Esame di Laboratorio del 19/06/2023

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

Un numero felice è definito tramite il seguente processo: partendo con un qualsiasi numero intero positivo, si sostituisca il numero con la somma dei quadrati delle sue cifre, e si ripete il processo fino a quando si ottiene `1`, oppure si entra in un ciclo che non include mai `1`. I numeri per cui tale processo dà `1` sono numeri *felici*, mentre quelli che non danno mai `1` sono numeri *infelici*. È possibile dimostrare che, se nella sequenza si raggiunge il `4`, il numero è infelice. Inoltre, la sequenza di ogni numero infelice raggiungerà sicuramente il numero `4`. Possiamo estendere il concetto allo `0`, che ovviamente genera la sequenza composta solo di `0` e quindi possiamo considerarlo infelice.

Si prenda per esempio 103, le operazioni da eseguire per verificare se è felice sono le seguenti:

$$
\begin{aligned}
1^2 + 0^2 + 3^2 &= 10 \\
1^2 + 0^2 &= 1
\end{aligned}
$$

Una volta raggiunto il valore 1, possiamo affermare che il numero 103 è *felice*. Diversamente, il numero 16 è infelice in quanto la procedura di sostituzione incontra un ciclo che non contiene il valore 1 (e contiene il 4):

$$
\begin{aligned}
1^2 + 6^2 &= 37 \\
&\dots \\
4^2 + 2^2 &= 20 \\
2^2 + 0^2 &= 4 \\
4^2 &= 16 \\
1^2 + 6^2 &= 37 \\
&\dots
\end{aligned}
$$

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

```
ishappy <n>
```

Il programma prende in input un numero intero positivo, `n`, e **verifica ricorsivamente** se questo è felice. Il programma stampa su `stdout` l'esito della verifica utilizzando la stringa `"Felice"` o `"Infelice"`.

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`.

Non saranno considerate valide soluzioni che non fanno uso della ricorsione per verificare se il numero è felice.

Esercizio 2

Dato un vettore di `n` elementi, `v`, si definisce sottovettore di `v` ogni vettore `vsub` di lunghezza arbitraria (minimo 0, massimo `n`) contenente *elementi contigui* di `v`. Tra tutti i `vsub`, si definisce *sottovettore ottimo* o *sottovettore di somma massima* quello avente elementi di somma massima.

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

```
extern int OptimalSubarray(const int *v, size_t n);
```

La funzione prende in input un vettore di numeri interi, `v`, contenente `n` elementi, e implementa un algoritmo di backtracking che trova il sottovettore ottimo di `v` e ritorna la somma dei suoi elementi. Il sottovettore di 0 elementi ha somma 0.

Se `n = 0`, la funzione ritorna 0.

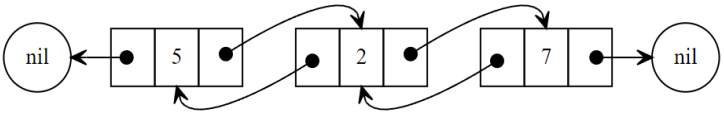
Seguono alcuni esempi:

1. Se `v = { 1, 2, 3, 4, 5, 6 }`, la funzione deve tornare `21`, infatti il sottovettore ottimo contiene tutti gli elementi di `v`;
2. Se `v = { 1, 2, 1, -100, 5, 6 }`, la funzione deve tornare `11`, infatti il sottovettore ottimo contiene solo gli ultimi due elementi di `v`;
3. Se `v = { -1, -2, -3, -4, -5, -6 }`, la funzione deve tornare `0`, infatti il sottovettore ottimo non contiene elementi di `v`.

Esercizio 3

Una lista di interi doppiamente concatenata può essere utilizzata per rappresentare un numero intero non negativo in base 10 *grande a piacere*: ciascuno degli elementi della lista contiene una cifra del numero. Le cifre sono memorizzate nella lista a partire da quella più significativa.

Utilizzando questa convenzione, il numero 527 viene rappresentato mediante la lista:



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

```
extern Item* DListSum(const Item *i1, const Item *i2);
```

La funzione prendere in input due liste di interi doppiamente concatenate, `i1` e `i2`, che rappresentano rispettivamente i numeri interi non negativi in base 10 `n1` e `n2`. La funzione deve ritornare una *nuova lista*, anch'essa doppiamente concatenata, contenente il risultato della somma `n1 + n2`.

