UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Algorithms and Data Structures

# *Homework 1*

Anno Accademico 2024/2025

Studenti
**Filomena Viglotti**
**matr. M63001734**
**Antonio Sirignano**
**matr. M63001732**

# Indice

# Capitolo 1

# Esercizio 1 - Notazione asintotica

## 1.1 Esercizio 1.1

- $f(n) = O(f^2(n))$

L'affermazione **non è sempre vera**.

Si può dimostrare ciò tendendo in considerazione due casi:

1. $f(n)$ è una funzione non negativa **crescente**
2. $f(n)$ è una funzione non negativa **decrescente**

Nel caso 1 si può considerare a titolo d'esempio la funzione

$$f(n) = e^n$$

Applicando la definizione di notazione $O$, si ha che

$$0 \leq e^n \leq c \cdot e^{2n}$$

Induttivamente, valutiamo il parametro $c$.

Utilizziamo il caso base $n = 0$. Si ottiene

$$1 \leq c \cdot 1 \implies c \geq 1$$

Supposta la validità al passo $n$, bisogna dimostrare che ciò è valido anche per $n+1$. Si ottiene

$$
\begin{aligned}
& e^{n+1} \leq c \cdot e^{2n+2} \\
& e \cdot e^{n} \leq c \cdot e^{2} \cdot e^{2n} \\
& 1 \cdot e^{n} \leq c \cdot e \cdot e^{2n} \\
& \implies 1 \leq c \cdot e \implies c \geq \frac{1}{e}
\end{aligned}
$$

Questo conferma che la funzione ha un limite asintotico superiore da cui si può dire che

$$f(n) = O(f^2(n))$$

Nel caso 2 si prenda in esame la funzione

$$f(n) = e^{-n}$$

Volendo applicare la definizione di notazione $O$, e ci si accorge che in questo caso non è valida, poiché

$$0 \leq e^{-2n} \leq e^{-n}$$

Non essendoci un limite asintotico superiore, si ha

$$f(n) \neq O(f^2(n))$$

Si è scelto anche in questo caso $c = 1$ al caso base, applicando l'induzione al analogamente al caso 1.

## 1.2 Esercizio 1.2

- $f(n) + O(f(n)) = \Theta(f(n))$

Per ogni coppia di funzioni $f(n)$ e $g(n)$, si ha $f(n) = \Theta(g(n))$, se e soltanto se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

L'equazione è corretta perchè se si considera una funzione $f(n)$ e le si somma una funzione appartenente a $O(f(n))$, la crescita della somma sarà comunque limitata dal termine $f(n)$ poichè $O(f(n))$ rappresenta una funzione che cresce al massimo quanto $f(n)$.

$O(f(n))$ può essere una costante moltiplicata per $f(n)$ o un termine che diventa trascurabile

rispetto a $f(n)$ quando $n$ cresce. Quindi $f(n)+O(f(n))$ sarà sempre asintoticamente equivalente a $f(n)$ poichè la somma di $f(n)$ con un termine che cresce al massimo quanto $f(n)$ è ancora $\Theta(f(n))$. Quindi l'equazione

$$f(n) + O(f(n)) = \Theta(f(n))$$

è sempre vera, perchè $O(f(n))$ rappresenta una funzione che non cresce più velocemente di $f(n)$, e quindi la somma è dominata da $f(n)$.

## 1.3 Esercizio 1.3

- $f(n) = \Omega(g(n))$ e $f(n) = o(g(n))$

Le due affermazioni non sono mai vere contemporaneamente.
Questo si dimostra sulla base delle definizioni di $O$ e $\Omega$. Infatti

$$\Omega(g(n)) = \{f(n) : \exists\ c > 0, n_0 > 0 : 0 \leq cg(n) \leq f(n), \forall n \geq n_0\}$$

$$o(g(n)) = \{f(n) : \forall c, \exists\ n_0 > 0 : 0 \leq f(n) < cg(n), \forall n \geq n_0\}$$

Mettendo a sistema le condizioni derivanti da queste definizioni, si ottiene

$$\begin{cases} 0 \leq cg(n) \leq f(n), & \forall n \geq n_0 \\ 0 \leq f(n) < cg(n), & \forall n \geq n_0 \end{cases}$$

Ovvero due condizioni che non si verificano mai contemporaneamente.
Se si avesse avuto $f(n) = \Omega(g(n))$ e $f(n) = O(g(n))$, sarebbe stato vero nel caso $f(n) = g(n)$; avendo una notazione di tipo $o$, non si verifica mai.

