1. Tempo disponibile 120 minuti.
2. Non è possibile consultare appunti, slide, libri, persone, siti web, ecc.
3. Scrivere in modo leggibile, su ogni foglio, nome, cognome e numero di matricola.
4. Le soluzioni agli esercizi che richiedono di progettare un algoritmo devono:
   - spiegare a parole l'algoritmo (se utile, anche con l'aiuto di esempi o disegni),
   - fornire e commentare lo pseudo-codice (indicando il significato delle variabili),
   - calcolare la complessità (con tutti i passaggi matematici necessari),
   - se l'esercizio ammette più soluzioni, a soluzioni computazionalmente più efficienti e/o concettualmente più semplici sono assegnati punteggi maggiori.

IMPORTANTE: Risolvere gli esercizi 1–2 e gli esercizi 3–4 su fogli separati. Infatti, al termine, dovrete consegnare gli esercizi 1–2 separatamente dagli esercizi 3–4.

1. Calcolare la complessità $T(n)$ nel caso pessimo del seguente algoritmo MYSTERY1 quando
   a. la struttura dati T è un Albero Binario di Ricerca
   b. la struttura dati T è un Albero AVL

Algorithm 1: MYSTERY1(INT n)

```
T = Tree() // Struttura dati albero
for i = 1, ··· , n do
    key = i
    data = MYSTERY2(i)
    INSERT(T, key, data)

function MYSTERY2(INT n) → INT
if n ≤ 0 then
    return 1
else
    return MYSTERY2(n/2) + n^2
```

Soluzione. Analizziamo prima il costo di MYSTERY2. Tale funzione richiama se stessa una volta su input $n/2$ e le altre operazioni nella chiamata hanno costo costante. L'equazione di ricorrenza di MYSTERY2

$$T'(n) = \begin{cases} 1 & n \leq 1 \\ T'(n/2) + 1 & n > 1 \end{cases}$$

può essere risolta con il Master Theorem

$$\alpha = \log_2 2 = 1 = \beta \Rightarrow T'(n) = \Theta(n^\alpha \log n) = \Theta(\log n)$$

La complessità di MYSTERY1 dipende dal costo del suo ciclo for interno. Tale ciclo viene eseguito n volte per i che va da 1 ad n. In ogni iterazione esegue una chiamata a MYSTERY2 con input i ed un inserimento in una struttura dati di tipo albero. Poiché il costo di MYSTERY2 è $\Theta(\log n)$ il suo contributo alla complessità è dato da:

$$\Theta(\log 1) + \cdots + \Theta(\log n) = \Theta\left(\sum_{i=1}^{n} \log i\right) = \Theta\left(\log \prod_{i=1}^{n} i\right) = \Theta\left(\log n!\right) = \Theta\left(n \log n\right)$$

Il costo pessimo di n inserimenti in una struttura dati albero dipende dal tipo di struttura dati

a. Se T è un Albero Binario di ricerca, un inserimento ha costo pessimo $\Theta(n)$, dove n è il numero di nodi nell'albero. Quindi n inserimenti hanno costo pessimo

$$\Theta(1) + \cdots + \Theta(n) = \Theta\left(\sum_{i=1}^{n} i\right) = \Theta\left(n^2\right)$$

In questo caso il costo pessimo di MYSTERY1 è

$$T(n) = \Theta(n \log n) + \Theta(n^2) = \Theta(n^2)$$

b. Se T è un Albero AVL, un inserimento ha costo pessimo $\Theta(\log n)$, dove n è il numero di nodi nell'albero. Quindi n inserimenti hanno costo pessimo

$$\Theta(\log 1) + \cdots + \Theta(\log n) = \Theta\left(\sum_{i=1}^{n} \log i\right) = \Theta\left(\log \prod_{i=1}^{n} i\right) = \Theta\left(\log n!\right) = \Theta\left(n \log n\right)$$

In questo caso il costo pessimo di MYSTERY1 è

$$T(n) = \Theta(n \log n) + \Theta(n \log n) = \Theta(n \log n)$$

2. Considerare una Tabella Hash T di dimensione $m = 11$, inizialmente vuota. La funzione hash è definita sul metodo della divisione

$$h'(k) = k \mod m$$

e abbiamo la seguente sequenza di operazioni:

