Introduzione agli Algoritmi
Seconda Prova intermedia a canali unificati con spunti per la soluzione

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

29 Maggio 2025

Esercizio 1 (10 punti): Si risponda alle seguenti domande, dando una giustificazione:

1. a. Qual è la limitazione inferiore al costo computazionale di un algoritmo basato sui confronti?

   b. È possibile progettare un algoritmo di ordinamento il cui costo computazionale nel caso peggiore sia minore di questa limitazione inferiore? Perché?

2. In un Max-heap contenente n chiavi:

   a. In quale nodo si trova la chiave minima?

   b. Qual è la sua altezza?

3. Data una lista concatenata semplice L (in cui cioè ogni nodo contiene un campo informativo e un puntatore al nodo successivo) qual è il costo computazionale per cancellare l'ultimo elemento della lista?

4. Dato un albero binario con n nodi:
   a. Quali sono i valori che può assumere la sua altezza h?
   b. Se l'albero fosse ternario, cioè ogni nodo avesse al più 3 figli, quali valori potrebbe assumere la sua altezza?
5. Dato un albero binario di ricerca qualunque:
   a. Quanto costa determinare la chiave di valore minimo?
   b. Che proprietà deve avere il nodo in cui si trova la chiave di valore massimo?

Le seguenti risposte vogliono solo dare un'idea della soluzione e non vanno considerate come esaustive:

1. **La limitazione inferiore al costo computazionale di un algoritmo basato sui confronti è $\Omega(n \log n)$, che si dimostra sfruttando il concetto di albero delle decisioni. Esistono algoritmi che richiedono costo inferiore nel caso peggiore (ad esempio Counting Sort), perché non si basano sui confronti e quindi richiedono che siano verificate delle ipotesi aggiuntive.**
2. **Assumendo di trovarsi in presenza di un heap massimo (quello utilizzato per l'algoritmo di HeapSort):**
   - **La chiave massima si trova nella radice dell'heap;**
   - **La chiave minima si trova in una delle foglie dell'albero e non è possibile determinare la sua posizione esatta;**
   - **L'altezza è sempre logaritmica. Più precisamente:**

$$h = \lfloor \log_2 n \rfloor$$

infatti, un heap binario è un albero binario completo o quasi completo, ovvero un albero in cui tutti i livelli, tranne eventualmente l'ultimo, sono completamente riempiti, e i nodi dell'ultimo livello sono inseriti da sinistra verso destra.
In un albero binario di questo tipo:

- **Il numero massimo di nodi fino ad altezza h è $2^{h+1} - 1$.**
- **Pertanto, per un dato numero di nodi n, l'altezza dell'heap è il massimo valore intero h tale che:**

$$2^h \leq n < 2^{h+1}$$

che equivale a:

$$h = \lfloor \log_2 n \rfloor$$

3. **Il costo computazionale per cancellare l'ultimo elemento di una lista concatenata semplice è $O(n)$ dove n è il numero di elementi nella lista. Questo perché in una lista concatenata semplice:**

   - **Ogni nodo ha un puntatore solo verso il nodo successivo.**
   - **Non esiste un puntatore diretto al nodo precedente.**

   Per cancellare l'ultimo nodo:

   (a) **È necessario scorrere tutta la lista partendo dalla testa.**

   (b) **Occorre trovare il penultimo nodo, cioè quello il cui puntatore punta all'ultimo nodo.**

   (c) **Solo allora si può modificare il puntatore del penultimo nodo per renderlo nullo.**

Poiché bisogna attraversare l'intera lista per trovare il nodo precedente all'ultimo, il tempo richiesto cresce linearmente con il numero di nodi.

4. **(a) Albero binario: In un albero binario con n nodi, l'altezza h dell'albero può assumere i seguenti valori:**

   - **Il valore minimo di altezza si verifica quando l'albero è completamente bilanciato, ossia quando ogni livello dell'albero è pieno tranne l'ultimo. In questo caso, l'altezza è circa $\log_2 n$, quindi: $h_{\textsf{min}} = \lfloor \log_2 n \rfloor$; in altre parole, $h = \Omega(\log n)$.**
   - **Il valore massimo di altezza si verifica quando l'albero è completamente sbilanciato e si presenta come una lista (ogni nodo ha al più un figlio). In questo caso, l'altezza è pari al numero totale di nodi meno uno, cioè: $h_{\textsf{max}} = n - 1$; in altre parole, $h = O(n)$.**

   Pertanto, i valori che può assumere l'altezza h di un albero binario con n nodi sono compresi tra:

   $$h_{\textsf{min}} = \lfloor \log_2 n \rfloor \quad \textsf{e} \quad h_{\textsf{max}} = n - 1.$$

   In altre parole, h sarà un $\Omega(\log n)$ ed un $O(n)$.

   (b) Albero ternario: Se l'albero è ternario, cioè ogni nodo ha al massimo 3 figli, la situazione è simile, ma con una base diversa per il calcolo dell'altezza:

   - **Il valore minimo di altezza si verifica quando l'albero è completamente bilanciato, cioè ogni nodo ha esattamente 3 figli fino all'ultimo livello. In questo caso, l'altezza è circa $\log_3 n$,**

quindi: $h_{\textsf{min}} = \lfloor \log_3 n \rfloor$**; in altre parole** $h = \Omega(\log n)$ **(la base del logaritmo non è importante nella notazione asintotica, poiché influisce per una costante moltiplicativa).**

- **Il valore massimo di altezza si verifica quando l'albero è completamente sbilanciato e ogni nodo ha al massimo un figlio. In questo caso, l'altezza è pari a** $n-1$**; in altre parole** $h = O(n)$**.**

