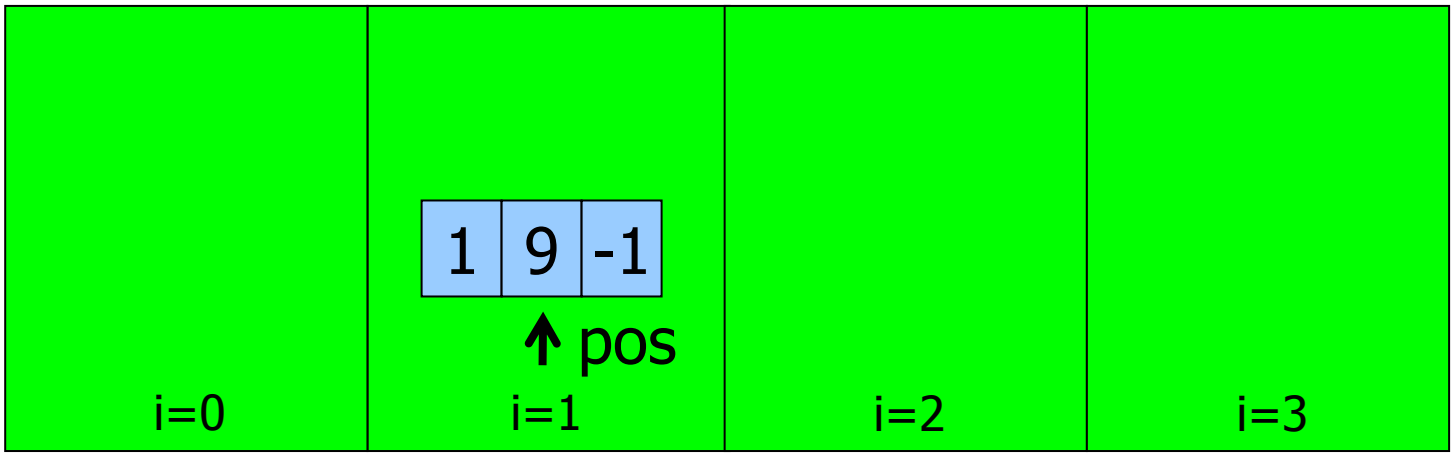
sol[pos] = val[i]


i
val=
1
9
5
4
pos=0 e start=0
1
-1
-1
pos
i=0
i=1
i=2
i=3
ricorsione
con pos=1 e start=1

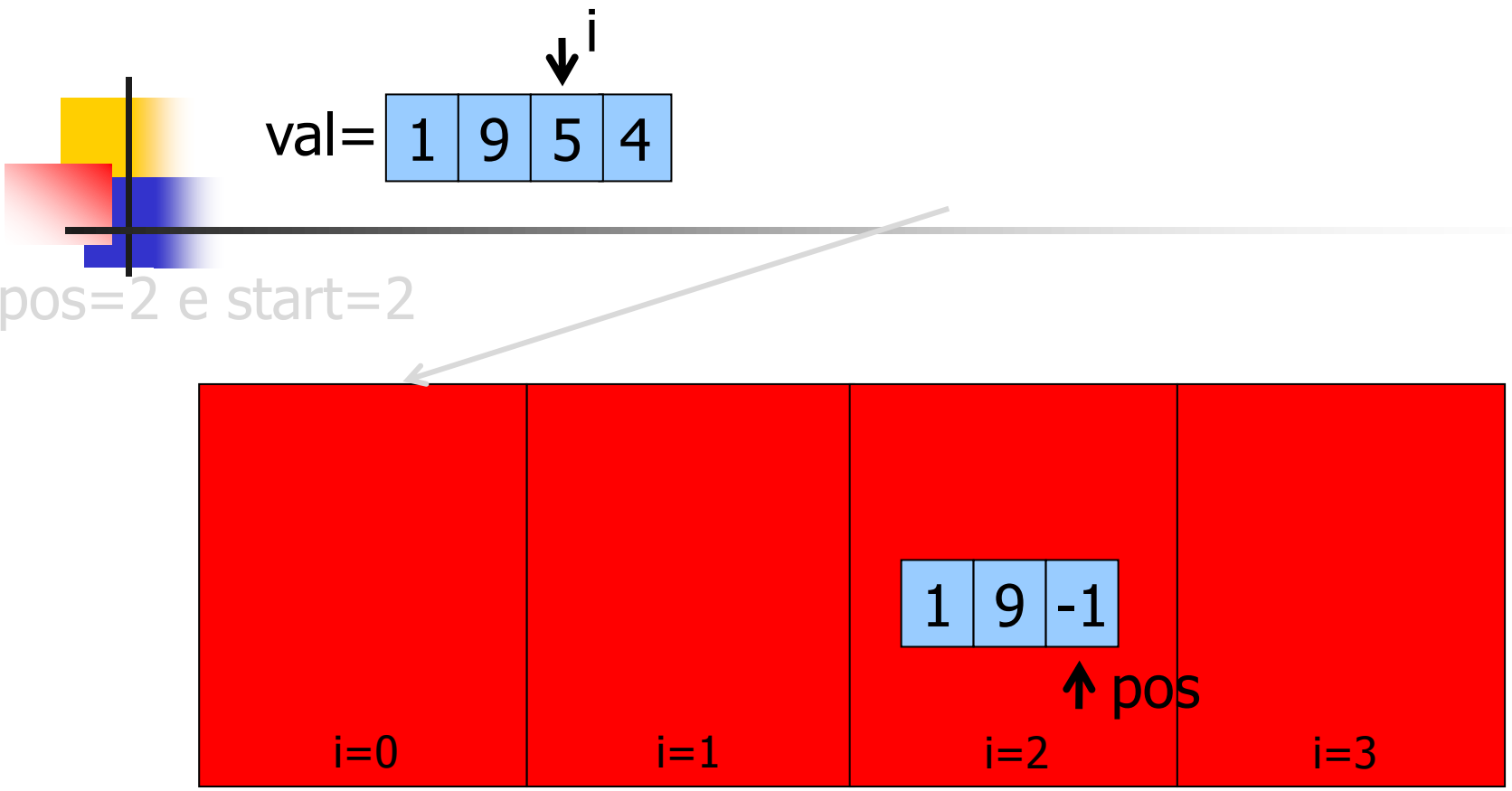


sol[pos] = val[i]

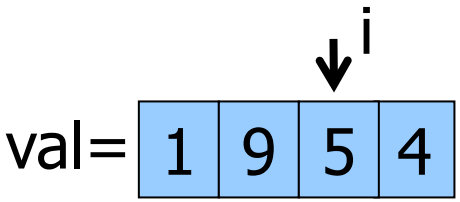


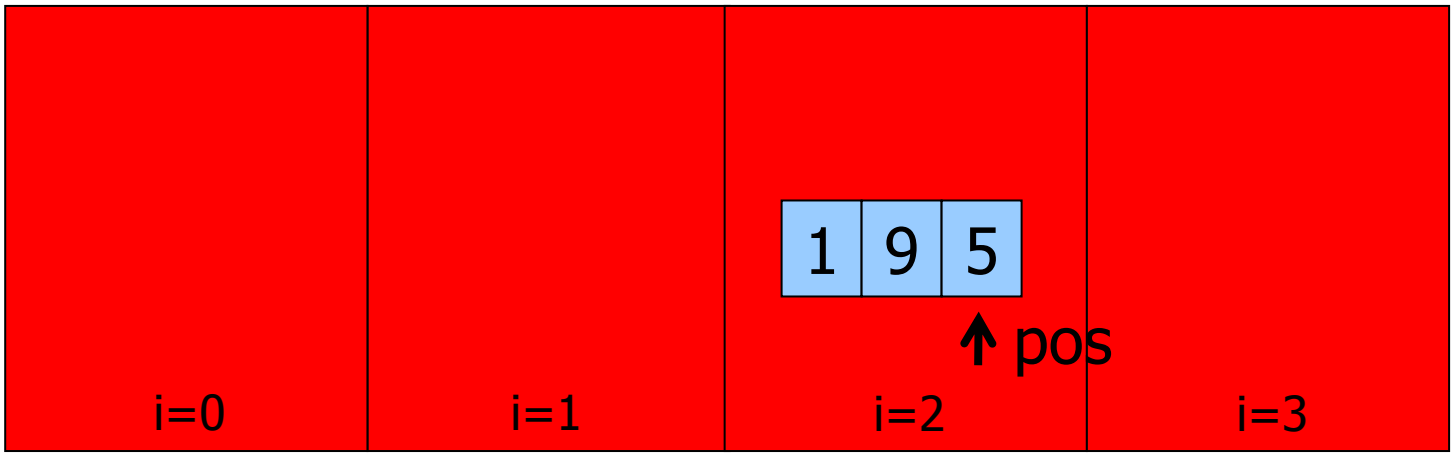


ricorsione
con pos=2 e start=2

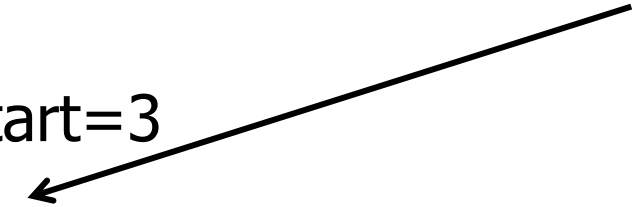


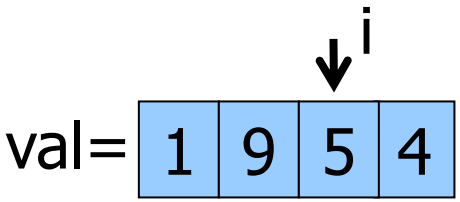
sol[pos] = val[i]

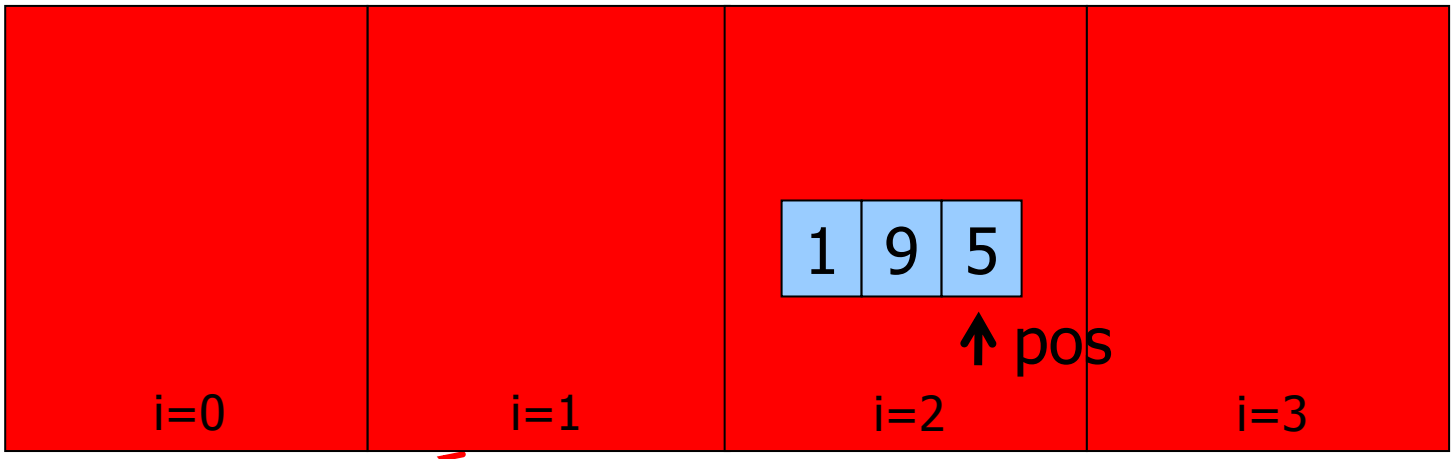




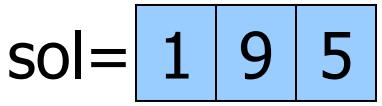
ricorsione
con pos=3 e start=3

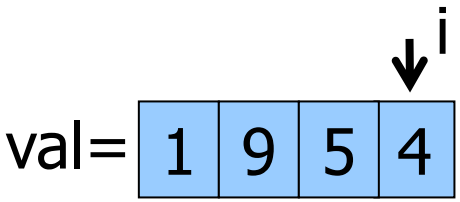


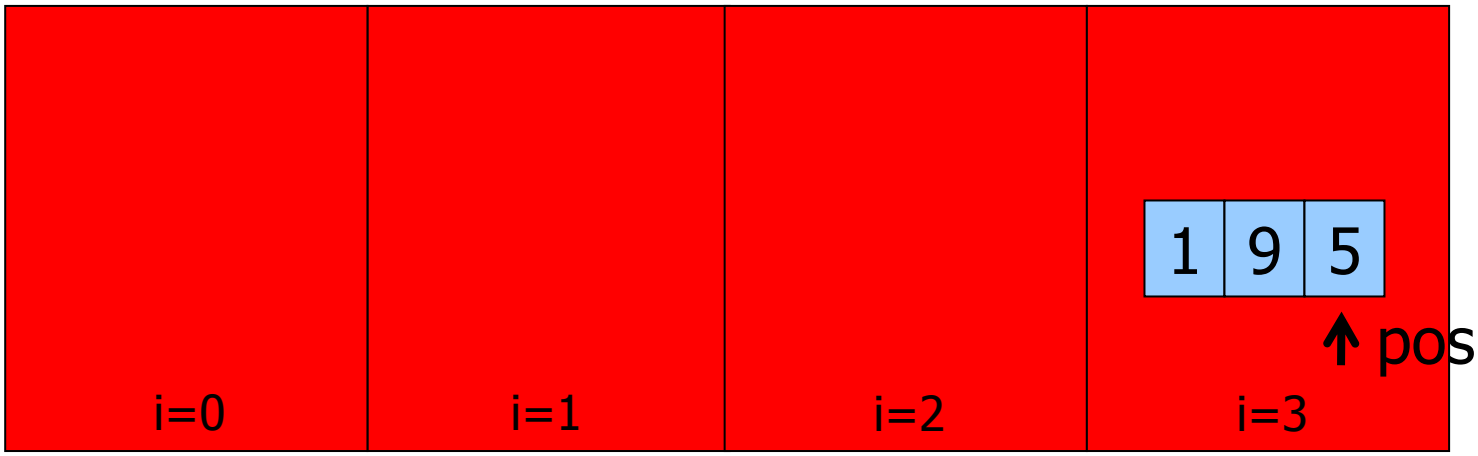




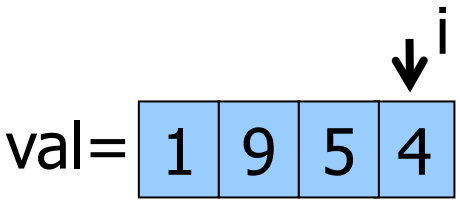
terminazione: visualizza, aggiorna count ritorna

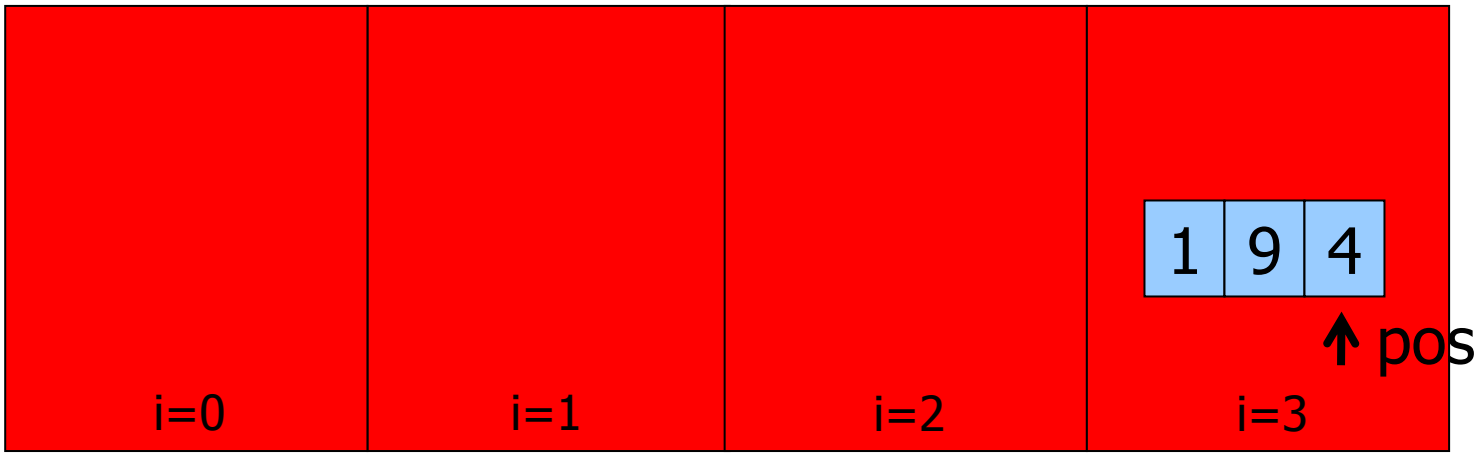




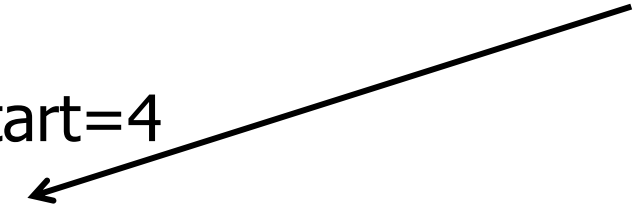


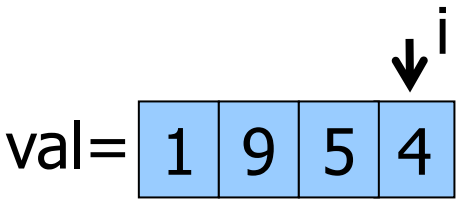
sol[pos] = val[i]

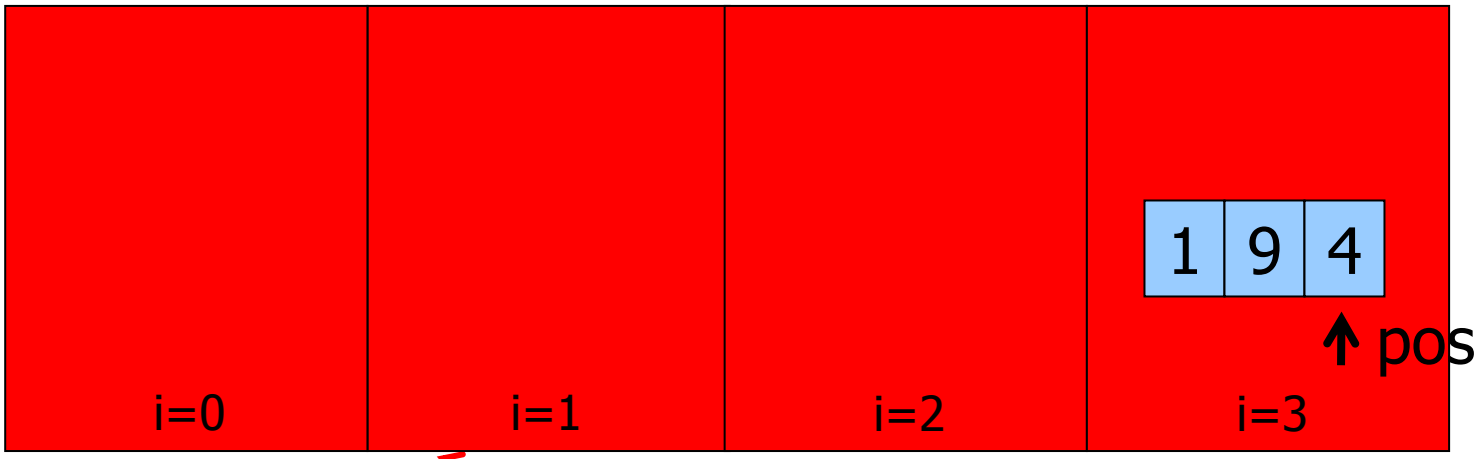




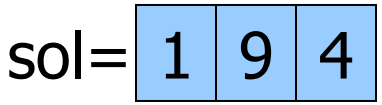
ricorsione
con pos=3 e start=4

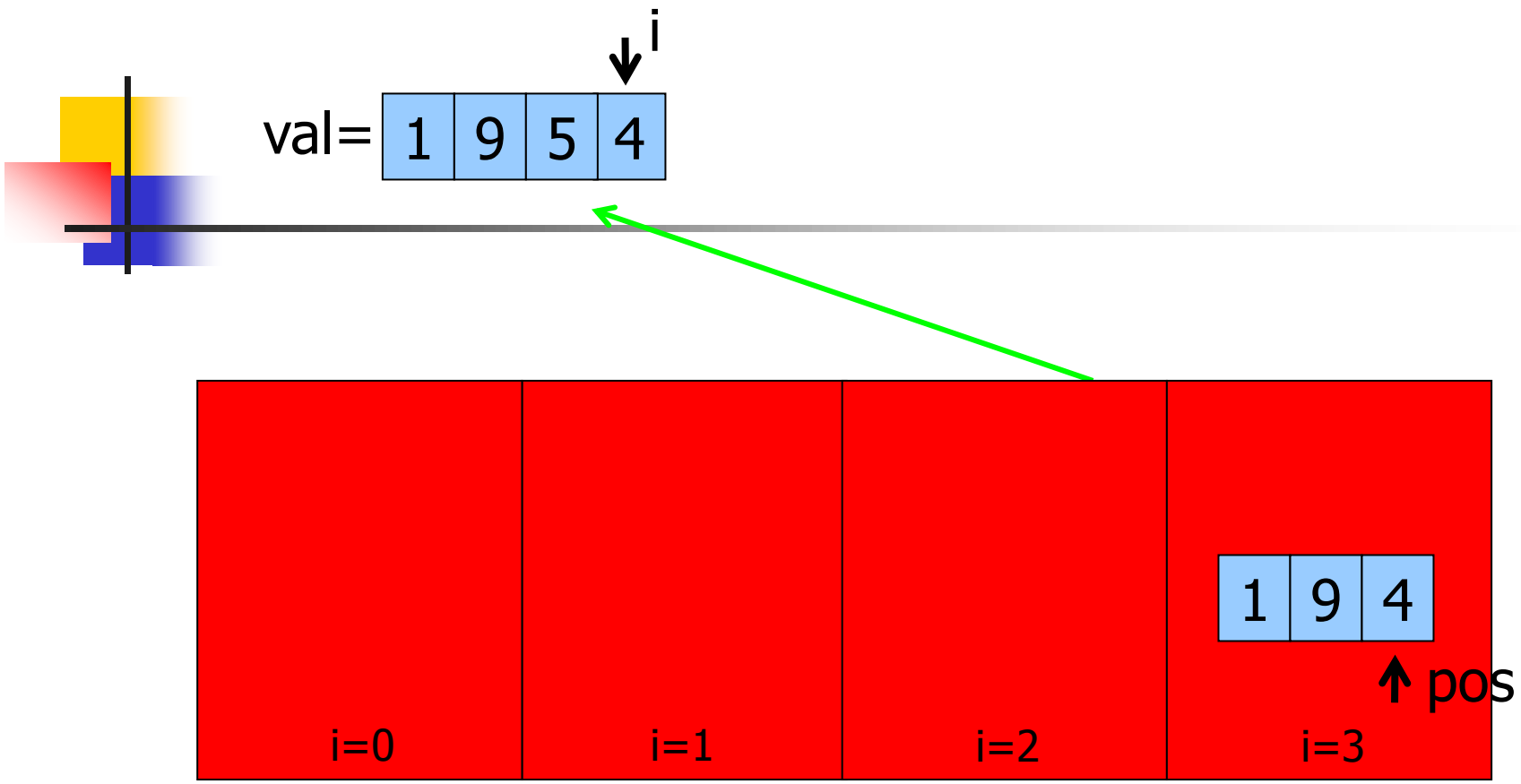




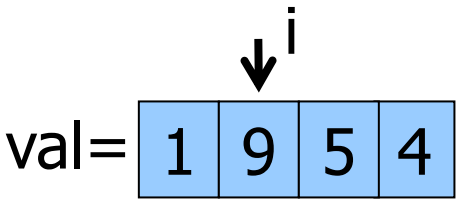


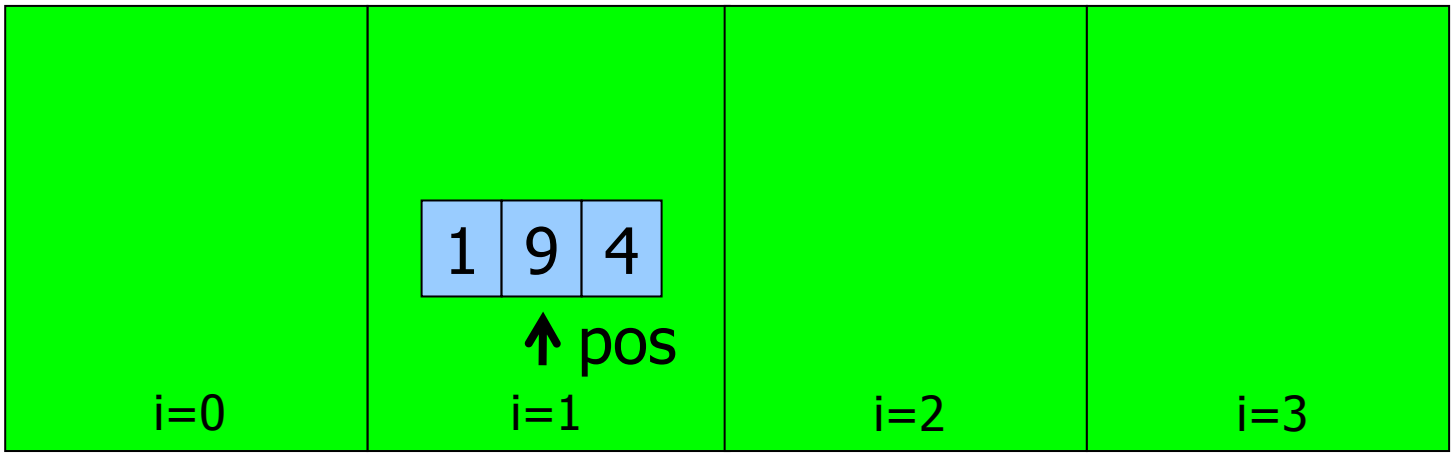
terminazione: visualizza, aggiorna count ritorna