# Capitolo 2

# Esercizio 2 - Complessità

## 2.1 Esercizio 2.1

- É vero che $2^{n+1} = O(2^n)$?

Procediamo applicando la definizione della notazione $O$

$$2^{n+1} \leq c \cdot 2^n$$

Consideriamo il caso base $n = 0$:

$$2^1 \leq c \cdot 1 \implies c \geq 2$$

Supposta vera al passo $n$, procediamo nel validare il passo $n + 1$

$$2^{n+2} \leq c \cdot 2^{n+1}$$

$$2 \cdot 2^{n+1} \leq c \cdot 2 \cdot 2^n$$

Il che è vero se si scegliesse ad esempio $c = 2$

## 2.2 Esercizio 2.2

- É vero che $2^{2n} = O(2^n)$

Per mostrare ciò applichiamo la definizione di $O$:

$$0 \leq \ 2^{2n} \leq c \cdot 2^n$$

Si valuta il caso base $n = 0$

$$1 \leq c \cdot 1 \implies c \geq 1$$

Supposto valido per $n - 1$, deve essere verificato al passo $n$:

$$2^{2n} \leq c \cdot 2^n$$

$$2^n \cdot 2^n \leq c \cdot 2^n$$

Per ogni $n$, non esiste una costante $c$ che sia maggiore di $2^n$: l'espressione è quindi falsa.

# Capitolo 3

# Esercizio 3

## 3.1 Esercizio 3.1

- $T(n) = 2T(\frac{n}{3}) + n \log n$

Per fornire il limite inferiore e superiore per T(n), si parte dall'applicazione del terzo caso del Teorema dell'Esperto. Dapprima si verificano le condizioni che soddisfano la tesi:

1. Scelto $\epsilon = 0.1$, si verifica banalmente che

$$n \log n = \Omega(n^{\log_3 2+\epsilon}) == \Omega(n^{\log_3 2+0.1}) = \Omega(n^{0.631+0.1})$$

2. Si deve verificare che, scelto $c < 1$, $af(\frac{n}{b}) \leq cf(n)$, per $n$ sufficientemente grande:

$$3 \cdot \frac{n}{3} \log\left(\frac{n}{3}\right) \leq c \cdot n \log n$$

   Passando al limite per $n$ che tende ad infinito

$$\lim_{n\to\infty} n \log\left(\frac{n}{3}\right) \leq c \cdot \lim_{n\to\infty} n \log n$$
$$\implies \lim_{n\to\infty} \frac{n \log\left(\frac{n}{3}\right)}{n \log n} \leq c$$

   Il che è verificato, poiché il denominatore tende più velocemente a $\infty$.

Si conclude quindi che

$$T(n) = \Theta(f(n)) = \Theta(n \log n)$$

## 3.2 Esercizio 3.2

- $T(n) = 3T(\frac{n}{5} + \log^2(n))$

In tal caso è possibile applicare nuovamente il Teorema dell'Esperto, al caso 3.

Le condizioni che si verificano sono

1. La funzione in esame è

$$f(n) = O(n^{log_5 3+\epsilon}) = O(n^{0.68+\epsilon}) = O(n^1), \qquad \text{con } \epsilon = 0.32$$

2. La seconda condizione da verificare è la seguente:

$$\begin{aligned} & 3\log^2\frac{n}{5} \leq c\log^2 n \\ \implies & 3\frac{\log^2\frac{n}{5}}{\log^2 n} \leq c \\ \implies & 3\lim_{n\to\infty}\frac{\log^2\frac{n}{5}}{\log^2 n} \leq c \end{aligned}$$

   Il denominatore tende più velocemente ad infinito rispetto al numeratore. Quindi la condizione è verificata.