1) insert 17 2) insert 4 3) insert 7
4) insert 6 5) insert 28 6) insert 39

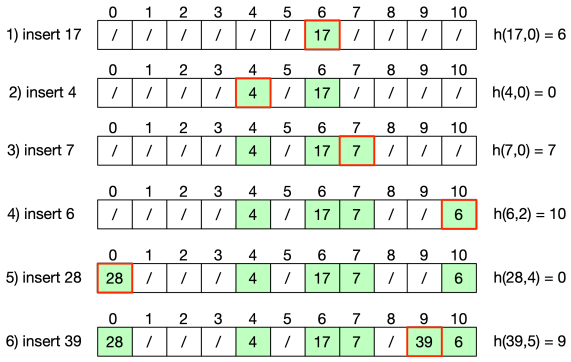
a) Mostrare lo stato di T dopo l'esecuzione delle operazioni precedenti, assumendo che le collisioni in T siano gestite con indirizzamento aperto e ispezione quadratica

$$h(k, i) = (h'(k) + i^2) \mod m$$

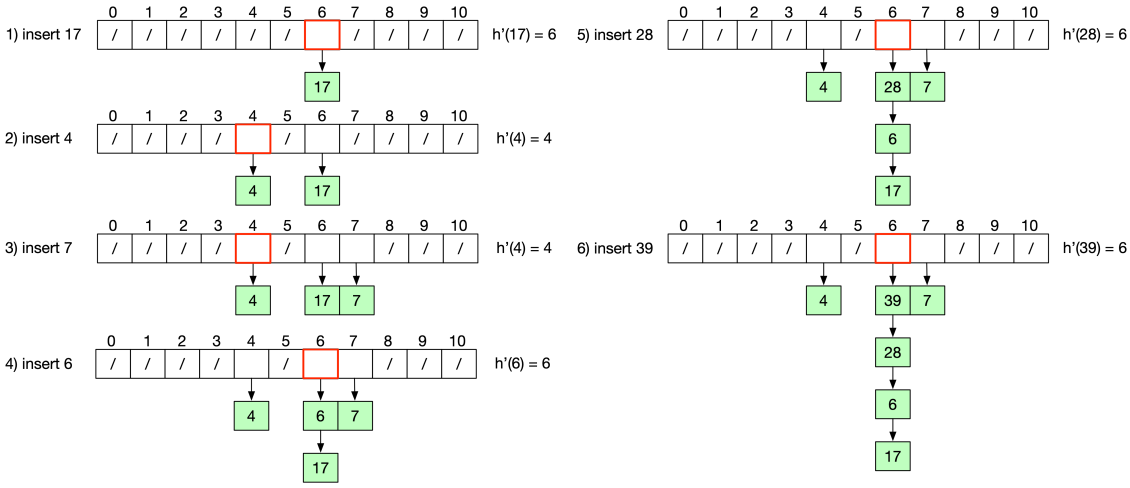
b) Mostrare lo stato di T dopo l'esecuzione delle operazioni precedenti, assumendo che le collisioni in T siano gestite con concatenamento

Soluzione.

a) Indirizzamento aperto



b) Concatenamento



3. Progettare un algoritmo efficiente che dati un array V di numeri ordinati in modo non decrescente e due numeri A e B, con $A \leq B$, restituisce la quantità di numeri in V che sono strettamente minori di A oppure strettamente maggiori di B. Ad esempio, se $V = [1, 3, 3, 5, 9, 11]$, $A = 3$ e $B = 6$, l'algoritmo deve restituire 3, visto che i valori da contare sono 1, 9, e 11.