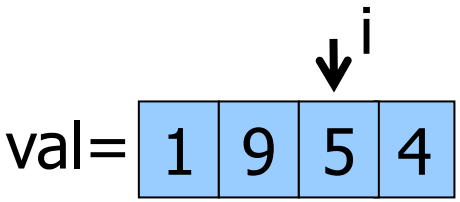


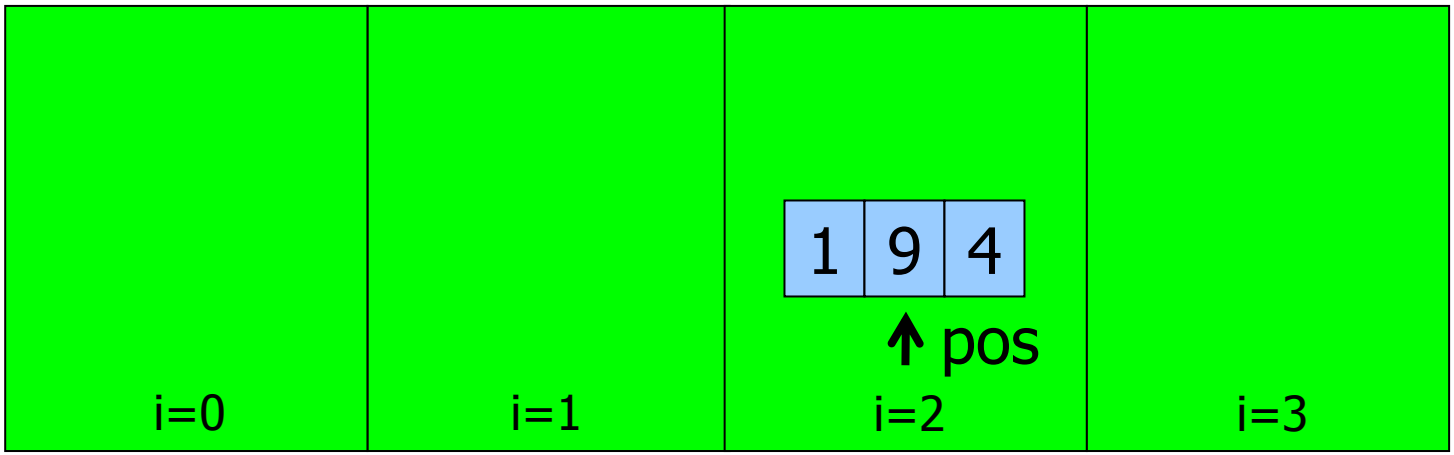


ciclo for terminato, ritorna

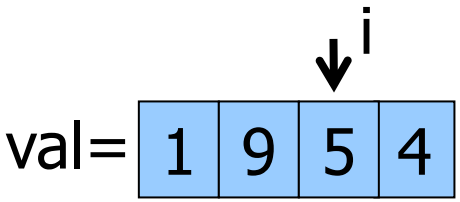

i
val=
1
9
5
4

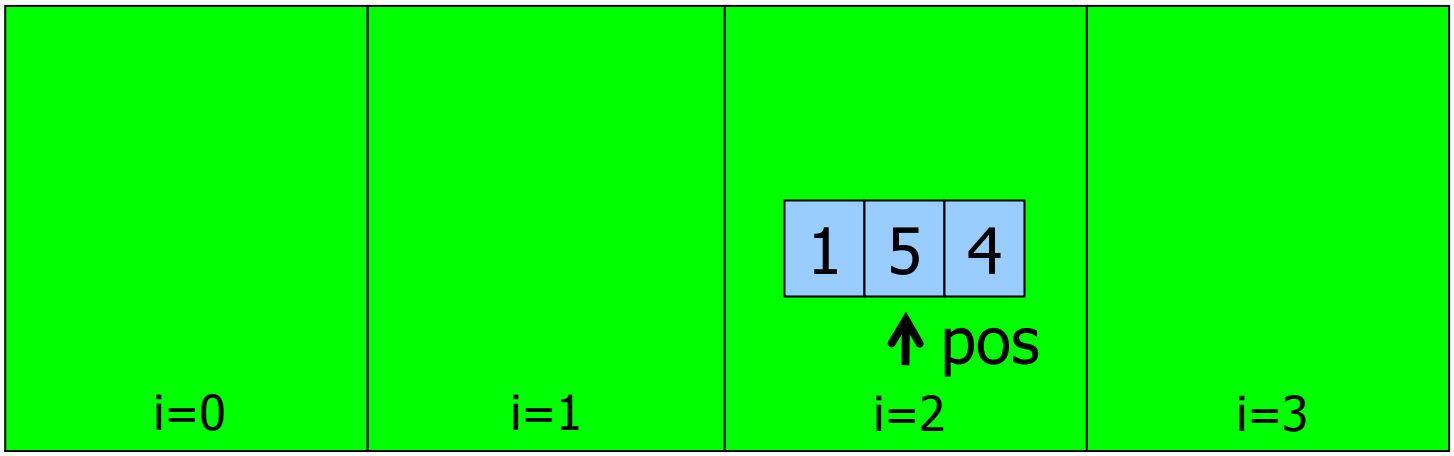

1
9
4
pos
i=0
i=1
i=2
i=3



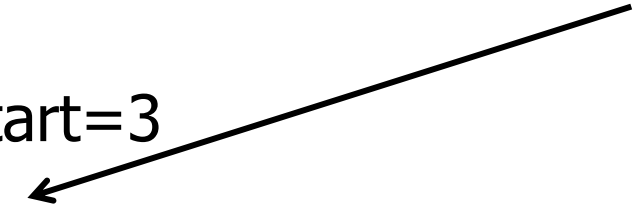


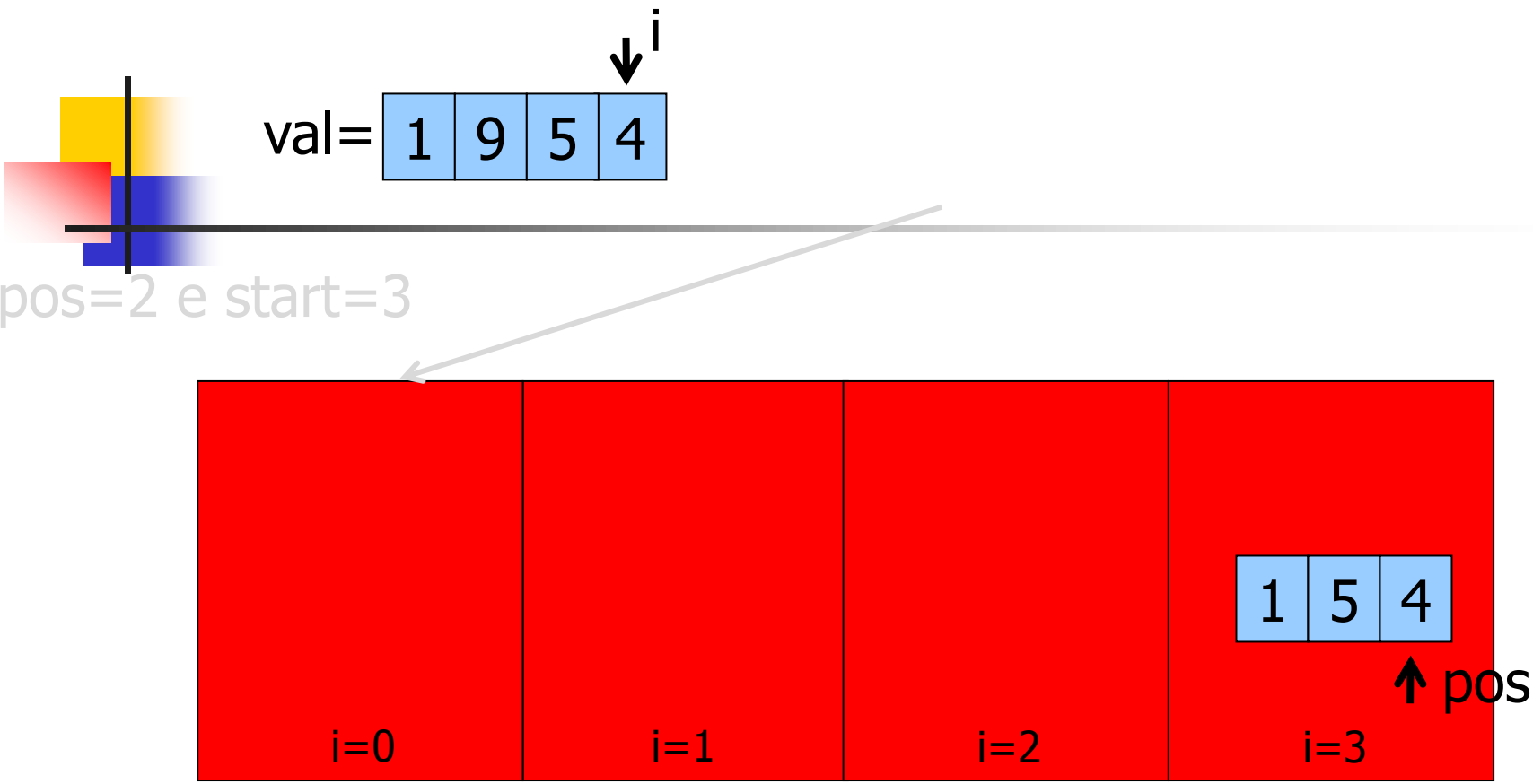
sol[pos] = val[i]



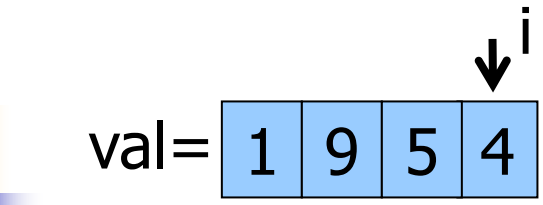


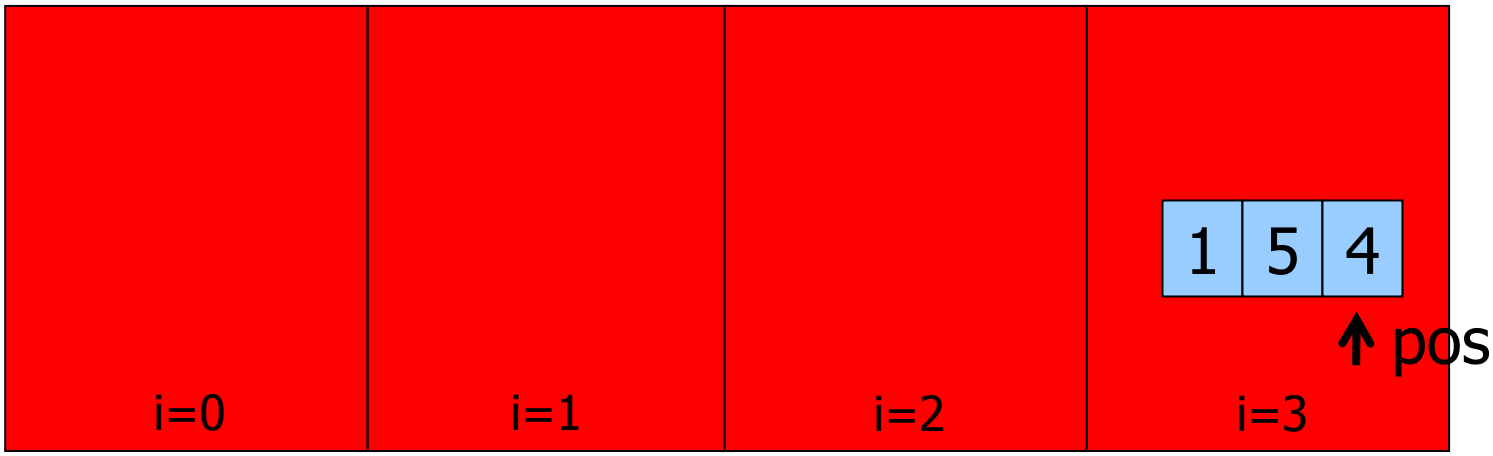
ricorsione
con pos=2 e start=3



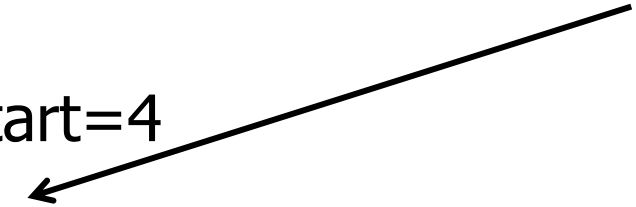


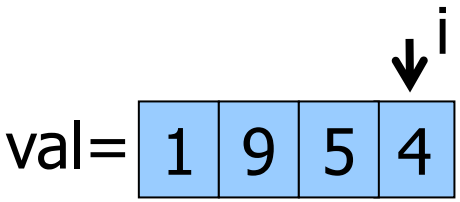
sol[pos] = val[i]

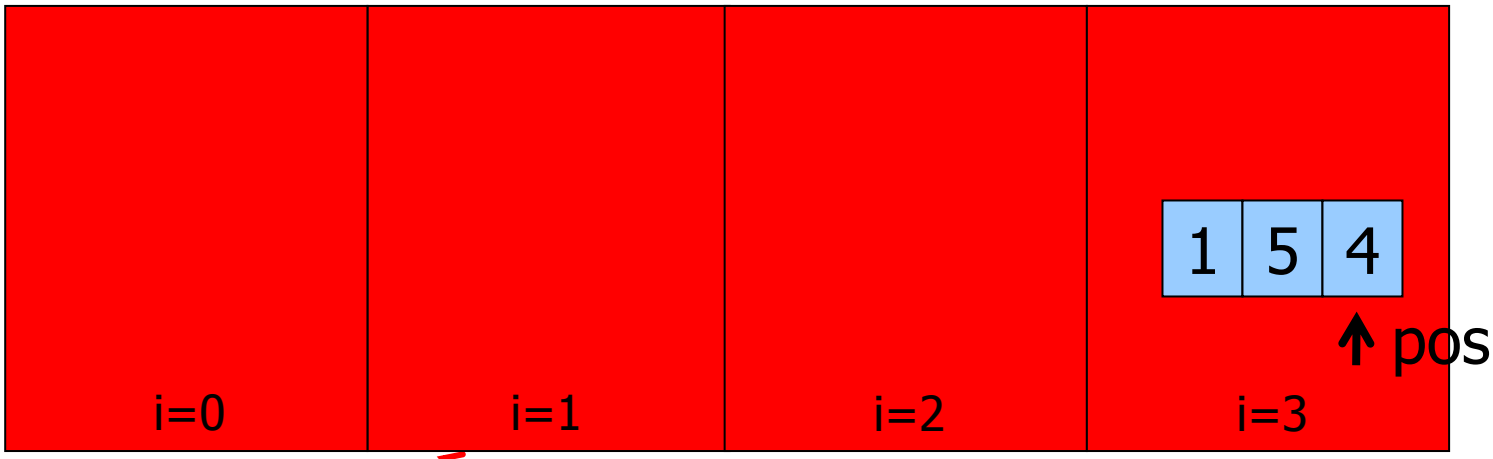




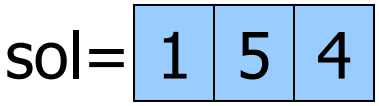
ricorsione
con pos=3 e start=4

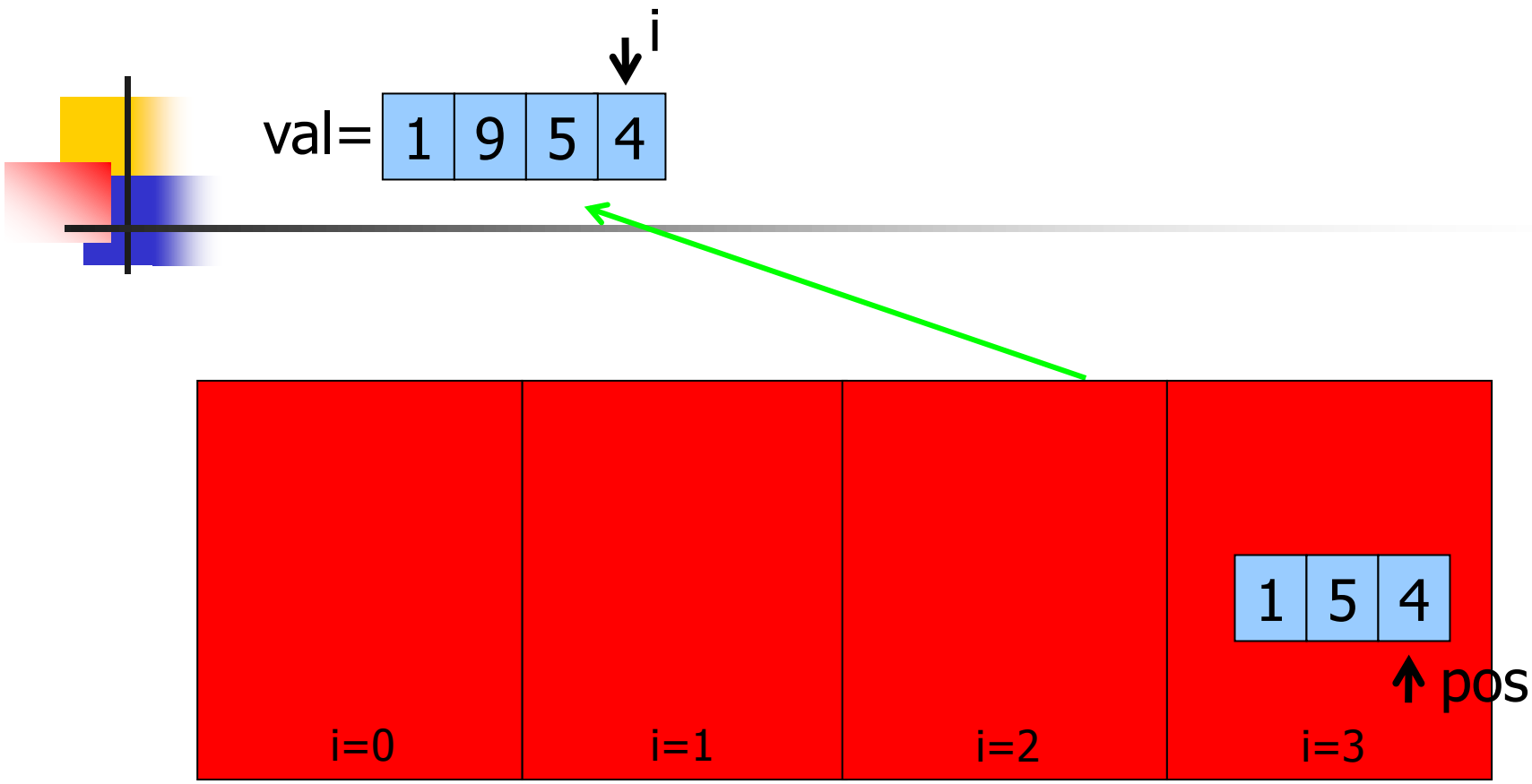




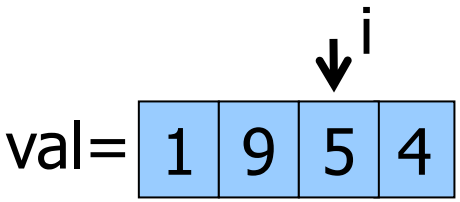


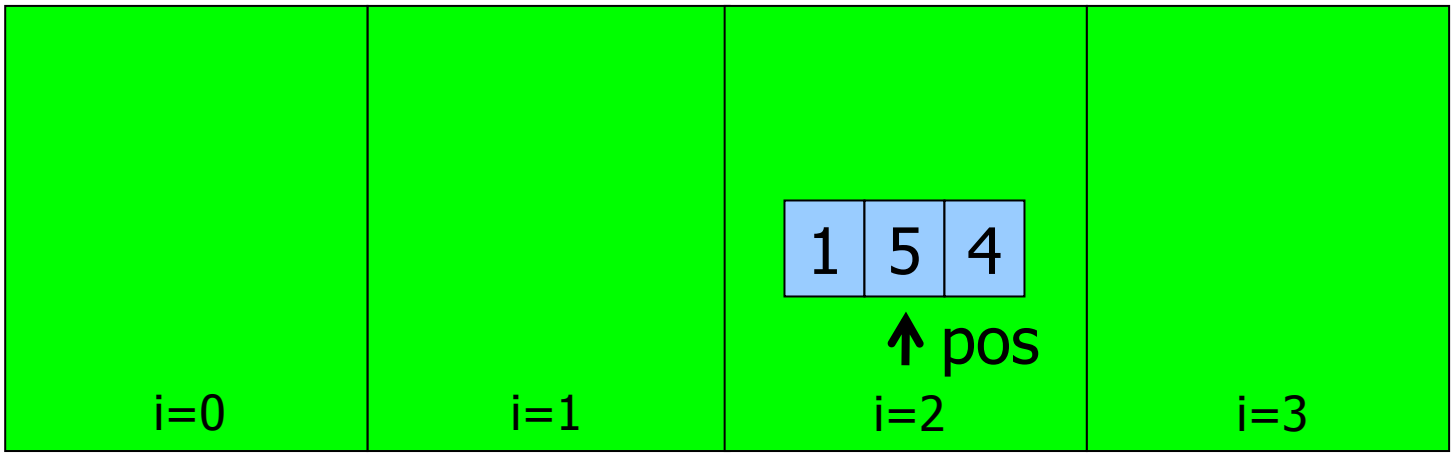
terminazione: visualizza, aggiorna count ritorna

