



**UNIMORE**  
UNIVERSITÀ DEGLI STUDI DI  
MODENA E REGGIO EMILIA

# Dispense per il Laboratorio di Fondamenti di Informatica II e Lab

Federico Bolelli

## **Esercitazione 03: Backtracking - Parte 1**

Ultimo aggiornamento: 25/03/2021

# Backtracking - SubsetK

Nel file `subset.c` implementare la definizione della funzione:

```
extern int SubsetK(int n, int k);
```

La funzione accetta come parametri due numeri interi positivi, `n` e `k`, e deve stampare su `stdout` tutti i sottoinsiemi di cardinalità `k` dell'insieme contenente i primi `n` numeri naturali a partire da 0. La funzione ritorna quindi il numero di sottoinsiemi possibili.

Ad esempio, se `n = 4` e `k = 2`, la funzione deve stampare tutti i sottoinsiemi di cardinalità 2 dell'insieme `{ 0, 1, 2, 3 }`, ovvero:

`{ 2 3 }`, `{ 1 3 }`, `{ 1 2 }`, `{ 0 3 }`, `{ 0 2 }`, `{ 0 1 }`,

e ritornare 6. Si ricordi che, in un insieme:

- Non possono essere presenti elementi ripetuti;
- Gli elementi non hanno un ordine, quindi `{ 1 2 }` e `{ 2 1 }` sono lo stesso insieme.

Si scriva un opportuno `main()` di prova per testare la funzione.

# Backtracking - SubsetK

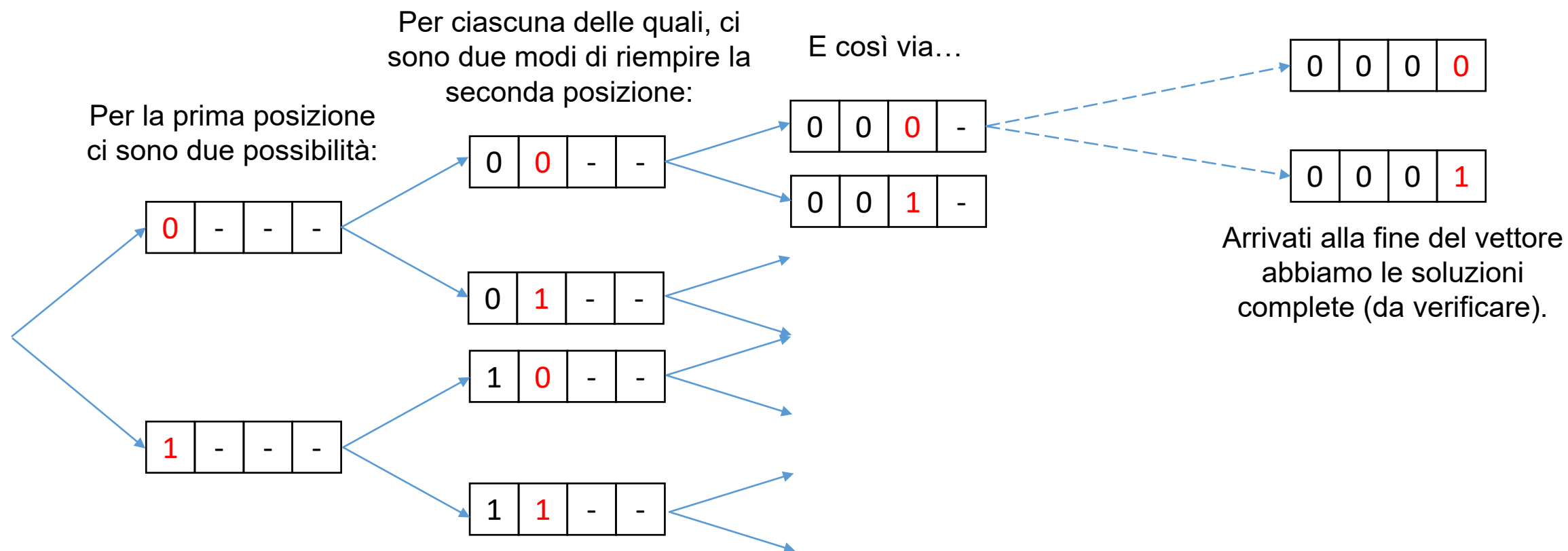
- Come nella maggior parte dei problemi di backtracking, l'obiettivo è quello di costruire passo passo e in maniera incrementale tutte le possibili soluzioni, ovvero tutti i possibili **vettori soluzione**.
- In questo caso specifico, un vettore soluzione deve poter rappresentare un sottoinsieme di  $\{0, \dots, n-1\}$ .
- Il modo più semplice è utilizzare un vettore di **bool** di dimensione  $n$ , che chiamiamo  $v_{curr}$ , vettore soluzione corrente.
- $v_{curr}[i] = 1$  se  $i$  è compreso nel sottoinsieme;
- $v_{curr}[i] = 0$  se  $i$  *non* è compreso nel sottoinsieme.
- Ad esempio per  $n = 4$  e  $k = 2$ 
  - $v_{curr}$ :