Si conclude che

$$T(n) = \Theta(f(n)) = \Theta(\log^2 n)$$

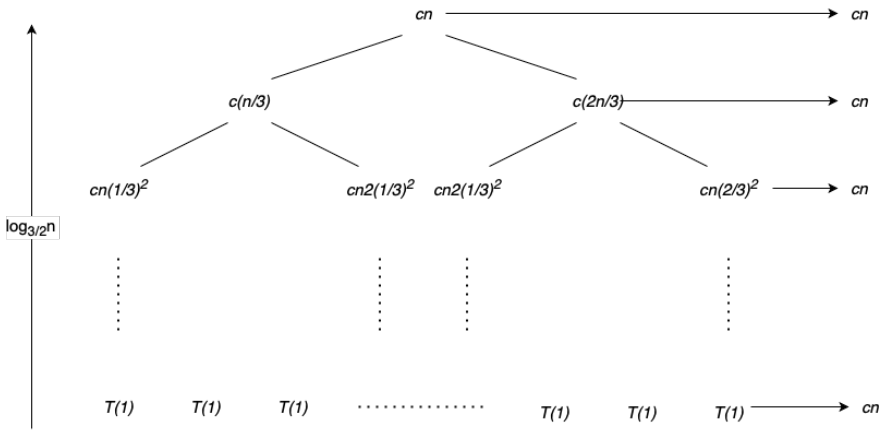
# Capitolo 4

# Esercizio 4 - Ricorrenze

## 4.1 Esercizio 4.1

- $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + cn$

L'albero di ricorrenza per tale funzione è il seguente




Ogni livello ha un costo costante pari ad $cn$.

L'altezza dell'albero si può stimare il fattore di riduzione della dimensione del problema: si nota che il problema si riduce come segue:

$$n \to \frac{2n}{3}$$

Quindi, l'altezza dell'albero è pari ad:

$$h \approx \log_{\frac{3}{2}} n = O(\log n)$$

Moltiplicando l'altezza dell'albero per il costo di ogni singolo livello, si ottiene il costo complessivo:

$$T(n) = cn \times \log n = \Omega(n \log n)$$

Questo perché $cn \times \log n$ è almeno $\Omega(n \log n)$, ovvero:

$$0 \leq d \cdot n \log n \leq cn \cdot \log n$$

dove $d$ è una costante arbitraria positiva tale che

$$0 < d \leq c$$

In conclusione è verificato che $T(n) = \Omega(n \log n)$.

# Capitolo 5

# Problema 1

Si deve fornire un algoritmo in grado di cercare il valore massimo in un vettore unimodale.

```
1: procedure FindMax(A):
2:     left ← 1
3:     right ← A.length
4:     mid_position ← 0
5:     while left < right do
6:         mid_position = ⌊(left + right)/2⌋
7:         if A[mid_position] < A[mid_position + 1] then
8:             left ← mid_position + 1
9:         else if A[mid_position] > A[mid_position + 1] then
10:            right ← mid_position
11:        end if
12:    end while
13: return A[left]
14: end procedure
```

Il ciclo *while* in questo caso va trattato in modo simile ad una ricorrenza: ad ogni iterazione del ciclo, la dimensione dell'array si dimezza e di conseguenza viene dimezzato il numero di iterazioni del ciclo, il quale al più viene eseguito $\log(\frac{n}{2})$ volte.
Supposto quindi un costo costante $c$ per ogni iterazione, si ha:

$$T(n) = c \cdot \log \frac{n}{2} = c \cdot \log n - c \cdot \log 2 = O(\log n)$$

É riportato di seguito il codice dell'algoritmo scritto in Python:

```python
# L'algoritmo deve trovare il massimo in un array unimodale
# (SUGGERIMENTO: simile a ricerca binaria)

A = [1, 2, 4, 24, 27, 60, 56, 40, 39, 27, 13, 12, 6, 2]
```

```
5
6  def find_max(A):
7      left = 0
8      right = len(A)-1
9      mid_position = 0
10
11     while left < right:
12         mid_position = (left+right)//2
13         if A[mid_position] < A[mid_position+1]:
14             left = mid_position+1
15         elif A[mid_position] > A[mid_position+1]:
16             right = mid_position
17
18     return A[left]
19
20 print(find_max(A))
```

# Capitolo 6

# Problema 2

Si vuole fornire un algoritmo *divide et impera* che permetta di trovare il prefisso più lungo in comune tra l'insieme di stringhe fornito input.
Per tale problema si deve far in modo di riordinare il vettore di parole in ingresso in ordine alfabetico, per poi confrontare la prima parola con l'ultima parola. Così facendo la complessità si basa solo sull'algoritmo di ordinamento scelto, poiché si può considerare non rilevante ai fini della complessità il confronto tra le due parole. Lo pseudocodice è il seguente:

```
1:  procedure MaxPrefix(A):
2:      i ← 1
3:      prefix ← ""
4:      MergeSort(A, 0, A.lenght)
5:      while i < len(A[1] and i < len(A[A.lenght] do
6:          if A[0][i] ≠ A[A.lenght][i] then return prefix
7:          end if
8:          prefix ← prefix + A[0][i]
9:      end while
10: return prefix
11: end procedure
```

Avendo utilizzato come algoritmo di ordinamento il Merge Sort, la complessità dell'algoritmo sarà:

$$T(n) = \Theta(n \log n) + k \cdot \Theta(1)$$

dove con $k$ si indica la lunghezza di una singola parola.