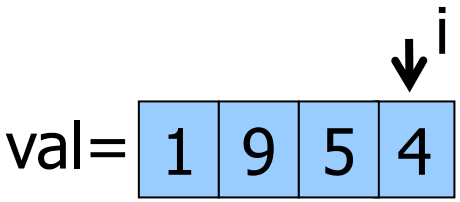


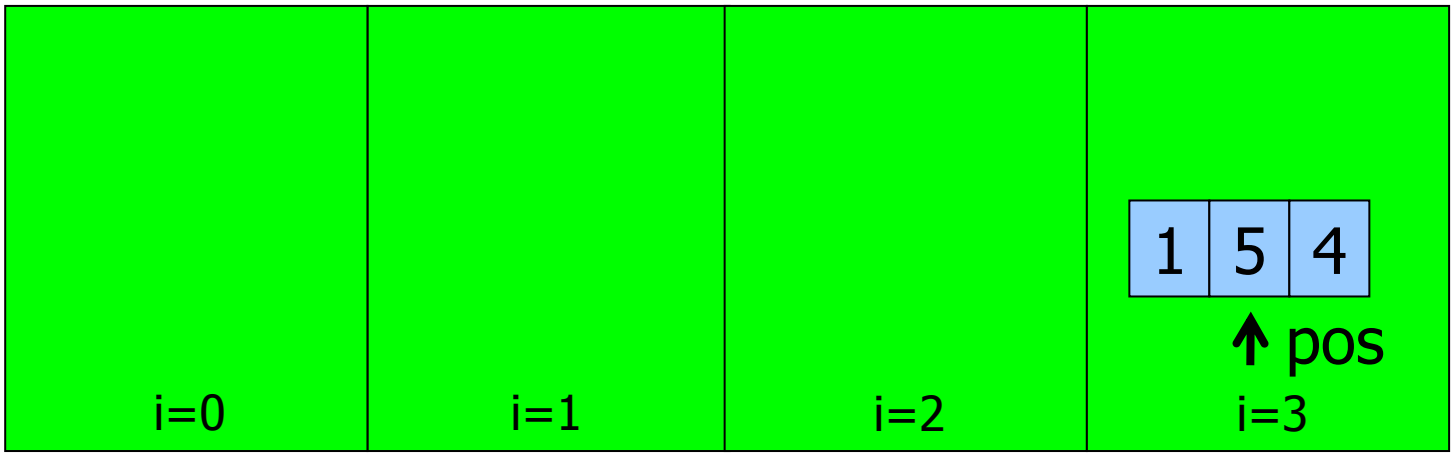


ciclo for terminato, ritorna

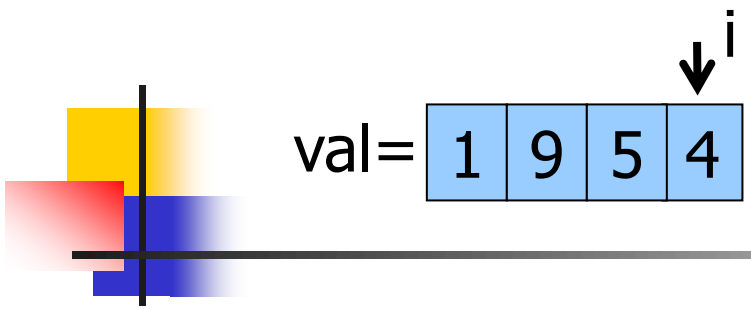

i
val=
1
9
5
4

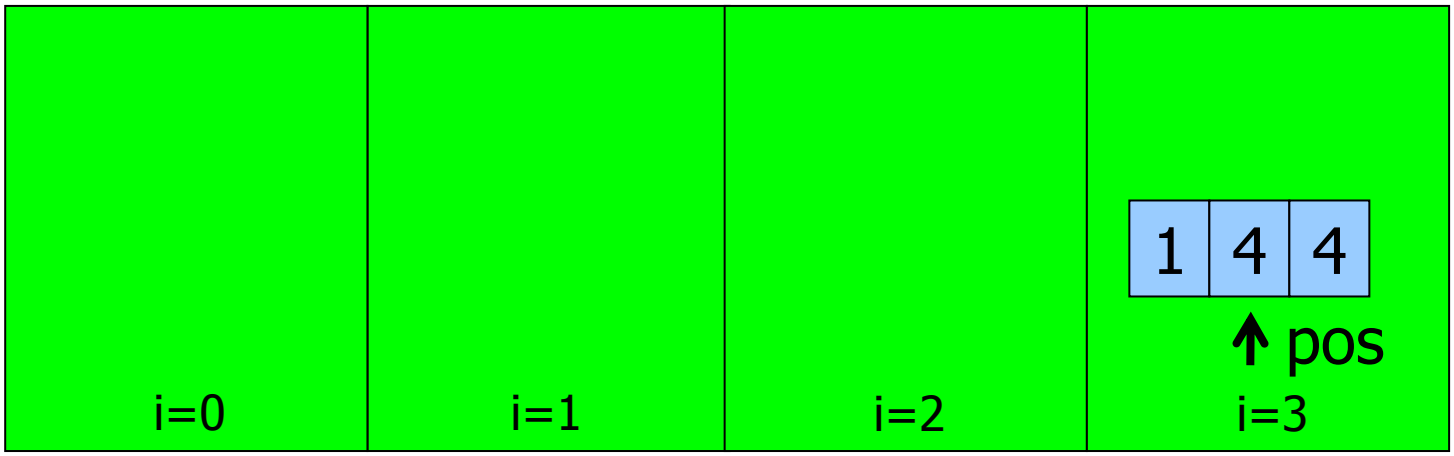

1
5
4
pos
i=0
i=1
i=2
i=3



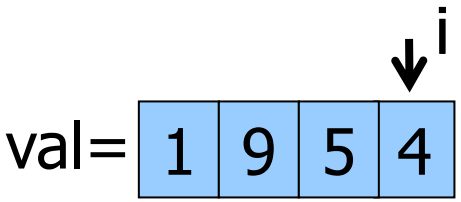


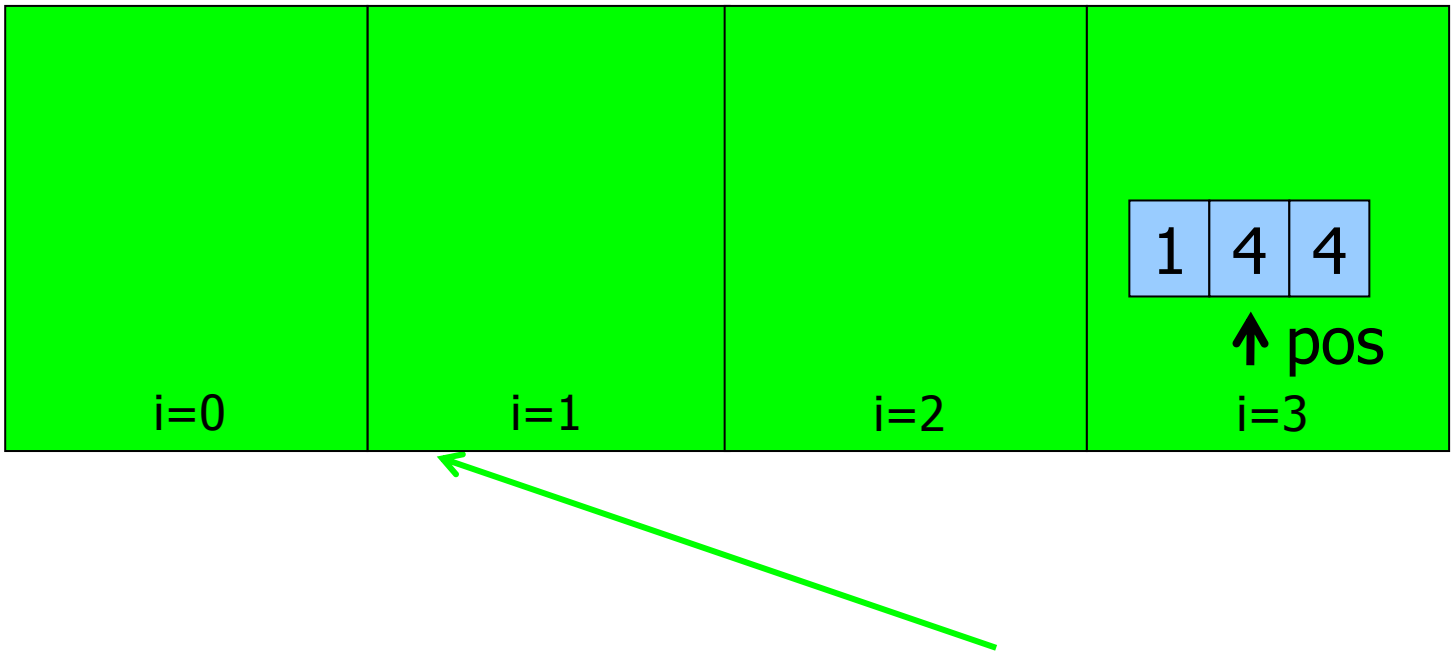
sol[pos] = val[i]



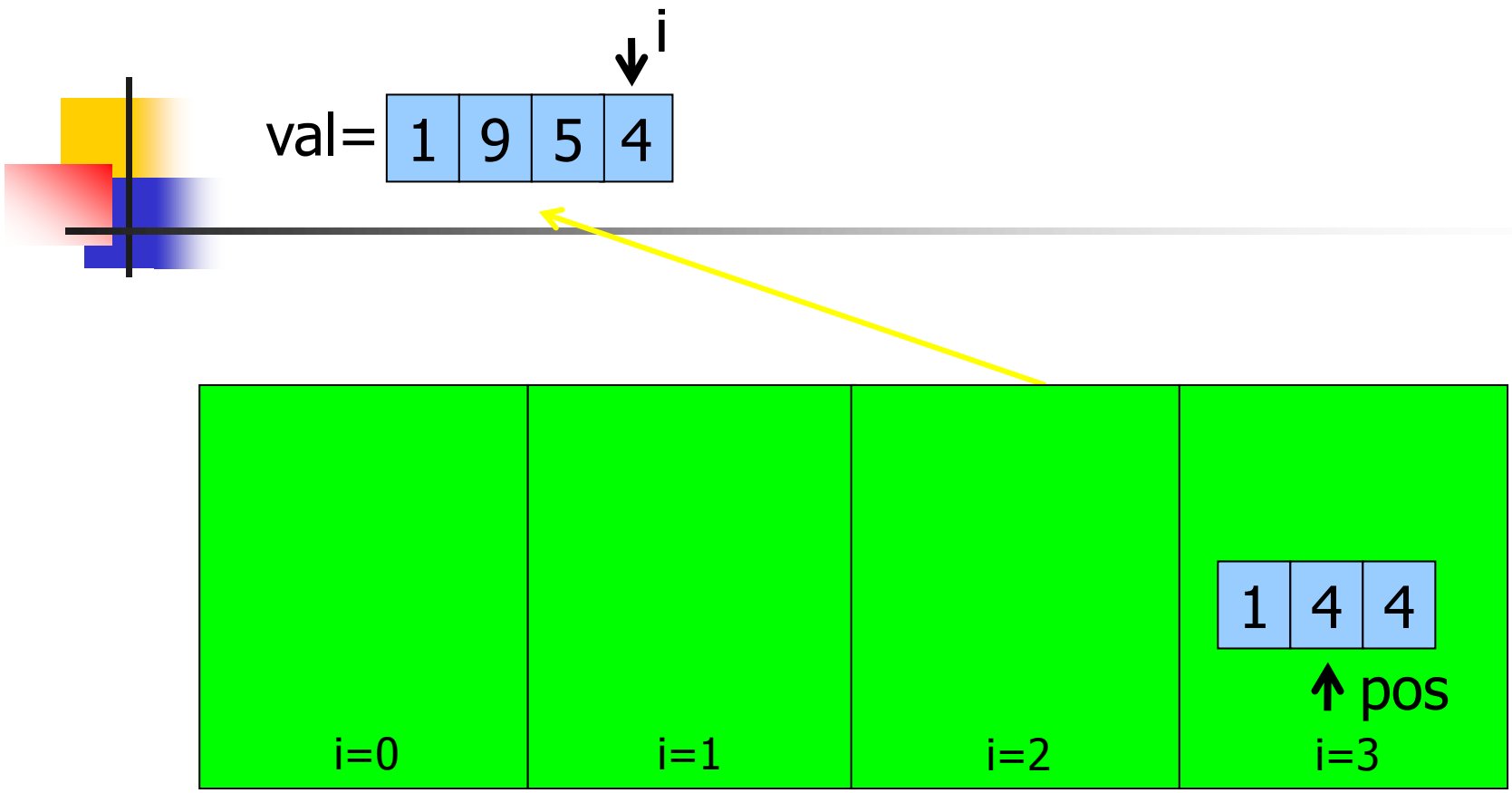


ricorsione
con pos=2 e start=4

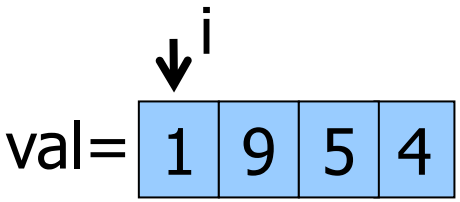


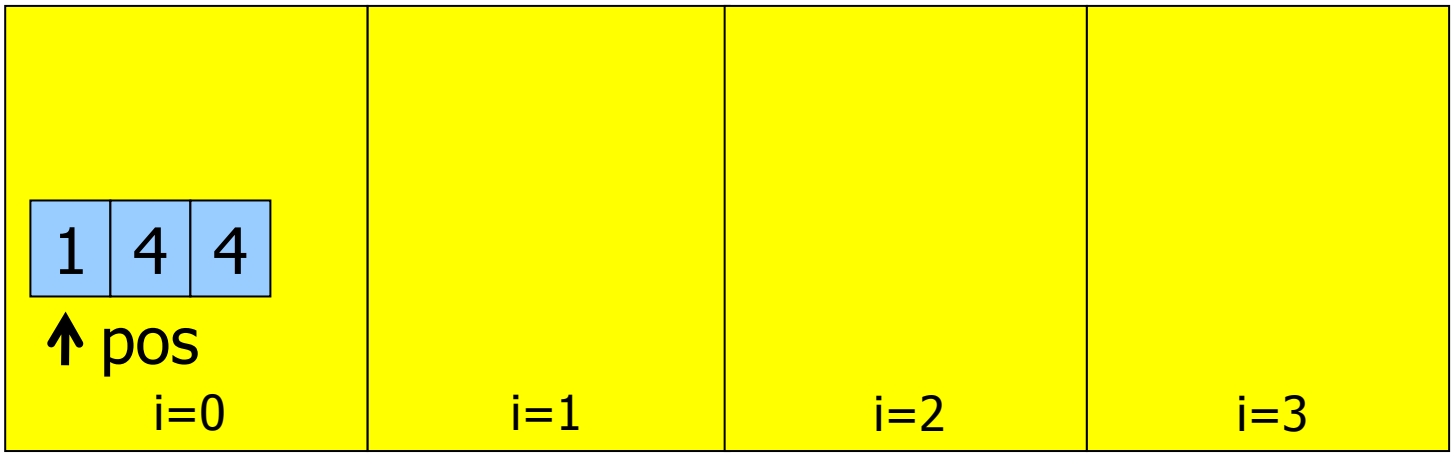


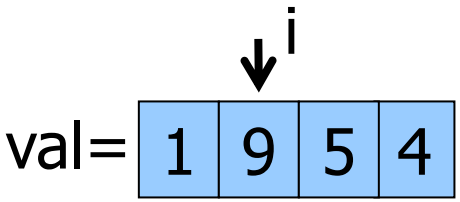
il ciclo for non inizia nemmeno
ritorna

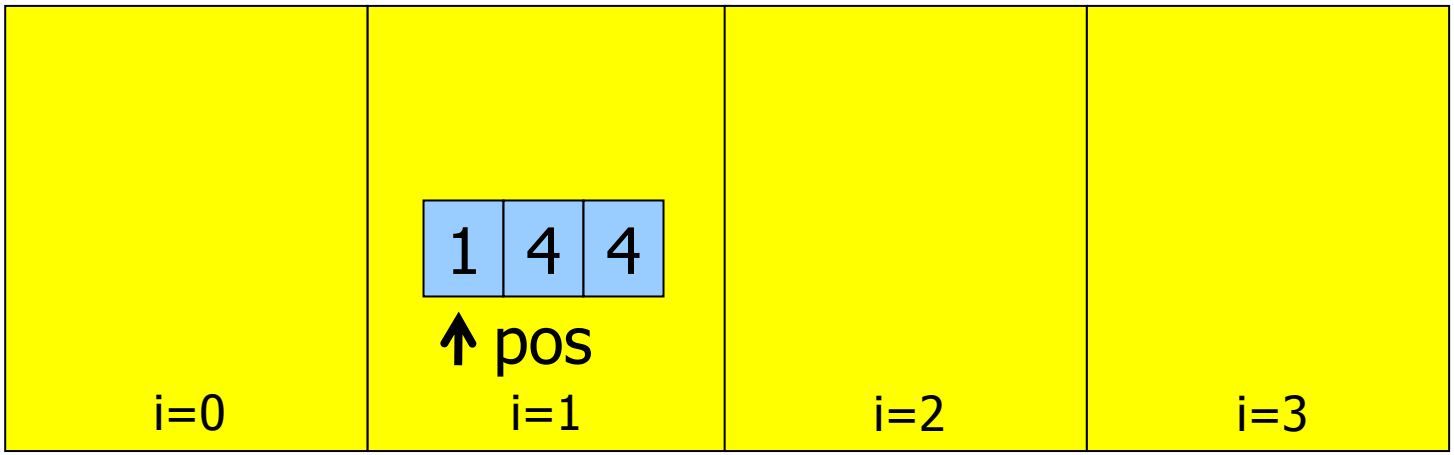


ciclo terminato, ritorna

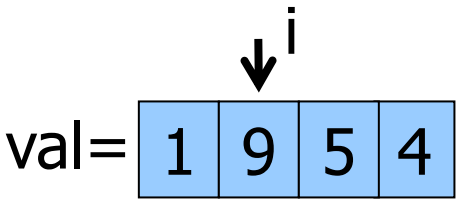

i
val=
1
9
5
4

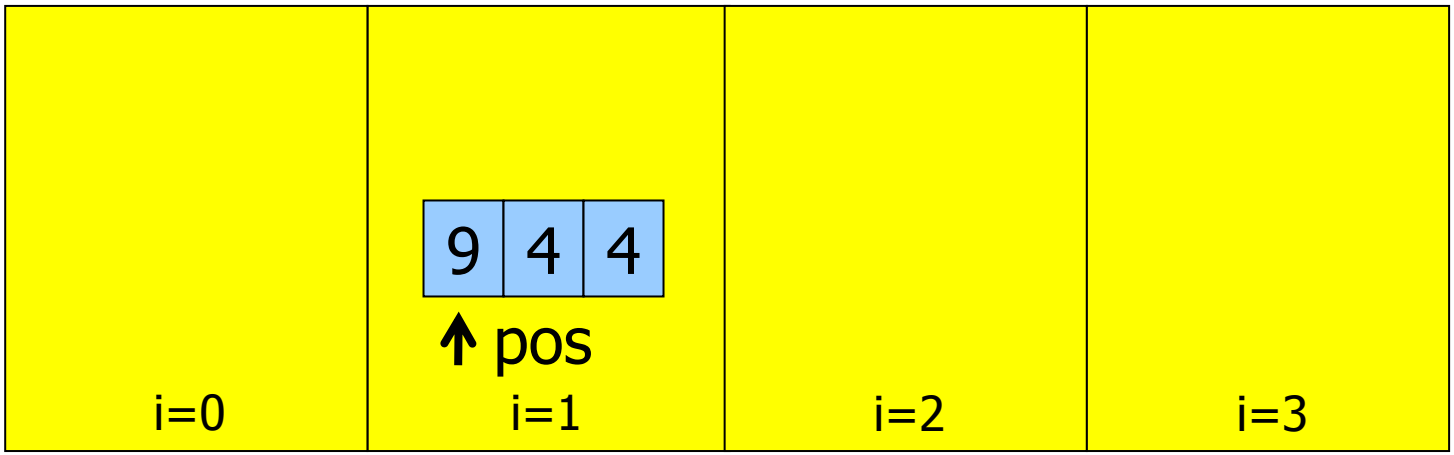

1
4
4
pos
i=0
i=1
i=2
i=3



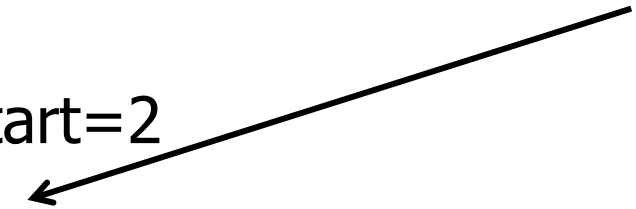


sol[pos] = val[i]

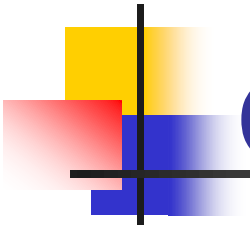




ricorsione
con pos=1 e start=2



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 ogni volta che il ciclo for delle scelte termina.

`count` registra il numero di soluzioni.

01calcolocombinatorio

```
val = malloc(n * sizeof(int));
sol = malloc(k * sizeof(int));
```

```
int comb_r(int pos,int *val,int *sol,int n,int k,
           int start, int count) {
  int i, j;
  if (pos >= k) {
    for (i=0; i<k; i++)
      printf("%d ", sol[i]);
    printf("\n");
    return count+1;
  }
  for (i=start; i<n; i++) {
    sol[pos] = val[i];
    count = comb_r(pos+1, val, sol, n, k, start, count);
    start++;
  }
  return co   ;
}
```

terminazione

iterazione sulle scelte

scelta: sol[pos] riempito con i valori possibili di `val` da `start` in poi

ricorsione su prossima posizione

aggiornamento di start

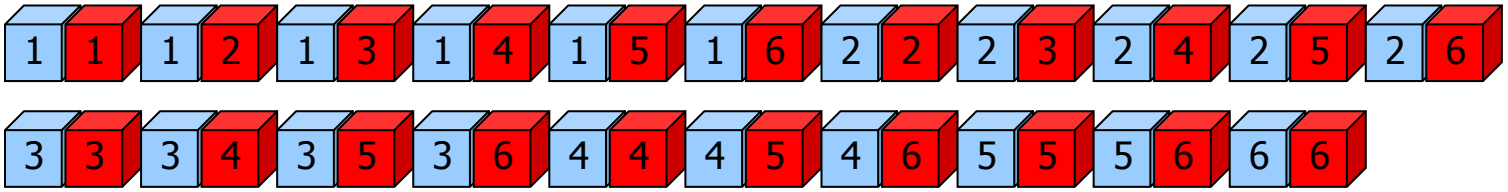
Esempio

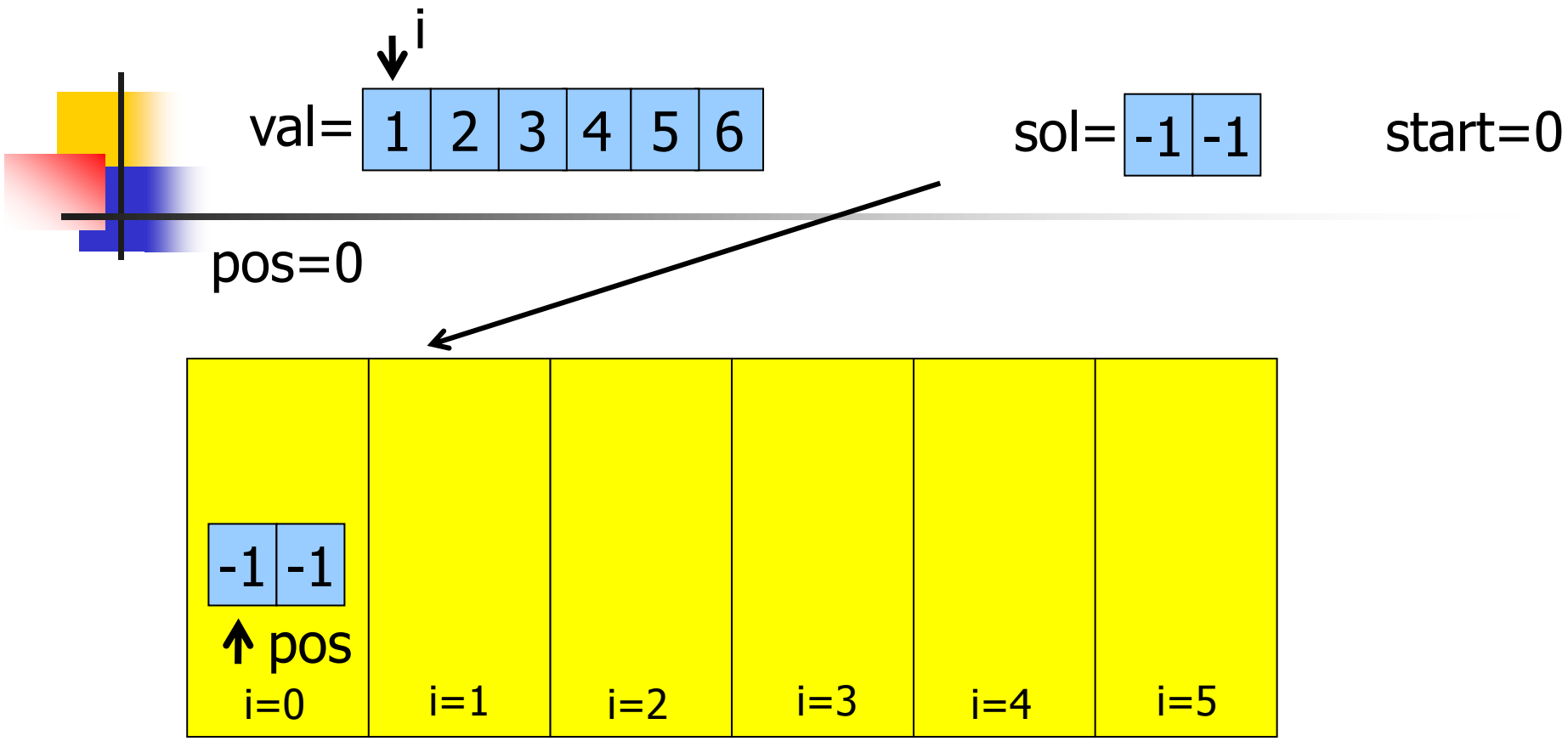
Lanciando contemporaneamente 2 dadi , quante sono le composizioni con cui si possono presentare le facce?

