# Esercitazione di Laboratorio: Backtracking Parte 2

## Esercizio 1

All'interno di un magazzino ci sono `n` cartoni. Ogni cartone possiede un peso in grammi, un'altezza in centimetri e un limite massimo di peso che può sostenere sopra di sé, anch'esso espresso in grammi.

Si creino i file `torrecartoni.h` e `torrecartoni.c` che consentano di utilizzare la struttura:

```
typedef struct {
    unsigned p; // Peso
    unsigned a; // Altezza
    unsigned l; // Limite
} Cartone;
```

e la funzione:

```
extern void TorreCartoni(const Cartone *c, size_t n);
```

Dato un vettore di `n` cartoni, la funzione implementa un algoritmo di *backtracking* per individuare la configurazione che massimizza l'altezza di una pila di cartoni, rispettando il vincolo che nessun cartone abbia sopra di sé un peso superiore al limite consentito.

È possibile che esistano più soluzioni equivalentemente ottime, la funzione deve considerarne una. Non è detto che tutti i cartoni del magazzino possano essere impilati.

La funzione deve stampare su `stdout` la soluzione trovata, utilizzando il formato dell'esempio che segue.

Sia dato il vettore `c = {{ .p=10,.a=20,.l=40 }, { .p=10,.a=10,.l=8 }, { .p=9,.a=3,.l=5 }}`, che rappresenta i pacchi:

| Indice | Peso | Altezza | Limite |
|---|---|---|---|
| 0 | 10 | 20 | 40 |
| 1 | 10 | 10 | 8 |
| 2 | 9 | 3 | 5 |

La funzione deve produrre l'output:

```
1
0
Altezza: 30cm
```

In questo caso la torre migliore ha altezza 30 cm ed è formata da due pacchi: quello di indice 0 alla base e quello di indice 1 in cima.

In pratica, ogni cartone viene rappresentato nell'output da un numero che corrisponde al suo indice nel vettore `c`. Dall'alto verso il basso, il primo indice rappresenta la testa della torre, l'ultimo la base. **L'ordine con cui posiziono i cartoni nella torre è importante!**

**Suggerimento:** costruisci la torre a partire dalla cima!

## Esercizio 2

Giovanni deve percorrere $m$ chilometri in motocicletta. Prima di partire si segna la posizione delle $n$ stazioni di servizio:

$s_0, s_1, \ldots, s_{n-1}$

presenti lungo il percorso. Tali posizioni sono identificate dalle distanze (in chilometri):

$d_0, d_1, \ldots, d_{n-1}$

dove $d_0$ è la distanza dal punto di partenza alla stazione $s_0$, e per $i = 1, \ldots, n-1$, $d_i$ è la distanza fra le stazioni $s_{i-1}$ e $s_i$. Inoltre, per $i = 0, \ldots, n-1$, `p[i]` indica il prezzo (al litro) del carburante nella stazione $s_i$. La motocicletta consuma 0.05 litri per chilometro e ha un serbatoio di 30 litri inizialmente pieno. Giovanni decide di riempire totalmente il serbatoio ogni volta che si ferma in una stazione di servizio.

Nel file `stazioniservizio.c` si implementi la definizione della seguente procedura di backtracking:

```
void StazioniServizio(double m, const double *d, const double *p, size_t n);
```

Dati i km totali da percorrere, `m`, e gli array delle distanze e dei prezzi `d` e `p`, ciascuno contenente `n` elementi, la funzione deve individuare in quali delle stazioni di servizio Giovanni deve fermarsi per minimizzare la spesa per il carburante, pur percorrendo tutti gli `m` km.

Si ignorino i litri di carburante che rimangono nel serbatoio al termine del viaggio.

La funzione deve stampare su `stdout` la soluzione ottima (se esiste), ovvero la sequenza di stazioni in cui occorre fermarsi per spendere il meno possibile e percorrere gli `m` km. Il formato dell'output dovrà essere lo stesso dell'esempio che segue.

Date le stazioni:

```
0: km 260.0000, prezzo 35.0000
1: km 284.0000, prezzo 35.0000
2: km 308.0000, prezzo 33.0000
3: km 332.0000, prezzo 29.0000
4: km 356.0000, prezzo 23.0000
```

e dato `m=1540`, la funzione deve stampare:

```
0 2 3
Spesa totale: 1913.200000 euro
```

Se il problema non ammette soluzione la funzione deve stampare su `stdout` la stringa "Non esistono soluzioni".

## Esercizio 3

Scrivere un programma a linea di comando con la seguente sintassi:

```
tiroallafune <n1> <n2> <n3> ... <nn>
```

Il programma accetta come parametri $n$ numeri interi che rappresentano gli elementi di un insieme $I$. Utilizzando un algoritmo di backtracking, il programma deve dividere $I$ in due sottoinsiemi disgiunti, $A$ e $B$,

rispettivamente di dimensione $n/2$ e $n - n/2$, tali che sia minima la differenza in valore assoluto tra la somma dei valori di $A$ e la somma dei valori di $B$.

Ad esempio, l'insieme `I = {1, 4, 6, -8, 14}` deve essere diviso nei sottoinsiemi `{4, 6}` e `{1, -8, 14}`: le rispettive somme sono `10` e `7`, e la differenza, `3`, è la minore possibile.

Il risultato deve essere stampato su `stdout` secondo la seguente sintassi:

```
{ 4, 6 }, { 1, -8, 14 }
```