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
|---|---|---|---|

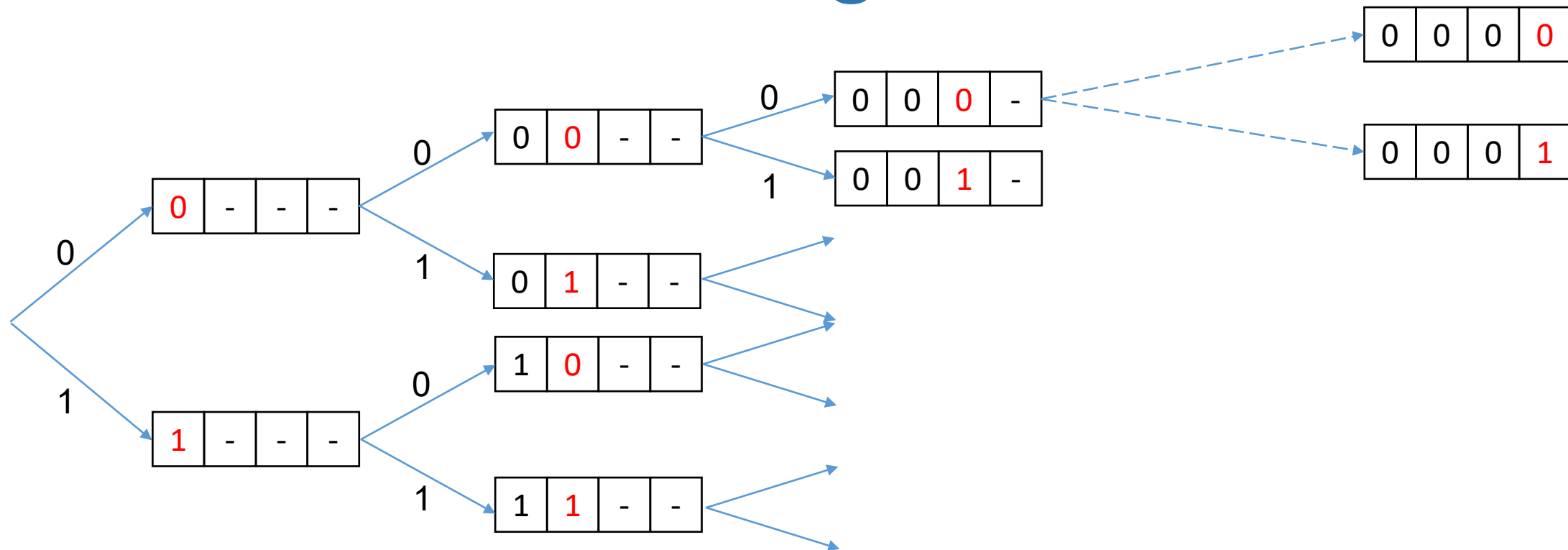
 rappresenta il sottoinsieme  $\{1, 3\}$

# Backtracking - SubsetK

- Tutte le possibili configurazioni di `vcurr`, comprese quelle sbagliate secondo la consegna, costituiscono lo **spazio delle soluzioni**.
- Cerchiamo di esplorare lo spazio delle soluzioni di `SubsetK(4, 2)`.



