Domanda 37 Scrivere una funzione `IsABR(A)` che dato in input un array di interi `A[1..n]`, interpretato come albero binario (come nel caso degli heap, ogni `A[i]` è un nodo con figlio sinistro e destro `A[2i]` e `A[2i+1]`) verifica se `A` è un albero binario di ricerca. Valutarne la complessità.

Soluzione: Una versione iterativa, di tempo $O(n \log n)$ consiste nel verificare che ogni nodo sia minore o uguale dei nodi dei quali è nel sottoalbero sinistro e maggiore o uguale di quelli in cui è nel sottoalbero destro

```
IsABR(A)

  n =  A.length
  i = n
  isABR = true
  while ((i > 1) and isABR)
    j = i
    while j>1                         # risale l'albero
         if even(j)                   # i nodi di indice pari sono figli sx
           isABR = isABR and A[i] <= A[j/2]
         else                         # quelli di indice dispari figli dx
           isABR = isABR and A[i] >= A[j/2]
         j = j/2
    i = i-1
  return isABR
```

E varie versioni ricorsive, la prima per ogni nodo verifica che sia maggiore o uguale al massimo del sottoalbero destro e minore o uguale del minimo del sottoalbero destro. Dato che prima controlla che tali sottoalberi siano ABR può cercare minimo e massimo in tempo $\log n$ cosa che assicura che il costo complessivo è lineare. Infatti $T(n) = 2T(n/2) + 2 \log n$ che secondo il master theorem porta a $T(n) = \Theta(n)$.

```
 IsABRO(A,n)
  return IsABRO_rec(A,n,1)

IsABRO_rec(A,n,i):
  if i > n
    return True
  else:
    isABRL = IsABRO_rec(A,n,2*i)
    isABRR = IsABRO_rec(A,n,2*i+1)
    M = max(A,n,2*i)                  # massimo del sottoalbero sx (assumendolo
                                        ABR, gia' controllato!)
    m = min(A,n,2*i+1)                # minimo del sottoalbero dx (assumendolo
                                        ABR, gia' controllato!)
    return isABRL and isABRR
                  and A[i] >= M
                  and A[i] <= m


Tmin(A,n,i):
  if i>n:
    return +INFTY
  else:
    while (2*i <=n):
      i = 2*i    # scende a sx

    return A[i]
```

Esercizio 11 Sia dato un albero i cui nodi contengono una chiave intera $x.key$, oltre ai campi $x.l$, $x.r$ e $x.p$ che rappresentano rispettivamente il figlio sinistro, il figlio destro e il padre. Si definisce *grado di squilibrio* di un nodo il valore assoluto della differenza tra la somma delle chiavi nei nodi foglia del sottoalbero sinistro e la somma delle chiavi dei nodi foglia del sottoalbero destro. Il grado di squilibrio di un albero è il massimo grado di squilibrio dei suoi nodi.

Fornire lo pseudocodice di una funzione `sdegree(T)` che calcola il grado di squilibrio dell'albero T (si possono utilizzare funzioni ricorsive di supporto). Valutare la complessità della funzione.

Soluzione:

```
// computes the sum of the leaf nodes of the subtree and the sdegree
// for the node x (returns two values)

sdegree(x)

  if (x == nil)
     sum = 0
     degree = 0
  elif (x.left == nil) and (x.right = nil)      # leaf
     sum = x.key
     degree = 0
  else
     suml, degreel = sdegree(x.l)
     sumr, degreer = sdegree(x.r)
     sum = suml + sumr
     degree = max { degreel, degreel, abs(suml - sumr) }

  return sum, degree
```

Domanda 23 Scrivere una funzione `IsMaxHeap(A)` che dato in input un array di interi `A[1..n]` che verifica se `A` è organizzato a max-heap e ritorna un corrispondente valore booleano. Valutarne la complessità.