Nel caso alla funzione vengano passate le liste `[9, 5, 4]` e `[6, 9]`, la lista risultante ritornata dalla funzione deve essere `[1, 0, 2, 3]`, in quanto $954 + 69 = 1023$.

Una lista vuota corrisponde al numero `0`, quindi se una delle due liste di input è vuota la funzione deve ritornare una copia dell'altra, se entrambe le liste di input sono vuote la funzione deve ritornare una lista vuota.

Attenzione: la lista potrebbe rappresentare un numero veramente grande, non potete pertanto utilizzare alcun tipo di dato standard per rappresentare `n1`, `n2` e `n1 + n2`.

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 `mergetree.c` definire la funzione corrispondente alla seguente dichiarazione:

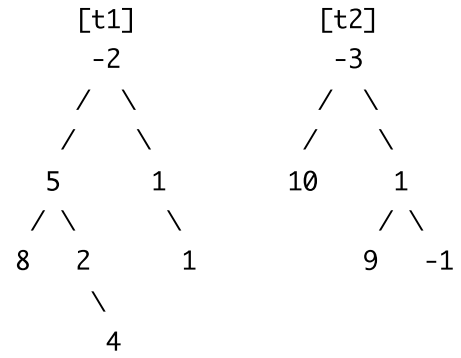
```c
extern Node* MergeTree(const Node *t1, const Node* t2);
```

La funzione prende in input due alberi binari `t1` e `t2` e ritorna un *nuovo* albero rappresentante il *merge* tra `t1` e `t2`. Il *merge* di due alberi binari può essere definito come la somma degli elementi che, rispetto alla radice, si trovano nelle stesse posizioni. Nel caso in cui solo uno dei due alberi ha un nodo in una certa posizione, il nodo (e l'eventuale sottoalbero che questo rappresenta) viene semplicemente copiato.

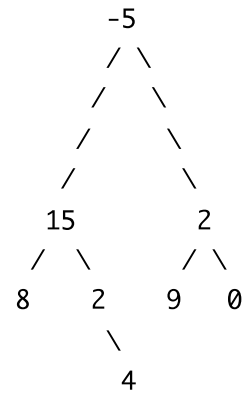
In altre parole, la funzione `MergeTree` deve eseguire le seguenti operazioni:

- Se entrambi i puntatori `t1` e `t2` sono `NULL`, restituisce `NULL` perché non c'è alcun nodo da unire;
- Se solo `t1` è `NULL`, crea una copia di `t2` e la restituisce;
- Se solo `t2` è `NULL`, crea una copia di `t1` e la restituisce;
- Se entrambi i puntatori sono non `NULL`, crea un nuovo nodo con valore pari alla somma dei valori dei nodi corrispondenti nei due alberi (`t1` e `t2`), quindi invoca ricorsivamente la funzione `MergeTree` sui sottoalberi sinistro e destro di `t1` e `t2`.

Dati ad esempio:



Il risultato del *merge* è il seguente:



La funzione `MergeTree()` deve restituire un puntatore al nodo radice del **nuovo** albero binario che rappresenta il merge dei due alberi. Si faccia attenzione al fatto che gli alberi passati potrebbero non avere lo stesso numero di livelli.

Se entrambi gli alberi passati alla funzione sono `NULL`, deve essere ritornato `NULL`.

Suggerimento: per semplificare la risoluzione dell'esercizio si suggerisce di utilizzare la funzione ausiliaria `TreeCopy()` che dato un albero binario ne restituisce una copia. Di seguito è riportata una sua possibile implementazione (che potete copiare, incollare e usare nella vostra soluzione):

```
Node *TreeCopy(const Node *t) {
    if (TreeIsEmpty(t)) {
        return NULL;
    }

    Node* n = TreeCreateRoot(TreeGetRootValue(t), NULL, NULL);
    n->left = CopyTree(TreeLeft(t));
    n->right = CopyTree(TreeRight(t));

    return n;
}
```

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

```
typedef int 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

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

