Introduzione agli Algoritmi
Esame Scritto a canali unificati con spunti per la soluzione

docenti: T. Calamoneri, A. Monti
Sapienza Università di Roma

16 Settembre 2025

Esercizio 1 (10 punti): Nella funzione riportata qui sotto, i valori delle costanti a, b e c sono determinati dalla propria *matricola* e corrispondono rispettivamente **alla prima, seconda e terza cifra** che compaiono nella propria matricola.

```
def es1(n):
    if n <= 1:
        return 1
    s = 1
    for i in range(c+1):
        s = s*n
    tot = 0
    while tot < s :
        tot += 1
    for i in range (a+1):
        tot += (c+1)*es1(n//(b+2))
    return tot
```

Si sostituiscano i parametri a, b e c con i valori corrispondenti dati dalla propria matricola e si risponda alle seguenti domande:

1. Qual è l'equazione di ricorrenza che descrive il costo computazionale della funzione $es1(n)$? Perché?
2. È possibile applicare il teorema principale per risolverla? motivare accuratamente la risposta.
3. Risolvere l'equazione di ricorrenza con il metodo dell'albero, dandone la rappresentazione grafica e dettagliando i calcoli.

1. **La funzione esegue $a+1$ chiamate ricorsive su input di dimensione $n//(b+2)$. Il costo delle istruzioni al di fuori delle chiamate ricorsive è dato da**
   - **un ciclo for che itera $c+1$ volte. Con costo $\Theta(c+1) = \Theta(1)$ (essendo c costante).**
   - **un ciclo $while$ che esegue $s = n^{c+1}$ iterazioni con costo $\Theta(n^{c+1})$.**

   Ne segue la sequente relazione di ricorrenza (nella forma generale)

   $T(n) = (a+1)T\left(\frac{n}{b+2}\right) + \Theta(n^{c+1}),$ **con caso base** $T(0) = T(1) = \Theta(1)$.

2. **Possiamo applicare il teorema principale in quanto ne valgono tutte le ipotesi, e confrontiamo $f(n) = \Theta(n^{c+1})$ con $n^{log_{b+2}(a+1)}$. A seconda di valori di a, b e c si rientra in uno dei seguenti casi:**

$$T(n) = \begin{cases} \Theta(n^{log_{b+2}(a+1)}) & \textbf{se } log_{b+2}(a+1) > c+1 \textbf{ caso 1 del teor.} \\ \Theta(n^{c+1}\log n) & \textbf{se } log_{b+2}(a+1) = c+1 \textbf{ caso 2 del teor.} \\ \Theta(n^{c+1}) & \textbf{se } log_{b+2}(a+1) < c+1 \textbf{ caso 3 del teor.} \end{cases}$$

Si ricordi che nel terzo caso del teorema c'è da verificare anche la condizione aggiuntiva e che, ovviamente, ciascuno si troverà a gestire un solo caso, che varierà a seconda della propria matricola.

3. **Si omette qui la soluzione tramite metodo dell'albero.**

Esercizio 2 (10 punti): Sia dato un array A di n interi, organizzato rispetto alla propria matricola come segue:

- se la terza cifra è pari o nulla, i valori in A letti da sinistra a destra sono dapprima crescenti e poi decrescenti;
- se la terza cifra è dispari, i valori in A letti da sinistra a destra sono dapprima decrescenti e poi crescenti.

Le due sequenze, quella crescente e quella decrescente, che costituiscono A contengono sempre almeno due elementi ciascuna. Non serve memorizzare la matricola, ma basta usare la terza cifra opportunamente per definire il proprio input.

Progettare un algoritmo **iterativo** che restituisca il minimo ed il massimo di A in tempo $O(\log n)$.

Esempio: supponendo che la propria matricola sia 1234567:
un input ammissibile è A : 10 7 6 3 5 8 9 12.
Supponendo che la propria matricola sia 0123456:
un input ammissibile è A : 3 5 8 10 12 9 7 6.
In entrambi i casi l'output sarà la coppia $(3, 12)$.

Dell'algoritmo proposto:

1. si scriva lo pseudocodice opportunamente commentato in modo da chiarire il senso delle istruzioni,
2. si giustifichi formalmente il costo computazionale,

3. se ne dettagli il funzionamento su un array A di lunghezza $n = 12$ in cui la sequenza crescente contenga almeno 9 interi e siano necessarie almeno tre iterazioni del ciclo principale.

