



UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dispense del Corso di Laboratorio di
Fondamenti di Informatica II e Lab

Esercitazione 04: Backtracking

Ultimo aggiornamento: 26/03/2019

Backtracking: Torre di Cartoni 1/5

- Esercizio 1 (Torre di Cartoni):

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 espresso in grammi. Si utilizza la seguente struttura:

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

Backtracking: Torre di Cartoni 2/5

Nel file `torrecartoni.c` si implementi la definizione della procedura ricorsiva `TorreCartoni`:

```
void TorreCartoni(int n, cartone *c, int s, torre *cur,  
                 torre *best, bool *usati)
```

Dato un array di n cartoni, la funzione deve individuare la configurazione che massimizzi l'altezza massima di una pila di cartoni, rispettando il vincolo che nessun cartone abbia sopra di sé un peso superiore al limite consentito.

N.B. è possibile che esistano più soluzioni ottime, se ne consideri solo una. Ovviamente, non è detto che tutti i cartoni del magazzino possano essere impilati.

Backtracking: Torre di Cartoni 3/5

La procedura accetta i seguenti parametri:

- `n`: numero di cartoni disponibili (dimensione dell'array `c`);
- `c`: array di cartoni;
- `s`: posizione attuale, ovvero a che livello dell'albero di backtrack si trova la funzione corrente;
- `cur`: torre corrente;
- `best`: torre migliore;
- `usati`: array binario che indica quali cartoni sono già stati utilizzati nella soluzione corrente (ad esempio 1 = usato, 0 = disponibile).
All'inizio dovranno essere tutti disponibili;

N.B. la funzione di backtracking potrebbe trovare la soluzione anche senza utilizzare l'array `usati`.

Backtracking: Torre di Cartoni 4/5

`torre` è una `struct` così definita:

```
typedef struct {  
    unsigned a;  
    int *c_ids;  
    int size;  
} torre;
```

dove `a` è l'altezza della `torre`, `c_ids` è un array che contiene gli indici dei cartoni utilizzati per costruire la torre e `size` la sua dimensione. Ad esempio, se nell'array `c_ids` di una torre ci sono i valori 0, 2, 1 significa che la torre è costruita con i pacchi in posizione 0, 2 e 1 dell'array di cartoni `c`.

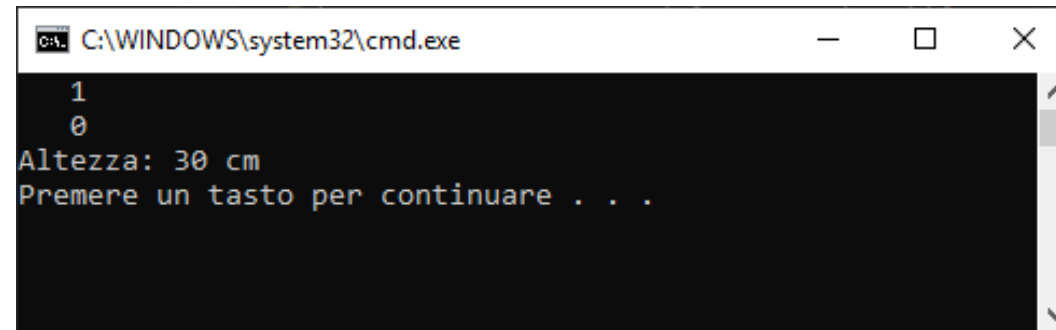
Quale dovrà essere la capacità di `c_ids`?

Backtracking: Torre di Cartoni 5/5

Nel `main()`, chiamare la funzione ricorsiva e mostrare su standard output la soluzione ottima, ovvero la sequenza di cartoni che occorre impilare per ottenere l'altezza massima. Ad esempio, con

$$c = \{ \{ .p=10, .a=20, .l=40 \}, \{ .p=10, .a=10, .l=8 \}, \{ .p=9, .a=3, .l=5 \} \}$$

L'output dovrà essere il seguente:



```

C:\WINDOWS\system32\cmd.exe
1
0
Altezza: 30 cm
Premere un tasto per continuare . . .
  