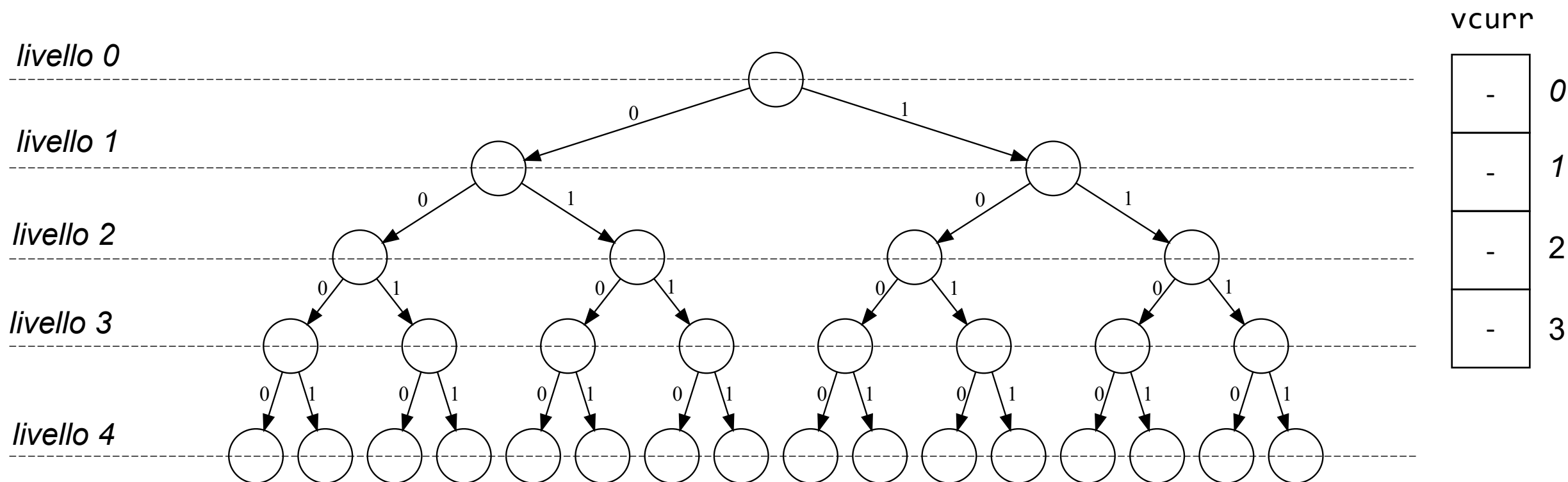
# Backtracking - SubsetK



- Lo spazio delle soluzioni è quindi rappresentato come un **albero**:
  - Ogni **ramo** rappresenta la **scelta** di un valore da inserire in una certa posizione;
  - Ogni **nodo NON foglia** corrisponde ad una **soluzione parziale** al problema;
  - Ogni **nodo foglia** rappresenta una **possibile soluzione** che andrà verificata;

# Backtracking - SubsetK

- Ruotiamo l'albero di 90° per visualizzarlo meglio.
- Ogni livello corrisponde ad una posizione di `vcurr`, ad eccezione dell'ultimo.
- Per ogni livello, tranne l'ultimo, è necessario compiere una scelta.



# Backtracking - SubsetK

- Ogni nodo dell'albero è la radice di un sottoalbero, che è a sua volta un albero.
- Questa proprietà permette di esplorare facilmente un albero tramite una funzione **ricorsiva**: se posso invocarla sulla radice, allora posso invocarla su qualunque nodo, che a sua volta è la radice di un albero più piccolo.