Modello: combinazioni con ripetizione

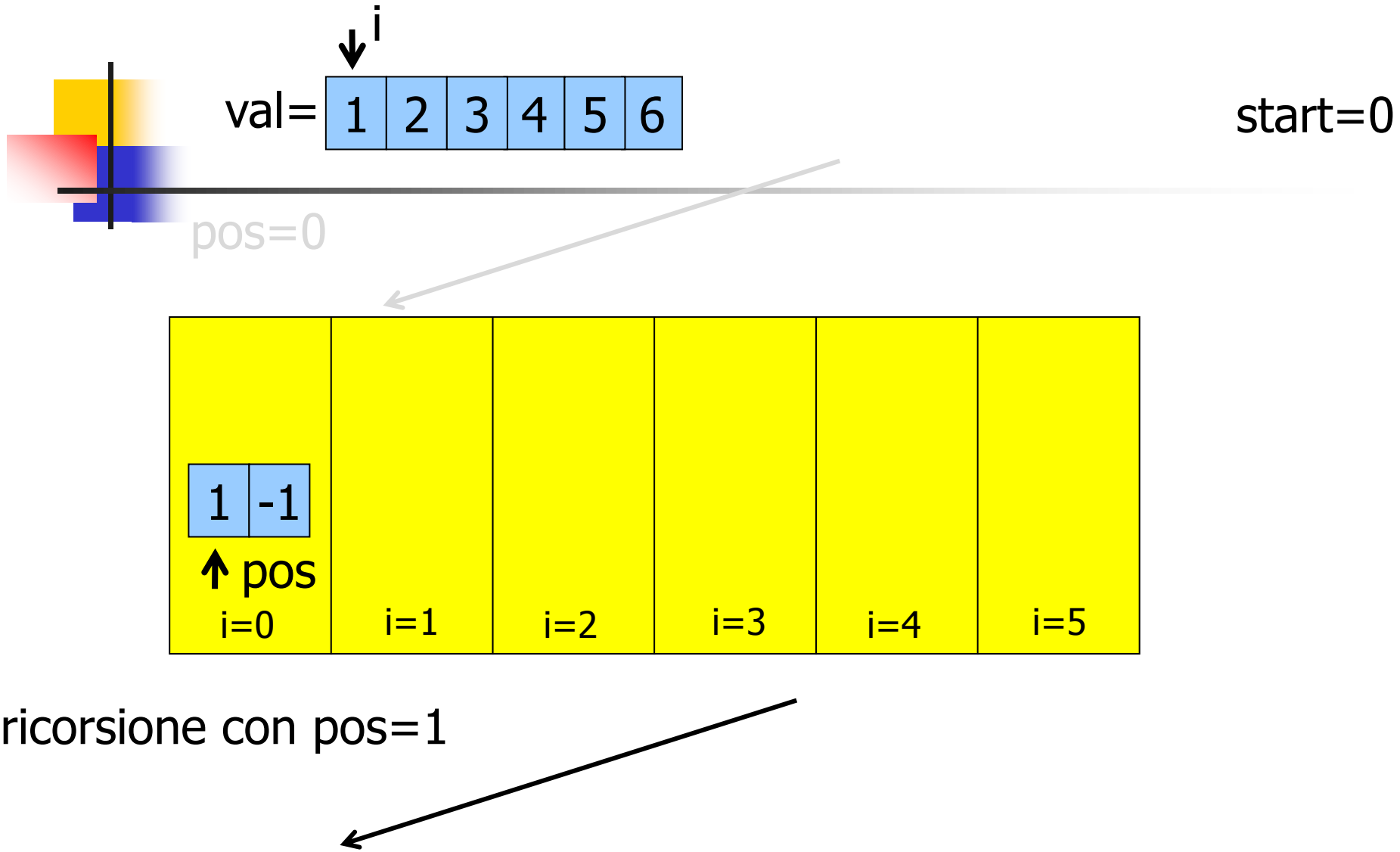
$C'_{6,3} = (6 + 2 - 1)!/2!(6-1)! = 21$

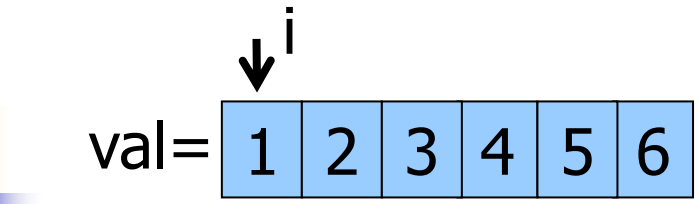
Soluzione





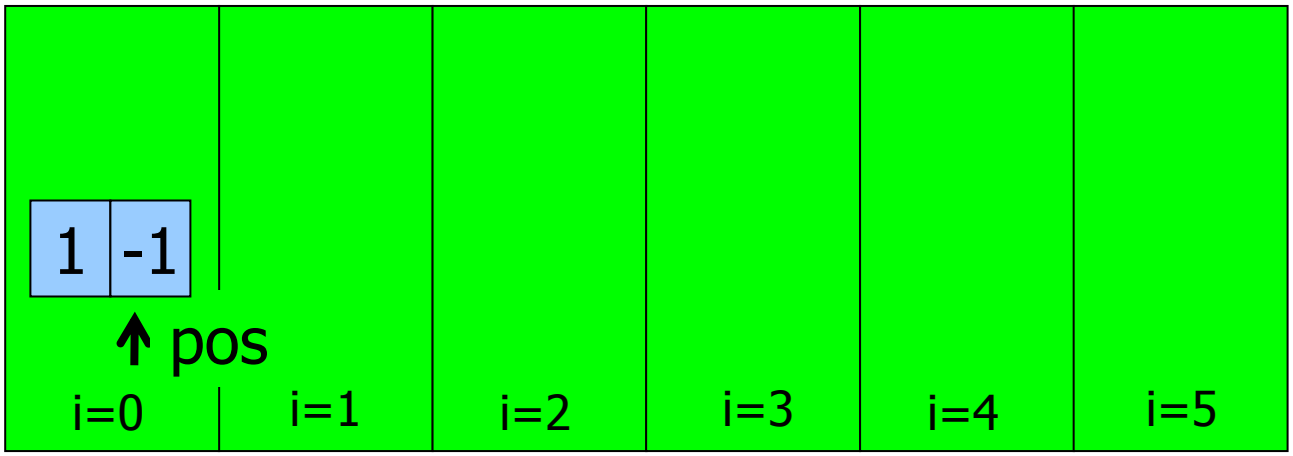
sol[pos] = val[i]


i
val=
1
2
3
4
5
6
start=0
pos=0
1
-1
pos
i=0
i=1
i=2
i=3
i=4
i=5
ricorsione con pos=1

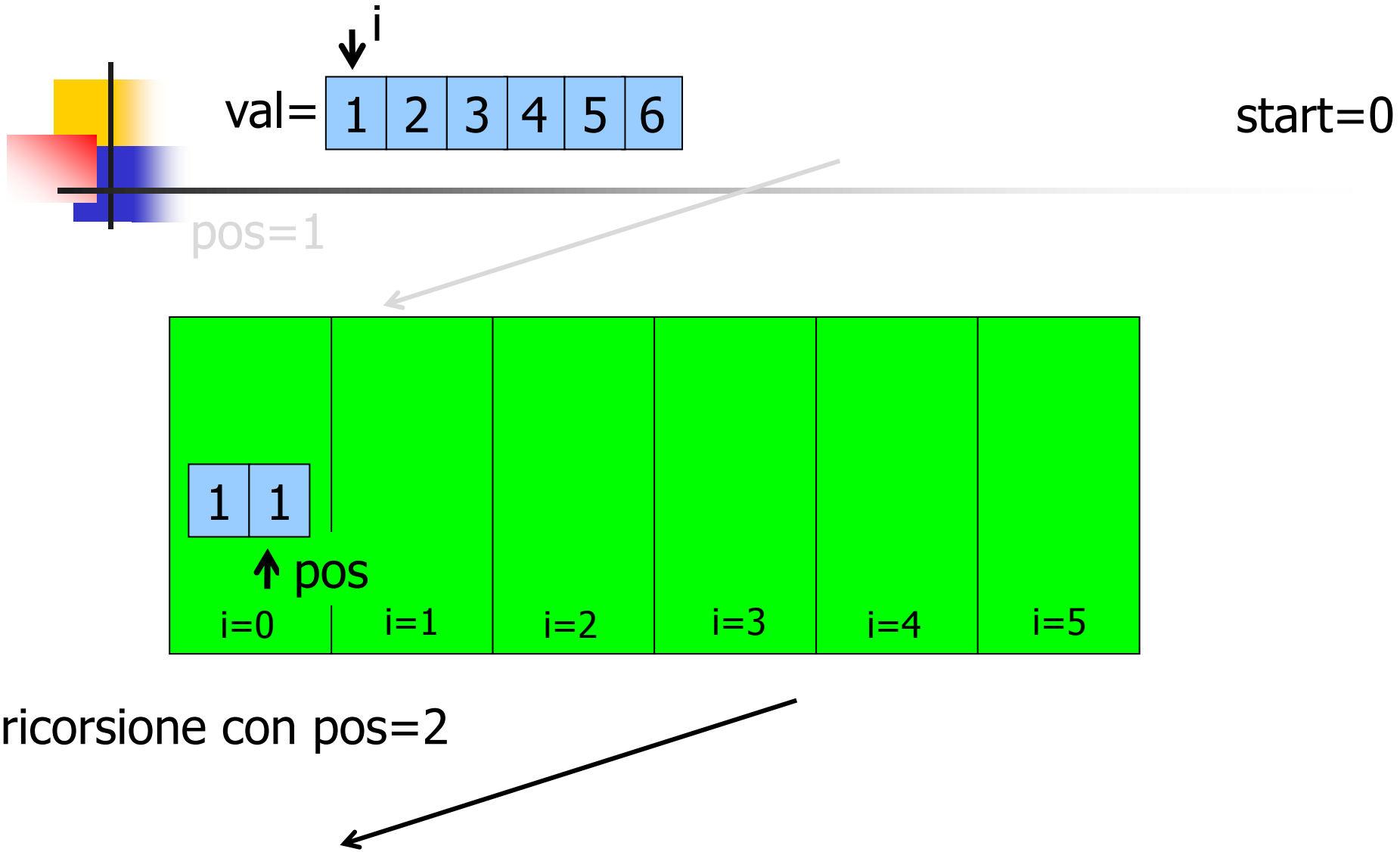


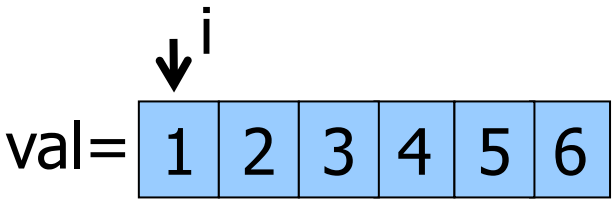
start=0

pos=1

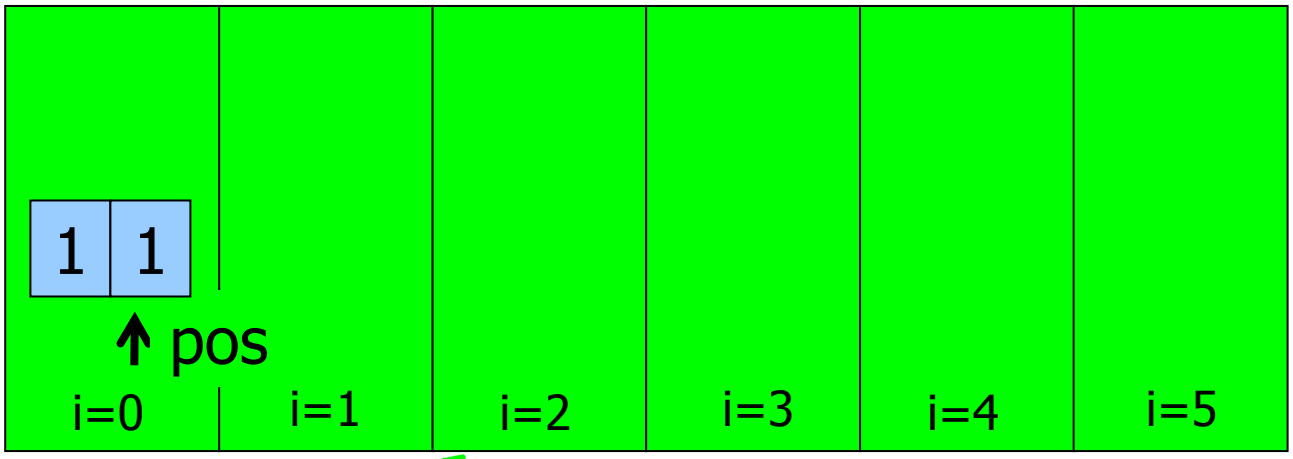


sol[pos] = val[i]


i
val=
1
2
3
4
5
6
start=0
pos=1
1
1
pos
i=0
i=1
i=2
i=3
i=4
i=5
ricorsione con pos=2

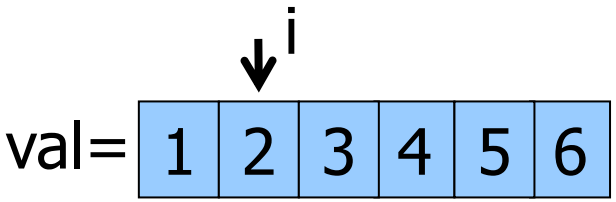


start=1

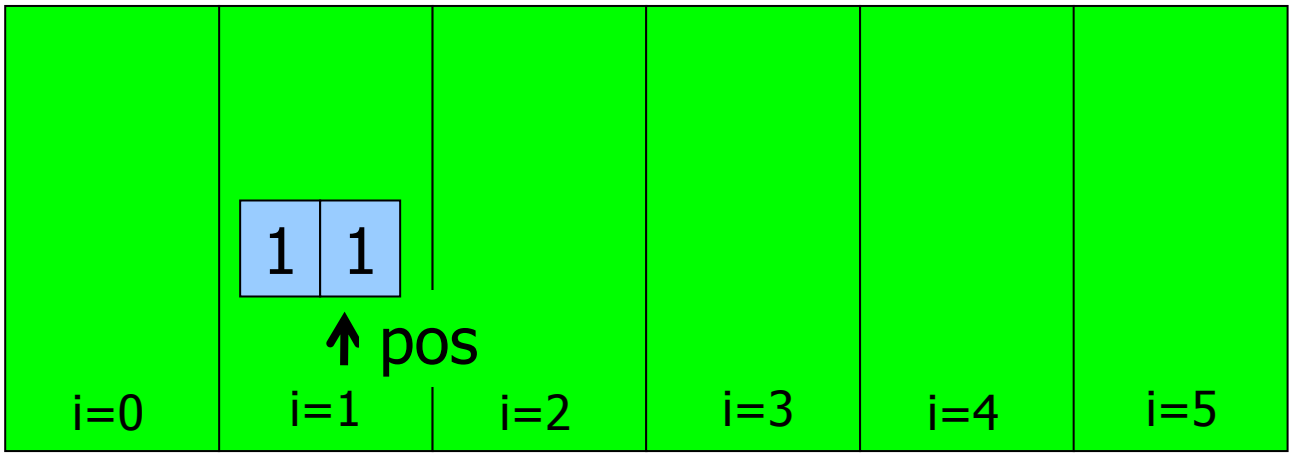


terminazione: visualizza, aggiorna count
ritorna e aggiorna start

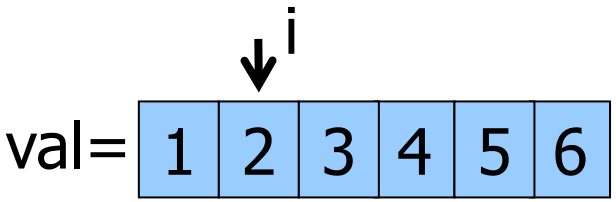
sol= 1 1



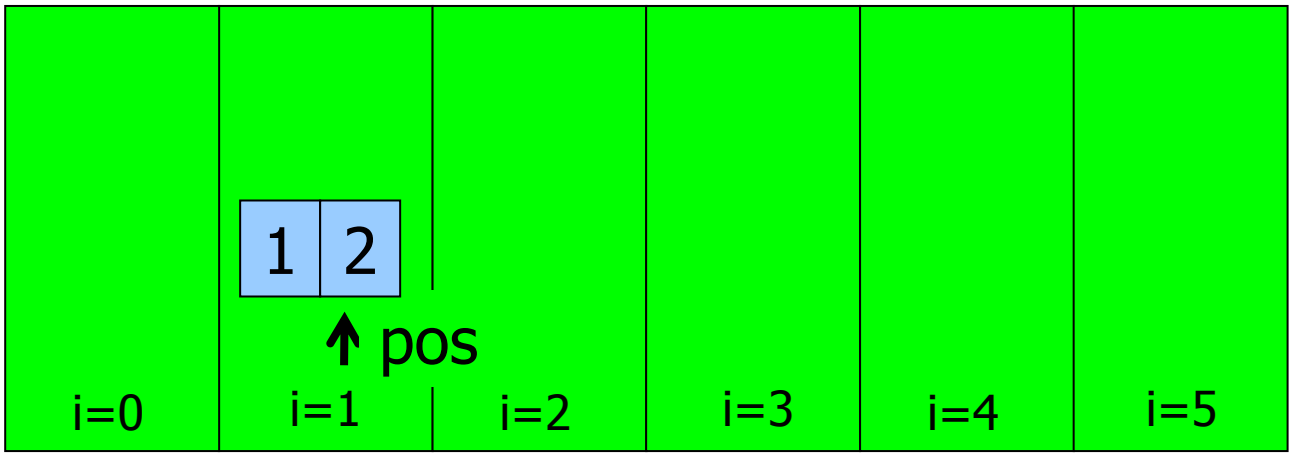
start=1



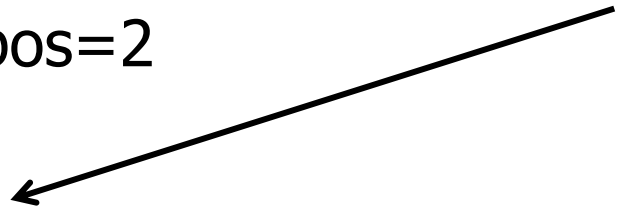
sol[pos] = val[i]

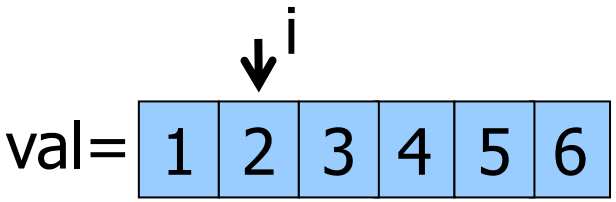


start=1



ricorsione con pos=2

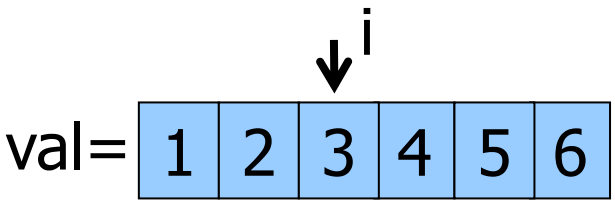




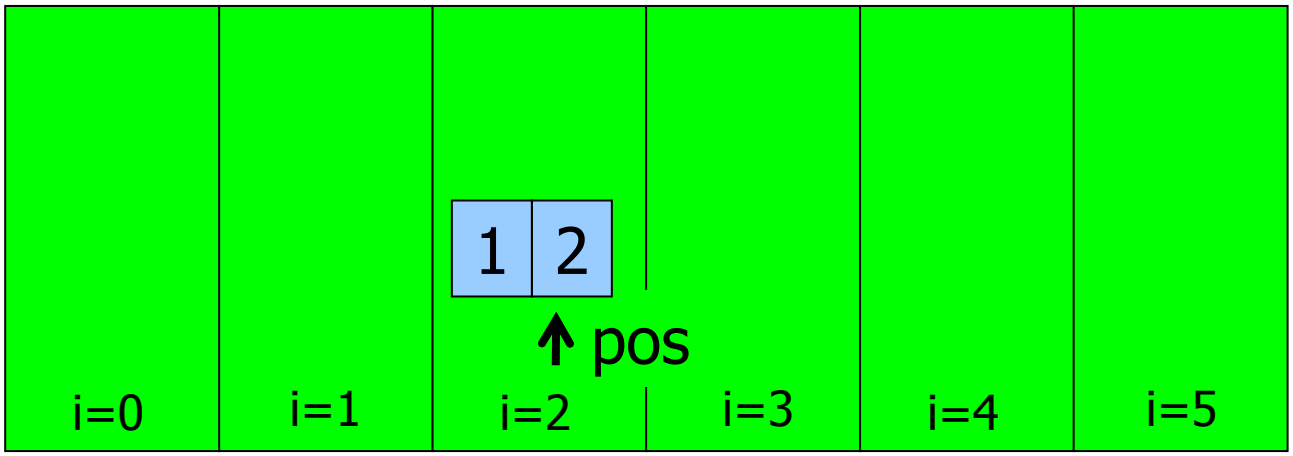
start=2

terminazione: visualizza, aggiorna count
ritorna e aggiorna start

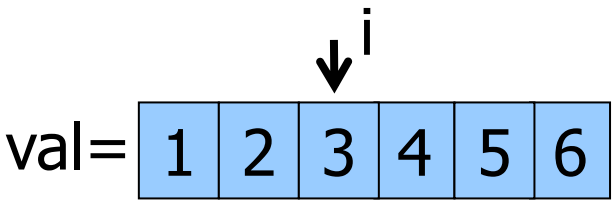
sol= 1 2



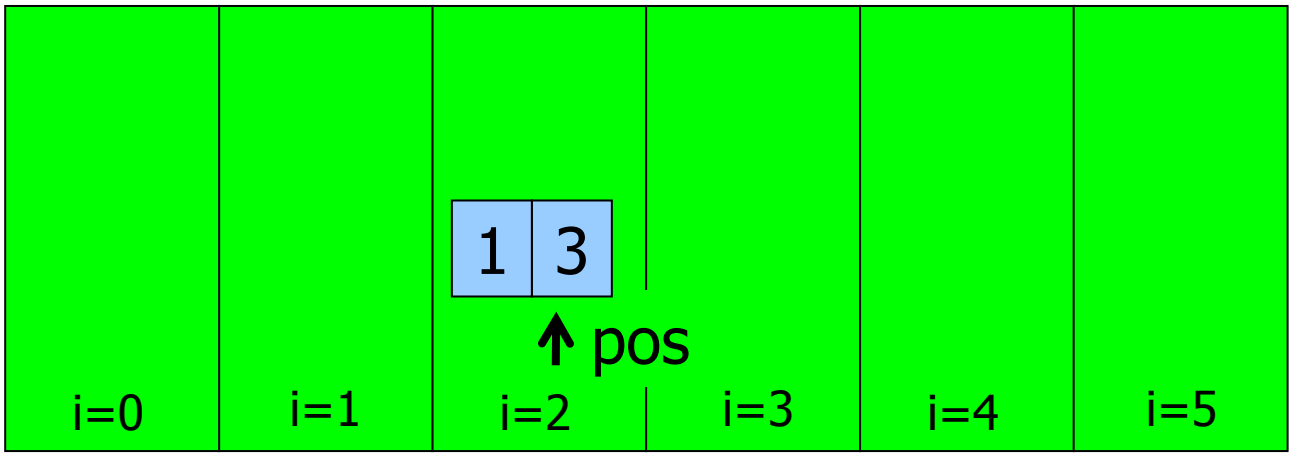
start=2



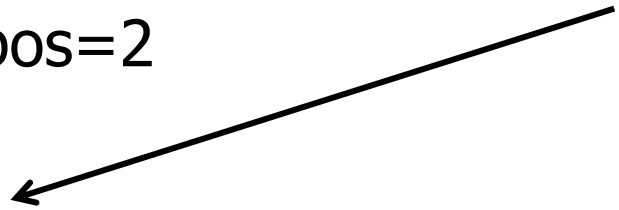
sol[pos] = val[i]