Soluzione: Versione ricorsiva

```
IsMaxHeap(A,i,j) // verifica se l'array A[i,j] e' un max-heap
                    (o meglio, se contiene un max-heap radicato in i,
                     dato che al passo ricorsivo non tutto [i,j]
                     conterra' il sottoalbero scelto
  l = 2*i
  r = 2*i+1
  if l<j
    leftOk = A[i] >= A[l] and IsMaxHeap(l, j)
  else
    leftOk = true

  if r<j
    rightOk = A[i] >= A[r] and IsMaxHeap(r, j)
  else
    rightOk = true

  return leftOk and rightOk
```

Per quanto riguarda la complessità si osservi che $T(n) = c+2T(n/2)$ per un'opportuna costante c e quindi, utilizzando il master theorem, si deduce che la complessità è $O(n)$.

Versione iterativa

```
IsMaxHeap(A,n) // verifica se l'array A[1..n] e' un max-heap
  i=2
  while (i <= n) and (A[i] <= A[i/2])
      i++
  return (i==n+1)
```

Domanda 14 Dare una soluzione asintotica per la ricorrenza $T(n) = 3T(n/2) + n(n+1)$.

Soluzione: Si usa il master theorem con $a = 3$, $b = 2$, $f(n) = n(n+1)$. Si deve confrontare $f(n)$ con $n^{\log_b a}) = n^{\log_2 3}$ ed essendo che $\log_2 3 < 2$ per qualunque $0 < \epsilon < 2 - \log_2 3$ si vede facilmente che $f(n) = \Omega(n^{\log_2 3+\epsilon})$.

Per concludere che $T(n) = \Theta(f(n)) = \Theta(n^2)$ usando il caso 3 del master theorem occorre dimostrare la regolarità di $f(n)$, ovvero che $af(n/b) < kf(n)$ per $0 < k < 1$, asintoticamente. Si imposta

$$af(n/b) = 3n/2(n/2+1) < kn(n+1)$$

e si vede che $n/2 + 1 < 5n/8$ per $n > n_0 = 8$, quindi sotto questa condizione

$$af(n/b) = 3n/2(n/2+1) < 15/16n(n+1)$$

ovvero $k = 15/16$ va bene.

Domanda 17 Dare la definizione di $\Omega(f(n))$. Mostrare che se $f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n))$ allora $f(n) = \Omega(h(n))$.

Soluzione: Si ha che

$$\Omega(f(n)) = \{g(n) \mid \exists c > 0.\ \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ 0 \leq cg(n) \leq f(n)\}.$$

Se $f(n) = \Omega(g(n)$ e $g(n) = \Omega(h(n))$ allora esistono $c_1, c_2 > 0$, $n_1, n_2 \in \mathbb{N}$ tali che per ogni $n \geq n_1$

$$0 \leq c_1 g(n) \leq f(n) \tag{1}$$

e per ogni $n \geq n_2$

$$0 \leq c_2 h(n) \leq g(n) \tag{2}$$

Ne consegue che per ogni $n \geq \max\{n_1, n_2\}$, moltiplicando (2) per c_1 si ha

$$0 \leq c_1 c_2 h(n) \leq c_1 g(n) \leq f(n)$$

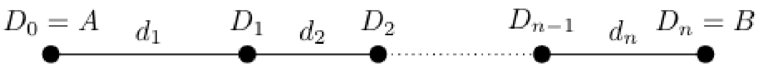
ovvero, indicato con $n_0 = \max\{n_1, n_2\}$ e $c = c_1 c_2$, per ogni $n \geq n_0$,

$$0 \leq ch(n) \leq f(n)$$

ovvero $f(n) = \Omega(h(n))$.

