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 $T(n)$ del seguente algoritmo MYSTERY1:

Algorithm 1: MYSTERY1(INT n) $\rightarrow$ INT

```
x = 1
for i = 1, ··· , n do
    x = x * n
    MYSTERY2(x)
return x

function MYSTERY2(INT n) → INT
if n ≤ 0 then
    return 1
else
    return 2^100 × MYSTERY2(n/2)
```

Soluzione. Analizziamo prima la complessità di MYSTERY2, che effettua una sola chiamata ricorsiva con valore in input dimezzato. L'equazione di ricorrenza di MYSTERY2 è quindi

$$T'(n) = \begin{cases} 1 & n \leq 0 \\ T'(n/2) + 1 & n > 0 \end{cases}$$

Tale equazione di ricorrenza può essere risolta con il Master Theorem

$$\alpha = \log_2 1 = 0 = \beta \Rightarrow T'(n) = \Theta(n^\alpha \log n) = \Theta(\log n)$$

La funzione MYSTERY1 richiama iterativamente MYSTERY2 per n volte, ogni volta con potenze crescenti di n come valore in input. Poiché MYSTERY2 ha un costo logaritimo, eseguire n iterazioni del ciclo for in MYSTERY1 ha il seguente costo

$$
\begin{aligned}
T(n) &= \Theta(\log n) + \Theta(\log n^2) + \cdots + \cdots \Theta(\log n^n) \\
&= \Theta\left(\log n + 2\log n + \cdots + n \log n\right) \\
&= \Theta\left(\log n \cdot \sum_{i=1}^{n} i\right) \\
&= \Theta\left(\log n \cdot \frac{n(n+1)}{2}\right) \\
&= \Theta(n^2 \log n)
\end{aligned}
$$

2. Consideriamo un albero binario di ricerca (BST) T contenenti chiavi intere. Ideare un algoritmo che, preso in input T ed un intero x, rimuova da T tutti i nodi con chiave $\geq x$. L'albero finale deve essere un valido BST. Analizzare la complessità computazionale della soluzione proposta nel caso ottimo e pessimo (fornire esempi di caso ottimo e pessimo).

   Soluzione. Potremmo usare iterativamente una funzione che rimuove un singolo nodo in BST ma questo comporterebbe un algoritmo poco efficiente rispetto ad una soluzione ad-hoc per il nostro problema. Infatti, analizzando attentamente il problema, possiamo vedere che, grazie alla topologia di un BST, abbiamo la necessità di gestire unicamente due casi distinti:

   1. Se $T.key < x \Rightarrow$ non è necessario rimuovere il nodo T e non è necessario cercare nel suo sottoalbero sinistro (in tale sottoalbero tutte le chiavi sono sicuramente $< x$). Quindi, richiamiamo ricorsivamente la nostra procedura sul sottoalbero destro e facciamo puntare $T.right$ al nodo ritornato da tale chiamata ricorsiva. Ritorniamo infine il puntatore a T
   2. Se $T.key \geq x \Rightarrow$ dobbiamo rimuovere il nodo T e tutto il suo sottoalbero destro (in tale sottoalbero tutte le chiavi sono sicuramente $\geq x$). Assumendo che il nostro linguaggio di programmazione sia dotato di garbage collector, non dobbiamo esplicitamente deallocare T ed il suo sottoalbero destre, semplicemente richiamiamo ricorsivamente la procedura sul sottoalbero sinistro e ne ritorniamo il puntatore. In questo modo, il padre di T (eventualmente NIL) punterà al nodo ritornato da tale chiamata ricorsiva.

Algorithm 2: REMOVEGT(BST T, INT x) $\rightarrow$ BST

```
if T == NIL then
    return NIL
else if T.key < x then
    T.right = REMOVEGT(T.right, x)
    return T
else
    return REMOVEGT(T.left, x)
```

   Sia n il numero di nodi in T.