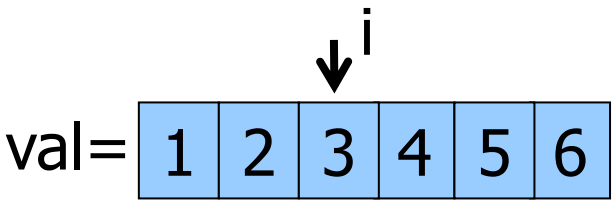


start=2

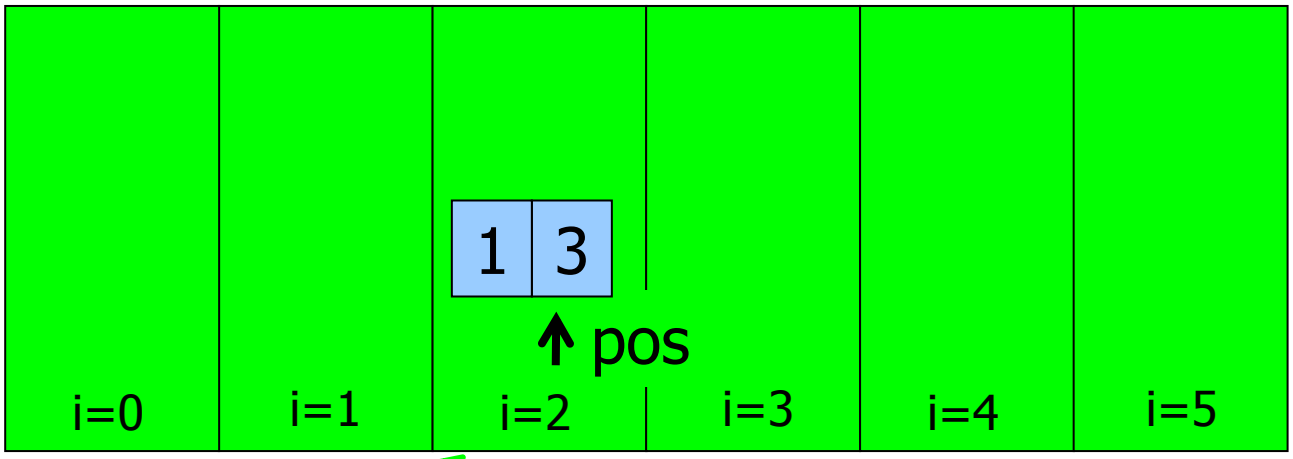


ricorsione con pos=2



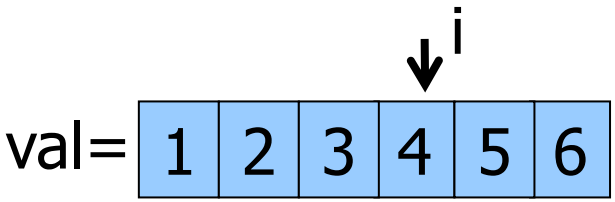


start=3

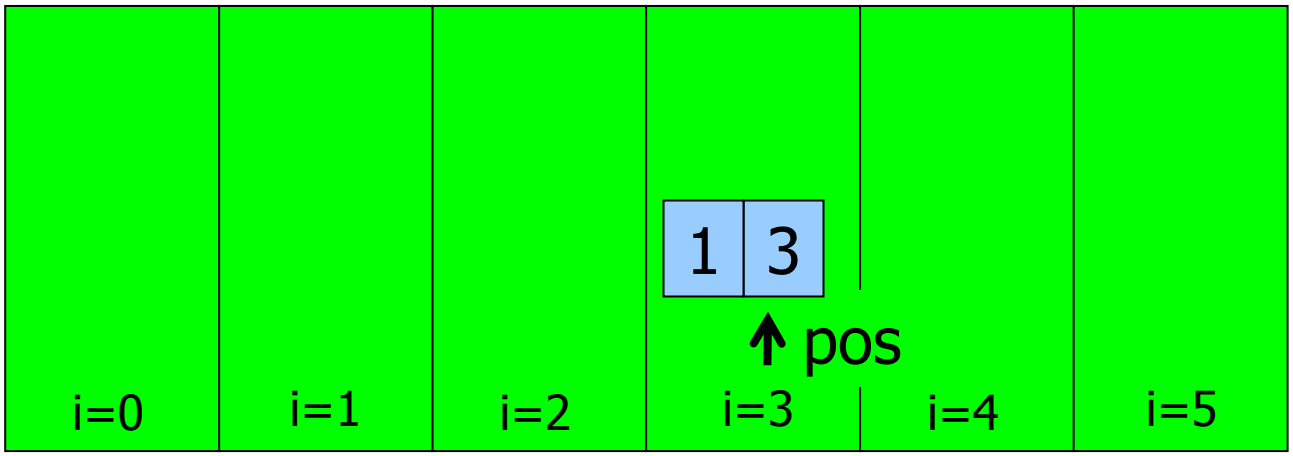


terminazione: visualizza, aggiorna count
ritorna e aggiorna start

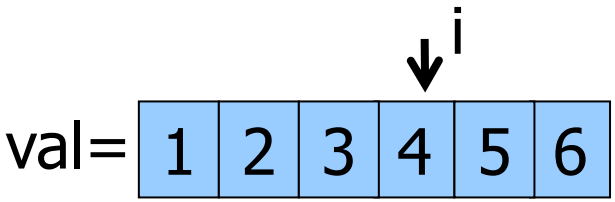
sol= 1 3



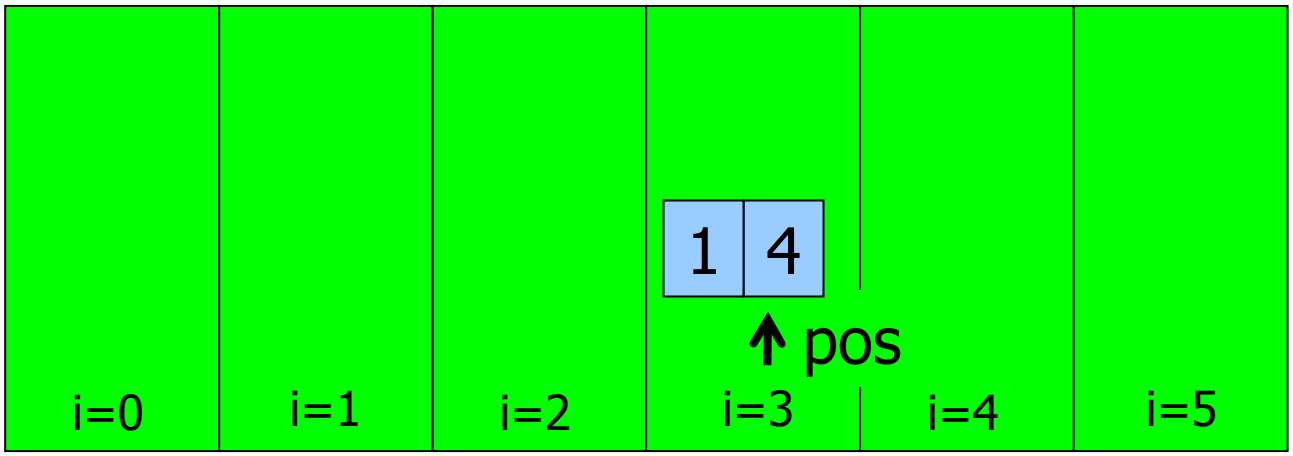
start=3



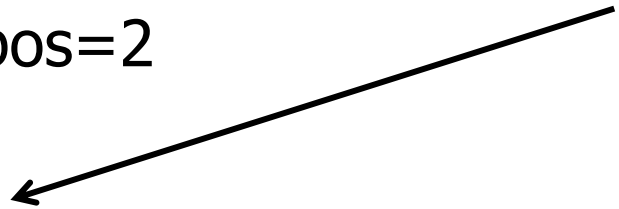
sol[pos] = val[i]

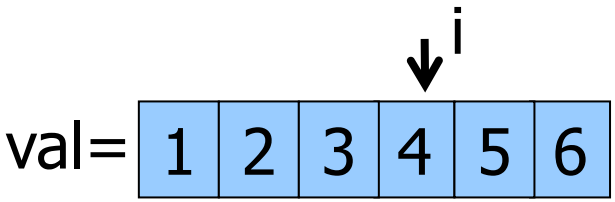


start=3

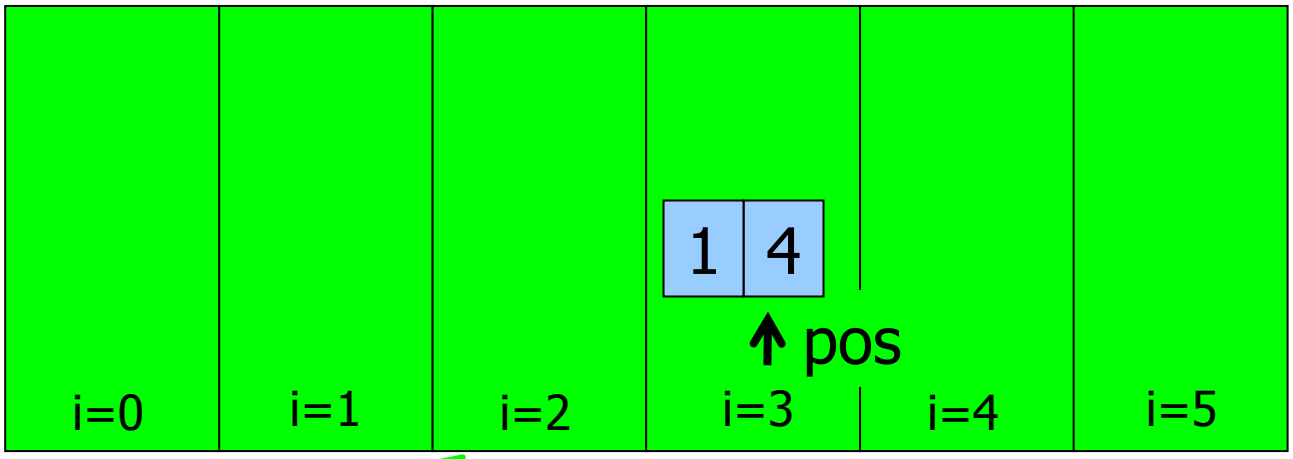


ricorsione con pos=2



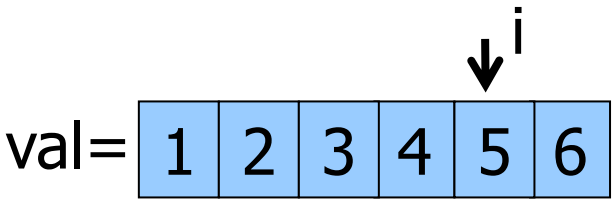


start=4

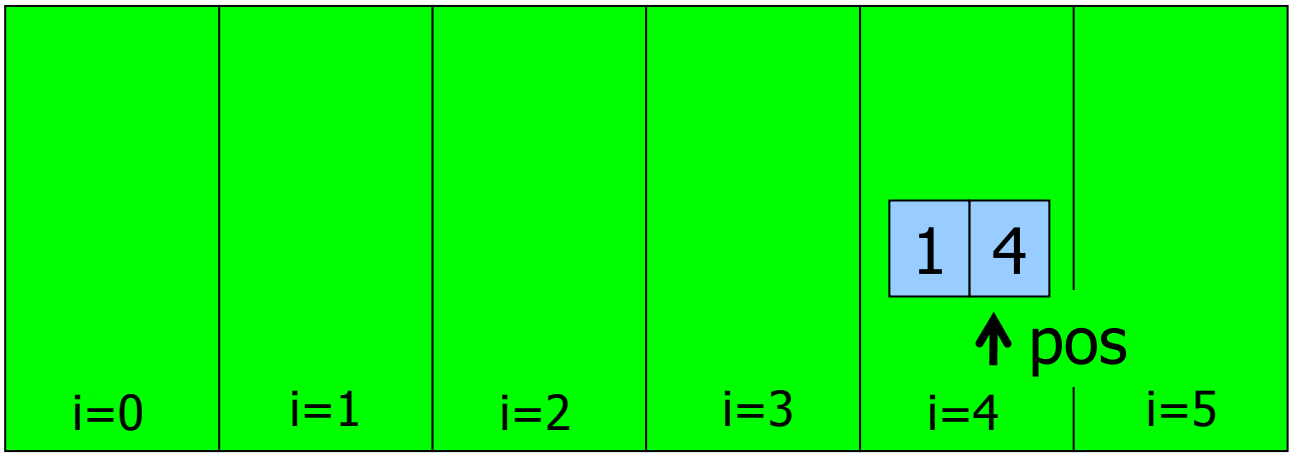


terminazione: visualizza, aggiorna count
ritorna e aggiorna start

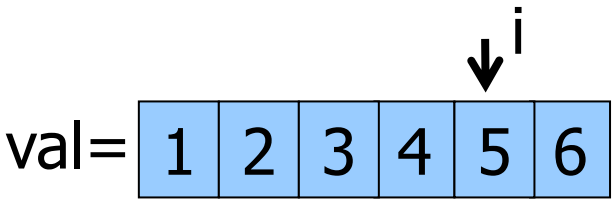
sol= 1 4



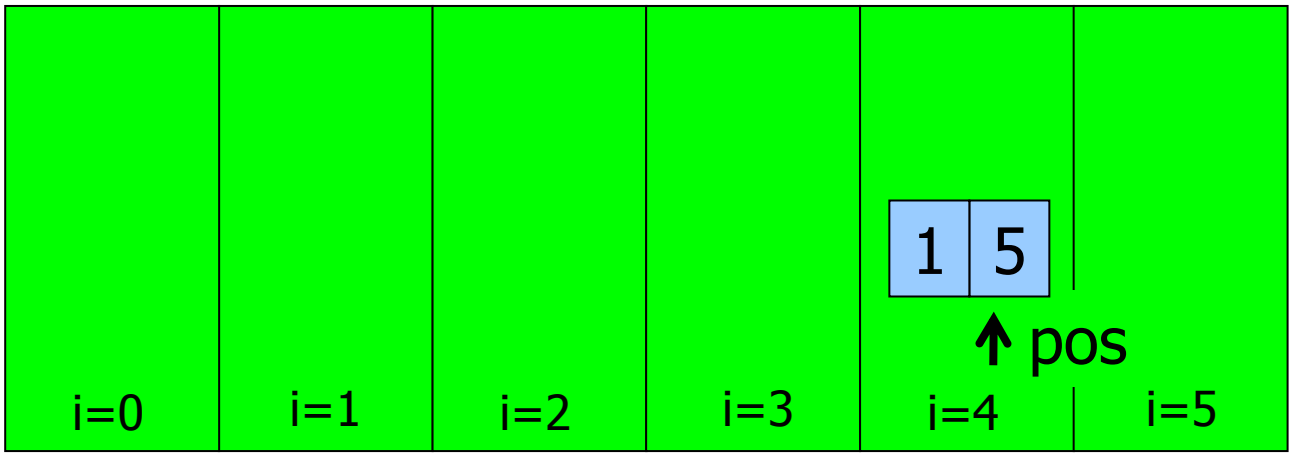
start=4



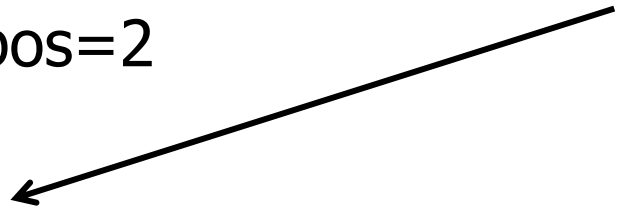
sol[pos] = val[i]

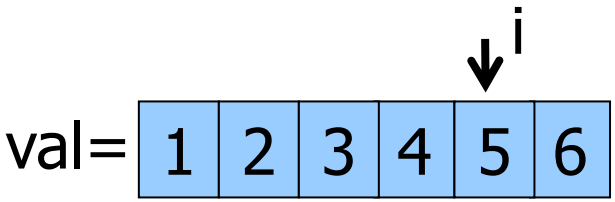


start=4

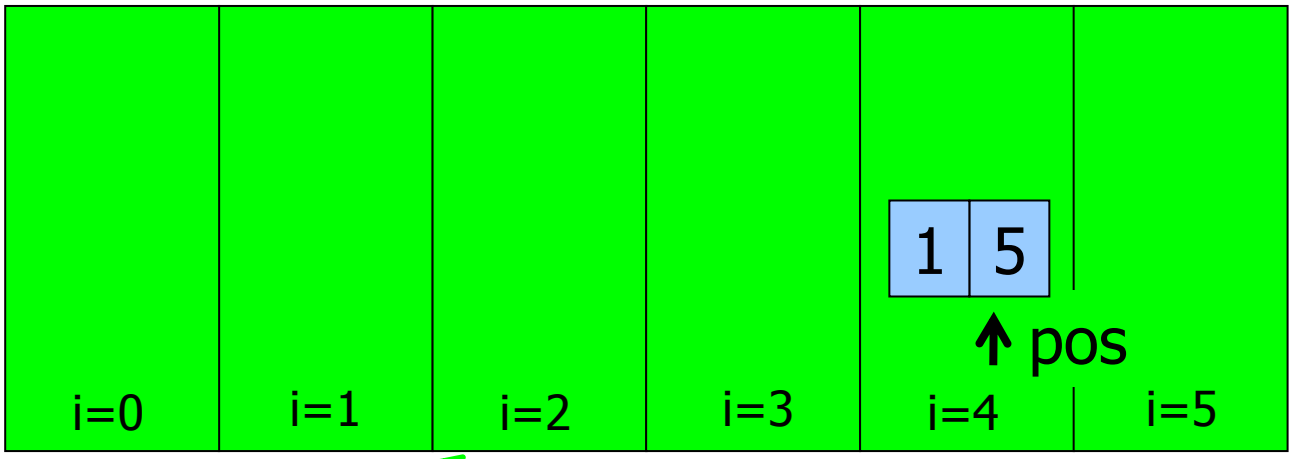


ricorsione con pos=2



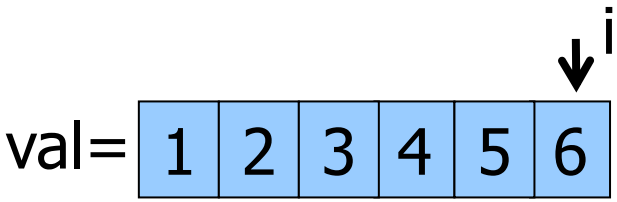


start=5

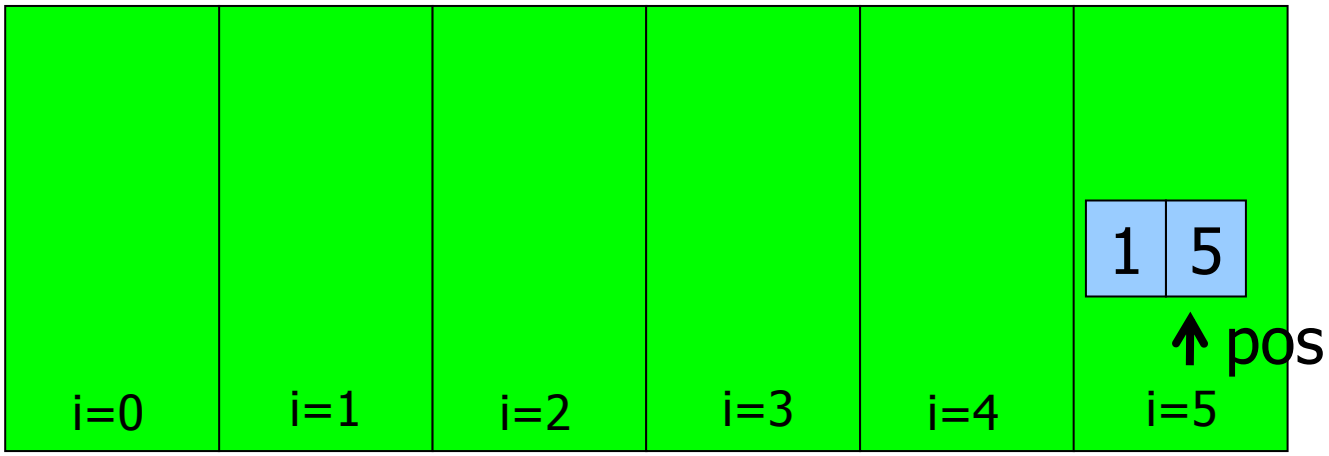


terminazione: visualizza, aggiorna count
ritorna e aggiorna start

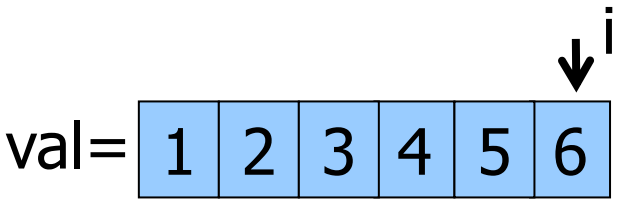
sol= 1 5



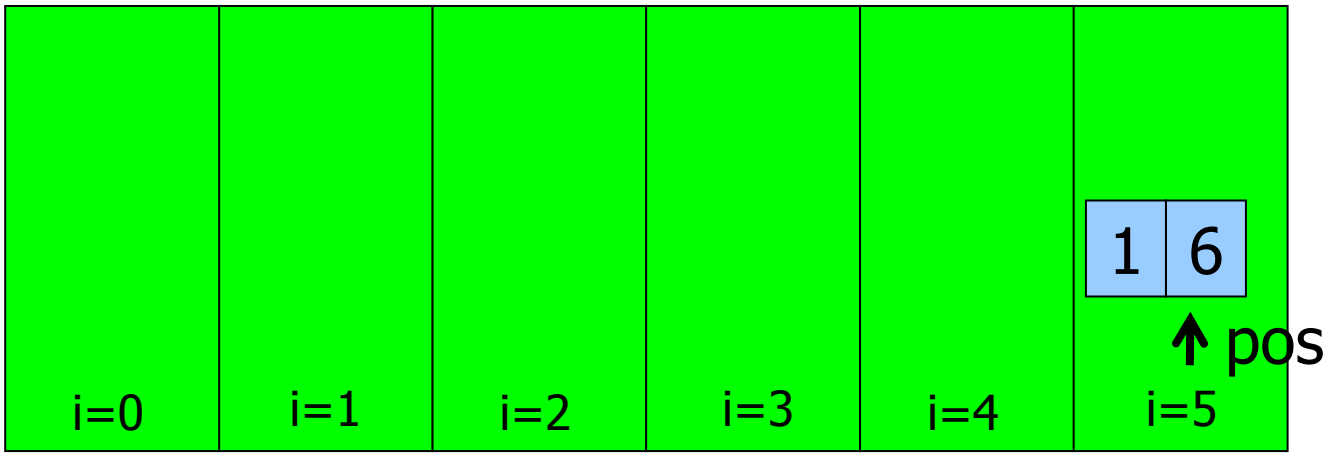
start=5



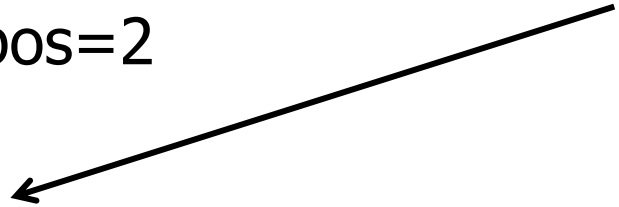
sol[pos] = val[i]

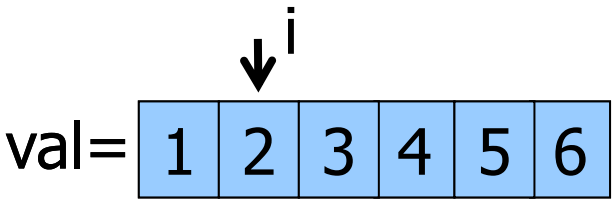


start=5

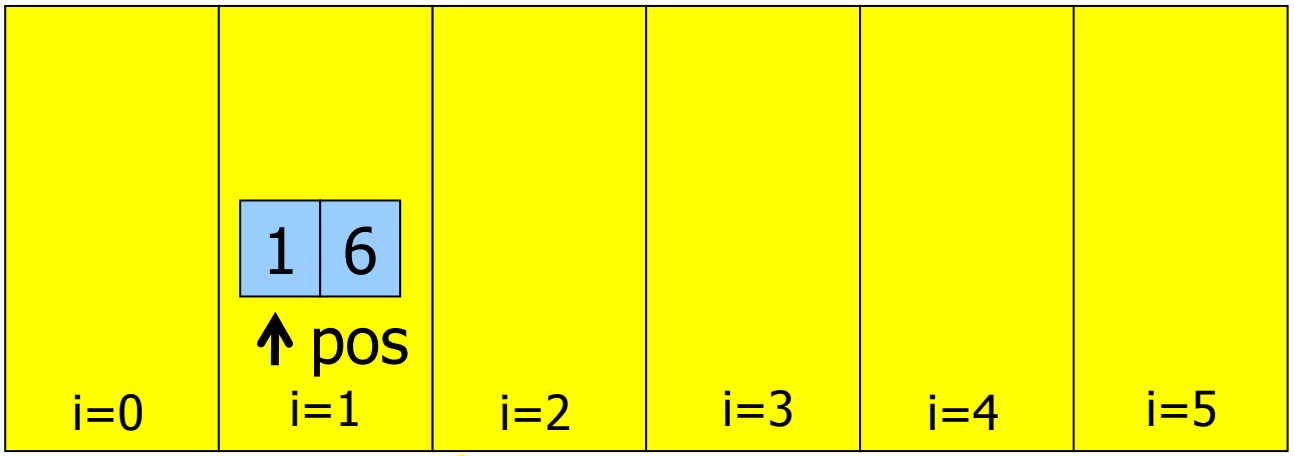


ricorsione con pos=2



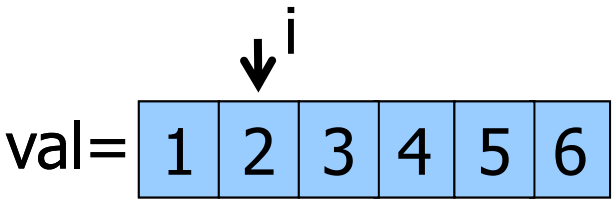


start=1

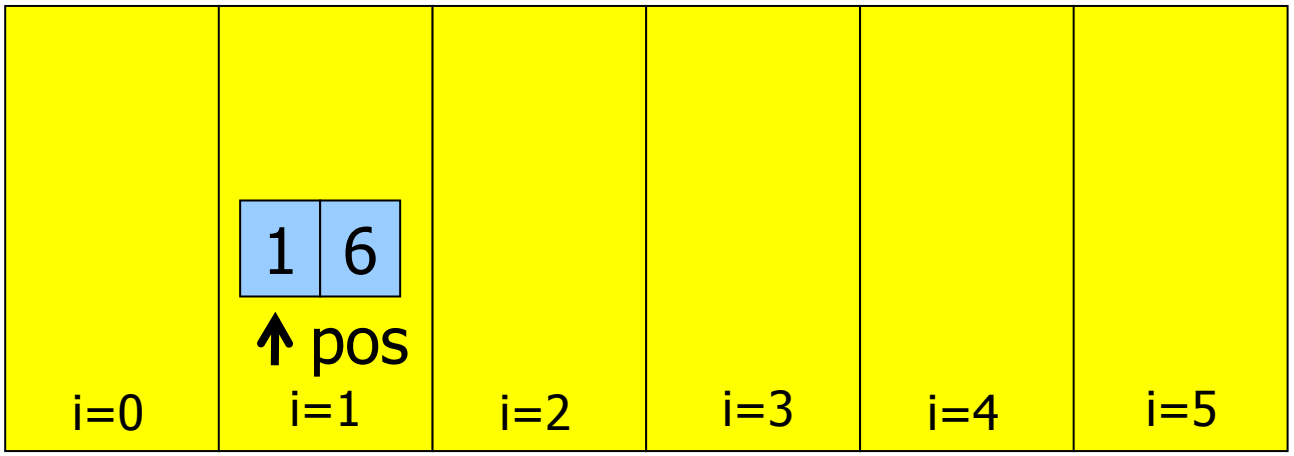


terminazione: visualizza, aggiorna count
ritorna e aggiorna start

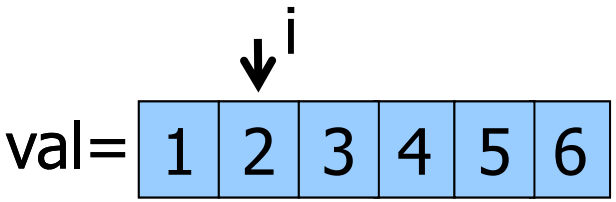
sol= 1 6



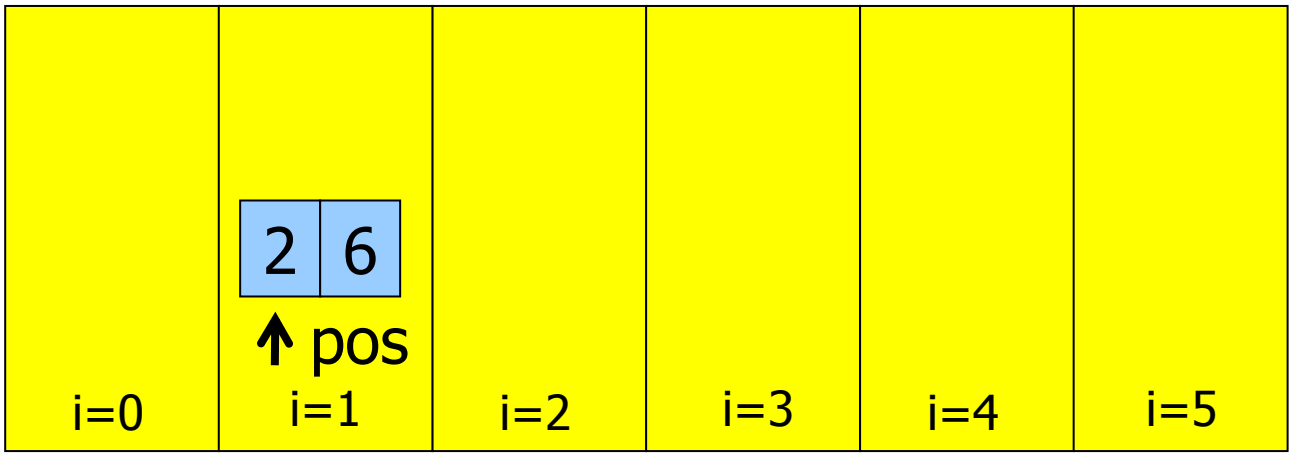
start=1



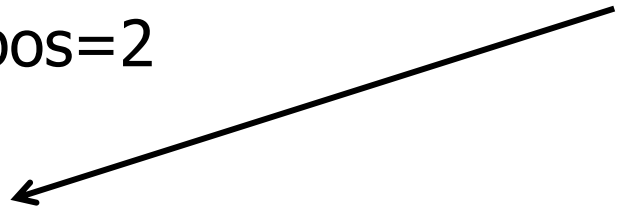
sol[pos] = val[i]



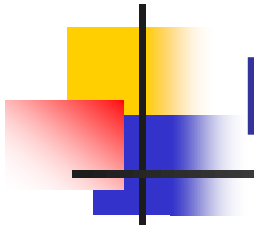
start=1



ricorsione con pos=2



etc. etc.



