# Teoria

**1.** Dato il seguente array bidimensionale:

```
double M[3][4];
```

Scrivi un'espressione che, usando l'aritmetica dei puntatori, acceda all'elemento `M[1][2]`.

**2.** Data la seguente funzione ricorsiva:

```
int g(int n) {
  if (n <= 2) return 1;
  return g(n-1) + g(n-3);
}
```

- Qual è il parametro su cui viene fatta la ricorsione?
- Qual è la misura di complessità del problema?
- Spiegare perché questa misura decresce ad ogni chiamata ricorsiva.
- Determinare il fattore minimo di decrescita della misura di complessità ad ogni chiamata ricorsiva.

**3.** Assumendo che un int occupi 4 byte, cosa stampa il seguente codice?

```
int mat[2][3] = {{1,2,3}, {4,5,6}};
printf("%d", *(*(mat+1)+1));
```

**4.** Cosa stamperanno le seguenti istruzioni? Spiegare perché.

```
printf("%d\n", ***r);
printf("%p\n", **r);
printf("%p\n", *q);
```

Se dopo le dichiarazioni iniziali eseguiamo le seguenti istruzioni:

```
int y = 20;
*p = y;
```

Come cambierà la rappresentazione in memoria? Cosa stamperà ora `printf("%d\n", x);`?

**5.** Analizzare il seguente codice:

```
int* createDanglingPointer() {
    int local = 42;
    return &local;
}
```

```
int main() {
    int *ptr = createDanglingPointer();
    printf("%d\n", *ptr);
    return 0;
}
```

a) Spiegare perché questo codice crea un dangling pointer.

b) Quali sono i potenziali problemi che possono derivare dall'uso di questo dangling pointer?

c) Come si potrebbe modificare la funzione `createDanglingPointer()` per evitare il dangling pointer?

**6.** Considerare il seguente codice:

```
int *p, *q;
p = (int*)malloc(sizeof(int));
*p = 10;
q = p;
free(p);
*q = 20;
```

a) Dopo l'esecuzione di free(p), cosa rappresenta q?

b) Quali sono i rischi associati all'uso di q dopo la chiamata a free(p)?

c) Come si potrebbe modificare questo codice per evitare potenziali problemi di memoria?

**7.** Dato il seguente codice:

```
int matrix[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
int *ptr = &matrix[1][2];
printf("%d", *(ptr+2));
```

Cosa verrà stampato?

a) 7

b) 8

c) 9

d) 10

# Esercizi

**1.** Implementare una funzione ricorsiva che inverta una lista concatenata. La funzione deve avere la seguente firma:

```
struct Node* reverse_list(struct Node* head);
```

Dove Node è definito come:

```
struct Node {
  int data;
  struct Node* next;
};
```

Esempio:

Input: 1 -> 2 -> 3 -> 4 -> NULL

Output: 4 -> 3 -> 2 -> 1 -> NULL

Implementare la funzione in modo efficiente, discutendo la strategia utilizzata e analizzando la complessità computazionale.

**2.** Implementare una funzione che trovi il più lungo cammino in un albero binario. La funzione deve avere la seguente firma:

```
int longest_path(BST* root);
```

Dove BST è definito come:

```
typedef struct BST {
  int value;
  struct BST* left;
  struct BST* right;
} BST;
```

La funzione deve restituire la lunghezza del cammino più lungo dalla radice a una foglia.

Esempio:

Per l'albero:

```
  1
 /\
2  3
/\  \
4  5  6
  \
   7
```

La funzione deve restituire 4 (cammino 1 -> 2 -> 5 -> 7)

Scrivere la funzione in modo ricorsivo, specificando PRE e POST condizioni e discutendone brevemente la correttezza.

**3.** Scrivere una funzione che, dato un testo e un pattern, trovi tutte le occorrenze del pattern nel testo in modo efficiente. La funzione deve utilizzare un algoritmo che precalcoli informazioni sul pattern per evitare confronti non necessari durante la ricerca nel testo.

Ad esempio, se abbiamo: Testo: "ABABDABACDABABCABAB" Pattern: "ABABCABAB"

La funzione dovrebbe restituire l'indice 10, che è la posizione di inizio dell'unica occorrenza completa del pattern nel testo.

La funzione deve avere la seguente firma:

```
int find_pattern(const char* text, const char* pattern);
```

**4.** Implementare una funzione che riorganizzi un albero binario di ricerca sbilanciato:

Scrivere una funzione che, dato un albero binario di ricerca potenzialmente sbilanciato, lo riorganizzi in modo da ridurne l'altezza, mantenendo la proprietà di albero binario di ricerca.

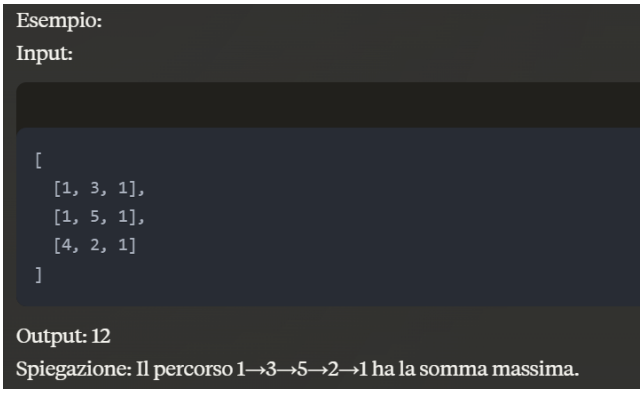
Ad esempio, dato il seguente albero sbilanciato:


La funzione dovrebbe trasformarlo in:


```
BST* reorganize_bst(BST* root);
```

**5.** Implementare una funzione per trovare il percorso con la somma massima in una matrice:

Scrivere una funzione che, data una matrice di numeri interi, trovi il percorso dalla cella in alto a sinistra alla cella in basso a destra che massimizza la somma dei valori nelle celle attraversate. È possibile muoversi solo verso destra o verso il basso.



La funzione deve avere la seguente firma:

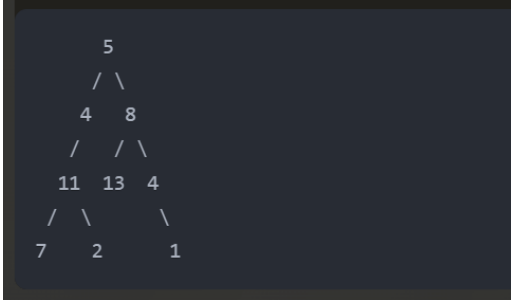
```
int max_path_sum(int** matrix, int rows, int cols);
```

**6.** Implementare una funzione per trovare il percorso radice-foglia con somma specificata:

```
bool has_path_sum(TreeNode* root, int sum);
```

Dato un albero binario e una somma, determinare se l'albero ha un percorso radice-foglia tale che la somma dei valori lungo il percorso sia uguale alla somma data.

Esempio:




sum = 22
Output: true (5 + 4 + 11 + 2 = 22)

7. Implementare una funzione per trovare il più basso antenato comune di due nodi in un albero binario:

```
BST* lowest_common_ancestor(BST* root, BST* p, BST* q);
```

Dato un albero binario e due nodi p e q, trovare il più basso antenato comune (LCA) dei due nodi. L'LCA è definito come il nodo più basso nell'albero che ha sia p che q come discendenti.

Esempio:


p = 5, q = 1
Output: 3

p = 5, q = 4
Output: 5