Esercitazione di Laboratorio: Backtracking Parte 1

Esercizio 1

k amici trascorrono una giornata al mare. Giunti in spiaggia decidono di affittare un ombrellone ciascuno: tutti vogliono affittarne uno in prima fila, senza però essere vicini tra loro. La prima fila, contenente n ombrelloni, è tutta libera.

Nel file `ombrelloni.c` s'implementi la definizione della funzione:

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

La funzione accetta come parametri due numeri interi che rappresentano rispettivamente il numero di amici e il numero di ombrelloni in prima fila.

La funzione deve, utilizzando un algoritmo di *backtracking*, visualizzare su `stdout` tutti i possibili modi in cui è possibile posizionare i k ragazzi nella fila di n ombrelloni rispettando il vincolo di NON adiacenza. Ad esempio, con k=2 e n=4 l'output dovrà essere il seguente:

```
   1) 0 1 0 1↲
   2) 1 0 0 1↲
   3) 1 0 1 0↲
```

Nello specifico occorre stampare ogni soluzione su di una riga separata. Ogni riga è così formata:

1. I primi quattro caratteri sono utilizzati per stampare il numero della soluzione. Se le cifre del numero sono meno di 4 devono essere precedute da caratteri <spazio> di *padding*, affinché il numero e gli eventuali spazi che lo precedono occupino complessivamente 4 caratteri;
2. Seguono i caratteri `)` e <spazio>;
3. Segue la soluzione rappresentata mediante i caratteri `0` e `1`, dove `0` rappresenta "ombrellone vuoto" e `1` "ombrellone occupato". Questi numeri devono essere separati dal carattere <spazio>.

L'ordine con cui vengono stampate le soluzioni non è importante.

Se k<0 o n<0, o se non esistono soluzioni, la funzione non deve stampare nulla e deve ritornare 0. In tutti gli altri casi la funzione ritorna il numero di soluzioni trovate.

Suggerimento

È evidente che, per come definita, la funzione `Ombrelloni()` non ha abbastanza parametri per risolvere il problema mediante *backtracking*. Se ancora non sei sicuro di come definire la funzione ricorsiva puoi utilizzare, ad esempio, la seguente dichiarazione:

```
void OmbrelloniRec(int k, int n, int i, bool *vcurr, int cnt, int *nsol);
```

Dove:

1. k è il numero di ragazzi da posizionare;
2. n è il numero di posti disponibili in prima fila;
3. `i` è la posizione attuale, ovvero il livello nell'albero delle soluzioni;
4. `vcurr` è un array che indica lo stato degli ombrelloni in prima fila (ad esempio 1 = occupato, 0 = libero) e rappresenta la soluzione (eventualmente parziale) corrente. All'inizio tutti gli ombrelloni dovranno essere liberi.

5. `cnt` è un contatore per memorizzare il numero di ragazzi posizionati nella soluzione corrente. A seconda di come viene impostata, la soluzione potrebbe anche ignorare questo parametro;
6. `nsol` è il numero totale di soluzioni trovate.

Cercate di visualizzare lo spazio di ricerca delle soluzioni prima di procedere con l'implementazione:

1. Quante e quali scelte posso fare ad ogni passo della ricorsione? Quanti figli ha/può avere ogni nodo dell'albero?
2. Quali sono le foglie dell'albero? Ovvero, quali sono i casi di terminazione (casi base) della ricorsione?

Esercizio 2

Ogni anno che passa, Babbo Natale fatica sempre più a caricare la slitta dei regali. Per aiutarlo, si implementi nel file `babbonatale.c` la definizione della procedura corrispondente alla seguente dichiarazione:

```
extern void BabboNatale(const int *pacchi, size_t pacchi_size, int p)
```

Data la portata massima della slitta in kg, `p`, e `pacchi_size` regali, ognuno di peso `pacchi[i]`, la procedura deve, utilizzando un algoritmo di *backtracking*, individuare quali regali occorre caricare per massimizzarne il numero totale, senza sforare la portata.

Quindi, la procedura stampa su `stdout` la soluzione trovata, ovvero la sequenza di regali che occorre caricare.

Ad esempio, se `p = 20` e `pacchi = { 10, 11, 1, 3, 3 }` l'output dovrà essere:

```
0 1 1 1 1
```

Dove `0` significa regalo NON caricato e `1` significa regalo caricato. La soluzione dell'esempio prevede quindi di caricare nell'ordine i pacchi di peso 11, 1, 3 e 3. Il formato dell'output dovrà essere uguale a quello di esempio.

Si noti che la soluzione potrebbe non essere unica, ma la procedura deve limitarsi a trovare una tra quelle ottime.

Suggerimento

È evidente che, per come definita, la funzione `BabboNatale()` non ha abbastanza parametri per risolvere il problema mediante *backtracking*. Se ancora non sei sicuro di come definire la funzione ricorsiva puoi utilizzare, ad esempio, la seguente dichiarazione:

```
void BabboNataleRec(const int *pacchi, size_t pacchi_size, int p, int i,
                    bool *vcurr, bool *vbest, int *max, int cnt, int sum)
```

Dove:

- `pacchi` è un vettore contenente i pesi dei regali da caricare;
- `pacchi_size` dimensione del vettore pacchi in numero di elementi;
- `p` è la portata massima della slitta;
- `i` è la posizione attuale. Si noti che questo valore corrisponde al livello dell'albero di backtrack che la funzione corrente sta esplorando;
- `vcurr` è un vettore che indica i regali attualmente caricati nella soluzione corrente (ad esempio `1` = caricato, `0` = NON caricato);
- `vbest` è un vettore che indica i regali caricati nella miglior soluzione fino ad ora trovata (ad esempio `1` = caricato, `0` = NON caricato);

- `max` è il numero di regali caricati nella soluzione `vbest`;
- `cnt` è il numero di regali caricati nella soluzione `vcurr`;
- `sum` è la somma dei pesi dei regali caricati nella soluzione `vcurr`.

Si noti che la funzione di backtracking potrebbe trovare la soluzione anche senza utilizzare i parametri `max`, `cnt` e `sum`.