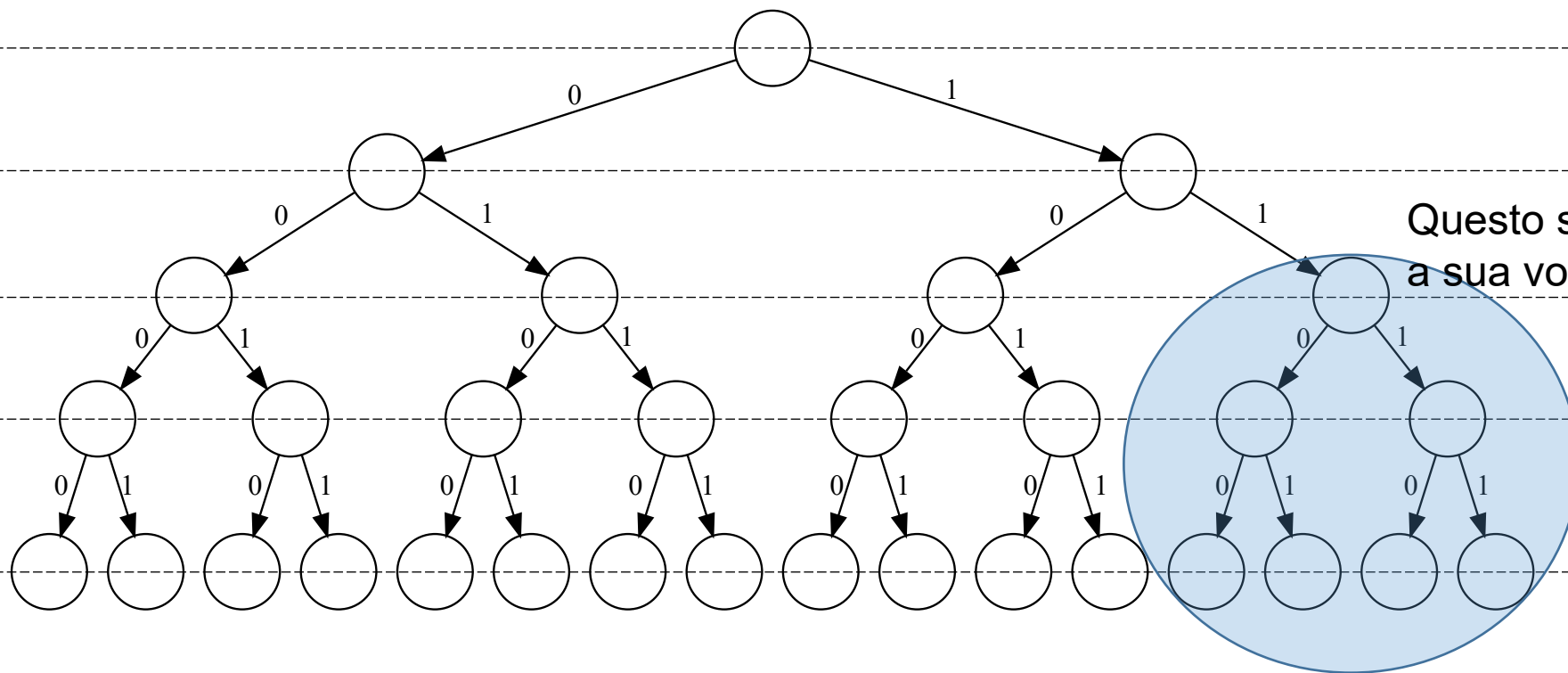
*livello 0*

*livello 1*

*livello 2*

*livello 3*

*livello 4*



# Backtracking - SubsetK

- Passiamo al lato pratico e cerchiamo di implementare una soluzione.
- Ci bastano i parametri della funzione definita dal testo?

