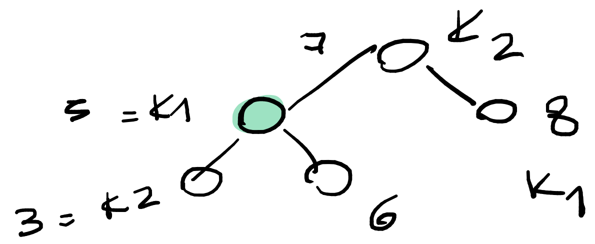
Esercizio 1 (9 punti) Scrivere una funzione `Anc(T,k1,k2)` che dato un albero binario di ricerca `T` nel quale tutte le chiavi sono distinte, e due chiavi `k1`, `k2` presenti in `T`, verifica se il nodo contenente `k1` è antenato del nodo contenente `k2`. Valutare la complessità della funzione.

BST → TUTTE CHIAVI UNICHE

```
Anc(t, k1, k2)

x = T.root

if (x == nil) return nil

else
while (x.key <> k1) and (x.key <> k2)
        if(x.key > k2)
                x = x.right
        else
                x = x.left

// confrontare rispetto a k1
if (x.key == k1)
        return true
else
        return false
```



Soluzione: È sufficiente osservare che, dato che le chiavi sono uniche e `k1`, `k2` sono certamente contenute in `T`, il nodo che contiene `k1` è antenato di quello che contiene `k2` se e solo se cercando `k2` a partire dalla radice di `T` incontro la chiave `k1`. Si assume che un nodo sia antenato di sé stesso.

La funzione può dunque essere realizzata come una semplice variante della ricerca negli alberi binari di ricerca.

```
Anc(T, k1, k2)
   x = T.root

   while (x.key <> k1) and (x.key <> k2)
       if (k2 < x.key)
          x = x.left
       else
          x = x.right

   return (x.key == k1)
```

Se l'albero ha altezza h, nel caso peggiore non trovo `k1` e la chiave `k2` è una foglia a profondità h, che quindi raggiungo in h iterazioni. Pertanto la complessità è $O(h)$.

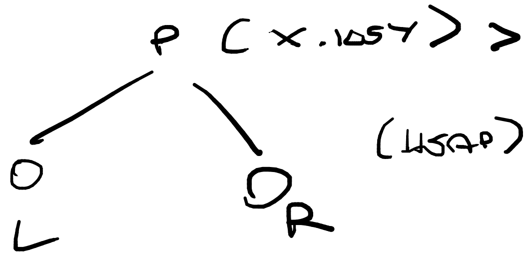
Domanda B (7 punti) Scrivere una funzione `toTree(A)` che dato un array `A` organizzato a max-heap (dimensione A.heapSize), lo trasforma in un albero binario realizzato con strutture linked, ancora organizzato a max-heap e ritorna la radice di tale albero. Il nuovo albero è costituito da nodi `x` con i campi `x.p` (parent), `x.k` (chiave), `x.l` e `x.r` (figlio sinistro e figlio destro). Per allocare un nuovo nodo si assuma di avere a disposizione un costruttore `node()`. Valutare la complessità.

(x.L ⇒ 2i
x.R ⇒ 2i + 1)

A → A.HEAPSIZE

X → P / L / R / K

```
toTree(A)
T.root = A[0]
x = T.root

for i = 0 to A.heapsize
        y = node()
        y.p = x
        y.key = A[i]

        if(y.p > y.key)
                y.l = A[2 * i]
                y.r = A[2 * i+1]
        else
                y.key = y.p
return y
```



```
toTree(A)
   T.root = toTreeRec(A,1,nil)
   return T

// toTreeRec(A,i,x):
// dato un max-heap A ritorna la radice di un albero, copia "linked" del
// sottoalbero di A radicato in i, l'argomento x e' il nodo padre
toTreeRec(A,i,x)
   if i <= A.heapSize
      y = node()                 // crea un nuovo nodo
      y.key = A[i]               // la chiave e' A[i]
      y.p = x                    // il parent e' x
      left  = 2*i                // costruisce i sottoalberi sx e dx,
      right = 2*i+1              // che saranno figli sx e dx di y
      y.l = toTreeRec(A,left, y)
      y.r = toTreeRec(A,right,y)
   else
      return nil
```

