DIETI UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica

CORSO DI
ALGORITMI E STRUTTURE DATI
Prof. ROBERTO PIETRANTUONO
Homeworks set #1

HomeWork Set 1 - Bruno/Carleo

VINCENZO LUIGI BRUNO e CRISTINA CARLEO

1.1 Notazione asintotica e crescita delle funzioni

1)
f1(n) = $n^{0.999999}$ log n
f2(n) = 10000000n
f3(n) = 1.000001^n
f4(n) = n^2

- $f_1(n)$ è $O(n^{0.999999} \lg n)$, dove al crescere di n, l'esponente < 1 non arriva $O(n \lg n)$
- $f_2(n)$ è $O(n)$ in quanto n è moltiplicato per un coefficiente.
- $f_4(n) = O(n^2)$ è quadratica.
- $f_3(n)$ è $O(1.000001^n)$, per n piccoli $f_3(n) \approx 1$ mentre per $n \to \infty$ $f_3(n)$ è un esponenziale

$$f_1(n) \leq f_2(n) \leq f_4(n) \leq f_3(n)$$

2)
f1(n) = $2^{2^{1000000}}$
f2(n) = 2^{10000n}
f3(n) = $\binom{n}{2}$
f4(n) = $n\sqrt{n}$

- $f_1(n)$ è classificato come costante: $O(1)$
- $f_4(n)$ invece cresce come: $O(n^{\frac{3}{2}})$
- $f_3(n)$ dalla formula di Gauss troviamo che:

$$\frac{n!}{2 \cdot (n-2)!} = \frac{n \cdot (n-1) \cdot (n-2)!}{2 \cdot (n-2)!} = \frac{n \cdot (n-1)}{2} = O(n^2)$$

- f_2 invece cresce sponenzialmente: $O(2^n)$

Dunque:

$$f_1(n) \leq f_4(n) \leq f_3(n) \leq f_2(n)$$

---

3)
f1(n) = $n^{\sqrt{n}}$
f2(n) = 2^n
f3(n) = $n^{10} \cdot 2^{n/2}$
f4(n) = $\sum_{i=1}^{n}(i+1)$

- $f_4(n) = n + \sum_{i=1}^{n} i = n + \frac{n(n-1)}{2} = O(n^2)$
- $f_1(n) = n^{\sqrt{n}} = 2^{\sqrt{n} \cdot \log(n)}$ è inferiore alla complessità esponenziale ma superiore della quadratica
- $f_3(n) = n^{10} \cdot 2^{(\frac{n}{2})} = 2^{\log(n^{10})} \cdot 2^{(\frac{n}{2})} = 2^{(\frac{n}{2})+10\log(n)}$
- $f_2(n) = O(2^n)$

Dunque:

$$f_4(n) \leq f_1(n) \leq f_3(n) \leq f_2(n)$$

1.2. Notazione asintotica e crescita delle funzioni

- $f(n) = \frac{(n^2-n)}{2}$ e $g(n) = 6n$
  Visto che $6n \leq c(n^2) \rightarrow 6 \leq cn$ **vale che** $g(n) = O(f(n))$, mentre **non vale il viceversa.**
- $f(n) = n + 2\sqrt{n}$ e $g(n) = n^2$
  Visto che $n + 2\sqrt{n} \leq cn^2 \rightarrow \frac{1}{n} + \frac{2}{\sqrt{n}} \leq c$ **vale che** $f(n) = O(g(n))$, mentre **non vale il viceversa**
- $f(n) = n \log n$ e $g(n) = n\frac{\sqrt{n}}{2}$
  Visto che $n \log n \leq cn^{\frac{3}{2}} \rightarrow \log n \leq c\sqrt{n}$ **vale che** $f(n) = O(g(n))$, mentre **non vale il viceversa.**
- $f(n) = n + \log n$ e $g(n) = \sqrt{n}$
  Visto che $\sqrt{n} \leq cn \rightarrow \frac{1}{\sqrt{n}} \leq c$ **vale che** $g(n) = O(f(n))$ mentre **non vale il viceversa.**
- $f(n) = 2(\log n)^2$ e $g(n) = \log n + 1$
  Visto che $\log n \leq c \log^2 n \rightarrow \frac{1}{\log n} \leq c$ **vale che** $g(n) = O(f(n))$, mentre **non vale il viceversa.**

- $f(n) = 4n \log n + n$ e $g(n) = \frac{n^2 - n}{2}$

  Visto che $n \log n \leq cn^2 \rightarrow \log n \leq cn$ **vale che** $f(n) = O(g(n))$**,** mentre **non vale il viceversa**