# Backtracking - SubsetK

- Evidentemente no!
- Avremo sicuramente bisogno di aggiungere qualcuno, come ad esempio:
  - `vcurr`: vettore soluzione corrente, di lunghezza `n`
  - `i`: livello dell'albero
  - `nsol`: per contare il numero di soluzioni trovare.
- Quindi ricorriamo ad una funzione ausiliaria che possiamo ad esempio chiamare `SubsetKRec()`

# Backtracking - SubsetK

```
#include <stdbool.h>

void SubsetKRec(int n, int k, bool *vcurr, int i, int *nsol) {}

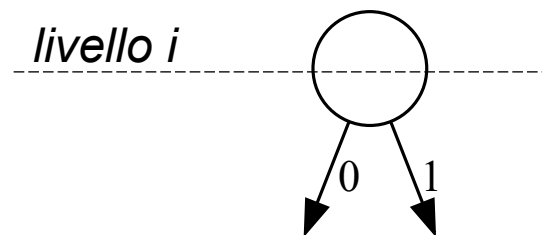
int SubsetK(int n, int k) {
    bool *vcurr = malloc(sizeof(bool)*n);
    int nsol = 0;

    SubsetKRec(n, k, vcurr, 0, &nsol);

    free(vcurr);
    return nsol;
}
```

# Backtracking - SubsetK

- Concentriamoci ora sulla funzione ricorsiva `SubsetKRec()`
- Un'istanza della funzione ricorsiva può essere vista come un nodo dell'albero che rappresenta lo spazio delle soluzioni.
- La funzione ha due possibilità: scegliere o non scegliere il numero  $i$  per costruire la soluzione corrente.
- Fare due scelte significa invocare se stessa ricorsivamente due volte.



# Backtracking - SubsetK

- Invocare ricorsivamente la funzione significa scendere di un livello nell'albero delle soluzioni, quindi non devo dimenticare di incrementare la *i* di 1:

```
static void SubsetKRec(int n, int k, bool *vcurr, int i, int *nsol)
{
    vcurr[i] = 0;
    SubsetKRec(n, k, vcurr, i + 1, nsol);

    vcurr[i] = 1;
    SubsetKRec(n, k, vcurr, i + 1, nsol);
}
```

# Backtracking - SubsetK

- Cosa manca perché la funzione ricorsiva sia ben strutturata e quindi abbia una fine?

# Backtracking - SubsetK

- Esatto: il caso base!
- Il caso in cui il sottoalbero corrente è una foglia!
- Le foglie si trovano al livello  $i=n$  dell'albero.

# Backtracking - SubsetK

```
if (i == n) {  
    int cnt = 0;  
    for (int j = 0; j < n; ++j) {  
        if (vcurr[j]) {  
            cnt++;  
        }  
    }  
  
    if (cnt == k){  
        (*nsol)++;  
        printf("{ ");  
        for (int j = 0; j < n; ++j) {  
            if (vcurr[j]) {  
                printf("%i ", j);  
            }  
        }  
        printf("}, ");  
    }  
  
    return;  
}
```

