# Soluzioni Esercitazione IV ASD 2023-24

Lorenzo Cazzaro    Riccardo Maso    Alessandra Raffaetà

**Esercizio 1**. Sia **arr** un array di lunghezza **n** di numeri naturali positivi (> 0) e sia **k** un numero naturale. Si consideri il problema di determinare una **sottosequenza crescente di lunghezza massima**. Ad esempio, dato l'array *[10, 22, 9, 33, 21, 50, 41, 60, 80]*, una sottosequenza crescente di lunghezza massima è *[10, 22, 33, 50, 60, 80]*, che ha lunghezza 6.

1. Definire in modo ricorsivo la lunghezza massima di una sottosequenza crescente;
2. tradurre tale definizione in un algoritmo di programmazione dinamica con il metodo bottom-up che determina la lunghezza massima di una sottosequenza crescente;
3. trasformare l'algoritmo di modo che determini anche una sottosequenza crescente di lunghezza massima;
4. valutare le complessità degli algoritmi.

I prototipi delle funzioni sono:

```
int lunglongestIncreasingSubsequence(vector<int>& arr)
vector<int> longestIncreasingSubsequence(vector<int>& arr)
```

**Soluzione:**

1. Sia $lis(i)$ la funzione che restituisce la lunghezza massima di una sottosequenza crescente composta da elementi con indici da 0 a $i$ che termina con l'elemento in posizione $i$ (incluso)

$$\texttt{lis}(i) = \begin{cases} 1 & \text{if } i = 0 \\ 1 + max\{lis(j) \mid arr[j] < arr[i] \land 0 \leq j < i\} & \text{if } 0 < i < n \end{cases}$$

Per ottenere la lunghezza massima di una sottosequenza crescente si deve calcolare:

$$max\{lis(j) \mid 0 \leq j < n\}$$

.

2.
```cpp
#include <vector>
#include <algorithm>

using namespace std;

//funzione che restiuisce la lunghezza della sottosequnza crescente piu'
    lunga
int lunglongestIncreasingSubsequence(vector<int>& arr){
    vector<int> dp(arr.size(), 1);
    //itero l'indice della fine della sequenza
    for (int i = 1; i < dp.size(); i++) {
        //itero l'indice dell'inizio della sequenza
        for (int j = 0; j < i; j++) {
            // se il primo elemento e' piu' piccolo dell'ultimo
            if (arr[j] < arr[i]) {
                // salvo il massimo valore
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
    }

    // estraggo la lunghezza della sottosequnza crescente piu' lunga
    int maxLength = 0;
    for (size_t i = 0; i < dp.size(); i++)
        if(dp[i] > maxLength)
            maxLength = dp[i];

    return maxLength;
}
```

3.
```cpp
#include <vector>
#include <algorithm>

using namespace std;

//funzione che restiuisce la sottosequenza crescente di lunghezza massima
vector<int> longestIncreasingSubsequence(vector<int>& arr) {
    vector<int> dp(arr.size(), 1);
    vector<int> prec(arr.size(), -1);

    //itero l'indice della fine della sequenza
    for (int i = 1; i < dp.size(); i++) {
        //itero l'indice dell'inizio della sequenza
        for (int j = 0; j < i; j++) {
            // se il primo elemento e' piu' piccolo dell'ultimo
            if (arr[j] < arr[i]) {
                // controllo quale dei due valori e' maggiore e lo salvo
                if (dp[i] < dp[j] + 1){
                    dp[i] = dp[j] + 1;
                    // salvo l'elemento che precede i nella sottoquenza
                    prec[i] = j;
                }
            }
```

```
        }
    }

    // salvo l'indice della sottosequenza crescente di lunghezza massima
    int maxIndex = 0;
    for (size_t i = 0; i < dp.size(); i++)
        if(dp[i] > dp[maxIndex])
            maxIndex = i;

    // ricostruisco il risultato
    vector<int> result;
    while (maxIndex >= 0){
        result.push_back(arr[maxIndex]);
        //ad ogni iterazione mi sposto al predecessore
        maxIndex = prec[maxIndex];
    }

    //inverto l'array in quanto gli elementi sono stati inseriti partendo
     dall'ulitmo
    for (int i = 0; i < result.size() / 2; i++) {
        int temp = result[i];
        result[i] = result[result.size() - i - 1];
        result[result.size() - i - 1] = temp;
    }

    return result;
}
```

4. Per calcolare la complessità temporale asintotica nel caso peggiore della funzione `lunglongestIncreasingSubsequence` si determina prima di tutto il costo dei due cicli `for` annidati. In totale vengono eseguite $\sum_{k=1}^{n-1} k = \frac{(n-1)n}{2}$ iterazioni e ciascuna iterazione prevede l'esecuzione di istruzioni dal costo costante. Il ciclo `for` alla fine della funzione itera $n$ volte ed esegue operazioni di costo costante, dunque ha complessità $\Theta(n)$.
   Quindi la complessità asintotica di `lunglongestIncreasingSubsequence` è $T(n) = \Theta(n^2) + \Theta(n) = \Theta(n^2)$.