9 Greedy

Esercizio 33 Si supponga di voler viaggiare dalla città A alla città B con un'auto che ha un'autonomia pari a d km. Lungo il percorso si trovano $n-1$ distributori $D_1, \ldots, D_{n-1}$, a distanze di $d_1, \ldots, d_n$ km ($d_i \leq d$) come indicato in figura



L'auto ha inizialmente il serbatoio pieno e l'obiettivo è quello di percorrere il viaggio da A a B, minimizzando il numero di soste ai distributori per il rifornimento.

i. Introdurre la nozione di soluzione per il problema e di costo della una soluzione. Mostrare che vale la proprietà della sottostruttura ottima e individuare una scelta che gode della proprietà della scelta greedy.

ii. Sulla base della scelta greedy individuata al passo precedente, fornire un algoritmo greedy `stop(d, n)` che dato in input l'array delle distanze `d[1..n]` restituisce una soluzione ottima.

iii. Valutare la complessità dell'algoritmo.

Soluzione. In generale, la specifica del problema consiste in una sequenza di soste possibili $D_0(= A), D_1, \ldots, D_n(= B)$. Per una coppia di soste D_i, D_j con $i \leq j$ poniamo $d_{i,j} = \sum_{h=i+1}^{j} d_h$. Quindi una soluzione del problema è una sottosequenza di soste che porta dal punto iniziale al punto finale, senza percorrere tratti di lungheza maggiore di d ovvero:

$$S = D_{i_0} \ldots D_{i_k}$$

(con $i_0 < i_1 < \ldots < i_k$) tali che $D_{i_0} = A$ e $D_{i_k} = B$, per ogni $j \in \{0, \ldots, k-1\}$ vale $d_{i_j i_{j+1}} \leq d$. Il costo $c(S)$ è il numero $k - 1$ di soste.

Sottostruttura ottima. Sia $S = D_{i_0} \ldots D_{i_k}$ una soluzione ottima per il problema $D_0, D_1, \ldots, D_n$. Se consideriamo il sottoproblema di andare da D_{i_1} a D_n, ovvero $D_{i_1}, D_{i_1+1} \ldots D_n$, allora $S_1 = D_{i_1}, \ldots, D_{i_k}$ è una soluzione ottima. Infatti, è chiaramente una soluzione. Inoltre, se ci fosse una soluzione migliore S'_1, con un numero inferiore di soste per il sottoproblema, ovvero $c(S'_1) < c(S_1)$, aggiungendo la sosta D_{i_0} otterremmo una soluzione S' migliore di S per il problema originale dato che $c(S') = c(S'_1) + 1 < c(S_1) + 1 = c(S)$.

Significa che, avendo scelto bene prima,
aggiungendo altre soste, la scelta rimane minima.

Scelta greedy. Per il problema $D_0, D_1, \ldots, D_n$, si fissa necessariamente $i_0 = 0$, e la scelta greedy consiste nel raggiungere la sosta più lontana a distanza minore o uguale di d, ovvero definire $i_1 = \max\{j \mid d_{0,j} \leq d\}$.

Data una soluzione ottima per il problema $S = D_{j_0} \ldots D_{j_k}$, certamente $j_0 = 0 = i_0$ e $j_1 \leq i_1$. Quindi è immediato verificare che anche $S' = D_{j_0} D_{i_1} D_{j_2} \ldots D_{j_k}$ è una soluzione per il problema $D_0, D_1, \ldots, D_n$, ed è ottima, dato che $c(S') = c(S)$. (Si noti che, più precisamente, in prima istanza si potrebbe pensare che D_{i_1} potesse essere anche oltre D_{j_2}, ma questo darebbe una soluzione migliore di quella ottima, portando ad un assurdo).

Algoritmo. Ne segue l'algoritmo che riceve in la sequenza di distanze nella forma di un array $d[1, n]$ e restituisce un array $S[1..n-1]$ con le soste scelte (la prima e l'ultima sono scelte sempre, non serve indicarlo).