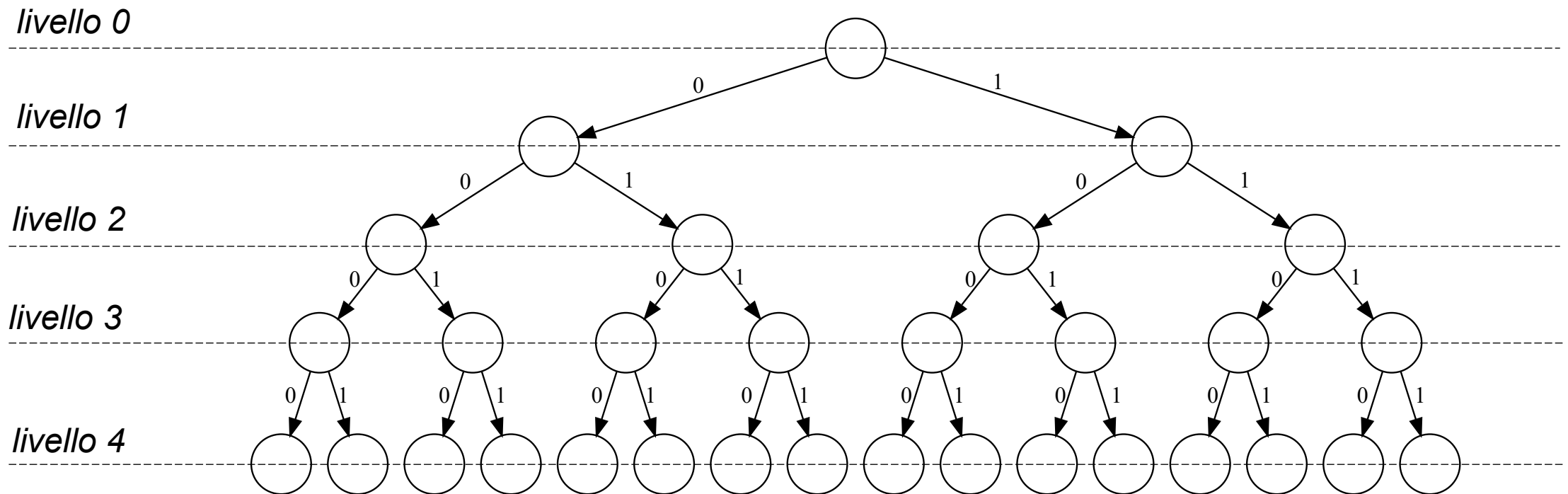
Verifico da quanti elementi è composto l'insieme/soluzione corrente.

Se è formata da esattamente k elementi allora è una soluzione valida e posso stamparla a video, ad esempio usando il formato dell'esempio nel testo dell'esercizio.

Termino la ricorsione.

# Backtracking - SubsetK

- Eseguiamo passo passo le chiamate ricorsive e cerchiamo di seguire l'esecuzione sull'albero delle soluzioni:



# Backtracking - SubsetK

- È una buona idea quella di conteggiare gli elementi a 1 di `vcurr` tutte le volte che incontro il caso base?
- Evidentemente no! Come potrei risolvere?

# Backtracking - SubsetK

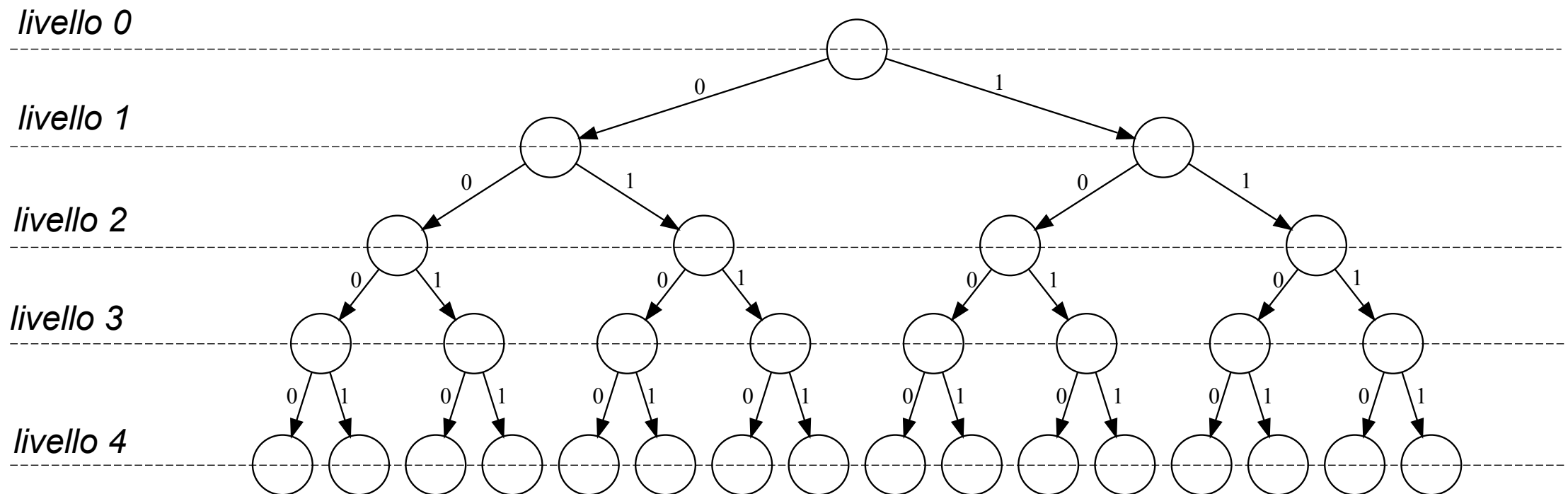
- Potrei ad esempio aggiungere un parametro alla chiamata ricorsiva, così da memorizzare il numero di elementi utilizzati fino ad ora per la costruzione della soluzione corrente.
- Vediamo come ...