Si tratta di una visita dell'albero e quindi la complessità è $\Theta(n)$ dove n è il numero di elementi del max-heap.

```
toTree(A)
    n = A.heapSize
    create array N[1..n]   // N[i] = nodo corrispondente ad A[i]

    // 1) crea tutti i nodi
    for i = 1 to n
        N[i] = node()
        N[i].key = A[i]
        N[i].p = nil
        N[i].l = nil
        N[i].r = nil

    // 2) collega parent e figli
    for i = 1 to n
        left = 2*i
        right = 2*i + 1

        if left ≤ n
            N[i].l = N[left]
            N[left].p = N[i]

        if right ≤ n
            N[i].r = N[right]
            N[right].p = N[i]

    T.root = N[1]
    return T
```

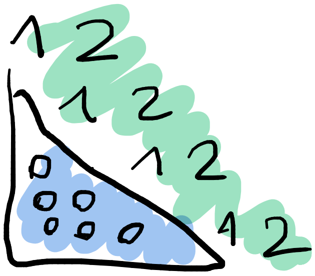
Esercizio 2 (9 punti) Data una stringa $X = x_1, x_2, \ldots, x_n$, si consideri la seguente quantità $\ell(i,j)$, definita per $1 \le i \le j \le n$:

$$\ell(i,j) = \begin{cases} 1 & \text{se } i = j \\ 2 & \text{se } i = j-1 \\ 2 + \ell(i+1, j-1) & \text{se } (i < j-1) \text{ e } (x_i = x_j) \\ \sum_{k=i}^{j-1} (\ell(i,k) + \ell(k+1,j)) & \text{se } (i < j-1) \text{ e } (x_i \ne x_j). \end{cases}$$

1. Scrivere una coppia di algoritmi INIT_L(X) e REC_L(X, i, j) per il calcolo memoizzato di $\ell(1, n)$.
2. Si determini la complessità *al caso migliore* $T_{\text{best}}(n)$, supponendo che le uniche operazioni di costo unitario e non nullo siano i confronti tra caratteri.

```
- INIT_L(X)

n = length(X)
// casi base

for i = 1 to (n - 1)
        L[i, i] = 1
        L[i, i+1] = 2
L[n, n] = 1

for i = 1 to n - 2
        for j = i + 2 to n
                L[i, j] = 0

return REC_L(X, 1, n)

- REC_L (X, i, j)
        if L[i, j] == 0
                if (X[i] == X[j])
                        L[i, j] = 2 +
                        REC_L[i+1, j-1]
                else
                        for k = i to j - 1
                                L[i, j] = REC_L(X, i, k) + REC_L(X, k+1, j) + L[i,j]
return L[i, j]
```

→ $j = i + 1$ (POSSIBILE)



$$\begin{cases} 2 + \ell(i+1, j-1) & \text{se } (i < j-1) \text{ e } (x_i = x_j) \\ \sum_{k=i}^{j-1} (\ell(i,k) + \ell(k+1,j)) & \text{se } (i < j-1) \text{ e } (x_i \ne x_j) \end{cases}$$

→ LCS ....

$O\lfloor n/2 \rfloor$ $\quad O\lfloor n/2 \rfloor$ $\quad 2 = 1 + 1$

BOTTOM-UP → ITERATIVO → 1 ALG.

TOP-DOWN → MEMOIZZAZIONE → RICORSIVO
→ 2 ALGORITMI

$$T_{best}(n) = 2 + T_{best}(n-2) \simeq O(n)$$

max/min → COMPUTE → BOTTOM_UP

INIT/REC → MEMOIZZAZIONE

$O(n)$ / $O\frac{(n-2)(n-1)}{2}$ / $O(n-2)(n-1)$

Domanda A (7 punti) Si consideri la funzione ricorsiva `search(A,p,r,k)` che dato un array A, ordinato in modo decrescente, un valore k e due indici p, q con $1 \leq p \leq r \leq A.length$ restituisce un indice i tale che $p \leq i \leq r$ e $A[i] = k$, se esiste, e altrimenti restituisce 0.

Domanda B (6 punti) Si consideri una tabella hash di dimensione $m = 8$, e indirizzamento aperto con doppio hash basato sulle funzioni $h_1(k) = k \ mod \ m$ e $h_2(k) = 1 + 2(k \ mod \ (m - 2))$. Si descriva in dettaglio come avviene l'inserimento della sequenza di chiavi: 13, 29, 19, 27, 8.