1.3. Notazione asintotica e crescita delle funzioni

- $2^{n+1} = O(2^n)$ è **vero**, poichè $2^{n+1} = 2 \cdot 2^n = O(2^n)$
- $2^{2n} = O(2^n)$ è **falso,** poichè $2^{2n} = 4^n$ e $2^n = o(4^n)$
  - Supponiamo che $2^{2n} = O(2^n)$. Allora esiste una costante c tale che per un certo $n \geq n_0$, $2^{2n} \leq c \cdot 2^n$. Dividendo entrambi i lati per 2^n, otteniamo $2^n < c$. Non ci sono quindi valori per c e n_0 che possano rendere questo
  - vero, quindi l'ipotesi è falsa e $2^{2n}! = O(2^n)$

1.4. Ricorrenze

- $T(n) = 2T(n/2) + O(\sqrt{n})$
  - Usando il metodo dell'esperto risulta che $a = 2$, $b = 2$, $f(n) = O(\sqrt{n})$ e $n^{\log_b a} = n$, .

    Dal primo caso del teorema dell'esperto risulta che

    $O(\sqrt{n}) = O(n^{1-\epsilon})$ con $\epsilon = \frac{1}{2}$. Quindi $T(n) = \Theta(n)$
  - Applicando l'albero di ricorrenze sostituendo $O(\sqrt{n})$ con $c\sqrt{n}$. Notiamo che ciascun nodo ha il doppio dei nodi del livello precedente, dunque il costo per ogni singolo nodo è $c\sqrt{\frac{n}{2^i}}$

$$
\begin{aligned}
T(n) = \sum_{i=0}^{\log_2 n - 1} 2^i \cdot c\sqrt{\frac{n}{2^i}} + 2^{\log_2 n} T(1) = \\
= \sum_{i=0}^{\log_2 n - 1} 2^i \cdot c\sqrt{\frac{n}{2^i}} + \Theta(n^{\log_2 2}) = \\
= c\sqrt{n} \sum_{i=0}^{\log_2 n - 1} \sqrt{2}^i + \Theta(n) = \\
== c\sqrt{n} \frac{\sqrt{2}^{\lg_2(n)} - 1}{\sqrt{2} - 1} + \Theta(n) = \\
== c\sqrt{n}(1 + \sqrt{2})(\sqrt{n} - 1) + \Theta(n)
\end{aligned}
$$

  Dunque il risultato è $\Theta(n)$.

- $T(n) = T(\sqrt{n}) + \Theta(\log(\log(n)))$

  Faccio una sostituzione di variabili: $n = 2^m$. La ricorrenza diventa dunque:

  $T(2^m) = T(2^{m/2}) + \Theta(\log(m))$. A questo punto, considerando $T(2^m) = S(m)$ si ottiene:

  $S(m) = S(\frac{m}{2}) + \Theta(\log(m))$.

  Per risolvere questa ricorrenza non è possibile utilizzare il terzo caso del teorema dell'esperto, infatti $\lg(n)$ non è polinomialmente più grande di n^0.

  Utilizzando il metodo dell'albero di ricorrenza si ottiene la seguente espressione:

$$
\begin{aligned}
S(m) = S\left(\frac{m}{2}\right) + c\log_2(m) = \\
= \sum_{i=0}^{\log_2 m - 1} c\log_2\left(\frac{m}{2^i}\right) + \Theta(n^{\log_2 1})
\end{aligned}
$$

  Ma visto che per la proprietà dei logaritmi: $\log_2\left(\frac{m}{2^i}\right) = \log_2 m - \log_2 2^i = \log_2 m - i$:

$$S(m) = c \sum_{i=0}^{\log_2 m - 1} \log_2 m - c \sum_{i=0}^{\log_2 m - 1} i =$$
$$= c \log_2(m) \log_2(m) - c \frac{\log_2 m (\log_2(m) + 1)}{2} =$$
$$\implies S(m) = \Theta(\log_2^2(m))$$

Invertiamo ora la trasformazione utilizzata: $m = \log_2(n)$:

$$T(n) = \Theta(\log_2^2(\log_2 n))$$

- $T(n) = 10T(\frac{n}{3}) + 17n^{1.2}$

  $17n^{1.2-\epsilon} = O(n^{\log_3(10)-\epsilon}) \approx O(n^{2.09-\epsilon})$ che è uguale a $O(n^{1.2})$ con $\epsilon = 0.89$.