# Backtracking - SubsetK

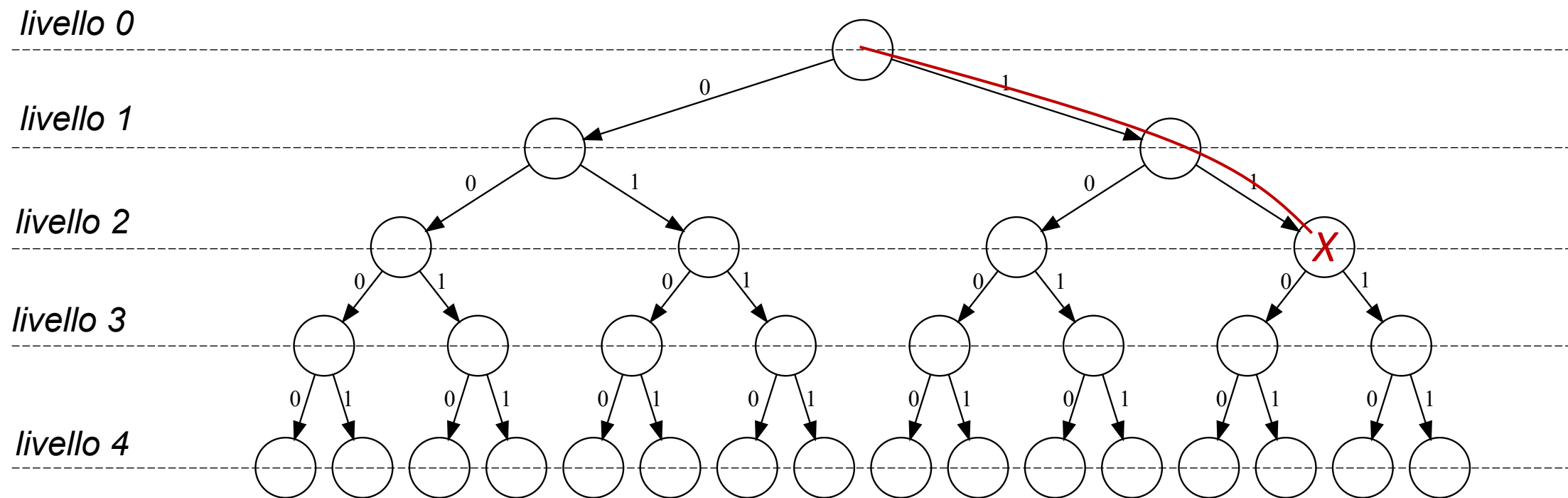
```
static void SubsetKRec(int n, int k, bool *vcurr, int i, int *nsol, int cnt) {  
    if (i == n) {  
        if (cnt == k) {  
            // Stampa della soluzione corrente e incremento di nsol  
        }  
        return;  
    }  
  
    vcurr[i] = 0;  
    SubsetKRec(n, k, vcurr, i + 1, nsol, cnt);  
    vcurr[i] = 1;  
    SubsetKRec(n, k, vcurr, i + 1, nsol, cnt + 1);  
}  
  
int SubsetK(int n, int k) {  
    // [...]  
    SubsetKRec(n, k, vcurr, 0, &nsol, 0);  
    // [...]  
}
```

# Backtracking - SubsetK

- È sempre necessario arrivare ad una foglia per capire se la soluzione corrente è valida o meno?