   La funzione `longestIncreasingSubsequence` presenta gli stessi cicli analizzati precedentemente nella prima funzione. Inoltre, sono presenti un ciclo `while` che itera un numero di volte pari alla lunghezza della sottosequenza trovata, che sarà al massimo lunga $n$ la sua complessità è quindi $\mathcal{O}(n)$. L'ultimo ciclo `for` prevede un numero di iterazioni pari alla metà della lunghezza della sottosequenza, che sarà al massimo lunga $n$, e il suo corpo presenta istruzioni dal costo costante. La sua complessità è quindi $\mathcal{O}(n)$.

   Quindi, la complessità asintotica di `longestIncreasingSubsequence` è $T(n) = \Theta(n^2) + \Theta(n) + \mathcal{O}(n) + \mathcal{O}(n) = \Theta(n^2)$.

**Esercizio 2**. Data una griglia di monete **n x m**, dove ogni cella contiene un certo numero di monete, trova il massimo numero di monete che si possono raccogliere partendo dall'angolo in alto a sinistra della griglia e spostandosi solo verso destra o verso il basso fino all'angolo in basso a destra.

1. Dare una caratterizzazione ricorsiva della soluzione;
2. tradurre tale definizione in un algoritmo di programmazione dinamica con il metodo **top-down** che risolve il problema;
3. valutare la complessità dell'algoritmo.

Il prototipo della funzione è

```
int massimaRaccoltaMonete(vector<vector<int>>& arr)
```

**Soluzione:**

1. Sia `arr` la griglia di monete e sia `gain(i, j)` la funzione che restituisce il numero massimo di monete che si possono raccogliere partendo dalla cella in posizione `(i, j)` nella griglia e muovendosi verso l'angolo in alto a sinistra della griglia spostandosi solo verso sinistra o verso l'alto. La caratterizzazione ricorsiva è la seguente:

$$\mathtt{gain}(i,j) = \begin{cases} 0 & \text{if } i < 0 \ or \ j < 0 \\ \mathtt{arr}[i,j] + \max\{\mathtt{gain}(i-1,j), \mathtt{gain}(i,j-1)\} & \text{if } i \geq 0 \ and \ j \geq 0 \end{cases}$$

2.
```
#include <vector>
#include <algorithm>

using namespace std;

// Funzione ausiliaria per calcolare il massimo numero di monete raccolte
int massimaRaccoltaMoneteAux(vector<vector<int>>& arr, int i, int j,
    vector<vector<int>>& dp) {
    // fuori dalla griglia
    if (i < 0 || j < 0) {
        return 0;
    }

    //soluzione gia' calcolata
    if (dp[i][j] != -1) {
        return dp[i][j];
    }

    int sopra = 0, sinistra = 0;
    if (i > 0) {
        //Movimento verso l'alto
        sopra = massimaRaccoltaMoneteAux(arr, i - 1, j, dp);
    }
    if (j > 0) {
        //Movimento verso sinistra
```

```
        sinistra = massimaRaccoltaMoneteAux(arr, i, j - 1, dp);
    }

    dp[i][j] = arr[i][j] + max(sopra, sinistra);
    return dp[i][j];
}

int massimaRaccoltaMonete(vector<vector<int>>& arr) {
    int n = arr.size();
    int m = arr[0].size();
    //matrice di supporto per programmazione dinamica
    vector<vector<int>> dp(n, vector<int>(m, -1));

    return massimaRaccoltaMoneteAux(arr, n - 1, m - 1, dp);
}
```

3. La complessità temporale asintotica nel caso peggiore della funzione `massimaRaccoltaMonete` è determinata dalla complessità dell'inizializzazione della matrice `dp` e dalla complessità asintotica della funzione ausiliaria `massimaRaccoltaMoneteAux`. La complessità dell'inizializzazione della matrice `dp` è $\Theta(n \times m)$, in quanto deve essere inizializzata l'intera matrice. La complessità asintotica di `massimaRaccoltaMoneteAux` dipende dalle chiamate ricorsive effettuate. Ogni chiamata ricorsiva risolve un sottoproblema e salva la sua soluzione nella matrice `dp`. La soluzione del sottoproblema, escluse le sottochiamate ricorsive, avviene in tempo costante e i sottoproblemi sono $n \times m$. Dunque la funzione ha complessità asintotica $\Theta(n \times m)$. Per concludere, la complessità asintotica di `massimaRaccoltaMonete` è $T(n, m) = \Theta(n \times m) + \Theta(n \times m) = \Theta(n \times m)$.