Per scelta greedy, prendiamo la sosta
con distanza <= a quella attuale,
tale da capire caso per caso quale i conviene di più.

```
stop(d, n)
  dist = d[1]              // distanza percorsa
  for i=2 to n
    if dist + d[i] > d
      S[i-1]=1
      dist=d[i]
    else
      S[i-1] = 0

  return S
```

1 2 3 4 5 6 $\boxed{7}$

L'algoritmo ricalca proprio la selezione delle attività e:
- prende come distanza la prima, per inizializzazione
- se la somma tra la prima distanza e quella attuale è > della sequenza delle distanze, allora abbiamo la distanza buona e salviamo la sosta con la distanza ottima (sapendo che ci siamo fermati "prima", quindi S[i-1]=1)
- se invece non ci siamo fermati prima, avremo S[i-1] = 0

La complessità è $\Theta(n)$.

Esercizio 1 (9 punti) Realizzare una funzione `avgTree(T)` che dato un albero binario `T` con chiavi numeriche, verifica se, per ogni nodo che abbia discendenti, la chiave del nodo è maggiore o uguale della media delle chiavi dei discendenti e ritorna conseguentemente un valore booleano (la radice dell'albero è `T.root` e ogni nodo `x` ha i campi `x.left` `x.right` e `x.key`).

Soluzione: L'idea è quella di effettuare una visita dell'albero, in modo ricorsivo, raccogliendo per ogni sottoalbero le seguenti informazioni:

- se è soddisfatta la condizione sulla media richiesta
- la somma delle chiavi
- il numero dei nodi.

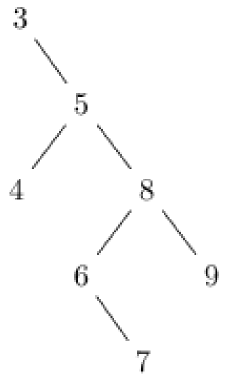
Quano mi trovo su di un nodo, analizzo i sottoalberi sinistro e destro. Quindi, avendo a disposizione la somma delle chiavi ed il numero dei nodi per i due sottoalberi posso verificare la condizione sulla media per il nodo in esame. Lo pseudocodice della soluzione può essere il seguente:

```
avgTree(T)
   v, n, s = avgTreeRec(T.root)
   return v


# avgTreeRec(x):
# verifica se il sottoalbero radicato in x e' un avgTree e ritorna tre valori:
# - un booleano,
# - il numero dei discendenti
# - la somma delle loro chiavi

avgTreeRec(x)

if x = nil
   return true, 0, 0
else
   # ispeziono i sottoalberi sinistro e destro
   vl, nl, sl = avgTreeRec(x.left)
   vr, nr, sr = avgTreeRec(x.right)

   # se non ci sono discendenti (nl+nr =0) oppure la chiave del nodo in
   # esame e' maggiore o uguale della media delle chiavi dei discendenti
   # la condizione e' soddisfatta
   v =   (nl+nr =0) or (x.key >= (sl+sr)/(nl+nr))

   # il numero di nodi dell'albero radicato in x e' dato dalla somma del
   # del numero di nodi nel sottoalbero dx e nel sottoalbero sx piu' uno
   # (il nodo x stesso)
   n =   nl + nr + 1

   # la somma delle chiavi dell'albero radicato in x e' dato dalla somma del
   # delle chiavi nel sottoalbero dx e in quello sx piu' x.key
   s =   sl + sr + x.key

   return v, n, s
```

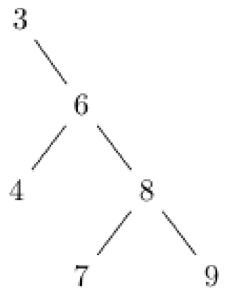
Domanda B (7 punti) Dare la definizione di albero binario di ricerca. Specificare l'albero ottenuto inserendo, con la procedura vista a lezione, a partire da un albero vuoto, i nodi aventi le seguenti chiavi:

$$3, 5, 8, 6, 9, 7, 4$$

Dall'albero così ottenuto si cancelli il nodo con chiave 5 e si indichi l'albero ottenuto. Sia per gli inserimenti che per la cancellazione, motivare sinteticamente il risultato ottenuto.