# Backtracking - SubsetK



*Arrivati al nodo X sappiamo già che questa è una soluzione valida ( $cnt==k$ ) quindi non ha senso proseguire!*

# Backtracking - SubsetK

- Possiamo fare il *pruning* di interi sottoalberi aggiungendo il caso base  $cnt == k$ .
- Ovviamente non sarà più necessario effettuare la stessa verifica dentro il caso base  $i == n$ , che diventerà una semplice terminazione della ricorsione.
- Se ad un certo punto ho costruito un insieme con  $k$  elementi allora ho trovato una soluzione valida e torno indietro (*backtrack*) per provarne altre.
- Se non ci sono più strade disponibili su questo ramo della ricorsione torno indietro (*backtrack*) per provarne altri.
- L'ordine con cui scrivo i casi base è importante.

# Backtracking - SubsetK

```
static void SubsetKRec(int n, int k, bool *vcurr, int i, int *nsol, int cnt) {  
  
    if (cnt == k) {  
        (*nsol)++;  
        // Stampa della soluzione corrente  
    }  
  
    if (i == n) { return; }  
  
    vcurr[i] = 0;  
    SubsetKRec(n, k, vcurr, i + 1, nsol, cnt);  
    vcurr[i] = 1;  
    SubsetKRec(n, k, vcurr, i + 1, nsol, cnt + 1);  
}
```

# Backtracking - SubsetK

- **Attenzione** al contenuto di `vcurr`!
- Quando faccio *pruning*, il vettore soluzione corrente `vcurr` è valido solo sino al livello *i-esimo*
- I valori contenuti ad indici successivi sono relativi a soluzione costruite in chiamate precedenti, quindi non devo considerarle nella stampa della soluzione corrente.
- Ci sono fondamentalmente due strade per evitare questo problema:
  - Riscrivo il ciclo `for` della stampa perché si fermi a `i` e non più ad `n`;
  - Dopo la chiamata ricorsiva che esplora il ramo 1 ripristino lo stato del vettore `vcurr` impostando `vcurr[i]` nuovamente a 0.

# Backtracking - SubsetK

- Riscrivo il ciclo for della stampa perché si fermi a  $i$  e non più ad  $n$ :

```
printf("{ ");  
for (int j = 0; j < i; ++j) {  
    if (vcurr[j]) {  
        printf("%i ", j);  
    }  
}  
printf("}, ");
```

# Backtracking - SubsetK

- Dopo la chiamata ricorsiva che esplora il ramo 1 ripristino lo stato del vettore `vcurr` impostando `vcurr[i]` nuovamente a 0.

```
vcurr[i] = 0;  
SubsetKRec(n, k, vcurr, i + 1, nsol, cnt);  
vcurr[i] = 1;  
SubsetKRec(n, k, vcurr, i + 1, nsol, cnt + 1);  
vcurr[i] = 0;
```

# Facciamo il Punto

Per risolvere un problema attraverso un algoritmo di *backtracking* è sempre opportuno:

1. Definire la lunghezza massima della sequenza che rappresenta una soluzione. **In questo esempio  $n$ ;**
2. Definire il dominio dei valori ammissibili per gli elementi della sequenza risolutiva. **In questo esempio 2, ovvero (1) prendo o (0) non prendo l'elemento  $i$ -esimo;**
3. Definire, per ogni posizione della sequenza risolutiva, le eventuali regole matematiche che la soluzione parziale deve soddisfare. In altre parole, occorre definire i vincoli che sussistono tra gli elementi della sequenza risolutiva. **In questo esempio il vincolo è che nella sequenza non ci siano più di due elementi a 1;**
4. Rappresentare lo spazio delle soluzioni che la funzione deve esplorare per individuare i vettori soluzione.