```
extern void CircleSort(int *v, size_t v_size);
```

La procedura prende in input un vettore di interi, `v`, e la sua dimensione, `v_size`. La procedura ordina *in place* e in senso crescente gli elementi del vettore utilizzando l'algoritmo *Circle Sort*.

L'algoritmo può essere visualizzato disegnando cerchi concentrici su un vettore di numeri interi (da qui il nome). Gli elementi del vettore che giacciono sullo stesso cerchio e che sono diametralmente opposti tra loro vengono confrontati e se trovati nell'ordine sbagliato (quello a sinistra è più grande di quello a destra) vengono scambiati.

Il processo di cui sopra viene ripetuto in modo ricorsivo sui sottovettori ottenuti dividendo un due parti il vettore di partenza. La ricorsione termina quando il sottovettore è formato da un solo elemento.

Quanto descritto sinora rappresenta una singola iterazione dell'algoritmo. Se durante un'iterazione non vengono effettuati scambi l'algoritmo termina, altrimenti si esegue una nuova iterazione.

In altre parole, l'algoritmo prevede tre passaggi:

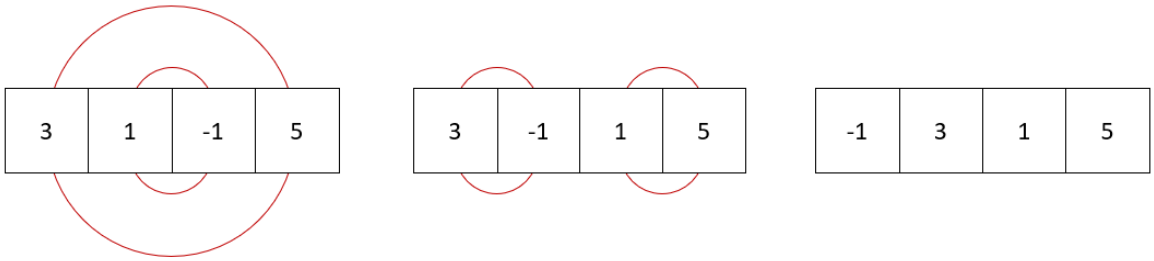
1. Si confronta il primo elemento con l'ultimo elemento, quindi il secondo elemento con il penultimo elemento, ecc. Ad ogni confronto vengono effettuati gli scambi necessari;
2. Ricorsivamente, il vettore viene "diviso" in due parti e su ognuna di esse vengono applicati i punti 1 e 2. Il processo ricorsivo termina quando il sottovettore è formato da un solo elemento.
3. Se nei punti precedenti è stato effettuato almeno uno scambio, questi vanno ripetuti.

Nel caso in cui `v_size` (o la dimensione di uno dei sottovettori) sia dispari:

1. Al termine del punto 1 è necessario aggiungere il confronto e l'eventuale swap tra `v[v_size/2]` e `v[v_size/2 + 1]`;
2. Lo split del punto 2 produrrà due sottovettori di dimensione differente, rispettivamente `v_size/2` e `v_size/2 + 1`.

Si consideri ad esempio il vettore `{3, 1, -1, 5}`, i confronti e gli eventuali scambi applicati dall'algoritmo *circle sort* sono i seguenti (nella figura avete una rappresentazione visiva della prima iterazione, punti 1 e 2):

1. Viene confrontato 3 con 5 e 1 con -1. Il secondo confronto genera uno scambio, il primo no. Si ottiene quindi il vettore `{3, -1, 1, 5}`
2. Il vettore `{3, -1, 1, 5}` viene visto come due sottovettori `{3, -1}, {1, 5}` su cui viene applicata la procedura di scambio circolare. La prima coppia genererà uno scambio, la seconda no.
3. I sottovettori `{-1, 3}, {1, 5}` vengono ulteriormente divisi in `{-1}, {3}, {1}` e `{5}`. Essendo elementi singoli la ricorsione termina e con lei la prima iterazione dell'algoritmo.
4. Nell'iterazione precedente è stato generato almeno uno scambio, quindi devo ripetere l'intero procedimento sul vettore `{-1, 3, 1, 5}`. Viene confrontato -1 con 5 e 3 con 1. Solo il secondo confronto genera uno scambio producendo `{-1, 1, 3, 5}`.
5. Si considerano i sottovettori `{-1, 1}` e `{3, 5}`. Entrambi sono già ordinati e nessuno scambio viene effettuato.
6. I sottovettori `{-1, 1}` e `{3, 5}` vengono ulteriormente divisi in `{-1}, {1}, {3}` e `{5}`. Essendo elementi singoli la ricorsione termina e con lei la seconda iterazione dell'algoritmo.
7. Anche la seconda iterazione ha prodotto una scambio, quindi occorrerà eseguirne una terza e ultima (il vettore è ordinato).



N.B. La divisione di un vettore in sottovettori è "virtuale", non vengono effettivamente creati nuovi vettori.