Essendo $k \cdot \Theta(1)$ trascurabile, si conclude che

$$T(n) = \Theta(n \log n)$$

e quindi, come detto in precedenza, la complessità è data solamente dall'algoritmo di ordinamento scelto.

É riportato di seguito il codice dell'algoritmo scritto in Python:

```
#Implementazione merge sort

def merge(A, p, q, r):
    n1 = q-p+1
    n2 = r-q

    L=[]
    R=[]

    for i in range(0, n1):
        L.append(A[p+i-1])
    for j in range(0, n2):
        R.append(A[q+j])

    L.append(chr(255) * 100)
    R.append(chr(255) * 100)

    i, j = 0, 0

    for k in range(p, r):
        if L[i] <= R[j]:
            A[k] = L[i]
            i = i+1
        else:
            A[k] = R[j]
            j = j+1

def merge_sort(A, p, r):
    if p<r:
        q = (p + r) // 2
        merge_sort(A, p, q)
        merge_sort(A, q+1, r)
        merge(A, p, q, r)


def max_prefix(A):
    i = 0
    prefix = ""
```

```
    merge_sort(A, 0, len(A))

    while i < len(A[0]) and i < len(A[-1]):
        if A[0][i] != A[len(A)-1][i]:
            return prefix
        prefix = prefix +  A[0][i]
        i=i+1

    return prefix


A = ["apple", "ape", "april", "applied"]
B = ["formica", "fortezza", "fortino", "forza", "formaggio"]
C = ["colonna", "collo", "colletto", "cornetto", "costo", "cono", "costa", "covid", "
    cobalto", "collana"]
D = ["ansia", "ansia"]


prefix_A = max_prefix(A)
prefix_B = max_prefix(B)
prefix_C = max_prefix(C)
prefix_D = max_prefix(D)

print(prefix_A)
print(prefix_B)
print(prefix_C)
print(prefix_D)
```

# Capitolo 7

# Problema 3

Si vuole fornire un algoritmo di inserimento in una struttura dati **Treap** nota, la quale prevede un inserimento che rispetti le proprietà dell'Albero di Ricerca Binario, e le proprietà del **Min-Heap order** dell'Heap sulla base di una priorità associata in maniera casuale a ciascun nodo. Ogni singolo nodo è una struttura i cui attributi sono:

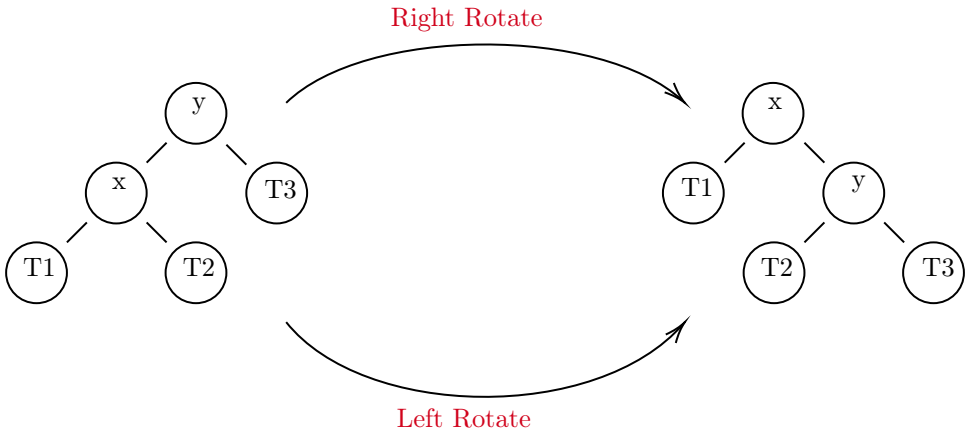
- `key`: rappresenta il valore del nodo
- `priority`: rappresenta la priorità del nodo associata casualmente
- `left`: rappresenta il figlio sinistro
- `right`: rappresenta il figlio destro

Dapprima si definiscono le funzioni `rightRotate()` e `leftRotate()` le quali consentono la rotazione necessarie per mantenere le proprietà dell'Heap, e funzionano come segue:

```
1: procedure RightRotate(y):
2:     x ← y.left
3:     T2 ← x.right
4:
5:     x.right ← y
6:     y.left ← T2
7: return x
8: end procedure
```

```
1: procedure LeftRotate(x):
2:     y ← x.right
3:     T2 ← y.left
4:
5:     y.left ← x
6:     x.right ← T2
7: return y
8: end procedure
```

Nel seguente grafico, si vede quello che succede quando vengono chiamate le funzioni precedenti.