**Esercizio 3**. Sia data una sequenza di città $c_1 c_2 \dots c_n$ collegate da un percorso ferroviario. Da ciascuna città $c_i$ è possibile raggiungere $c_j$, con $i < j$, con un treno diretto. Sapendo che per $i, j \in \{1, \dots, n\}$, $i < j$ il costo del biglietto del treno diretto da $c_i$ a $c_j$ risulta essere $b[i, j]$, determinare il costo minimo di un tragitto, con possibili cambi intermedi, da $c_1$ a $c_n$. Più in dettaglio:

1. Fornire una caratterizzazione ricorsiva del costo minimo `min[i]` di un tragitto dalla città $c_i$ alla città $c_n$;

2. tradurre tale definizione in un algoritmo di programmazione dinamica con il metodo **bottom-up** che determina il costo di un tragitto di costo minimo dalla città $c_1$ alla città $c_n$;

3. trasformare l'algoritmo in modo che determini anche la sequenza dei cambi necessari per ottenere il tragitto di costo minimo, privilegiando – a parità di costo – soluzioni che minimizzano il numero dei cambi;

4. valutare le complessità degli algoritmi.

I prototipi delle funzioni sono:

```
int minTrain(vector<vector<int>>& b, int n)
vector<int> minTrainSequence(vector<vector<int>>& b, int n)
```

**Soluzione:**

1. Sia `min[i]` la funzione che restituisce il costo minimo di un tragitto dalla città $c_i$ alla città $c_n$. La caratterizzazione ricorsiva è la seguente:

$$\texttt{min[i]} = \begin{cases} 0 & \text{if } i = n \\ min\{\texttt{b[i,j]} + \texttt{min[j]} \mid j \in \{i+1, \ldots, n\}\} & \text{if } i < n \end{cases}$$

2. 
```
#include <vector>
#include <algorithm>

using namespace std;

// Funzione che restituisce il minimo costo per andare dalla stazione 0
    alla stazione n-1
int minTrain(vector<vector<int>>& b, int n){
    vector<int> mincosto(n);
    int temp;
    mincosto[n-1] = 0;
    for (int i= n-2; i>= 0; i--){
        mincosto[i] = b[i][n-1];
        for (int j = i+1; j < n - 1; j++){
            temp = b[i][j] + mincosto[j];
            if (temp < mincosto[i])
                mincosto[i] = temp;
        }
    }
    return mincosto[0];
}
```

3. 
```
#include <vector>
#include <string>

using namespace std;

// Funzione che restituisce la sequenza dei cambi necessari per ottenere
    il tragitto di costo minimo per andare dalla stazione 0 alla stazione
     n-1
vector<int> minTrainSequence(vector<vector<int>>& b, int n){
    vector<int> mincosto(n);
    vector<int> cambi(n);
    vector<int> next(n);
    vector<int> sequenzacitta;
    int tempcosto;
    mincosto[n-1] = 0;
    cambi[n-1] = 0;
```

```
    for (int i= n-2; i>= 0; i--){
        mincosto[i] = b[i][n-1];
        cambi[i] = 0;
        next[i] = n-1;
        //compongo la sequenza a partire dall'indice i
        for (int j = i+1; j < n - 1; j++){
            tempcosto = b[i][j] + mincosto[j];
            if ((mincosto[i] > tempcosto) || (tempcosto == mincosto[i] &&
     (cambi[i] > cambi[j] + 1))){
                mincosto[i] = tempcosto;
                cambi[i] = cambi[j] + 1;
                next[i] = j;
            }
        }
    }

    int j = 0;
    while (j < n-1){
        sequenzacitta.push_back(j);
        j = next[j];
    }
    sequenzacitta.push_back(n-1);
    return sequenzacitta;
}
```

4. La complessità temporale asintotica nel caso peggiore della funzione `minTrain` è determinata dalla complessità asintotica dei cicli `for`. Vengono eseguite in totale $\sum_{i=1}^{n-1} i = \dfrac{(n-1)n}{2}$ iterazioni e ciascuna iterazione prevede l'esecuzione di istruzioni dal costo costante. Quindi, la complessità asintotica di `minTrain` è $T(n) = \Theta(n^2)$.

   La complessità temporale asintotica nel caso peggiore della funzione `minTrainSequence` è determinata dalla complessità asintotica dei due cicli `for` utilizzati per determinare il costo e la sequenza di città e dal costo dell'ultimo ciclo `while` per inserire in `sequenzacitta` le città:

   - La complessità asintotica derivante dai due cicli for è $\Theta(n^2)$.
   - La complessità dell'ultimo ciclo `while` è $\mathcal{O}(n)$, dato che `sequenzacitta` conterrà al massimo tutte le $n$ città.

   Quindi la complessità asintotica della funzione `minTrainSequence` è $T(n) = \Theta(n^2) + \mathcal{O}(n) = \Theta(n^2)$