Esame di Laboratorio del 05/07/2023

Note importanti:

- È considerato errore qualsiasi output non richiesto dagli esercizi.
- È consentito utilizzare funzioni ausiliarie per risolvere gli esercizi (in alcuni casi è caldamente consigliato o indispensabile!).
- Quando caricate il codice sul sistema assicuratevi che siano presenti tutte le direttive di include necessarie, comprese quelle per l'utilizzo delle primitive. Non dovete caricare l'implementazione delle primitive.
- È importante sviluppare il codice in Visual Studio (o altri IDE) prima del caricamento sul sistema, così da poter effettuare il debug delle funzioni realizzate!
- Su OLJ non sarà possibile eseguire più di una compilazione/test ogni 3 minuti, per un massimo di 4 compilazioni per esercizio. Il numero di sottomissioni, invece, non è sottoposto a vincoli temporali o quantitativi.

Esercizio 1

I numeri di Leonardo sono una sequenza di numeri data dalla relazione:

$$L(n) = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ L(n-1) + L(n-2) + 1, & n > 1 \end{cases}$$

Essi sono legati ai numeri di Fibonacci dalla relazione $L(n) = 2 * F(n+1) - 1, n \geq 0$

I primi numeri di Leonardo sono:

```
1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, 465, 753, 1219, 1973, 3193, ...
```

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

```
leonardo <n>
```

Il programma prende in input un numero intero positivo `n` e stampa a video i valori della sequenza di Leonardo da $L(0)$ a $L(n)$ inclusi, **calcolati ricorsivamente**.

Se `n < 0` o se il numero di parametri passati al programma è sbagliato, questo termina con codice `1` senza stampare nulla, in tutti gli altri casi il programma termina con codice `0` dopo aver stampato su `stdout`.

Il formato della stampa deve corrispondere aquello degli esempi riportati di seguito.

Non saranno considerate valide soluzioni che non fanno uso della ricorsione per il calcolo dei numeri di Leonardo.

Esempi:

```
comando: leonardo -1
output:
```

```
comando: leonardo 0
output: 1,
```

```
comando: leonardo 5
output: 1, 1, 3, 5, 9, 15,
```

Esercizio 2

Un'azienda vuole sostituire le lampadine ad incandescenza dei lampioni stradali con quelle a LED che sono più efficienti. Poiché le nuove lampadine sono anche più potenti, l'azienda pensa che alcuni lampioni non siano necessari e che potrebbe risparmiare ancora più energia non usandoli.

Modelliamo l'autostrada come una linea retta che misura M metri. Per convenzione, la strada parte al metro 0. L'x-esimo metro è un punto sulla strada che dista x metri dalla "partenza". Se una luce è posiziona al metro x e il LED ha un raggio di illuminazione pari a R metri, la strada sarà illuminata nel segmento da $max(0, x - R)$ a $min(M, x + R)$ inclusi. L'azienda vuole installare i LED in modo tale che ogni punto della strada sia illuminato, ovvero facendo sì che non ci siano zone d'ombra.

I lampioni lasciati senza lampadina non generano alcun tipo di illuminazione.

Nei file `optimal_illumination.c` definire la funzione corrispondente alla seguente dichiarazione:

```c
extern bool* OptimalIllumination(int M, int R, const int *light,
                                 size_t light_size);
```

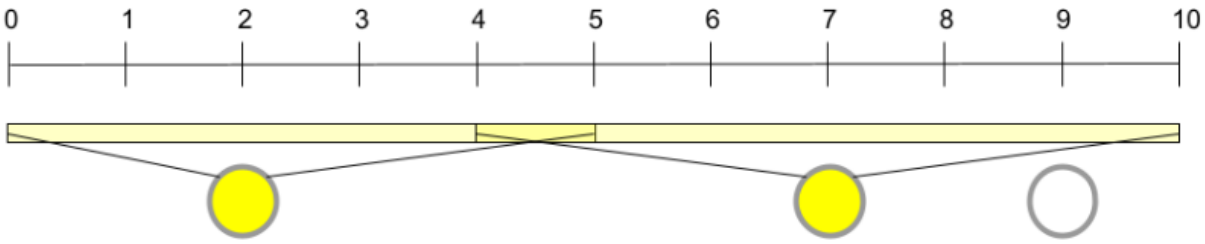
La funzione prende in input la lunghezza dell'autostrada in metri, `M`, il raggio di illuminazione delle nuove lampadine, `R`, e le posizioni di tutti i lampioni memorizzate (in ordine a partire da 0) nel vettore di interi `light` di dimensione `light_size`.