Esercizio 2 (9 punti) Data una stringa di numeri interi $A = (a_1, a_2, \ldots, a_n)$, si consideri la seguente ricorrenza $z(i,j)$ definita per ogni coppia di valori (i,j) con $1 \le i, j \le n$:

$$z(i,j) = \begin{cases} a_j & \text{if } i = 1, 1 \le j \le n, \\ a_{n+1-i} & \text{if } j = n, 1 < i \le n, \\ z(i-1,j) \cdot z(i,j+1) \cdot z(i-1,j+1) & \text{altrimenti.} \end{cases}$$

1. Si fornisca il codice di un algoritmo iterativo bottom-up Z(A) che, data in input la stringa A restituisca in uscita il valore $z(n,1)$.
2. Si valuti il numero esatto $T_Z(n)$ di moltiplicazioni tra interi eseguite dall'algoritmo sviluppato al punto (1).

```
Z(A)
n = length(A)

for i = 1 to n
        z[1, i] = a[i]
        z[i, n] = a[n + 1 - i]

for i = 1 to n - 1
        for j = n - 1 downto 1
                z[i, j] = z[i - 1, j] * z[i, j + 1] * z[i - 1, j + 1]
return z[n, 1]
```

1. Date le dipendenze tra gli indici nella ricorrenza, un modo corretto di riempire la tabella è attraverso una scansione "reverse column-major", in cui calcoliamo gli elementi della tabella in ordine decrescente di indice di colonna e, all'interno della stessa colonna, in ordine crescente di indice di riga. Il codice è il seguente.

```
Z(A)
n = length(A)
for i=1 to n do
   z[1,i] = a_i
   z[i,n] = a_{n+1-i}
for j=n-1 downto 1 do
   for i=2 to n do
      z[i,j] = z[i-1,j] * z[i,j+1] * z[i-1,j+1]
return z[n,1]
```

Si osservi che un altro modo corretto di riempire la tabella è attraverso una scansione "reverse diagonal", che scansiona per diagonali parallele alla diagonale principale partendo da quella contenente solo $z[1,n]$.

$$\sum_{s=1}^{n-1} \sum_{i=2}^{n} 2 = \sum_{s=1}^{(n-1)} 2(n-1) = 2(n-1)^2$$

$$[\quad]_{n-1 \times n-1}$$

Esercizio 2 (9 punti) Si consideri un file definito sull'alfabeto $\Sigma = \{a, b, c\}$, con frequenze $f(a), f(b), f(c)$. Per ognuna delle seguenti codifiche si determini, se esiste, un opportuno assegnamento di valori alle 3 frequenze $f(a), f(b), f(c)$ per cui l'algoritmo di Huffman restituisce tale codifica, oppure si argomenti che tale codifica non è mai ottenibile.

1. $e(a) = 0,\ e(b) = 10,\ e(c) = 11$

---

2. $e(a) = 1,\ e(b) = 0,\ e(c) = 11$
3. $e(a) = 10,\ e(b) = 01,\ e(c) = 00$

Esercizio 1 (9 punti) Realizzare una funzione `intersect(A1,A2,n)` che dati due array di interi `A1` e `A2`, organizzati a min-heap, con capacità n, restituisce un nuovo array `A`, ancora organizzato a min-heap, che contiene l'intersezione dei valori contenuti in `A1` e `A2`. Nel caso gli array contengano più occorrenze dello stesso valore v, l'intersezione mantiene il numero minimo di occorrenze di v (ad es. se `A1` contiene i valori `1,2,2,2` e `A2` contiene i valori `1,1,2,2` allora A conterrà `1,2,2`). Valutarne la complessità.

Esercizio 2 (9 punti) Supponiamo di avere un numero illimitato di monete di ciascuno dei seguenti valori: 50, 20, 1. Dato un numero intero positivo n, l'obiettivo è selezionare il più piccolo numero di monete tale che il loro valore totale sia n. Consideriamo l'algoritmo greedy che consiste nel selezionare ripetutamente la moneta di valore più grande possibile.

(a) Fornire un valore di n per cui l'algoritmo greedy *non* restituisce una soluzione ottima.

(b) Supponiamo ora che i valori delle monete siano 10, 5, 1. In questo caso l'algoritmo greedy restituisce sempre una soluzione ottima: dimostrare che ogni insieme ottimo M^* di monete di valore totale n contiene la scelta greedy.

50, 20, 1 $\quad$ $n = \underline{60}$ $\quad$ $\sum_{m=1}^{n} m[i] = n$

(a) → ?

[m[0] = 50 + 10 monete da 1] (11 monete)

>

[3 monete da 20] (3 monete)

OBS. → VALORE NON MULTIPLO (!)