   Soluzione. Si può procedere con il seguente algoritmo ricorsivo di tipo divide-et-impera in cui ogni chiamata ricorsiva che si occupa del sottovettore $V[i..j]$ evita di fare il passo ricorsivo se è possibile verificare immediatamente se tutti gli elementi del sottovettore $V[i..j]$ sono da contare (questo caso si verifica se $V[i] > B$ o $V[j] < A$) oppure nessuno degli elementi del sottovettore $V[i..j]$ sono da contare (questo caso si verifica se $V[i] \geq A$ e $V[j] \leq B$):

Algorithm 2: ContaArray($Number\ V[1..n], Number\ A, Number\ B, Integer\ i, Integer\ j$) $\rightarrow$ *Integer*

```
if i > j then
    return 0
else if V[i] > B or V[j] < A then
    return j − i + 1
else if V[i] ≥ A and V[j] ≤ B then
    return 0
else
    Intero m = Floor((i + j)/2)
    return ContaArray(V, A, B, i, m) + ContaArray(V, A, B, m+1, j)
```

   Tale algoritmo viene inizialmente invocato con $i = 1$ e $j = n$.

   Si consideri ora un certo livello di annidamento delle chiamate ricorsive: al più due istanze della funzione a tale livello effettueranno le 2 chiamate ricorsive (quella che contiene sia valori strettamente minori sia valori maggiori di A, e quella che contiene sia valori minori sia valori strettamente maggiori di B). Quindi ad ogni livello al più 4 istanze verranno eseguite. Il numero massimo di livelli di annidamenti è logaritmico in quanto la lunghezza dei sottovettori si dimezza ad ogni passaggio di livello. Il numero totale di istanze eseguite è quindi logaritmico. Visto che ogni istanza esegue un numero costante di operazioni, si ottiene il costo $T(n) = O(\log n)$.

4. Si consideri un grafo orientato pesato $G = (V, E, w)$ con pesi non negativi (ovvero $w(u, v) \geq 0$ per ogni $u, v \in V$) per cui esiste una funzione p che associa ad ogni vertice in V un peso $p(v)$ non negativo (ovvero $p(v) \geq 0$ per ogni $v \in V$). Dato un cammino $v_0, v_1, \ldots, v_n$ il relativo *costo complessivo* è dato dalla somma dei pesi dei vertici e degli archi attraversati (ovvero, il costo complessivo è $\sum_{i=0}^{n} p(v_i) + \sum_{i=1}^{n} w(v_{i-1}, v_i)$). Bisogna progettare un algoritmo che dato il grafo G, la funzione p, un vertice di partenza a ed un vertice di arrivo b, stampa il costo complessivo minimo per un cammino da a a b.

Cont.

Soluzione. È possibile risolvere il problema tramite una versione modificata dell'algoritmo di Dijkstra in cui al costo di ogni arco viene aggiunto il costo del vertice di arrivo dell'arco. Inoltre, la sorgente del cammino a non risulta a distanza 0 ma a distanza $p(a)$ per tener in considerazione anche il relativo costo iniziale $p(a)$.

L'Algoritmo 3 utilizza la solita convenzione secondo cui i vertici vengono rappresentati da numeri interi nell'intervallo $\{1 \dots |V|\}$; in questo modo le distanze stimate dei vertici possono essere rappresentate tramite un array D con indici che vanno da 1 a $|V|$. L'algoritmo ha il medesimo costo computazionale dell'algoritmo di Dijkstra, quindi $T(n, m) = O(m \log n)$ con $n = |V|$ e $m = |E|$.

Algorithm 3: CamminoSpesaMinima(Graph $G = (V, E, w)$, $p : V \to$ Number, Vertex a, Vertex b)

```
// inizializzazione strutture dati
n ← G.numNodi()
Number D[1..n]
for i ← 1 to n do
    D[i] ← ∞
D[a] ← p(a)
MinPriorityQueue[Int, Number] Q ← new MinPriorityQueue[Int, Number]()
Q.insert(a, D[a])

// esecuzione algoritmo di Dijkstra (modificato)
while not Q.isEmpty() do
    u ← Q.findMin()
    Q.deleteMin()
    if u = b then
        // costo minimo per b già calcolato
        print D[b]
        return
    for v ∈ u.adjacent() do
        if D[v] = ∞ then
            // prima volta che si incontra V
            D[v] ← D[u] + w(u, v) + p(v)
            Q.insert(v, D[v])
        else if D[u] + w(u, v) + p(v) < D[v] then
            // scoperta di un cammino migliore per raggiungere v
            D[v] ← D[u] + w(u, v) + p(v)
            Q.decreaseKey(v, D[v])

// b non è raggiungibile
print "città b non raggiungibile"
```

The End.