La funzione deve implementare un algoritmo di **backtracking** che trova il numero minimo di lampadine che l'azienda deve installare per illuminare l'intera superstrada. La funzione ritorna la soluzione ottima memorizzata in un vettore di `bool` allocato dinamicamente e avente dimensione `light_size`. In ogni posizione `i` del vettore di ritorno bisongna scrivere `1` se il lampione `light[i]` deve montare la lampadina, `0` altrimenti.

Se non esiste alcuna soluzione valida, la funzione ritorna `NULL`.

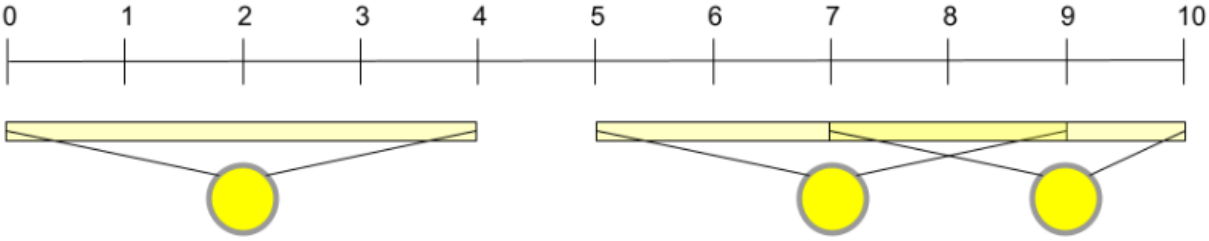
Seguono alcuni esempi:

1. `M = 10, R = 3, light = {2, 7, 9}`



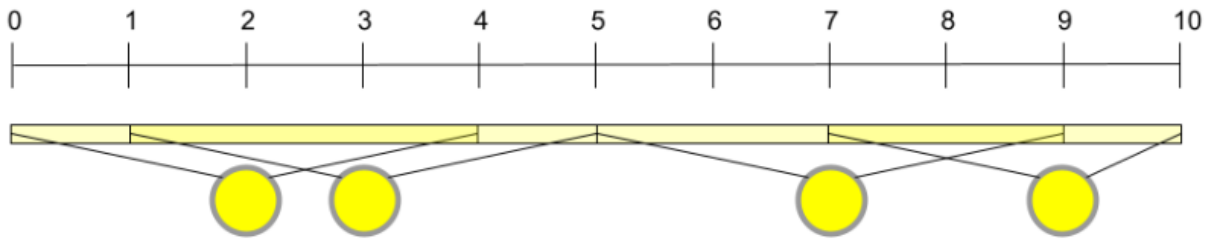
In questo caso è possibile illuminare tutta la strada accendendendo il primo e il secondo lampione, lasciando spento il terzo. Il vettore soluzione ritornato sarà quindi: `{1, 1, 0}`.

2. `M = 10, R = 2, light = {2, 7, 9}`



In questo caso NON è possibile illuminare tutta la strada, nemmeno accendendo tutti i lampioni. La funzione dovrà quindi ritornare NULL.

3. `M = 10, R = 2, light = {2, 3, 7, 9}`



In questo caso è possibile illuminare tutta la strada solamente accendendendo tutti i lampioni. Il vettore soluzione ritornato sarà quindi: `{1, 1, 1, 1}`.

Non saranno considerate valide sottomissioni che non usano il backtracking per individuare la soluzione ottima.

Esercizio 3

Una pulce salterina può saltare all'avanti o all'indietro in un sentiero. I salti della pulce sono codificati in una lista di interi doppiamente concatenata. Data la posizione iniziale della pulce (`Item` della lista) occorre determinare dove questa si fermarà (`Item` della lista). La codifica dei movimenti della pulce è la seguente:

- Se nella posizione corrente `curr` c'è un numero positivo, la pulce si muove all'avanti di tanti `Item` quanto specificato dal numero (`curr.value`);
- Se in `curr` c'è un numero negativo, la pulce si muove all'indietro di `abs(curr.value)` posizioni;
- Se nel movimento (all'avanti o all'indietro) la pulce raggiunge la fine del sentiero (la lista finisce), il movimento viene completato invertendo la direzione di moto. Ad esempio, se la pulce deve fare quattro salti all'avanti, ma nella lista c'è un solo `Item` rimanente, questa farà un salto all'avanti e tre all'indietro;
- Se `curr.value` è 0, la pulce non fa salti e il movimento termina;
- In ogni caso, la pulce non può fare più di `n` salti.