L’insieme delle parti

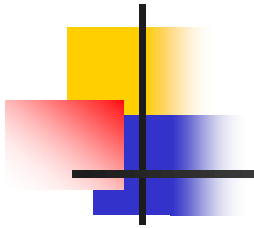
Sono possibili 3 modelli:

1. paradigma divide et impera
2. disposizioni ripetute
3. combinazioni semplici

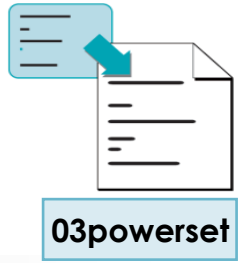
Divide et impera

- caso terminale: insieme vuoto
- caso ricorsivo: insieme delle parti per k-1 elementi unione o l'insieme vuoto o l'elemento k-esimo s_k
- iterazione su tutti gli elementi di S

$$\wp(S_k) = \begin{cases} \varnothing & \text{se } k = 0 \\ \{\ \wp(S_{k-1}) \cup s_k\ \} \cup \{\ \wp(S_{k-1})\ \} & \text{se } k > 0 \end{cases}$$



- 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 `count` rappresenta il numero totale di insiemi.



terminazione: non ci sono più elementi

```
int powerset(int pos, int *val, int *sol, int k,
             int start, int count) {
   int i;
   if (start >= k) {
      for (i = 0; i < pos; i++)
         printf("%d ", sol[i]);
      printf("\n");
      return count+1;
   }
   for (i = start; i < k; i++) {
      sol[pos] = val[i];
      count = powerset(pos+1, val, sol, k, i+1, count);
   }
   count = powerset(pos, val, sol, k, k, count);
   return count;
}
```

per tutti gli elementi da start in poi

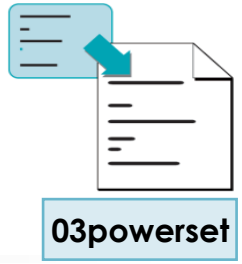
includi elemento e ricorri

non aggiungere nulla e ricorri

Disposizioni ripetute

Ogni sottoinsieme è rappresentato dal vettore della soluzione `sol` di k elementi:

- l'insieme delle scelte possibili per ogni posizione del vettore è {0, 1}, quindi n = 2. Il ciclo for è sostituito da 2 assegnazioni esplicite
- `sol[pos]=0` se l'oggetto pos-esimo non appartiene al sottoinsieme
- `sol[pos]=1` se l'oggetto pos-esimo appartiene al sottoinsieme
- nella stessa soluzione 0 e 1 possono comparire più volte



terminazione: stampa soluzione

```
void powerset(int pos, int *val, int *sol, int k)  {
  int j;
  if (pos >= k) {
    printf("{ \t");
    for (j=0; j<k; j++)
      if (sol[j]!=0)
        printf("%d \t", val[j]);
    printf("} \n");
    return;
  }
  sol[pos] = 0;
  powerset(pos+1, val, sol, k);
  sol[pos] = 1;
  powerset(pos+1, val, sol, k);
}
```

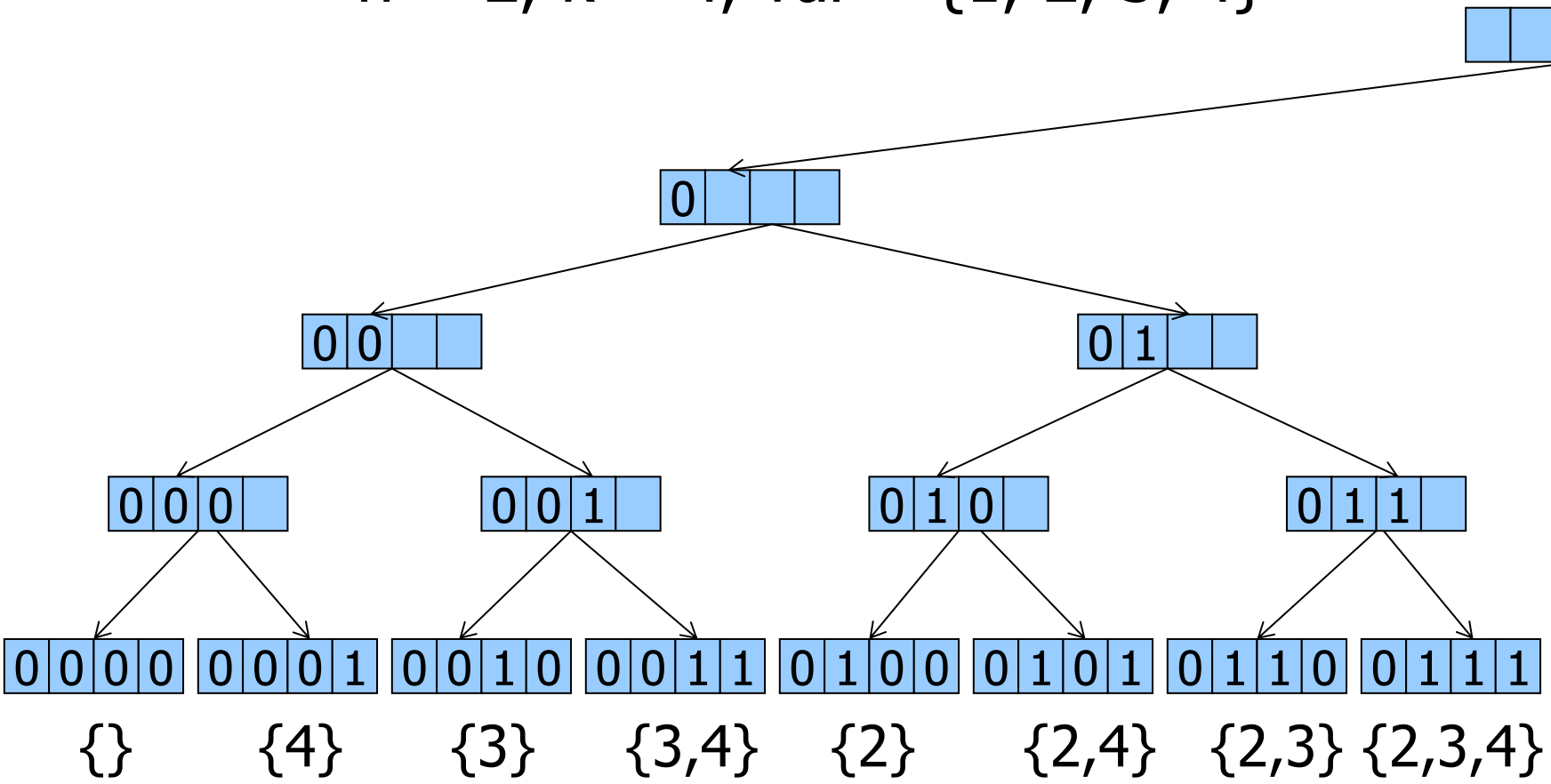
non prendere l'elemento pos

ricorri su pos+1

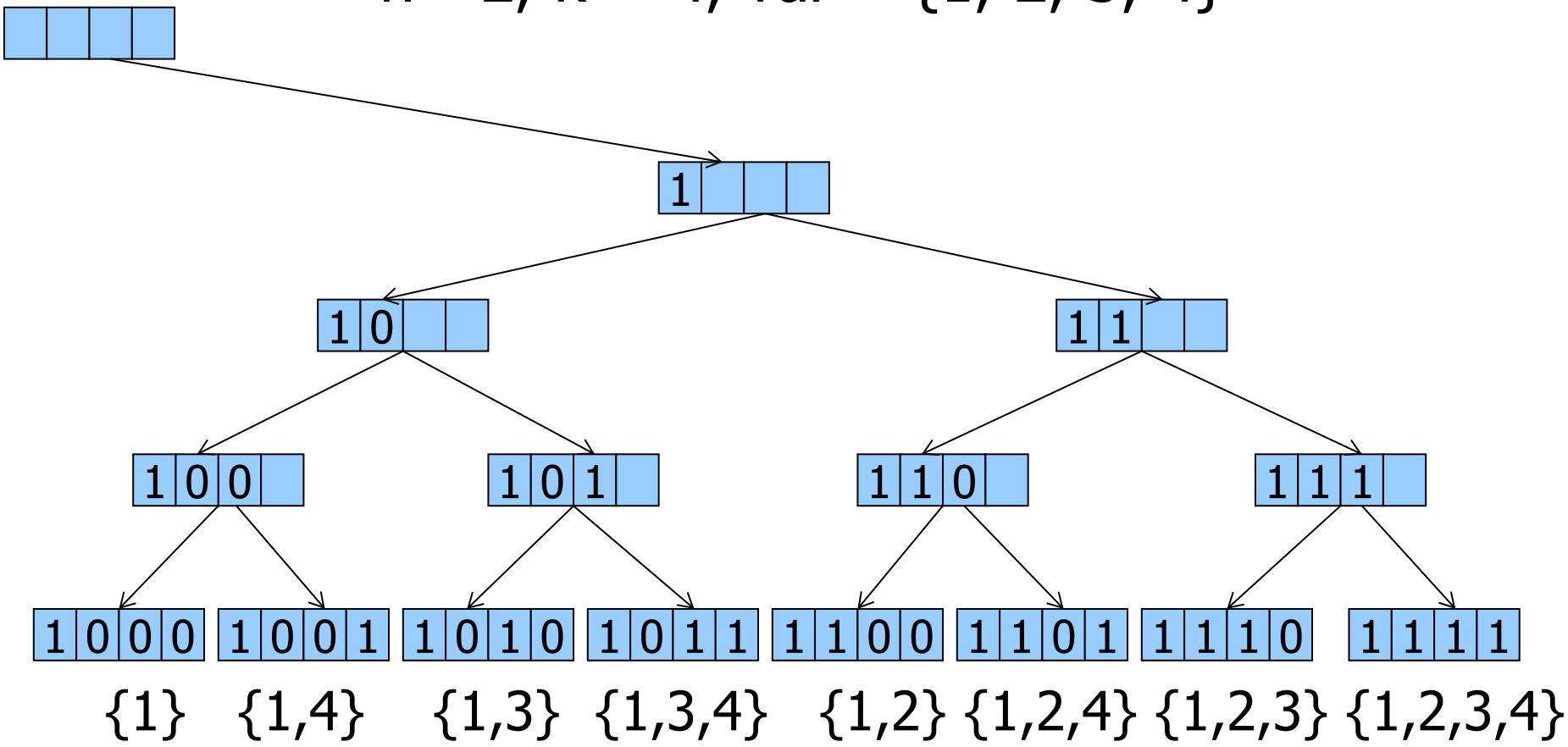
backtrack: prendi l'elemento pos

ricorri su pos+1

n = 2, k = 4, val = {1, 2, 3, 4}



n =2, k = 4, val = {1, 2, 3, 4}

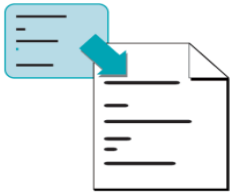


Combinazioni semplici

- Unione di insieme vuoto e insieme delle parti di dimensione 1, 2, 3, …., k
- Modello: combinazioni semplici di k elementi presi a gruppi di n

$$\wp(\mathrm{S}) = \{ \varnothing \} \cup \mathrm{U}_{n=1}^{k} \binom{k}{n}$$

- il wrapper si occupa dell'unione dell'insieme vuoto (non generato dalle combinazioni) e dell'iterare la chiamata alla funzione ricorsiva delle combinazioni.

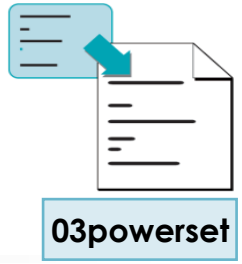


wrapper

```
int powerset(int* val, int k, int* sol){
   int count = 0, n;
   printf("{ }\n");
   count++;
   for(n = 1; n <= k; n++){
      count += powerset_r(val, k, sol, n, 0, 0);
   }
   return count;
}
```

insieme vuoto

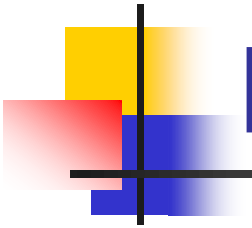
iterazione delle chiamate ricorsive



caso terminale: raggiunto numero prefissato di elementi

```
int powerset_r(int* val, int k, int* sol, int n,
               int pos, int start){
   int count = 0, i;
   if (pos == n){
      printf("{ ");
      for (i = 0; i < n; i++)
         printf("%d ", sol[i]);
      printf(" }\n");
      return 1;
   }
   for (i = start; i < k; i++){
      sol[pos] = val[i];
      count += powerset_r(val, k, sol, n, pos+1, i+1);
   }
   return count;
}
```

per tutti gli elementi da start in poi

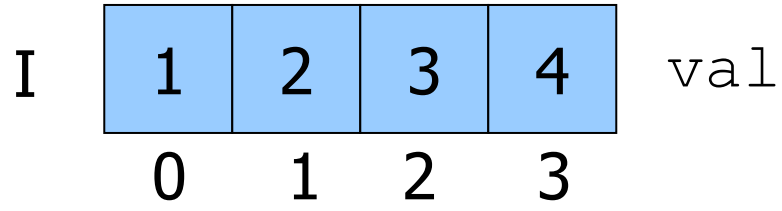


Partizioni di un insieme S

Rappresentazione delle partizioni:

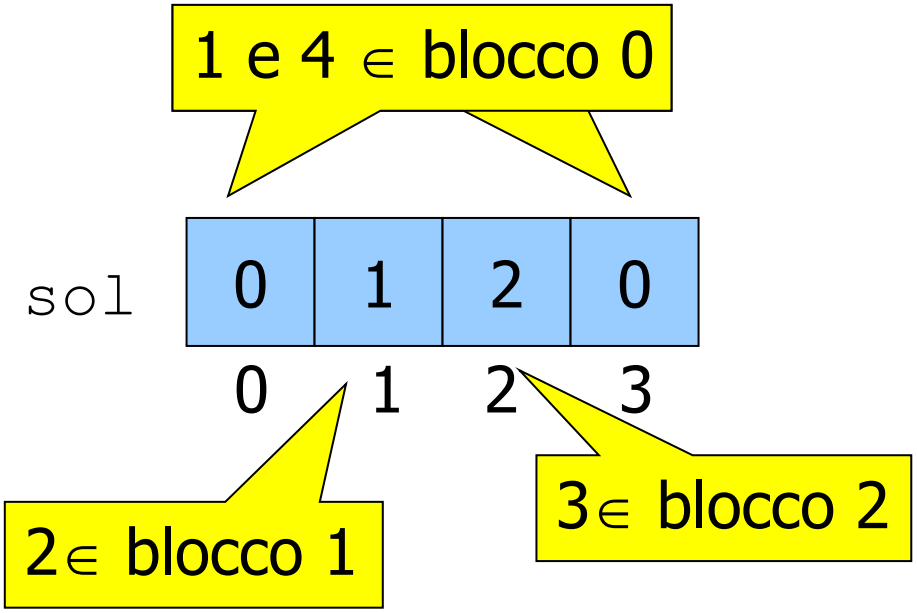
- dato l'elemento, si indica il blocco a cui appartiene univocamente
- dato il blocco, si elencano gli elementi (anche più d'uno) che vi appartengono.

La prima soluzione è preferita in quanto permette di usare vettori di interi.



Esempio

Se I = {1, 2, 3, 4}, n = 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=card(I), determinare:

- una partizione qualsiasi
- tutte le partizioni in k blocchi con k tra 1 e n
- tutte le partizioni in k blocchi.

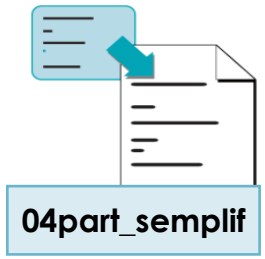
disposizioni ripetute

algoritmo di Er

Disposizioni ripetute

- Il numero di oggetti memorizzati nel vettore `val` è `n`
- Il numero delle scelte possibili per ogni oggetto è `k`, cioè il numero di blocchi. Si tratta di una generalizzazione del powerset rimuovendo il vincolo della scelta limitata a 0 oppure 1
- Il vettore `sol` di `n` celle contiene a quale blocco tra 0 e `k-1` appartiene l'oggetto di indice corrente
- Necessità di un controllo nella condizione di terminazione per evitare blocchi vuoti (calcolo delle occorrenze di ciascun blocco).

```
val = malloc(n*sizeof(int));
sol = malloc(n*sizeof(int));
```




```
void disp_ripet(int pos,int *val,int *sol,int n,int k) {
  int i, j, t, ok=1, occ[k];
  for (i=0; i<k; i++) occ[i]=0;
  if (pos >= n) {
    for (j=0; j<n; j++)
       occ[sol[j]]++;
    i=0;
    while ((i < k) && ok) {
        if (occ[i]==0) ok = 0;
        i++;
    }
    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 delle occorrenze

controllo occorrenze

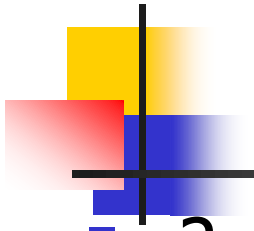
soluzione scartata

ricorsione

Algoritmo di Er (1987)

Calcolo di tutte le partizioni di `n` oggetti memorizzati nel vettore `val` in `k` blocchi con `k` tra 1 e `n`:

- indice `pos` per scorrere gli `n` oggetti e terminare la ricorsione quando `pos >= n`
- indice `k` per scorrere i blocchi utilizzabili in quel passo
- vettore `sol` di `n` elementi per la soluzione



- 2 ricorsioni:
  - attribuisco l'oggetto corrente a uno dei blocchi con indice tra 0 e `m` e ricorro sul prossimo oggetto
  - attribuisco l'oggetto corrente al blocco `m` e ricorro sul prossimo oggetto e su un numero di blocchi incrementato di 1.

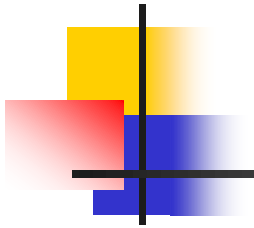
```
val = malloc(n*sizeof(int));
sol = malloc(n*sizeof(int));
```

```
void SP_rec(int n,int m,int pos,int *sol,int *val) {
  int i, j;
  if (pos >= n) {
    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++) {
    sol[pos] = i;
    SP_rec(n, m, pos+1, sol, val);
  }
  sol[pos] = m;
  SP_rec(n, m+1, pos+1, sol, val);
}
```

condizione di terminazione

ricorsione sugli oggetti

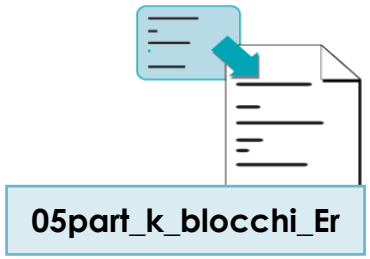
ricorsione su oggetti e blocchi



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.

```
val = malloc(n*sizeof(int));
sol = malloc(n*sizeof(int));
```




```
void SP_rec(int n,int k,int m,int pos,int *sol,int *val){
  int i, j;
  if (pos >= n) {
    if (m == k) {
      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;
    SP_rec(n, k, m, pos+1, sol, val);
  }
  sol[pos] = m;
  SP_rec(n, k, m+1, pos+1, sol, val);
}
```

condizione di terminazione

filtro

ricorsione sugli oggetti

ricorsione su oggetti e blocchi

Esplorazione dello spazio delle possibilità: singola soluzione

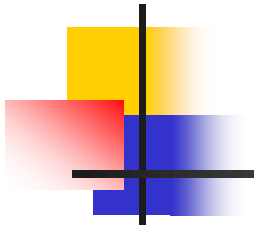
Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

Singola soluzione

La ricorsione è particolarmente utile quando si vogliono elencare tutte le soluzioni e questo è obbligatorio nei problemi di ottimizzazione.

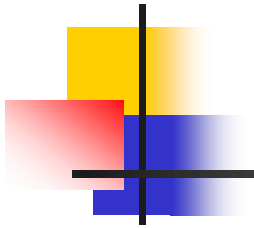
Se ne basta una sola, bisogna far sì che tutte le ricorsioni si chiudano, ricordando che ognuna torna a quella che l'ha chiamata.

E' infatti errato pensare di poter «forare» la catena delle ricorsioni, tornando subito a quella iniziale.



Soluzioni:

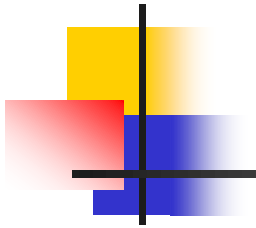
- uso di un flag:
  - variabile globale (soluzione sconsigliata)
  - parametro passato by reference
- funzione ricorsiva che ritorna un valore di successo o fallimento che viene testato.



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):

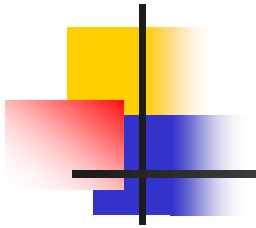
- nella condizione di terminazione, se la condizione di accettazione è verificata si ritorna 1, altrimenti si ritorna 0
- nel ciclo di scelta:
  - si effettua la scelta
  - si testa il risultato della chiamata ricorsiva: se c'è successo si ritorna 1
- terminato il ciclo di scelta: si ritorna 0.

nel main

```
if (funz_ric(.....  )==0)
  printf("soluzione non trovata\n");

int funz_ric(.....) {
 if (condizione di terminazione)
   if (condizione di accettazione) {
     .....
     return 1;
   else
     return 0;
   }
 for (ciclo sulle scelte) {
    scelta;
    if (funz_ric(.....))
      return 1;
    .....
  }
  return 0;
}
```

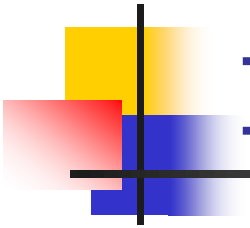
Problemi di ottimizzazione

Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino



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.

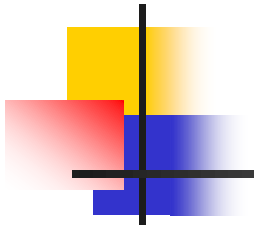


Il conto corrente

Input: vettore di interi di lunghezza nota n. Ogni intero rappresenta un movimento distinto su un conto bancario:

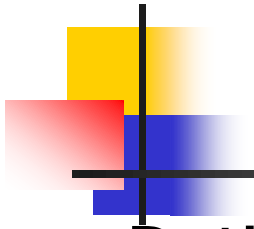
- >0: entrata
- <0: uscita.

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'ordinamento del vettore che minimizza la differenza tra saldo corrente massimo e saldo corrente minimo.



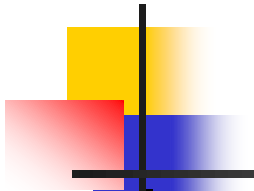
Dati n=10 e val={1,-2,3,14,-5,16,7,8,-9,120}, con

l'ordinamento

- {3,1,14,-2,-5,16,7,8,-9,120} il saldo max è 153, quello min è 3, la differenza è 150
  {120,1,3,-2,14,-5,16,-9,7,8} il saldo max è 153, quello min è 120, la differenza è 33 ed è una soluzione.

Modello:

- permutazioni semplici per enumerare gli ordinamenti



Algoritmo:

- algoritmo ricorsivo per le permutazioni semplici
- quando in condizione di terminazione:
  - calcolo di saldo max e min e della differenza corrente
  - confronto della differenza corrente con la minima sinora trovata
  - eventuale aggiornamento della soluzione.

Ipotesi:

- è noto un limite superiore alla massima differenza minima (= INT_MAX).