Pertanto, i valori che può assumere l'altezza h **di un albero ternario con** n **nodi sono compresi tra:**

$$h_{\textsf{min}} = \lfloor \log_3 n \rfloor \quad \textsf{e} \quad h_{\textsf{max}} = n - 1$$

In altre parole, anche in questo caso h **sarà un** $\Omega(\log n)$ **ed un** $O(n)$**.**

5. **In un albero binario di ricerca, la ricerca del minimo oppure del massimo si esegue in tempo** $O(h)$**, dove** h **è un** $\Omega(n)$ **ed un** $O(n)$**. La chiave di valore massimo è contenuta nel nodo più a destra privo di figlio destro.**

Esercizio 2 (10 punti):

Sia data una lista puntata semplice L con chiavi numeriche non negative. Definiamo *picchi* di una lista quei valori che sono preceduti e seguiti nella lista da valori più piccoli della loro metà. Per definizione, il primo e l'ultimo elemento della lista non possono essere picchi.

Ad esempio, nella lista $p \to 4 \to 9 \to 12 \to 36 \to 16 \to 23 \to 87 \to 34 \to 18 \to 64 \to 33$, 36 e 87 sono picchi, perché sia 12 che 16 sono minori di $36/2 = 18$ e sia 23 che 34 sono minori di $87/2 = 43,5$; 64 non è un picco perché, nonostante sia minore dei valori che lo precedono e seguono, non vale la stessa disuguaglianza per $64/2 = 32$.

Si progetti un algoritmo **iterativo** che, dato il puntatore p al primo elemento di L, restituisca il numero di picchi presenti nella lista.

Dell'algoritmo proposto:

a) si scriva lo pseudocodice opportunamente commentato, in modo da chiarire il senso delle istruzioni,

b) si giustifichi formalmente il costo computazionale.

Basta scorrere l'intera lista puntata e, per ciascun elemento, confrontare la metà del valore della sua chiave con i valori dell'elemento che precede e che segue.
Un possibile modo di procedere è quello di considerare tre puntatori che scorrono la lista su elementi consecutivi; se sono tutti e tre diversi da None, si verifica se la condizione è soddisfatta ed, in caso affermativo, si incrementa un contatore.

Il codice Python della procedura appena descritta è il seguente:

```
def es2(p):
    # inizializzo il contatore  dei picchi
    picchi = 0
    if p == None or p.next == None:
        # la lista di input ha meno di tre elementi
        return picchi
    # inizializzo i  tre puntatori consecutivi
    # ai primi  tre  nodi della lista
    a, b, c = p, p.next, p.next.next
    while c!= None:
        # scorro la lista
        if a.key < b.key/2 and b.key/2 > c.key:
```

```
        # il nodo puntato da b è un picco e va
        # incrementato il contatore  dei picchi
        picchi += 1
    a, b, c = b, c, c.next
# restituisco il numero di  picchi trovati
return picchi
```

È facile convincersi che il costro computazionale di questo algoritmo è $\Theta(n)$, ed è necessario giustificarlo formalmente calcolando il costo di ciascuna istruzione.

Esercizio 3 (10 punti): Sia T un albero binario qualunque memorizzato tramite record e puntatori. Ogni nodo dell'albero contenga un valore intero come chiave. Si progetti un algoritmo **ricorsivo** che, dato il puntatore r alla radice dell'albero, restituisca $True$ se l'albero è vuoto oppure se la chiave di ogni nodo interno dell'albero risulta essere la somma delle chiavi dei nodi foglia presenti nel suo sottoalbero, restituisca $False$ altrimenti. Il costo computazionale dell'algoritmo proposto deve essere $O(n)$, dove n è il numero di nodi dell'albero.

Dell'algoritmo proposto:

a) si scriva lo pseudocodice opportunamente commentato, in modo da chiarire il senso delle istruzioni,

b) si giustifichi formalmente il costo computazionale dando la ricorrenza che lo caratterizza e poi la sua soluzione.

NOTA BENE: nello pseudocodice della funzione ricorsiva **non** si deve far uso di variabili globali.

Durante la visita ciascun nodo restituisce $True$ se è radice di un albero che soddisfa le richieste del problema, $False$ altrimenti. Se il puntatore fa riferimento ad un albero vuoto allora la risposta è $True$, in caso contrario un nodo restituisce $True$ solo se la somma delle chiavi dei suoi figli dà la sua

chiave e riceve $True$ da entrambi i suoi sottoalberi.
Si osservi che non è necessario ritornare anche la somma delle chiavi nelle foglie, poiché, se la chiamata sul nodo puntato da p restituisce $True$, tale somma è semplicemente p.key.
Il codice Python della procedura appena descritta è il seguente:

```
def es3(p):
if p == None or (p.left==p.right==None):
        # l'albero è vuoto o è costituito da una foglia
        return True
    if es3(p.left) == False or es3(p.right) == False:
        # almeno uno dei due sottoalberi non rispetta
        # il vincolo del problema
        return False
    s = 0
    if p.left != None:
        s += p.left.key
    if p.right != None:
        s += p.right.key
    if p.key == s:
        # la somma s delle foglie dei due sottoalberi
        # coincide con il valore della radice p.key
        return True
    return False
```

La relazione di ricorrenza che descrive il costo dell'algoritmo è la seguente:

$$T(n) = T(h) + T(n-1-h) + O(1) \textbf{se } n > 1, O(1) \textbf{ altrimenti}$$

con $0 \leq k < n$. Risolta con il metodo di sostituzione, dà come risultato $\Theta(n)$ ed i calcoli vanno dettagliati.