```

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

Backtracking: Stazioni di Servizio 1/5

- Esercizio 2 (Stazioni di Servizio):

Giovanni deve percorrere m chilometri in motocicletta. Prima di partire si segna la posizione delle n stazioni di servizio s_0, s_1, \dots, s_{n-1} presenti lungo il percorso. Tali posizioni sono identificate dalle distanze (in chilometri) d_0, d_1, \dots, d_{n-1} dove d_0 è la distanza dal punto di partenza alla stazione s_0 , e per $i = 1, \dots, n - 1$, d_i è la distanza fra le stazioni s_{i-1} e s_i . Inoltre, per $i = 0, \dots, n - 1$, $p[i]$ 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.

Backtracking: Stazioni di Servizio 2/5

Nel file `stazioniservizio.c` si implementi la definizione della procedura ricorsiva `StazioniServizio`:

```
void StazioniServizio(double m, int n, double *d, double  
    *p, int s, double dist, piano *cur, piano *best)
```

Dati i km totali da percorrere `m` e gli array delle distanze e dei prezzi `d` e `p`, 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.

Backtracking: Stazioni di Servizio 3/5

La procedura accetta i seguenti parametri:

- `m`: km totali da percorrere;
- `n`: numero delle stazioni lungo il percorso (dimensioni degli array `d` e `p`);
- `d`: array delle distanze tra le stazioni;
- `p`: array dei prezzi dei carburanti nelle varie stazioni;
- `s`: posizione attuale;
- `dist`: distanza percorsa attualmente nella soluzione corrente;
- `cur`: piano attuale;
- `best`: piano migliore;

Backtracking: Stazioni di Servizio 4/5

`piano` è una `struct` così definita:

```
typedef struct {  
    double spesa;  
    int *stazioni;  
} piano;
```

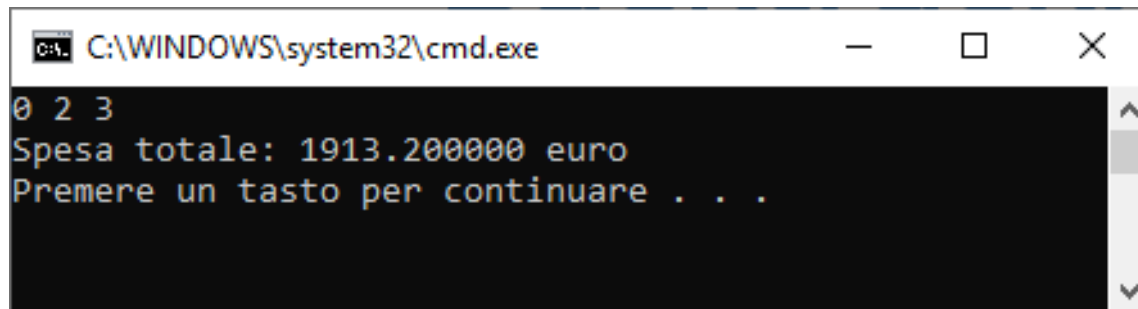
dove `spesa` è il costo totale del piano e `stazioni` è un array binario che contiene la lista delle fermate effettuate (1) e non (0).

Backtracking: Stazioni di Servizio 5/5

Nel `main()`, chiamare la funzione ricorsiva e mostrare su standard output la soluzione ottima (se esiste), ovvero la sequenza di stazioni in cui occorre fermarsi per spendere il meno possibile e percorrere gli `m` km. Date ad esempio le seguenti 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

l'output dovrà essere:



```
C:\WINDOWS\system32\cmd.exe
0 2 3
Spesa totale: 1913.200000 euro
Premere un tasto per continuare . . .
```

Nel caso in cui il problema non ammetta soluzione visualizzare in output la stringa "Non esistono soluzioni".