Si definisce ora la funzione `insert()`, la quale ci permette di inserire gli elementi nell'albero rispettando le proprietà:

```
1: procedure Insert(root, key):
2:     if root == NULL then return TreapNode(key)
3:     end if
4:     if key ≤ root.key then
5:         root.left ← insert(root.left, key)
6:         if root.left.priority < root.priority then
7:             root ←rightRotate(root)
8:         end if
9:     else
10:        root.right ← insert(root.right, key)
11:        if root.right.priority < root.priority then
12:            root ←leftRotate(root)
13:        end if
14:    end if
15: return root
16: end procedure
```

Si noti che nel caso in cui si volesse realizzare un Treap che rispetti le proprietà del Max Heap, allora nelle condizioni a rigo 6 e 11, il segno di $<$ va sostituito con $>$. Di seguito è presentato

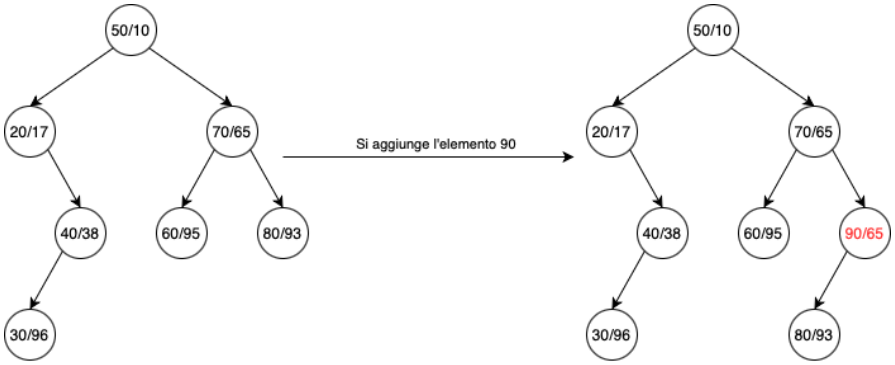
il codice in Python, in cui è anche presente la funzione `inorder()` che consente la stampa a video dell'albero.

```
import random
import string

class TreapNode:
    def __init__(self, key):
        self.key = key
        self.priority = random.randint(0,99)
        self.left = None
        self.right = None

def rightRotate(y):
    x = y.left
    T2 = x.right

    x.right = y
    y.left = T2

    return x

def leftRotate(x):
    y = x.right
    T2 = y.left

    y.left = x
    x.right = T2

    return y

def insert(root, key):
    if not root:
        return TreapNode(key)

    if key <= root.key:
        root.left = insert(root.left, key)

        if root.left.priority < root.priority:
            root = rightRotate(root)

    else:
```

```
        root.right = insert(root.right, key)

        if root.right.priority < root.priority:
            root = leftRotate(root)

    return root


def inorder(root):
    if root:
        inorder(root.left)
        print("key:", root.key, "| priority:", root.priority, end="")
        if root.left:
            print(" | left child:", root.left.key, end="")
        if root.right:
            print(" | right child:", root.right.key, end="")
        print()
        inorder(root.right)


if __name__ == '__main__':

    root = None
    root = insert(root, 50)
    root = insert(root, 30)
    root = insert(root, 20)
    root = insert(root, 40)
    root = insert(root, 70)
    root = insert(root, 60)
    root = insert(root, 80)

    inorder(root)

    root = insert(root, 90)

    print("Print after insert 90\n")
    inorder(root)
```

Nella seguente si mostra un un caso test: viene aggiunto ad un albero l'elemento 90.

Ovviamente si tenga a mente che ad ogni esecuzione potrebbe variare la posizione, poiché ad ogni chiave (numero di sinistra) è associata una randomicamente una priorità (numero di destra).


50/10
20/17
70/65
40/38
60/95
80/93
30/96
Si aggiunge l'elemento 90
50/10
20/17
70/65
40/38
60/95
90/65
30/96
80/93