```
val = malloc(n sizeof(int));
sol = malloc(n*sizeof(int));
mark = malloc(n*sizeof(int));
fin = malloc(n*sizeof(int));
```

08contocorrente

variabile globale

```
#include <stdio.h>
#include <stdlib.h>
#include <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, k, tot, *val, *sol, *mark, *fin;
  printf("Inserisci n: "); scanf("%d", &n);
  val=malloc(n*sizeof(int));
  sol=malloc(n*sizeof(int));
  mark=malloc(n*sizeof(int));
  fin=malloc(n*sizeof(int));
  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
  return 0;
}
```

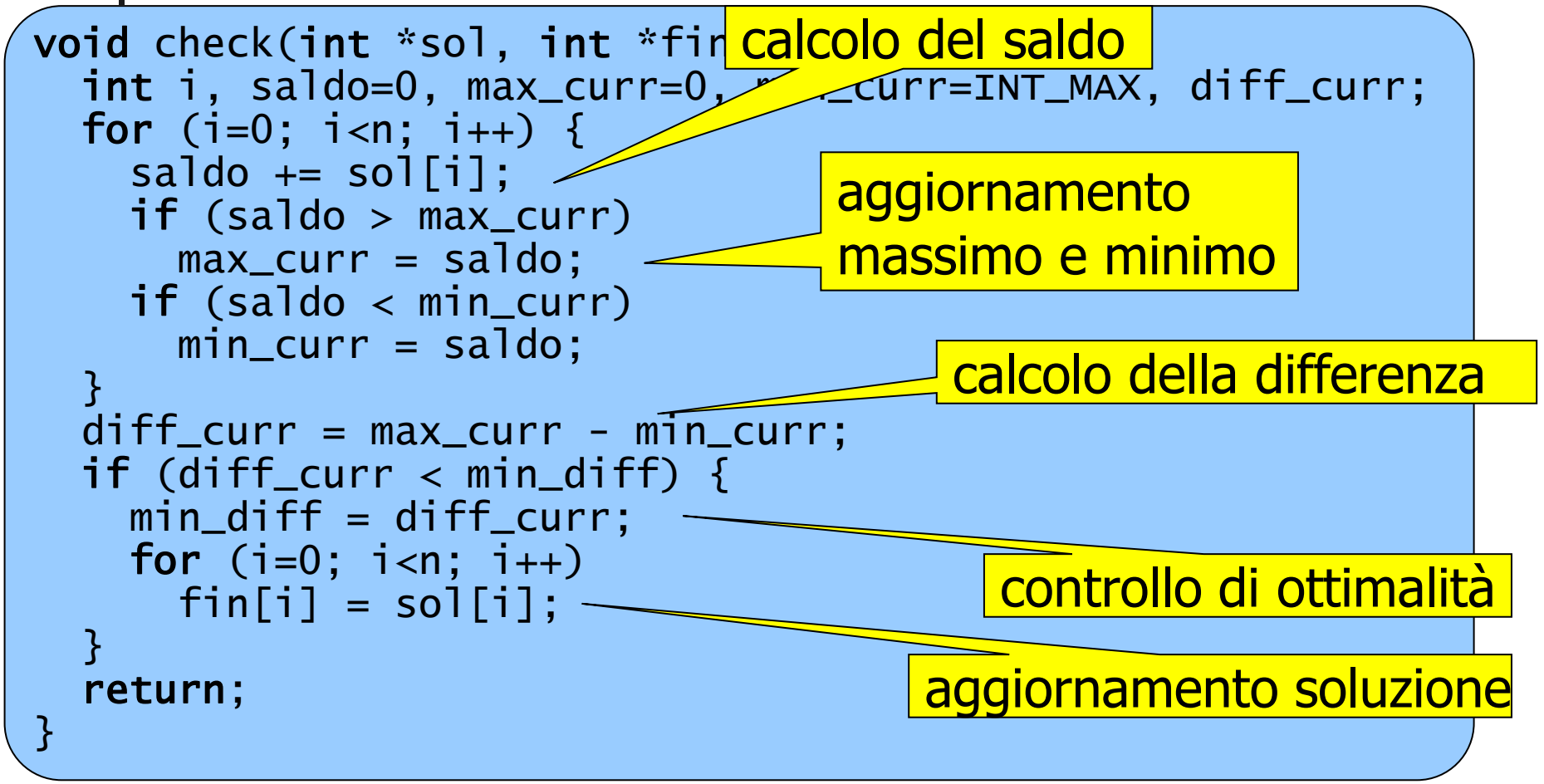
non serve il numero delle permutazioni

```
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 i, saldo=0, max_curr=0, m
_curr=INT_MAX, diff_curr;
for (i=0; i<n; i++) {
saldo += sol[i];
if (saldo > max_curr)
max_curr = saldo;
if (saldo < min_curr)
min_curr = saldo;
}
diff_curr = max_curr - min_curr;
if (diff_curr < min_diff) {
min_diff = diff_curr;
for (i=0; i<n; i++)
fin[i] = sol[i];
}
return;
}
calcolo del saldo
aggiornamento
massimo e minimo
calcolo della differenza
controllo di ottimalità
aggiornamento soluzione

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:

- $\sum_{j \in S} w_j x_j \leq$ cap
- $\sum_{j \in S} v_j x_j =$ 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

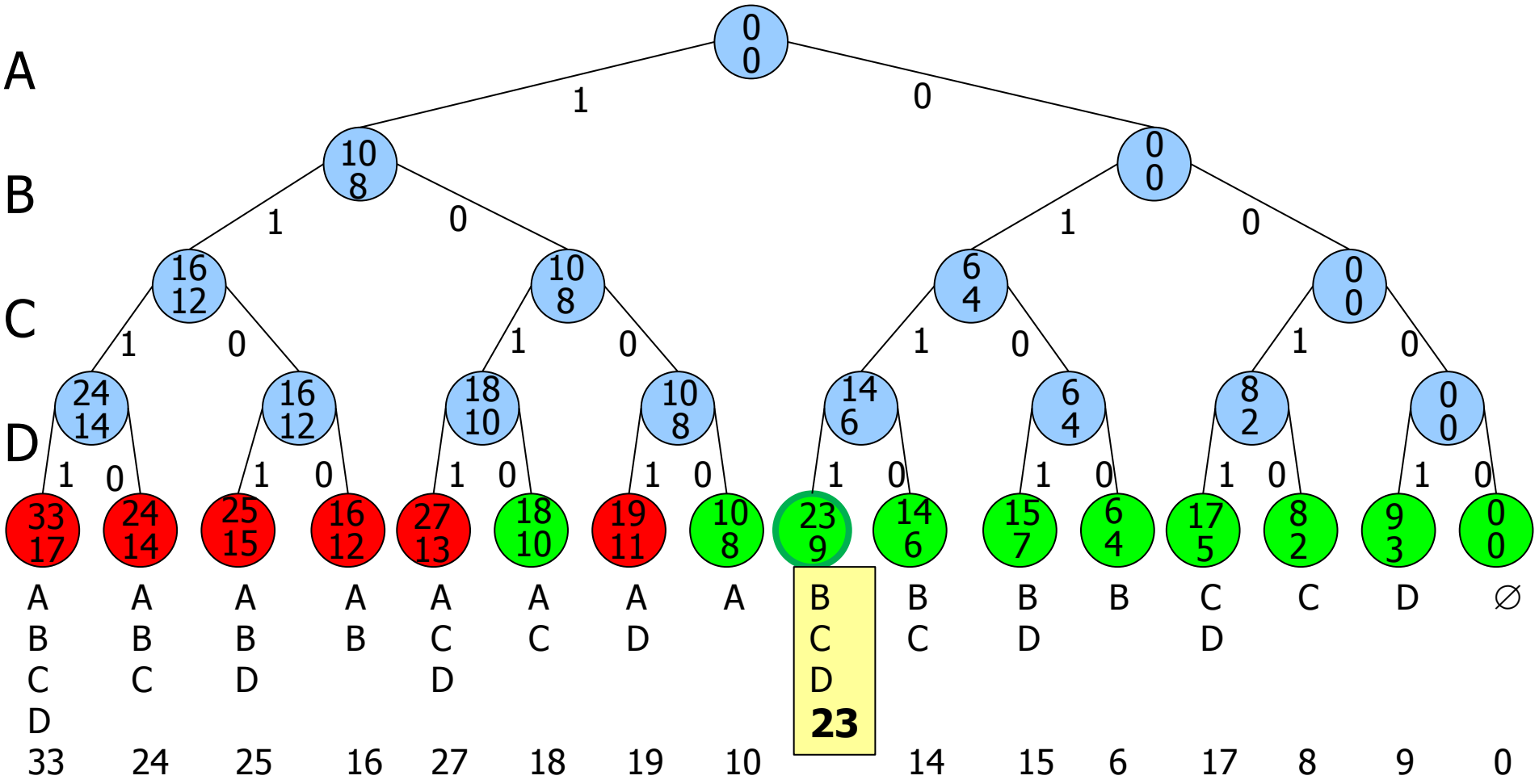
cap = 10

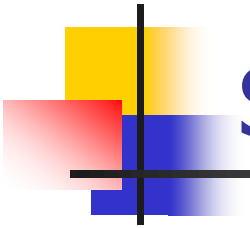
		A	B	C	D
Valore	v_i	10	6	8	9
Peso	w_i	8	4	2	3

Soluzione:

insieme {B, C, D} con valore massimo 23

Modello: disposizioni ripetute (powerset)





Strutture dati

- Strutture già viste per le disposizioni ripetute e inoltre:
- Variabili intere:
  - `cap` per la capacità dello zaino
  - `curr_value` per il valore corrente
  - `curr_cap` per la capacità usata correntemente
  - `bv` valore ottimo corrente
- Vettore di interi `best_sol` per la soluzione ottima corrente.

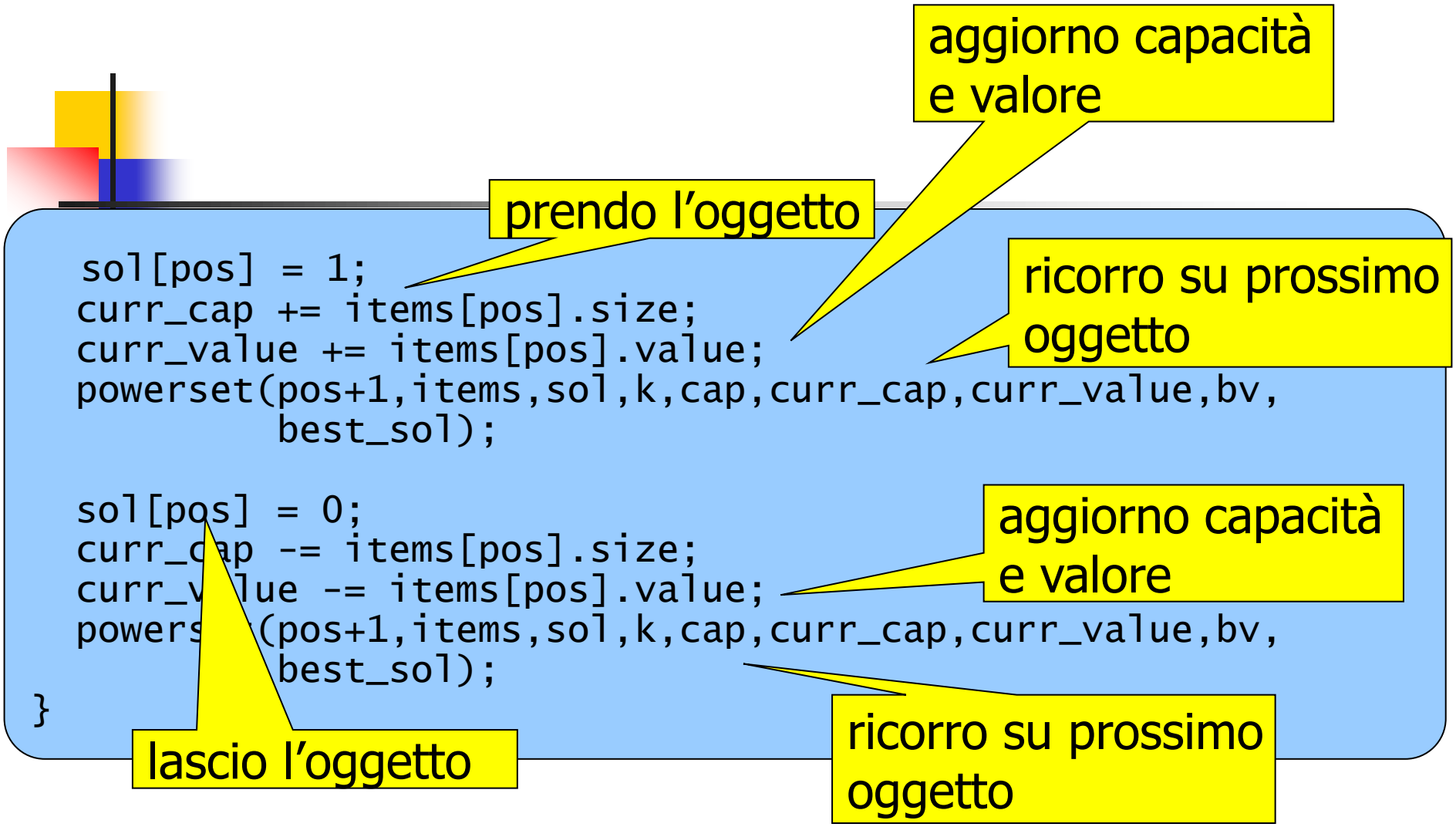
knapsack_no_pruning

```
void powerset(int pos,Item *items,int *sol,int k, int cap,
    int curr_cap,int curr_value,int *bv, int *best_sol) {

  int j;

  if (pos >= k) {
    if (curr_cap <= cap) {
      if (curr_value > *bv) {
        for (j=0; j<k; j++)
          best_sol[j] = sol[j];
        *bv = curr_value;
      }
    }
    return;
  }
```

condizione di terminazione

controllo accettabilità

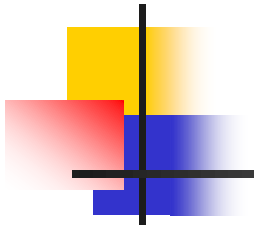
controllo ottimalità



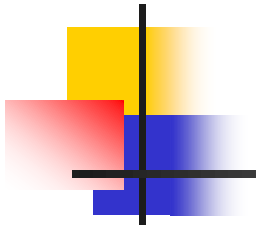

```
    sol[pos] = 1;
    curr_cap += items[pos].size;
    curr_value += items[pos].value;
    powerset(pos+1,items,sol,k,cap,curr_cap,curr_value,bv,
             best_sol);

    sol[pos] = 0;
    curr_cap -= items[pos].size;
    curr_value -= items[pos].value;
    powerset(pos+1,items,sol,k,cap,curr_cap,curr_value,bv,
             best_sol);
}
```

Il pruning dello spazio

Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

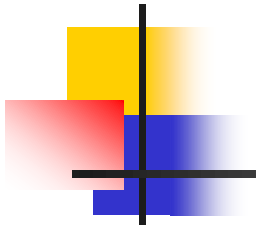


- Criterio di accettazione della soluzione espresso mediante vincoli
- Vincoli valutati:
  - direttamente nei casi terminali, senza specifica struttura dati
  - ad ogni chiamata ricorsiva, mediante struttura dati aggiornata dinamicamente
- Crescita molto rapida dello spazio delle soluzioni $\Rightarrow$ inapplicabilità dell'approccio enumerativo



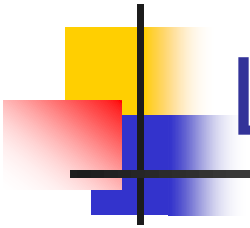
Pruning:

- riduzione dello spazio di ricerca
- nessuna perdita di esattezza
- 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.



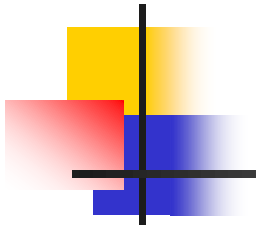
La somma di sottoinsiemi

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:

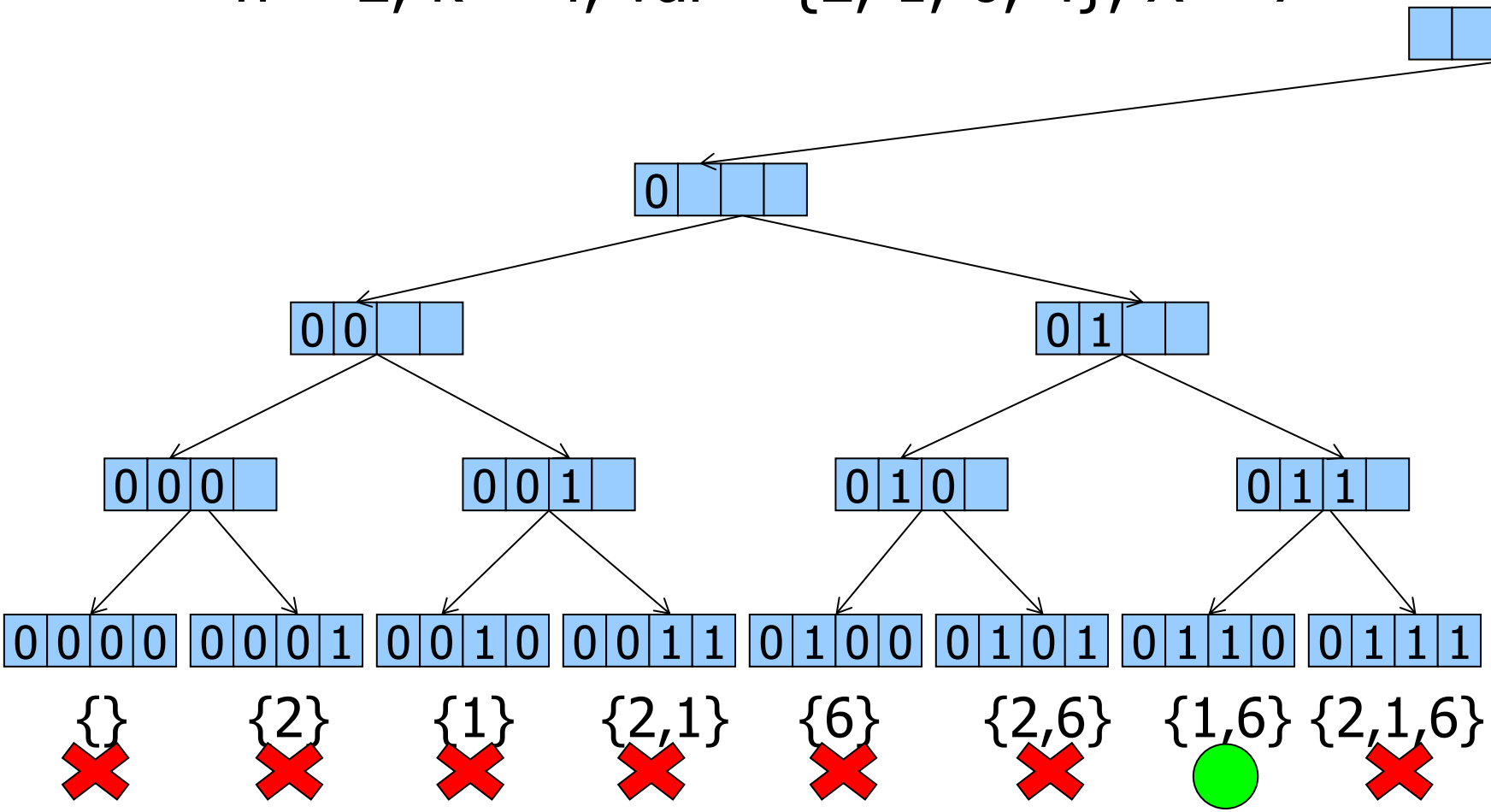
Y = { {1,2,4}, {1,6} }



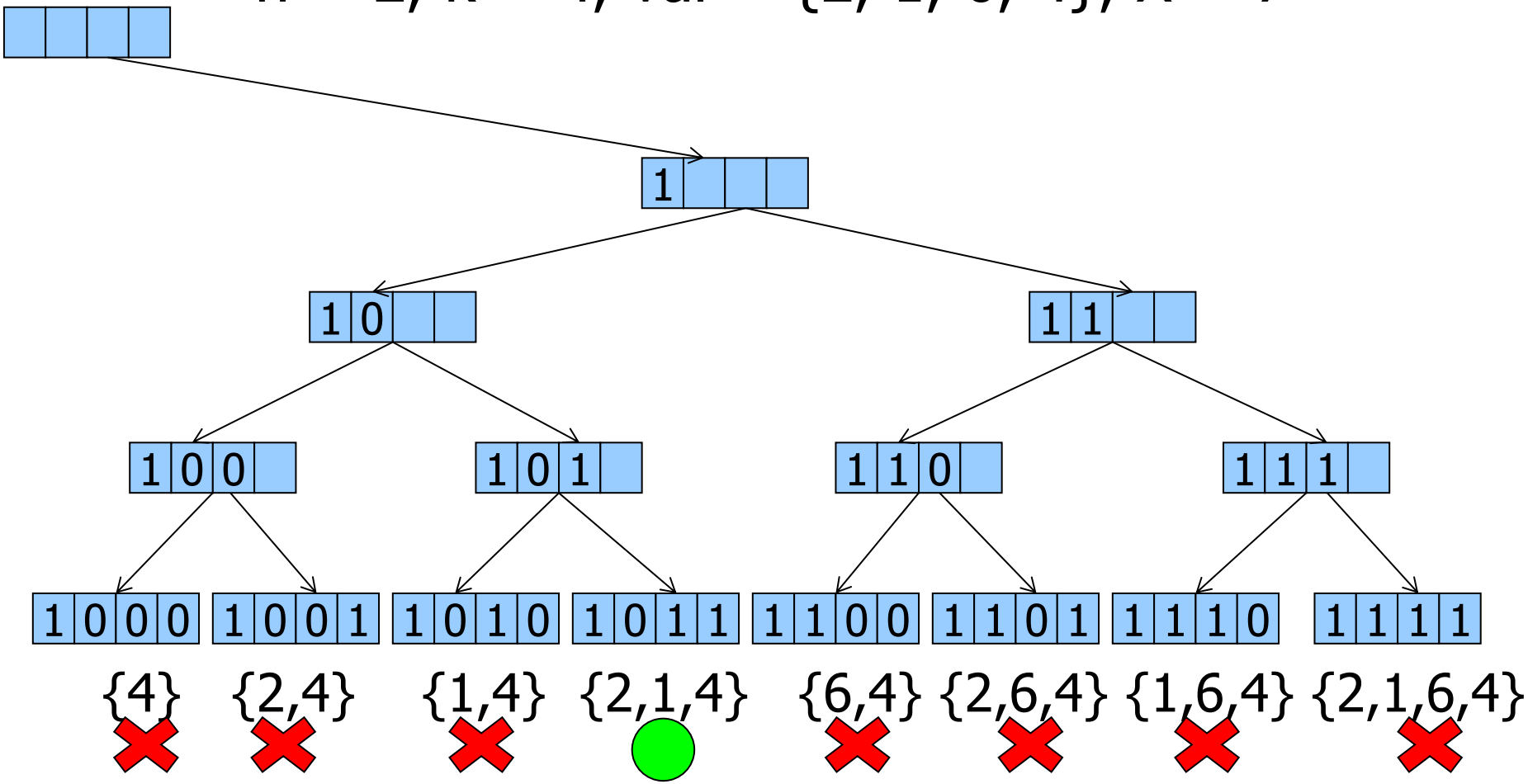
Approccio enumerativo (senza pruning):

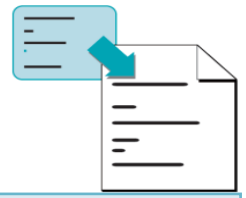
- calcolare il powerset $\wp$(val) (disposizioni ripetute)
- per ogni sottoinsieme (condizione di terminazione), verificare se la somma dei suoi elementi è X.

n = 2, k = 4, val = {2, 1, 6, 4}, X = 7



n = 2, k = 4, val = {2, 1, 6, 4}, X = 7





06simple_sum_of_subsets

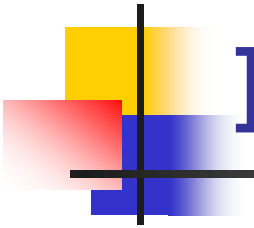
```
val = malloc(k * sizeof(int));
sol = malloc(k * sizeof(int));
```

```
void powerset(int pos,int *val,int *sol,int k,int X) {
  int j, out;
  if (pos >= k) {
    out = check(sol, val, X, k);
    if (out==1) {
}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;
  else
    return 0;
}
```

terminazione

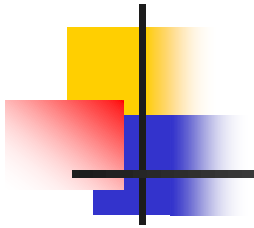
verifica soluzione

stampa soluzione



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)

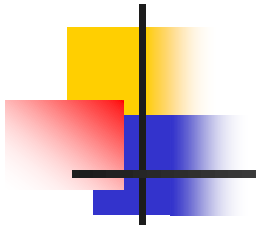


- Filtro dinamico sulle scelte: condizioni di accettazione che dipendono dalle scelte precedenti e dal problema (ad esempio la posizione di altri pezzi nel gioco)
- Validazione di una soluzione parziale: valutazione della speranza di raggiungere una soluzione o condizione sufficiente per decidere che 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_sum` è la somma corrente, inizialmente 0
- `tot`, inizialmente pari a `total`, somma di tutti i valori di `val`, contiene la somma dei valori non ancora presi, quindi ancora disponibili
- ad ogni passo si prende in considerazione un elemento di `val` solo se "promettente".