La cancellazione di 5 quindi produce:



Domanda A (7 punti) Definire formalmente la classe $O(f(n))$. Dimostrare che la seguente ricorrenza ha soluzione $T(n) = O(n)$

$$T(n) = \frac{1}{3}T(n-1) + 2n + 1$$

Soluzione: Per la definizione di $O(f(n))$, consultare il libro.

Si deve provare che asintoticamente, per un'opportuna costante $c > 0$

$$T(n) \leq cn$$

Si procede per induzione:

$$\begin{aligned}
T(n) &= \tfrac{1}{3}T(n-1) + 2n + 1 && \text{[per definizione della ricorrenza]} \\
&\leq \tfrac{1}{3}c(n-1) + 2n + 1 && \text{[per ipotesi induttiva } T(n-1) \leq c(n-1)\text{]]} \\
&= (\tfrac{1}{3}c + 2)n - (\tfrac{1}{3}c - 1) && \text{[assumendo } c \geq 3\text{]} \\
&\leq (\tfrac{1}{3}c + 2)n \\
&\leq cn
\end{aligned}$$

dove l'ultima disuguaglianza vale quando $c \geq 3$, consistentemente con l'assunzione. Risulta dunque dimostrata la tesi.

Esercizio 1 (9 punti) Realizzare una funzione booleana `checkSum(A,B,C,n)` che dati tre array `A[1..n]`, `B[1..n]` e `C[1..n]` con valori numerici, verifica se esistono tre indici i, j, k con $1 \leq i, j, k \leq n$ tali che $A[i] + B[j] = C[k]$. Valutarne la complessità.

Soluzione: L'idea consiste nell'ordinare i vettori B e C e a questo punto verificare per ogni elemento $A[i]$ del primo array se ne esiste uno del secondo $B[j]$ la cui somma produce un elemento $C[k]$ del terzo. Il fatto che gli array B e C siano ordinati permette di scorrerli una sola volta per ogni elemento del primo array.

```
checkSum(A,B,C,n):
    # ordina B e C
    Sort(B)
    Sort(C)

    i = 1
    found = false

    while (i <= n) and not found
        # per ogni A[i] verifica se per qualche elemento B[j] vale che
        # A[i]+B[j]=C[k] scorrendo B e C

        j = k = 1
        while (j <= n) and (k <= n) and (A[i] + B[j] <> C[k])
            if (A[i] + B[j] < C[k]):
               j++
            else
               k++
        if (j<=n and k<=n)
           found = true
        else
           i++

    return found
```

L'invariante del ciclo esterno è che se `found` è falsa, allora per ogni $i' < i$, e qualunque siano j' e k', vale $A[i'] + B[j'] \neq C[k']$. Se invece `found` è vera, allora $i, j, n \leq n$ e $A[i] + A[j] = A[k]$.

Per il ciclo interno, abbiamo che per ogni $j' < j$ e per ogni $k' < k$ vale $A[i] + B[j'] \neq C[k']$. Per il mantenimento dell'invariante, unico fatto da osservare è che quando $A[i]+B[j] < C[k]$ posso incrementare j mantenendo l'invariante. Infatti so che per ogni $j' < j$, dato che B è ordinato, vale $B[j'] \leq B[j]$ e quindi $A[i] + B[j'] \leq A[i] + B[j] < C[k]$. Il ragionamento quando $A[i] + B[j] > C[k]$ e incremento k è duale.

La complessità è $O(n^2)$ dato che ho due cicli annidati. In quello esterno, quando non esco i aumenta, quindi il numero di iterazioni è limitato da n. In quello interno, quando non esco, j oppure k aumentano, quindi il numero di iterazioni è limitato da $2n$. Complessivamente, nel caso peggiore, ho $n * 2n = 2n^2 = \Theta(n^2)$ iterazioni, ciascuna di costo costante.

A questo si somma il costo degli ordinamenti, che però posso realizzare in tempo $\Theta(n \log n)$ di modo che non incida sulla complessità asintotica.