E' facile convincersi che l'input costituito da due sequenze ordinate in senso opposto implica che uno tra massimo e minimo si troverà in prima o in ultima posizione, mentre l'altro in corrispondenza dell'intersezione delle due sequenze in A.
Ad esempio, se A è costituito da una sequenza crescente seguita da una decrescente, il minimo sarà min$\{A[0], A[n-1]\}$. Per trovare il massimo, sarà necessario modificare l'algoritmo di ricerca binaria, in modo da determinare l'elemento $A[m]$ che sia maggiore tanto di $A[m-1]$ che di $A[m+1]$.
Segue un codice Python per il caso di terza cifra della matricola pari o nulla. L'altro caso si può scrivere analogamente.

```
def trova_max_min(A):
    # caso in cui A è prima crescente e poi decrescente
    n = len(A)
    min=min(A[O], A[n-1])
    # il min è il primo o l'ultimo elemento di A
    max = Ric_Bin_Modificata(A, O, n-1)
    # il max viene trovato con una modifica della ric. bin. iter.
    return(min, max)


def Ric_Bin_Modificata(A, inizio, fine):
    while (inizio < fine):
       m=(fine+inizio)//2
       if m>O and m < len(A)-1 and A[m]>A[m-1] and A[m]>A[m+1]:
            # trovato il massimo
            return A[m]
```

```
    # il A[m] è maggiore di entrambi i suoi vicini è il max
    if A[m-1]<A[m]<A[m+1]:
      inizio = m+1
      # se A[m] è nella sequenza crescente andiamo a destra
    if A[m-1]>A[m]>A[m+1]:
      fine = m-1
      # se A[m] è nella sequenza decrescente andiamo a sinistra
  return A[m]
```

Sulla base di quanto studiato per l'algoritmo di ricerca binaria (che va qui brevemente esposto), questo algoritmo ha costo $O(\log n)$**.**

Esercizio 3 (10 punti): Sia dato un albero T memorizzato tramite record e puntatori; ogni nodo contenga un campo val, che memorizza una delle 10 cifre decimali, ed i due puntatori $left$ e $right$ ai figli sinistro e destro.
Si scriva un algoritmo **ricorsivo** che risponda *True* se esiste un cammino radice-foglia la somma delle cui chiavi sia pari alla somma delle cifre dalla propria matricola, *False* altrimenti. Il costo computazionale deve essere $O(n)$.

Dell'algoritmo proposto:

a) si descriva a parole l'algoritmo proposto.

b) si scriva lo pseudocodice,

b) si giustifichi il costo computazionale, pur senza risolvere formalmente la ricorrenza che lo caratterizza.

NOTA BENE: La funzione proposta deve prendere in input il puntatore alla radice dell'albero e non la matricola; inoltre **non** deve far uso di variabili globali.

a. L'idea è semplice: si effettua una visita dell'albero e, nella discesa, si accumula la somma delle cifre lungo i

cammini. Giunti ad una foglia, si controlla se la somma cumulata coincide con la somma delle cifre della matricola. In caso positivo si restituisce *True*, altrimenti *False*. Nella risalita, i nodi interni restituiscono l'OR di quanto ricevuto dai propri figli.
Non serve memorizzare l'intera matricola, ma direttamente la somma delle sue cifre.

b. Ecco un possibile pseudocodice a cui si accede la prima volta con il puntatore p al nodo corrispondente alla radice dell'albero, la somma delle cifre della matricola mat (che noi abbiamo inserito per generalizzare la soluzione ma che è superflua, visto che ognuno potrà usare una costante, che corrisponde alla somma delle cifre della propria matricola) e una variabile intera s contenente la somma corrente dei valori dei nodi lungo il cammino, inizializzata a zero:

```
def es(p, mat, s = 0)
    if p is None:
        return False
    if p.left is None and p.right  is None:
        if mat == s + p.val:
            return True
        return False
    return es(p.left, mat, s + p.val) or es(p.right, mat, s + p.val)
```

c. Il costo computazionale è quello di una visita, dove in ogni nodo viene fatto lavoro costante. Il costo di una visita di questo tipo è

$$T(n) = T(k) + T(n-k-1) + O(1)$$

con $0 \leq k < n$. Questa equazione di ricorrenza è ben nota ed ha come soluzione $O(n)$. Non è richiesto che la soluzione sia calcolata formalmente.