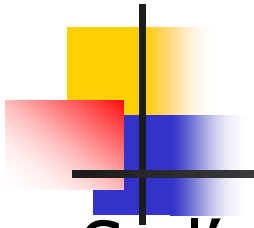
Un elemento di `val` è promettente se:

- la soluzione parziale + i valori che restano sono >= della somma cercata

```
p_sum + tot >= X
        &&
```

- la soluzione parziale + il valore di `val` è <= della somma cercata

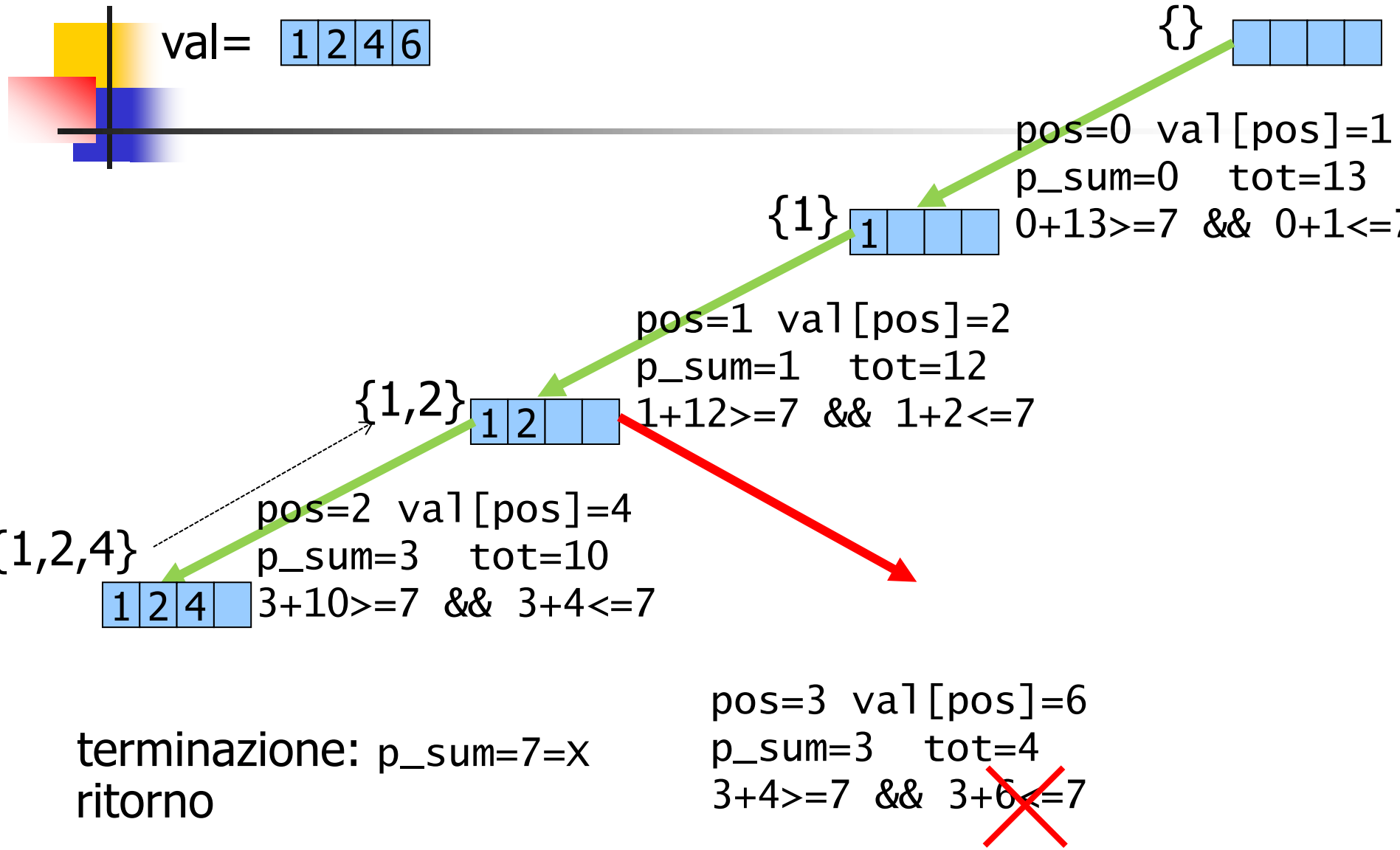
```
p_sum + val[pos]<= X
```



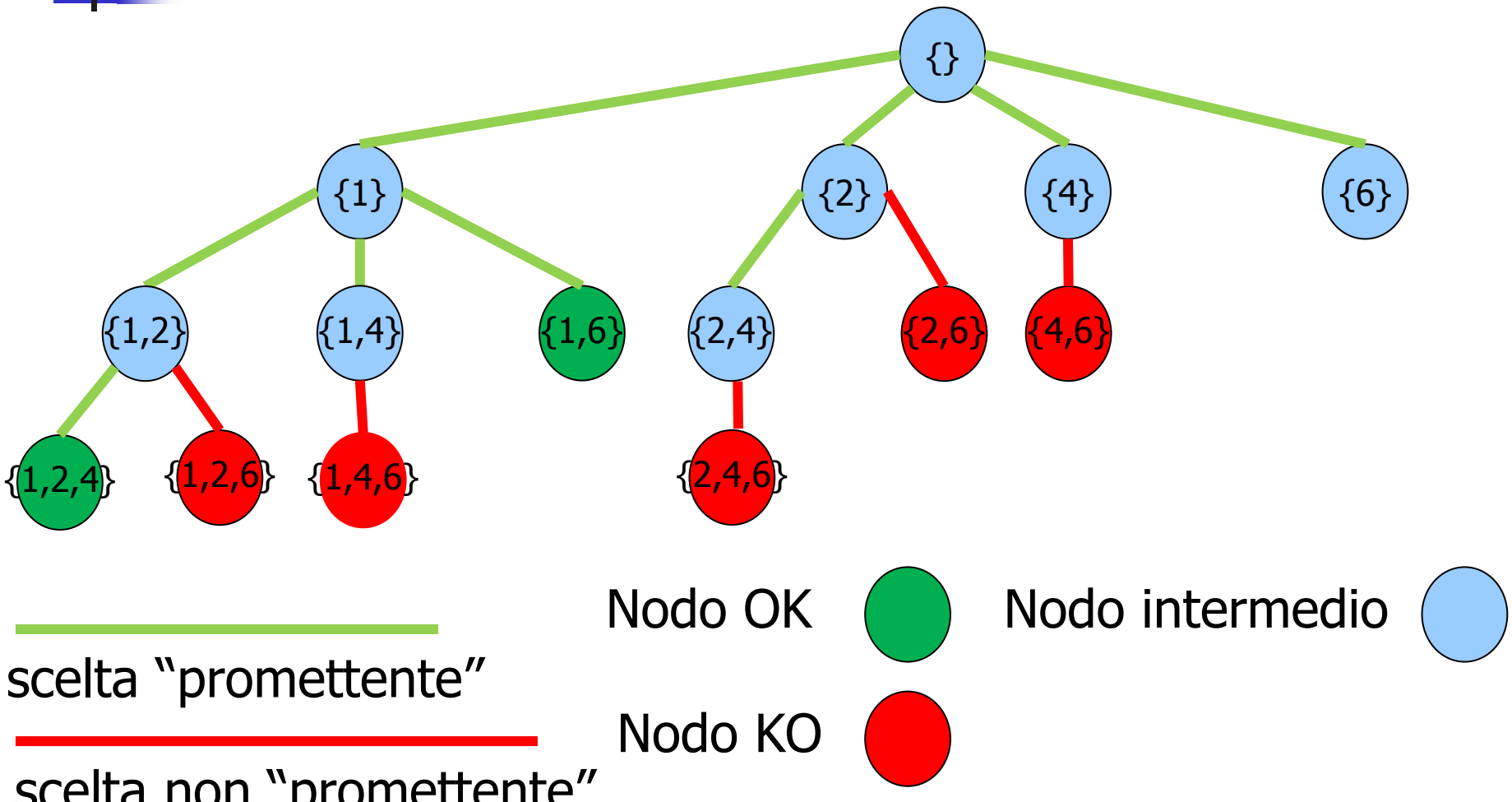
Se l'elemento è promettente:

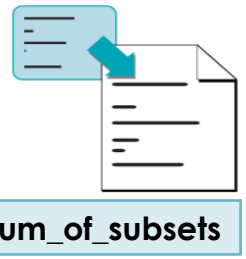
- lo si prende (`mark[pos]=1`)
- si ricorre sul prossimo (`pos+1`), aggiornando `p_sum` (`p_sum+val[pos]`) e `tot` (`tot-val[pos]`)
- in fase di backtrack, non lo si prende (`mark[pos]= 0`)
- si ricorre sul prossimo (`pos+1`), `p_sum` resta invariato, `tot` viene aggiornato (`tot-val[pos]`)

Promising: p_sum + tot >= X && p_sum + val[pos]<= X



terminazione: p_sum=7=X
ritorno


{}
{1}
{2}
{4}
{6}
{1,2}
{1,4}
{1,6}
{2,4}
{2,6}
{4,6}
{1,2,4}
{1,2,6}
{1,4,6}
{2,4,6}
Nodo OK
Nodo intermedio
Nodo KO
scelta "promettente"
scelta non "promettente"




```
val = malloc(k*sizeof(int));
sol = malloc(k*sizeof(int));
```

```
void sumset(int pos,int *val,int *mark,int p_sum,
            int tot,int X) {
  int j;
  if (p_sum==X) {
    printf("\n{\t");
    for(j=0;j<pos;j++)
      if(mark[j])
        printf("%d\t",val[j]);
    printf("}\n");
    return;
  }
  if(promising(val,pos,p_sum,tot,X)){
    mark[pos]=1;
    sumset(pos+1,val,mark,p_sum+val[pos],tot-val[pos],X);
    mark[pos]=0;
    sumset(pos+1,val,mark,p_sum, tot-val[pos],X);
  }
}
```

terminazione

stampa soluzione

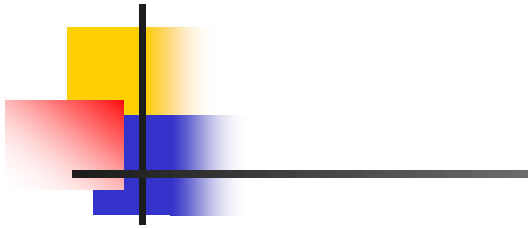
controlla se promettente

prendi

non prendere

ricorri

ricorri

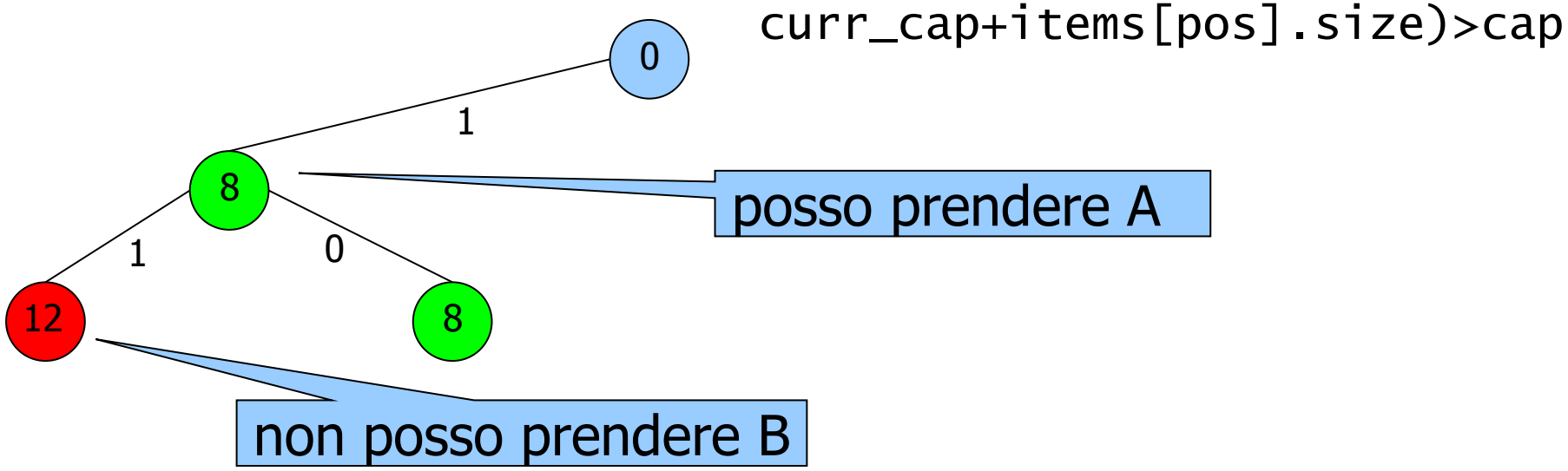


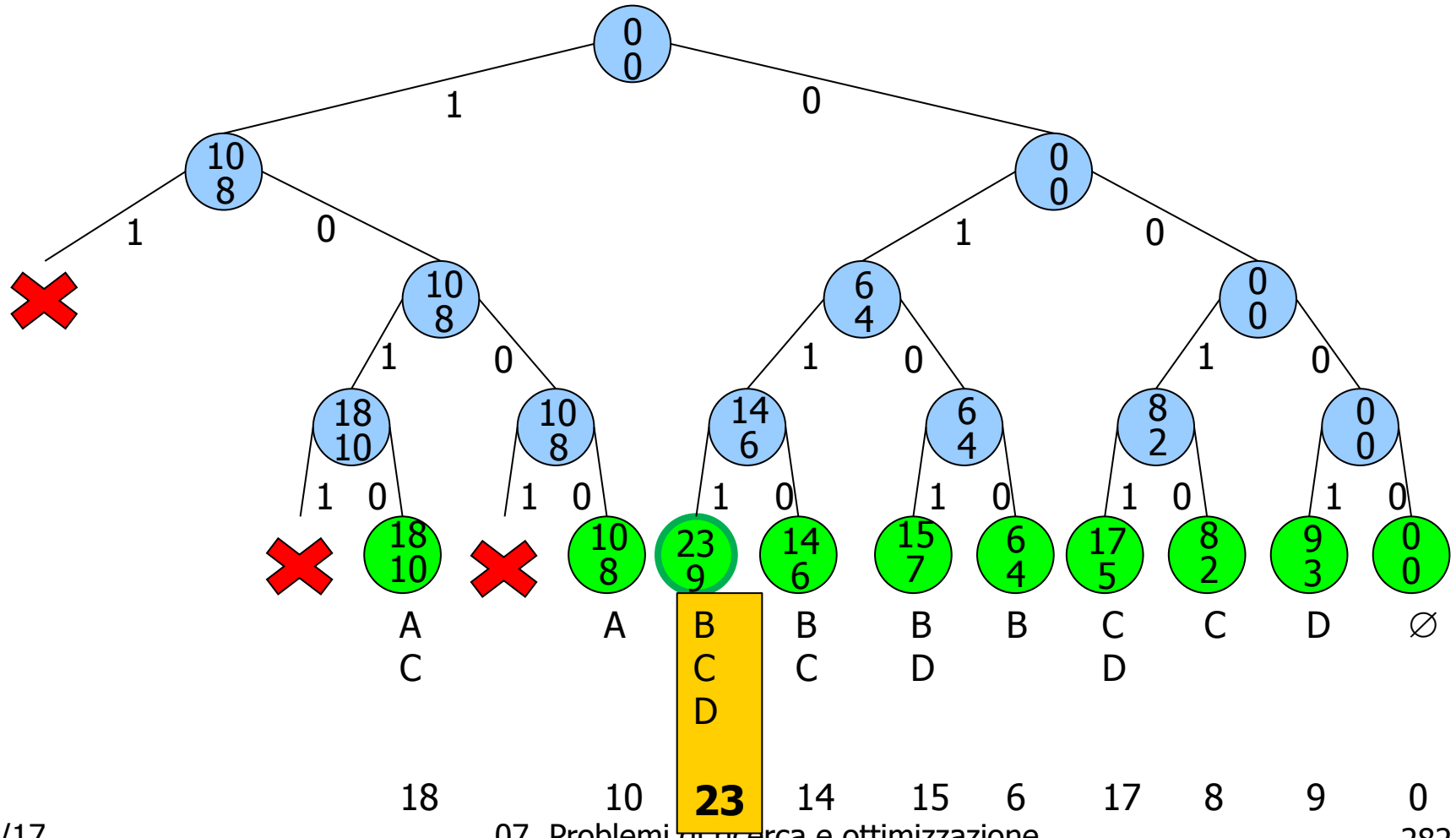
```
int promising(int *val,int pos,int p_sum,int tot,int X) {
    return (p_sum+tot > =X)&&(p_sum+val[pos]<=X);
}
```

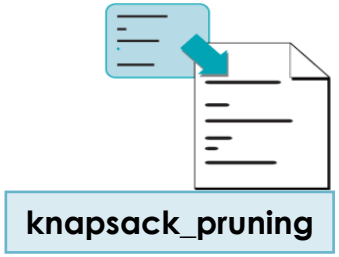
Lo zaino (discreto)

Approccio con pruning: disposizioni ripetute (powerset)

funzione di pruning: se, prendendo un oggetto, la capacità utilizzata eccede quella massima, l'oggetto non viene scelto




0 0
1
0
10 8
0 0
1
0
1
0
10 8
6 4
0 0
1
0
1
0
1
0
18 10
10 8
14 6
6 4
8 2
0 0
1 0
1 0
1 0
1 0
1 0
1 0
18 10
10 8
23 9
14 6
15 7
6 4
17 5
8 2
9 3
0 0
A C
A
B C D
B C
B D
B
C D
C
D
∅
18
10
23
14
15
6
17
8
9
0




```
void powerset(int pos,Item *items,int *sol,int k,int cap,
     int curr_cap,int curr_value,int *bv,int *best_sol) {
  int j;
  if (pos >= k) {
    if (curr_value > *bv) {
      for (j=0; j<k; j++)
        best_sol[j] = sol[j];
      *bv = curr_value;
    }
  }
        return;
}
  if ((curr_cap + items[pos].size) > cap) {
    sol[pos] = 0;
    powerset(pos+1,items,sol,k,cap,curr_cap,curr_value,
             bv,best_sol);
    return;
  }
```

condizione di terminazione

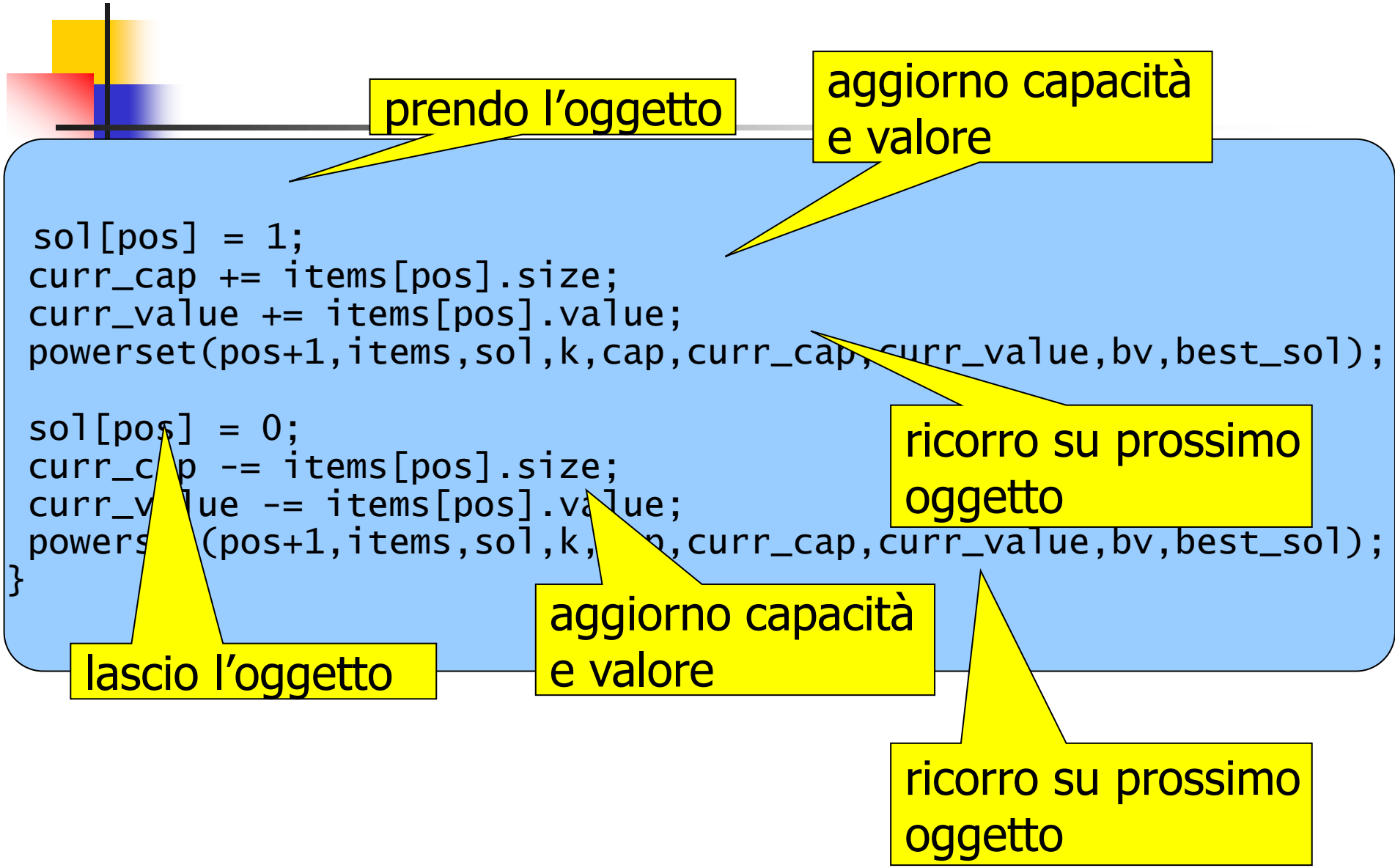
controllo ottimalità

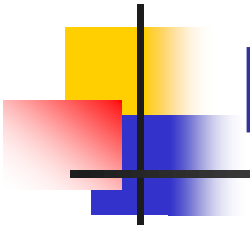
controllo pruning

lascio oggetto

ricorro su prossimo oggetto

ritorno


prendo l'oggetto
aggiorno capacità e valore
sol[pos] = 1;
curr_cap += items[pos].size;
curr_value += items[pos].value;
powerset(pos+1,items,sol,k,cap,curr_cap,curr_value,bv,best_sol);
sol[pos] = 0;
curr_cap -= items[pos].size;
curr_value -= items[pos].value;
powerset(pos+1,items,sol,k,cap,curr_cap,curr_value,bv,best_sol);
}
ricorro su prossimo oggetto
lascio l'oggetto
aggiorno capacità e valore
ricorro su prossimo oggetto



Riferimenti

- Backtracking
  - Bertossi 16
- Permutazioni e sottoinsiemi
  - Bertossi 16.3