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.

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.

```
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.

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 $1 \leq i \leq j \leq n$:

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

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.

1. Pseudocodice:

```
INIT_L(X)
 n <- length(X)
 if n = 1 then return 1
 if n = 2 then return 2
 for i=1 to n-1 do
   L[i,i] <- 1
   L[i,i+1] <- 2
 L[n,n] <- 1
 for i=1 to n-2 do
   for j=i+2 to n do
     L[i,j] <- 0
 return REC_L(X,1,n)



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

```
Il caso migliore `e chiaramente quello in cui tutti i caratteri sono uguali,
poich´e l’albero del la ricorsione in quel caso `e unario e i suoi nodi
interni corrispondono alle chiamate con indici (1, n),(2,n − 1),...,(k,n − k
+ 1), ognuno associato a un costo unitario. Ci sono al pi`u ⌊n/2⌋ di queste
chiamate, e quindi Tbest(n) = O(n)
```

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.

0	29
1	-
2	27
3	19
4	-
5	13
6	-
7	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 $1 \leq i, j \leq n$:

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

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).

Soluzione:

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 $z[1, n]$.

2. Ogni iterazione del doppio ciclo dell'algoritmo esegue due moltiplicazioni tra interi, e quindi

$$\begin{aligned} T_Z(n) &= \sum_{j=1}^{n-1} \sum_{i=2}^{n} 2 \\ &= \sum_{j=1}^{n-1} 2(n-1) \\ &= 2(n-1)^2. \end{aligned}$$

   Equivalentemente, basta osservare che l'algoritmo esegue due moltiplicazioni per ogni elemento di una tabella $(n-1) \times (n-1)$.

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. $e(a) = 10,\ e(b) = 01,\ e(c) = 00$

Soluzione:

1. Questa codifica viene restituita dall'algoritmo di Huffman quando $f(b), f(c) < f(a)$: in questo caso i nodi associati a b e c vengono uniti creando un nuovo nodo interno, che poi viene unito al nodo associato ad a. Quindi, per esempio, $f(a) = 40, f(b) = 25, f(c) = 35$.
2. La codeword $e(a) = 1$ è un prefisso della codeword $e(c) = 11$, cioè questa codifica non è libera da prefissi, e quindi non è una codifica di Huffman.
3. L'albero associato a questa codifica non è pieno perché c'è un nodo interno con un solo figlio (quello nel cammino che porta alla foglia associata al carattere a). Quindi questa codifica non è ottima e quindi non è una codifica di Huffman.

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.

Soluzione: Per costruire l'intersezione, si procede nel modo seguente. Posso estrarre gli elementi di ciascun min-heap, in ordine crescente, semplicemente con una sequenza di estrazioni di minimo, ciascuna delle quali ha costo logaritmico. Se da $A1$ estraggo v_1 e da A_2 estraggo v_2, se $v_1 = v_2$, inserisco il valore comune nell'intersezione. Se invece $v_1 < v_2$, certamente v_1 non è nell'intersezione, dato che tutti gli elementi rimanenti in $A2$ sono $\geq v_2$, quindi posso scartare v_1, estrarre un nuovo elemento da $A1$ e continuare. Se $v_1 > v_2$ procedo dualmente.

In questo modo ottengo gli elementi dell'intersezione in ordine crescente. L'ultima osservazione è che posso inserirli in questo modo in A, che risulterà un array crescente, che è un caso particolare di min-heap.

Lo pseudocodice può essere il seguente:

```
intersect(A1,A2,n)
   allocate A[1..n]
   i=0                          // ultimo elemento occupato in A
   while (A1.heapsize > 0) and (A2.heapsize > 0)
         v1 = Min(A1)
         v2 = Min(A2)
         if (v1 = v2)
            i++
            A[i]=1
            ExtractMin(A1)
            ExtractMin(A2)
         elseif (v1 < v2)
            ExtractMin(A1)
         else
            ExtractMin(A2)

   A.heapsize=i
   return A
```

La complessità di ogni `ExtractMin` è $O(\log n)$ mentre `Min` ha costo costante. Ol numero di estrazioni è chiaramente limitato da $2n$ e questo porta a dedurre che il costo complessivo è $O(n \log n)$.

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 $M^\star$ di monete di valore totale n contiene la scelta greedy.

Soluzione:

(a) Per esempio $n = 60$, perché la soluzione ottima è 3 monete da 20, mentre l'algoritmo greedy restituisce 11 monete (una da 50 e 10 da 1).

(b) Sia $M^\star$ una soluzione ottima. Sia x il valore maggiore tra 10, 5, e 1 che sia non superiore a n. Se $M^\star$ contiene una moneta di valore x, la proprietà è dimostrata. Altrimenti, sia $M \subseteq M^\star$ 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 $M' = M^\star \setminus M \cup X$, 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 $M^\star$: assurdo, quindi questo secondo caso non può verificarsi, e quindi $M^\star$ 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 $f(n) * f'(n) \in \Theta(g(n))$;

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 $f(n) + f'(n) \in \Theta(g(n))$.

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