  Per il primo caso del teorema dell'esperto possiamo affermare che $T(n) = \Theta(n^{\log_3(10)})$

  Usiamo il metodo dell'albero di ricorrenza:

$$T(n) = \sum_{i=0}^{\log_3 n - 1} 10^i \cdot c\left(\frac{n}{3^i}\right)^{1.2} + 10^{\log_3 n} T(1) =$$
$$= \sum_{i=0}^{\log_3 n - 1} (2,676)^i \cdot cn^{1,2} + \Theta(n^{\log_3 10}) =$$
$$= \left[\frac{(2,676)^{\log_3 n} - 1}{(2,676) - 1}\right] cn^{1,2} + \Theta(n^{\log_3 10})$$
$$\implies T(n) = \Theta(n^{\log_3 10})$$

- $T(n) = T(n/3) + T(2n/3) + cn$

  I rami più corti dell'albero hanno lunghezza $\log_3(n) + 1$ mentre i più lunghi hanno lunghezza $\log_{\frac{3}{2}}(n) + 1$. Inoltre ad ogni livello ci sono sempre cn operazioni pichè $T(n)$ è suddiviso in $T(\frac{2n}{3})$ e $T(\frac{n}{3})$. Andando a considerare solo i primi $\log_3(n) + 1$ livelli dell'albero (visto che vogliamo verificare il limite inferiore) si ottiene un limite inferiore di $T(n)$:

$$T(n) \geq \sum_{i=0}^{\log_3(n)} cn = cn(\log_3(n) + 1) = \Theta(n \log_3(n))$$
$$\implies T(n) = \Omega(n \log_3(n))$$

Problema 1.1

```
def merge(array, start, mid, end):

    #liste per memorizzare i due sotto-array da fondere
    leftElements = []
    rightElements = []

    for i in range(mid - start+1): #copio gli elementi da start a mid in leftElements
        leftElements.append(array[start+i])
    for j in range(end - mid):
        rightElements.append(array[mid+1+j])

    inv_count = 0 #per contare il numero di inversioni
    # indici per scorrere le due liste
    i=0
    j=0
    isCounted = False #per garantire che le inversioni vengano contate una sola volta
    #per ogni elemento in rightList

    #end+1 perchè vogliamo iterare fino al valore finale incluso
    for k in range(start, end+1):
        #se entrambi gli indici non hanno raggiunto la fine delle rispettive liste,
        #confronto gli elementi correnti
```

```
        if i < len(leftElements) and j < len(rightElements):

            #se l'elemento in rightElement è minore di quello in left allora...
            if rightElements[j] < leftElements[i] and not isCounted:
                inv_count += len(leftElements) - i #...tutti gli elementi rimanenti in leftElement sono maggiori dell'elemento corrente in
                #dunque vengono aggiunte al conteggio
                isCounted = True #le inversioni per l'elemento corrente in rightElement sono state contate

            #aggiungo all'array l'elemento più piccolo e incremento il rispettivo indice
            if leftElements[i] <= rightElements[j]:
                array[k] = leftElements[i]
                i = i+1
            else:
                array[k] = rightElements[j]
                j = j+1
                isCounted = False #le inversioni per il prossimo elemento in rightElements non sono state ancora contate

        else: #se uno degli indici ha raggiunto la fine della lista
            #copio il resto dell'altra lista nell'array

            if i <len(leftElements):
                array[k] = leftElements[i]
                i = i+1
            else:
                array[k] = rightElements[j]
                j = j+1

    return inv_count

def sortInvCount(array, start, end):

    inv_count = 0 #variabile per tenere traccia del numero di inversioni

    if start < end: #se ci sono almeno due elementi nell'array

        # modo sicuro per calcolare il punto medio dell'array
        # evitando un possibile overflow di intero
        mid = start + (end - start) // 2 #// divisione intera

        inv_count += sortInvCount(array, start, mid) # calcola ricorsivamente il numero di inversioni nella prima metà dell’array
        inv_count += sortInvCount(array, mid+1, end) # calcola ricorsivamente il numero di inversioni nella seconda metà dell’array
        inv_count += merge(array, start, mid, end) #fonde le due metà dell’array in un unico array ordinato e conta il numero di inversioni

    return inv_count

def readTest(path):

    testFile = open(path)

    testList = [] #lista per memorizzare tutti i test case
    tempList = [] #lista per memorizzare temporaneamente ciascun test case

    sizeOfSequence = 0 #variabile usata per tenere traccia del numero di elementi rimanenti nel test case corrente
    newTestCase = True #flag per indicare l'inizio di un nuovo test case

    for line in testFile:
        if newTestCase: #se il flag è vero la riga contiene la dimensione del prossimo test case
            sizeOfSequence = int(line) #la assegno castandola ad intero
            newTestCase = False
        else:
            tempList.append(int(line))
            sizeOfSequence -= 1
            if sizeOfSequence == 0: #quando finisco il numero di elementi del test case
                testList.append(tempList) #aggiungiamo la lista del singolo test case a quella di tutti i test case
                tempList = [] #svuoto la lista temporanea
                newTestCase = True #indico che la prossima riga rappresenterà l'inizio di un nuovo testcase

    return testList

if __name__ == "__main__":

    testList = readTest("./testCases1.txt")

    for test in testList:

        inv_count = sortInvCount(test, 0, len(test)-1)
        print(inv_count)
```

```
5
9
1
0
5
4
3
1
2
3
0
```