Nei file `pulce.h` e `pulce.c` si implementi la definizione della seguente funzione:

```
extern const Item* CalcolaPercorso(const Item *start, size_t n);
```

La funzione prendere in input l'indirizzo di un elemento di una lista doppiamente concatenata, `start`, che rappresenta il punto di partenza della pulce, e un intero senza segno, `n`, che rappresenta il numero massimo di salti che questa può fare. Usando la codifica sopra riportata, la funzione calcola il percorso della pulce e ritorna l'indirizzo dell'`Item` di destinazione.

La destinazione potrà essere un `Item` contenente il valore 0 o quello in cui si trova la pulce dopo aver effettuato esattamente `n` salti.

Se la lista di input è `NULL`, la funzione ritorna `NULL`.

Ad esempio, se alla funzione `CalcolaPercorso` passiamo l'indirizzo dell'elemento di valore 4 nella lista `[2, -3, 4, 1, 0, 5, -12, 3]` e n=13 la funzione restituirà l'indirizzo dell'elemento di valore 1. Infatti:

- La pulce farà 4 salti in avanti e si fermerà sul -12;
- Inizierà a saltare all'indietro fino ad arrivare all'inizio della lista (elemento di valore 2), quindi proseguirà all'avanti per completare i 12 salti, ma si fermerà sull'elemento di valore 1 prima di finire. Infatti, il numero massimo di salti consentito è 13.

Per la risoluzione di questo esercizio avete a disposizione le seguenti definizioni:

```c
typedef int ElemType;

struct Item {
    ElemType value;
    struct Item *next;
    struct Item *prev;
};
typedef struct Item Item;
```

e le seguenti funzioni primitive e non:

```c
ElemType ElemCopy(const ElemType *e);
void ElemSwap (ElemType *e1, ElemType *e2)
void ElemDelete(ElemType *e);
void ElemWrite(const ElemType *e, FILE *f);
void ElemWriteStdout(const ElemType *e);

Item *DListCreateEmpty(void);
Item *DListInsertHead(const ElemType *e, Item* i);
bool DListIsEmpty(const Item *i);
const ElemType *DListGetHeadValue(const Item *i);
Item *DListGetTail(const Item *i);
Item* DListGetPrev(const Item* i);
Item *DListInsertBack(Item *i, const ElemType *e);
void DListDelete(Item *item);
void DListWrite(const Item *i, FILE *f);
void DListWriteStdout(const Item *i);
```

Trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file `elemtype.h`, `elemtype.c`, `doublelist.h` e `doublelist.c` scaricabili da OLJ, così come la loro documentazione.

Esercizio 4

Nel file `colored_trees.c` definire la funzione corrispondente alla seguente dichiarazione:

```c
extern const Node* BiggestColoredTree(const Node *t);
```

La funzione prende in input un albero binario di `char`, `t`, e ritorna il puntatore al sottoalbero più grande (ovvero quello con il maggior numero di nodi) contenete nodi **tutti** dello stesso colore.

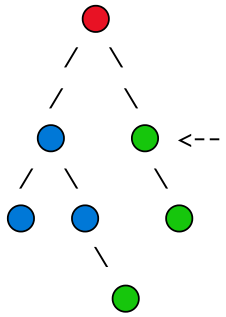
La proprietà "colore" di ogni nodo è memorizzata all'interno del campo `value`.

Se ci sono due o più sottoalberi che rispettano le proprietà sopra descritte, ovvero contenti nodi tutti dello stesso colore e aventi dimensione massima, la funzione ritorna l'albero di altezza massima. Se due o più soluzioni ottime sono sottoalberi della stessa altezza, la funzione ritorna quello che si trova "più a sinistra".

Se l'albero di input è vuoto, la funzione ritorna `NULL`.

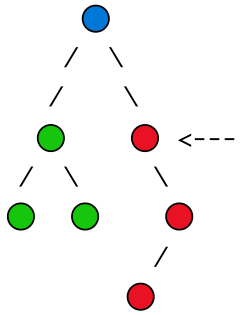
Attenzione: non occorre fare alcuna copia, la funzione ritorna il puntatore al nodo che rappresenta il sottoalbero individuato.

Negli esempi sotto riportati, il valore di `value` sarà rispettivamente `'r'` per il rosso, `'b'` per il blu e `'v'` per il verde. Dato ad esempio l'albero:



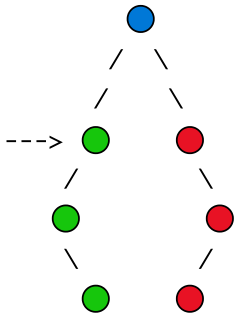
Il nodo che la funzione deve ritornare è quello puntato dalla freccia.

