Esercitazione di Laboratorio del 23/04/2021: Algoritmi Greedy

Esercizio 1

Nel file `monete.c` si implementi la definizione della funzione Monete:

```
extern int *Monete(int *tagli, size_t tagli_size, int budget);
```

La funzione accetta come parametri un array `tagli` di valori che rappresentano i tagli di monete disponibili (ad esempio 50, 20, 10, 5, 2 e 1 centesimo/i), la sua dimensione `tagli_size`, e un budget, `budget`, espresso in centesimi. La funzione deve trovare il numero minimo intero di monete necessarie per formulare il budget, allocare dinamicamente un array delle stesse dimensioni di `tagli` dove memorizzare il quantitativo di monete di ogni tipo necessarie per raggiungere il budget.

La funzione ritorna quindi l'array precedentemente allocato o `NULL` se il budget è minore o uguale a 0. Si utilizzi a tale scopo un algoritmo *greedy* basato sull'opportuna funzione di costo. Si supponga di disporre di infinite monete per ogni taglio e che i tagli disponibili permettano di costruire il budget.

Si assuma che l'array `tagli` preso in input dalla funzione sia ordinato in maniera decrescente.

Ad esempio, con

```
tagli = {50,20,10,5,2,1} e budget = 126
```

L'output della funzione dovrà essere il vettore

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

In questo caso la soluzione greedy ci dice che il numero minimo di monete necessarie per costruire il budget è 5: due pezzi da 50c, uno da 20c, uno da 5c e uno da 1c.

In questo esempio la soluzione greedy corrisponde a quella ottima. Riuscite a trovare un esempio in cui la soluzione dell'algoritmo greedy è peggiore di quella ottima?

Esercizio 2

Si creino i file `gioielli.c` e `gioielli.h` che consentano di utilizzare la `struct`:

```
typedef struct {
    int codice;
    float peso;
    float prezzo;
} Gioiello;
```

e la funzione:

```
extern Gioiello *CompraGioielli(const char *filename, float budget,
                                size_t *ret_size);
```

Dato il nome di un file di testo, `filename`, e un budget, `budget`, la procedura legge da file i gioielli disponibili e seleziona quelli da comprare in modo da massimizzare il peso complessivo dei gioielli acquistati, rispettando il budget. Si utilizzi a tale scopo un algoritmo *greedy*, definendo l'opportuna funzione di costo.

Un gioiello si deve comprare per intero, senza frazionamenti. Un gioiello si può acquistare una sola volta.

Il `codice` identifica il gioiello, `peso` il suo peso in grammi e `prezzo` il suo prezzo di vendita in euro. Il file `filename` memorizza i dati per righe. Ogni riga del file contiene il `codice`, il `peso` e il `prezzo` di un singolo gioiello separati da *whitespace*:

```
<codice><whitespace><peso><whitespace><prezzo>↲
```

Si assuma che il file sia formato correttamente.

La funzione deve ritornare un vettore di `Gioiello` allocato dinamicamente e contenente i gioielli che devono essere acquistati per massimizzare il peso complessivo rispettando il budget, oppure `NULL` se non è possibile aprire il file.

Al termine della funzione, la dimensione del vettore allocato dinamicamente dovrà essere salvata nell'area di memoria puntata da `ret_size`.

Ad esempio, dato un budget di 121€ e il file di gioielli seguente:

```
1 12 100
2 10 21
3 25 120
```

Il vettore ritornato dovrà contenere i gioielli (non necessariamente in questo ordine):

```
1 12 100
2 10 21
```

Come si può notare la soluzione greedy non corrisponde in questo caso all'ottimo assoluto!

Esercizio 3

Siano dati n sciatori di altezza $a_1, a_2, \ldots, a_n$ e n paia di sci di lunghezza $s_1, s_2, \ldots, s_n$. Si vuole assegnare ad ogni sciatore un paio di sci, in modo da minimizzare la differenza totale fra le altezze degli sciatori e la lunghezza degli sci. Quindi, se allo sciatore i-esimo viene assegnato il paio di sci $h(i)$, occorre minimizzare la seguente quantità:

$\sum_{i=1}^{n} |a_i - s_h(i)|$

A tale scopo, si creino i file `sciatori.h` e `sciatori.c` che consentano di definire la seguente struttura:

```
typedef struct {
    double a;
    double l;
} Sciatore;
```

e la funzione:

```
extern Sciatore *Accoppia(double *altezze, double *lunghezze,
                          size_t v_size);
```

La struct consente di rappresentare l'accoppiamento sciatore-sci memorizzando rispettivamente l'altezza dello sciatore `a` e la lunghezza degli sci che gli sono stati assegnati `l`. La funzione accetta come parametri due vettori di `double`, uno di altezze e uno di lunghezze, e la loro dimensione `v_size`. La funzione accoppia le altezze agli sci utilizzando un algoritmo greedy che ad ogni passo deve scegliere la coppia (altezza, lunghezza) con la minima differenza in valore assoluto. Un'altezza deve essere accoppiata con una e una sola lunghezza e viceversa. La funzione ritorna quindi un vettore di `Sciatore` allocato dinamicamente e contenente le coppie nell'ordine in sono state scelte.

Il vettore di altezze e quello di lunghezze hanno sempre la stessa dimensione, indentificata dal parametro `v_size`. Se `v_size` è 0 la funzione deve ritornare `NULL`.