COMPLESSITA' ASINTOTICA:

$O(n \lg n)$ poichè utilizziamo il merge sort per ordinare l'array.

Problema 1.2

```
def longestCommonPrefix(str1, str2): #prendo 2 stringhe in input da comparare

    result = ""

    minLenght = min(len(str1), len(str2))

    i = 0

    while i < minLenght: #scorro fintantochè non arrivo alla fine della stringa più corta

        if str1[i]!=str2[i]:#esco dal while appena un carattere non corrisponde
            break

        result += str1[i]
        i = i+1

    return result #restituisco il prefisso comune più lungo

def findLCP(array, start, end): #approccio divide et impera (in input do l'array di test, indice iniziale e finale dell'array di strighe)

    if start == end: #c'è solo una stringa nell'array
        return array[start] #restituisco la stringa come prefisso comune più lungo

    if(end > start): #se ci sono almeno 2 stringhe nell'array

        # modo sicuro per calcolare il punto medio dell'array
        # evitando un possibile overflow di intero
        mid = start + (end - start) // 2 #// divisione intera

        str1 = findLCP(array, start, mid) #Chiamiamo la funzione ricorsivamente per prima metà dell'array
        str2 = findLCP(array, mid+1, end) #Chiamiamo la funzione ricorsivamente per la seconda metà dell'array

        return longestCommonPrefix(str1, str2) #Troviamo il prefisso comune più lungo per le 2 stringhe

    return "" #se nessuna delle condizioni precedenti è soddisfatta -> restituiamo una stringa vuota

def readTest(path):

    testFile = open(path)

    testList = []   #lista per memorizzare i test case letti da file

    numeroTestCase = int(testFile.readline()) #leggo il numero di test

    tempList = []   #lista per memorizzare le stringhe di un singolo test case.

    for line in testFile:       #leggo una alla volta ogni riga del file
        if line.strip() == "END": #se la riga contiene "END" (con strip rimuovo gli spazi all'inizio e alla fine) significa che è la fine d
            testList.append(tempList)   #aggiungo la lista del singolo test case alla lista dei test case
            tempList = []   #svuoto la lista temporanea del singolo test case
        else:
            tempList.append(line.strip())   #aggiungo la riga a tempList

    return testList
```

```
if __name__ == "__main__":

    testList = readTest("./testCases.txt")

    for test in testList: #per ogni test case
        result = findLCP(test, 0, len(test) -1) #trovo il prefisso comune più lungo
        print(result)
```

In questo algoritmo viene discusso un approccio "divide et impera". Per prima cosa dividiamo gli array di stringhe in due parti. Poi facciamo lo stesso per la parte sinistra e successivamente per la parte destra. Lo faremo fino a quando tutte le stringhe non saranno di lunghezza 1. A questo punto, inizieremo a conquistare restituendo il prefisso comune delle stringhe di sinistra e di destra.

```
4
apple
applied
april
ape
END
veleno
velato
vero
vento
END
intonaco
intontito
integro
END
reale
resto
reame
resa
rito
END
```

COMPLESSITA' ASINTOTICA:

Se **n** è il numero di stringhe da confrontare, e **M** è la lunghezza del prefisso comune più lungo da confrontare, notiamo che la complessità è come quella del MergeSort tranne che per il confronto interno, che non è più fatto tra numeri, ma fra stringhe, dunque non è più costante ma è $O(m)$.

Dunque ci troviamo che la ricorrenza diventa del tipo: $T(n) = 2T(\frac{n}{2}) + O(m)$, dove il costo per livello dell'albero di ricorrenze è $2^i \cdot m$, dunque:

$$\sum_{i=0}^{\log n} 2^i \cdot m = m \sum_{i=0}^{\log n} 2^i = m \cdot \frac{2^{\log n+1} - 1}{2 - 1} = m \cdot (2n - 1)$$
$$\implies \Theta(n \cdot m)$$

Problema 1.3

L'idea fondamentale su cui si basa l'algoritmo di inserimento è quella di effettuare un normale inserimento dell'elemento senza rispettare il vincolo di priorità, ed in seguito correggere la posizione dell'elemento inserito.