Dato invece il seguente albero:



Il nodo che la funzione deve ritornare è quello puntato dalla freccia. Infatti, il sottoalbero verde contiene lo stesso numero di nodi, ma è più basso di quello rosso.

O ancora, dato l'albero:



Il nodo che la funzione deve ritornare è quello rappresentante il sottoalbero di colore verde, in quanto anche se equivalente al rosso si trova "più a sinistra".

Per la risoluzione di questo esercizio avete a disposizione le seguenti definizioni:

```c
typedef char ElemType;

struct Node {
    ElemType value;
    struct Node *left;
    struct Node *right;
};
typedef struct Node Node;
```

e le seguenti funzioni primitive e non:

```
int ElemCompare(const ElemType *e1, const ElemType *e2);
ElemType ElemCopy(const ElemType *e);
void ElemDelete(ElemType *e);
int ElemRead(FILE *f, ElemType *e);
int ElemReadStdin(ElemType *e);
void ElemWrite(const ElemType *e, FILE *f);
void ElemWriteStdout(const ElemType *e);

Node *TreeCreateEmpty(void);
Node *TreeCreateRoot(const ElemType *e, Node *l, Node *r);
bool TreeIsEmpty(const Node *n);
const ElemType *TreeGetRootValue(const Node *n);
Node *TreeLeft(const Node *n);
Node *TreeRight(const Node *n);
bool TreeIsLeaf(const Node *n);
void TreeDelete(Node *n);

void TreeWritePreOrder(const Node *n, FILE *f);
void TreeWriteStdoutPreOrder(const Node *n);
void TreeWriteInOrder(const Node *n, FILE *f);
void TreeWriteStdoutInOrder(const Node *n);
void TreeWritePostOrder(const Node *n, FILE *f);
void TreeWriteStdoutPostOrder(const Node *n);
```

Trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file `elemtype.h`, `elemtype.c`, `tree.h` e `tree.c` scaricabili da OLJ, così come la loro documentazione.

Esercizio 5

Si hanno a disposizione N segmenti di corda e li si vogliono unire per formarne uno unico. L'unione di due segmenti ha un costo pari alla somma delle loro lunghezze.

Nel file `connect_ropes.c` definire la funzione corrispondente alla seguente dichiarazione:

```
extern size_t ConnectRopes(Heap *ropes);
```

La funzione prende in input un *min-heap* contenente le lunghezze di segmenti di corda che deve unire per formare una corda unica.

I segmenti devono essere uniti minimizzando il costo complessivo della procedura, ovvero sommando iterativamente i due segmenti di corda più corti fino a quando non ne rimane uno solo. La funzione ritora quindi il costo di creazione della corda.

L'implementazione deve sfruttare le proprietà heap e avere un costo non superiore a $O(n * \log n)$.

La funzione può modificare l'heap di input. Se l'heap è vuota la funzione ritorna 0.

Per la risoluzione di questo esercizio avete a disposizione le seguenti definizioni:

```
typedef int ElemType;

struct Heap{
    ElemType *data;
    size_t size;
```

```
};
typedef struct Heap Heap;
```

e le seguenti funzioni primitive e non:

```
int ElemCompare(const ElemType *e1, const ElemType *e2);
ElemType ElemCopy(const ElemType *e);
void ElemDelete(ElemType *e);
void ElemSwap(ElemType *e1, ElemType *e2);
int ReadElem(FILE *f, ElemType *e);
int ReadStdinElem(ElemType *e);
void WriteElem(const ElemType *e, FILE *f);
void WriteStdoutElem(const ElemType *e);

int LeftHeap(int i);
int RightHeap(int i);
int ParentHeap(int i);
Heap* CreateEmptyHeap();
void InsertNodeMinHeap(Heap *h, const ElemType *e);
bool IsEmptyHeap(const Heap *h);
ElemType* GetNodeValueHeap(const Heap *h, int i);
void MoveUpMinHeap(Heap *h, int i);
void MoveDownMinHeap(Heap *h, int i);
void DeleteHeap(Heap *h);
void WriteHeap(const Heap *h, FILE *f);
void WriteStdoutHeap(const Heap *i);
```

Trovate le definizioni, le dichiarazioni e le rispettive implementazioni nei file `elemtype.h`, `elemtype.c`, `maxheap.h` e `maxheap.c` scaricabili da OLJ, così come la loro documentazione.