Limiti asintotici e ricorrenze

Esercizio 1

$T(n) = 3T(n/4) + n^2$

Soluzione: Utilizziamo il master theorem. Rispetto allo schema generale:

- $a = 3$ (numero di chiamate ricorsive)
- $b = 4$ (fattore di divisione del problema)
- $f(n) = n^2$ (costo delle operazioni non ricorsive)

Calcoliamo $\log_b(a) = \log_4(3) \approx 0.79$

Confrontiamo $n^{\log_b(a)}$ con $f(n) = n^2$:

- $\log_4(3) < 2$ quindi $f(n) = \Omega(n^{\log_b(a) + \epsilon})$ per $\epsilon > 0$
- È facile verificare che vale la condizione di regolarità $af(n/b) \leq cf(n)$

Quindi siamo nel caso 3 del master theorem e $T(n) = \Theta(n^2)$

Esercizio 2

$T(n) = T(n-1) + n \log n$

Soluzione: Dimostriamo per sostituzione che $T(n) = O(n^2)$.
Ipotizziamo che $T(n) \leq cn^2$ per qualche costante $c > 0$.

$T(n) = T(n-1) + n \log n$
$\leq c(n-1)^2 + n \log n$
$= cn^2 - 2cn + c + n \log n$
$\leq cn^2$ quando $n \log n \leq 2cn - c$

La disuguaglianza è verificata per $c \geq 1$ e n sufficientemente grande.

Esercizio 3

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

Soluzione: Dimostriamo per sostituzione che $T(n) = O(n \log n)$.
Ipotizziamo che $T(n) \leq cn \log n$ per qualche costante $c > 0$.

$T(n) = T(n/3) + T(2n/3) + n$
$\leq c(n/3)\log(n/3) + c(2n/3)\log(2n/3) + n$
$= cn/3(\log n - \log 3) + 2cn/3(\log n - \log(3/2)) + n$
$= cn \log n - cn/3 \log 3 - 2cn/3 \log(3/2) + n$
$\leq cn \log n$ quando $c \geq 3$

La disuguaglianza è verificata per $c \geq 3$ e n sufficientemente grande.

Divide et impera e Ricorsione

Esercizio 4

Soluzione:

```
def equilibrio(A, l, r):
    if r - l < 2:  # caso base: array troppo piccolo
        return -1

    mid = (l + r) // 2
    left_sum = sum(A[l:mid])
    right_sum = sum(A[mid+1:r])

    if left_sum == right_sum:
        return mid
    elif left_sum < right_sum:
        # prova nella metà destra
        return equilibrio(A, mid, r)
    else:
        # prova nella metà sinistra
        return equilibrio(A, l, mid)
```

Complessità: $T(n) = T(n/2) + \Theta(n)$ per il calcolo delle somme
Dal master theorem otteniamo $T(n) = \Theta(n \log n)$

Esercizio 5

Soluzione:

i. Dimostrazione: Per assurdo, supponiamo che non esistano picchi locali. Consideriamo il primo elemento A[1]: se non è un picco locale, allora A[1] < A[2]. Se A[2] non è un picco locale, allora A[2] < A[3], e così via. Ma questo implicherebbe una sequenza strettamente crescente infinita, impossibile in un array finito.

ii.

```
def picco(A, l, r):
    if r - l < 2:
        return -1

    mid = (l + r) // 2

    if A[mid-1] < A[mid] and A[mid] > A[mid+1]:
        return mid
    elif A[mid-1] > A[mid]:
        return picco(A, l, mid)
    else:
        return picco(A, mid, r)
```

iii. Complessità: $T(n) = T(n/2) + \Theta(1)$
Dal master theorem otteniamo $T(n) = \Theta(\log n)$

Alberi Binari di Ricerca e Alberi e ricorsione

Esercizio 6

Soluzione:

```
def updateSum(x):
    if x is None:
        return 0
    x.sum = x.key + updateSum(x.left) + updateSum(x.right)
    return x.sum

def insert(T, k):
    # Inserimento standard in BST
    if T is None:
        return Node(k)

    if k < T.key:
        T.left = insert(T.left, k)
    else:
        T.right = insert(T.right, k)

    # Aggiorna sum dopo l'inserimento
    T.sum = T.key + (T.left.sum if T.left else 0) + (T.right.sum if T.right
else 0)
    return T
```

Complessità:

- updateSum: Θ(n) dove n è il numero di nodi
- insert: O(h) dove h è l'altezza dell'albero

Esercizio 7

Soluzione:

```
def countNodesAtLevel(root, level):
    if root is None:
        return 0
    if level == 0:
        return 1
    return countNodesAtLevel(root.left, level-1) + 
countNodesAtLevel(root.right, level-1)

def maxComplete(root):
    level = 0
    while True:
        nodes = countNodesAtLevel(root, level)
        if nodes < 2**level:
            return level - 1
        level += 1
```

Correttezza: L'algoritmo conta il numero di nodi ad ogni livello e si ferma quando trova un livello incompleto.

Complessità: O(n) dove n è il numero di nodi dell'albero.

Esercizio 8

Soluzione:

```
def updateMaxPath(x):
    if x is None:
        return 0
    if x.left is None and x.right is None:
        x.maxPath = 0
        return 0

    leftPath = updateMaxPath(x.left)
    rightPath = updateMaxPath(x.right)
    x.maxPath = 1 + max(leftPath, rightPath)
```

```python
    return x.maxPath

def delete(T, k):
    if T is None:
        return None

    if k < T.key:
        T.left = delete(T.left, k)
    elif k > T.key:
        T.right = delete(T.right, k)
    else:
        # Nodo con un solo figlio o foglia
        if T.left is None:
            return T.right
        elif T.right is None:
            return T.left

        # Nodo con due figli
        temp = minValueNode(T.right)
        T.key = temp.key
        T.right = delete(T.right, temp.key)

    # Aggiorna maxPath dopo la cancellazione
    updateMaxPath(T)
    return T
```

Complessità:

- updateMaxPath: Θ(n) dove n è il numero di nodi
- delete: O(h) dove h è l'altezza dell'albero, più O(n) per l'aggiornamento di maxPath