Per preservare le proprietà dell'albero binario di ricerca utilizziamo le rotazioni.

```
from random import randint
RAND_MAX=100

class Node(object):
    #costruttore del nodo
    def __init__(self, key, data, priority=None):
        self.key = key
        self.data = data
        #assegno ad x una priorità casuale
        if priority is not None:
            self.priority = priority
        else:
```

```
            self.priority=randint(1, RAND_MAX)
        #creo dei puntatori ai figli
        self.left = None
        self.right = None

    def __repr__(self):
        return f"Key: {self.key}, Priority: {self.priority}"

class Treap(object):
    def __init__(self):
        self.root = None #Inizialmente è vuoto

    def insert(self, key, data, priority=None):
        node = Node(key, data, priority)
        self.root = self._insert(self.root, node)

    def _insert(self, root, node):
        if root is None:
            return node

        #Se la chiave è più piccola di quella della radice
        if node.key < root.key:

            #inseriamo nel sottoalbero di sinistra
            root.left = self._insert(root.left, node)
            #questo inserimento ha violato le proprietà dell'heap?
            if root.left.priority < root.priority:
                root = self._rightRotate(root)

        #altrimenti
        else:

            #inseriamo nel sottoalbero di destra
            root.right = self._insert(root.right, node)
            #questo inserimento ha violato le proprietà dell'heap?
            if root.right.priority < root.priority:
                root = self._leftRotate(root)

        return root

    def _rightRotate(self, root):
        #prendiamo il figlio sinistro e il figlio destro di questo
        #       root
        #      /     \
        #    fsx      fdx
        #   /   \
        #ffsx   ffdx

        fsx = root.left

        #Rotazione: disegniamo cosa deve succedere
        #      root         |      fsx
        #     /    \             /     \
        #   ffdx     fdx    |   ffsx    root
        #                           /     \
        #                   |       ffdx     fdx

        root.left = fsx.right
        fsx.right = root

        return fsx

    def _leftRotate(self, root):
        #prendiamo il figlio destro e il figlio sinistro di questo
        #       root
        #      /     \
        #    fsx      fdx
        #           /   \
        #          ffsx   ffdx

        fdx = root.right

        #Rotazione: disegniamo cosa deve succedere
        #      root          |       fdx
        #      /   \               /    \
        #   fsx     ffsx     |   root    ffdx
        #                       /  \
        #                   | fsx  ffsx
```

```
        root.right = fdx.left
        fdx.left = root

        return fdx

    def visualize(self):
        if self.root is None:
            print("Treap is empty")
            return

        current_level = 1
        current_nodes = [self.root]

        while current_nodes:
            #Stampo il livello
            print(f"Level {current_level}: ", end="")
            next_nodes = []

            for node in current_nodes:
                print(f"({node.key}, {node.priority})", end=" ")

                if node.left:
                    next_nodes.append(node.left)
                if node.right:
                    next_nodes.append(node.right)

            print()  # Newline
            current_level += 1
            current_nodes = next_nodes

def run():
    t = Treap()
    t.insert(4, 4)
    print(f"\nNuovo inserimento...")
    t.visualize()
    t.insert(5, 5)
    print(f"\nNuovo inserimento...")
    t.visualize()
    t.insert(6, 6)
    print(f"\nNuovo inserimento...")
    t.visualize()
    t.insert(1, 1)
    print(f"\nNuovo inserimento...")
    t.visualize()
    t.insert(2, 2)
    print(f"\nNuovo inserimento...")
    t.visualize()
    t.insert(3, 3)
    print(f"\nNuovo inserimento...")
    t.visualize()
    return t

if __name__ == "__main__":
    treap_instance = run()
```

```
PS C:\Users\ladyc\Desktop\Algorithms Data Structures> python [prova.py](http://prova.py/)

Nuovo inserimento...
Level 1: (4, 48)

Nuovo inserimento...
Level 1: (4, 48)
Level 2: (5, 99)

Nuovo inserimento...
Level 1: (4, 48)
Level 2: (6, 64)
Level 3: (5, 99)

Nuovo inserimento...
Level 1: (4, 48)
Level 2: (1, 64) (6, 64)
Level 3: (5, 99)
```

```
Nuovo inserimento...
Level 1: (4, 48)
Level 2: (2, 54) (6, 64)
Level 3: (1, 64) (5, 99)

Nuovo inserimento...
Level 1: (4, 48)
Level 2: (2, 54) (6, 64)
Level 3: (1, 64) (3, 82) (5, 99)
```