(ASSEGN. PARCHEGGI) → ORDINE

$\underline{a_1 - p_1}$

$\underline{a_2 - p_2}$

$a_3 - \underline{p_3}$

min Σ

Esercizio 2 (9 punti) Supponiamo di avere un numero illimitato di monete di ciascuno dei seguenti valori: 50, 20, 1. Dato un numero intero positivo n, l'obiettivo è selezionare il più piccolo numero di monete tale che il loro valore totale sia n. Consideriamo l'algoritmo greedy che consiste nel selezionare ripetutamente la moneta di valore più grande possibile.

(a) Fornire un valore di n per cui l'algoritmo greedy *non* restituisce una soluzione ottima.

(b) Supponiamo ora che i valori delle monete siano 10, 5, 1. In questo caso l'algoritmo greedy restituisce sempre una soluzione ottima: dimostrare che ogni insieme ottimo M^* di monete di valore totale n contiene la scelta greedy.

$M = [\underline{10}, 5, 1]$ → monete ...

max $\cup\, M^* \leq x$

(M^* → ottimo)

$M' \to M^* \setminus M \cup X$

(b) Sia M^* una soluzione ottima. Sia x il valore maggiore tra 10, 5, e 1 che sia non superiore a n. Se M^* contiene una moneta di valore x, la proprietà è dimostrata. Altrimenti, sia $M \subseteq M^*$ un insieme di (2 o più) monete di valore totale x (si osservi che tale insieme esiste sempre quando i valori delle monete sono 10, 5, 1); consideriamo $M' = M^* \setminus M \cup X$, dove X è l'insieme contenente una moneta di valore x. M' è un insieme di monete di valore totale n e di cardinalità inferiore a quella di M^*: assurdo, quindi questo secondo caso non può verificarsi, e quindi M^* contiene necessariamente una moneta di valore x.

Domanda A (8 punti) Definire formalmente la classe $\Theta(g(n))$. Dimostrare le seguenti affermazioni o fornire un controesempio:

i. se $f(n), f'(n) \in \Theta(g(n))$ allora $f(n) + f'(n) \in \Theta(g(n))$;

ii. $f(n), f'(n) \in \Theta(g(n))$ allora $f(n) * f'(n) \in \Theta(g(n))$;

$\Theta \rightarrow [f \le x \le g]$

VALE $\Theta\, g(n) \rightarrow \exists\ c_1, c_2 > 0$

$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$

VALE PER f' $\rightarrow 0 \le c_1' g(n) \le f'(n) \le c_2' g(n)$

$n_0, n_0' \dots \rightarrow \Theta(n) \rightarrow n = \max(n_0, n_0')$

$((c_1 + c_1')g(n) =$ max

$c_1 g(n) + c_1' g(n \le (f(n) + f'(n))$

$\le c_2 g(n) + c_2' g(n)$

$\rightarrow$ VERIFICA

$[0 \le \underline{f(n) + f'(n)} \le g(n) \le f(n) + f'(n)]$

RISCRITTURA...

Domanda A (8 punti) Definire formalmente la classe $\Theta(g(n))$. Dimostrare le seguenti affermazioni o fornire un controesempio:

1°

2°

i. se $f(n), f'(n) \in \Theta(g(n))$ allora $f(n) + f'(n) \in \Theta(g(n))$; → 4°

3°

ii. $f(n), f'(n) \in \Theta(g(n))$ allora $f(n) * f'(n) \in \Theta(g(n))$;

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

Per (i), siano $f(n), f'(n) \in \Theta(g(n))$. Per definizione, esistono $c_1, c_2 > 0$ e n_0 tali che per ogni $n \geq n_0$ vale:

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

e, analogamente esistono $c_1', c_2' > 0$ e n_0' tali che per ogni $n \geq n_0'$ vale:

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

Quindi, per ogni $n \geq \max\{n_0, n_0'\}$ abbiamo:

$$(c_1 + c_1')g(n) = c_1 g(n) + c_1' g(n \leq f(n) + f'(n) \leq c_2 g(n) + c_2' g(n) = (c_2 + c_2')g(n)$$

dato che $c_1 + c_1', c_2 + c_1' > 0$ questo conclude la prova che $f(n) + f'(n) \in \Theta(g(n))$.

$$c_1(n) + c_2 \leq f \cdot g \leq c_1' + c_2'$$

$$\leq n^2 \leq n$$

$$c_1 n \cdot c_2 n \leq f(n) + f'(n)$$

NO!

Per la parte (ii), l'affermazione è falsa. Basta considerare $f(n) = f'(n) = n \in \Theta(n)$, ma ovviamente $f(n) * f'(n) = n^2 \notin \Theta(n)$.