   - Costo nel caso ottimo: $O(1)$
     - se la chiave della radice è $< x$ e la radice non ha un figlio destro, oppure
     - la chiave della radice è $\geq x$ e la radice non ha un figlio sinistro
   - Costo nel caso pessimo: $O(n)$
     - tutte le chiavi sono $< x$ e l'albero è una lista (solo figli destri)

3. Dobbiamo comprare una certa quantità P di piatti pregiati. Esistono solo n rivenditori di tali piatti. L' i-esimo rivenditore, con $i \in \{1, \ldots, n\}$, ha un numero limitato $d[i]$ di piatti disponibili alla vendita e ognuno di questi $d[i]$ piatti ha un prezzo $p[i]$. Bisogna calcolare il budget minimo necessario per comprare i P piatti. Progettare quindi un algoritmo che, dato il numero intero

P, il vettore di interi $d[1 \dots n]$ (piatti disponibili presso ognuno degli n rivenditori), il vettore di numeri non negativi $p[1 \dots n]$ (costo di un piatto presso ognuno degli n rivenditori), restituisce la cifra minima necessaria per acquistare P piatti. L'algoritmo deve restituire $+\infty$ nel caso in cui i rivenditori non abbiano globalmente un numero sufficiente di piatti.

Soluzione. Il problema può essere risolto semplicemente comprando i piatti con il prezzo unitario minore. Un modo greedy per implementare questo approccio è di ordinare i rivenditori in ordine non decrescente di prezzo unitario, poi si iniziano a comprare i piatti disponibili partendo dal rivenditore con il prezzo unitario minimo, passando a rivenditori successivi (secondo l'ordinamento) fino al raggiungimento della quantità prevista. Tale soluzione prevede di ordinare l'intero elenco di rivenditori e avrà quindi un costo computazionale $\Omega(n \log n)$, considerando la complessità pseudo-lineare del problema dell'ordinamento. Nel caso in cui il numero di rivenditori utilizzati fosse molto inferiore rispetto al numero totale di rivenditori, risulterebbe inutile aver ordinato tutti i rivenditori in quanto si utilizzerebbe solo una minima parte iniziale del vettore ordinato dei rivenditori. In questi casi, risulta più efficiente procedere come riportato nell'Algoritmo PIATTI: preventivamente si popola un min-heap con i rivenditori considerando il loro ordinamento rispetto al prezzo unitario e poi si estrae solo la quantità di rivenditori sufficiente per comprare tutti i piatti richiesti. Sia q questa quantità di rivenditori utilizzati. Considerando il costo $O(n)$ per popolare il min-heap, ed il costo $O(\log n)$ per l'estrazione del minimo da un min-heap, tale algoritmo risulta avere costo computazionale $O(n + q \log n)$ che risulterebbe $O(n)$ nel caso in cui $q = O(\frac{n}{\log n})$ (in caso contrario, avremmo comunque $q = O(n)$ che implicherebbe un costo computazionale $O(n \log n)$ non superiore alla soluzione che preventivamente ordina tutti i rivenditori).

Algorithm 3: PIATTI(INT P, INT[1..n] d, REAL[1..n] p) $\rightarrow$ REAL

```
/* Inizializza res piatti residui, tot costo totale, A coppie prezzo/rivenditore (p[i], i) */
REAL res ← P, tot ← 0, daComprare
(REAL,INT)[1..n] A
for i ← 1 to n do
    A[i] ← (p[i], i)

/* Generazione dell'heap contenente le coppie (p[i], i) ordinate secondo il primo campo    */
minHeap[(REAL,INT)]H ← A.heapify()

/* Estrazione dall'heap dei rivenditori da cui comprare i piatti                           */
while res > 0 and (not H.empty()) do
    (REAL,INT) ( p, j ) ← H.findMin()
    H.deleteMin()
    daComprare ← min(res, d[j])
    res ← res − daComprare
    tot ← tot + (d[j] × daComprare)

/* Controllo se si sono comprati tutti i piatti                                            */
if res = 0 then
    return tot
else
    return +∞
```

4. Progettare un algoritmo che dato un grafo orientato pesato $G = (V, E, w)$ con pesi non negativi (ovvero, $\forall v_1, v_2 \in V.\, w(v_1, v_2) \geq 0$), un insieme di possibili vertici di partenza $S \subseteq V$, un vertice di arrivo $t \in V$, ed un numero non negativo K, verifica se esiste un cammino di peso non superiore a K che va da un qualche vertice iniziale al vertice di arrivo (ovvero, l'algoritmo restituisce *true* se esiste $s \in S$ tale da avere un cammino da s a t con peso minore o uguale a K, e restituisce *false* in caso contrario).

Cont.

Soluzione. Visto che i pesi sono non negativi, si può procedere ispirandosi all'algoritmo di Dijkstra. La differenza del presente problema rispetto al problema della ricerca del cammino di costo minimo da singola sorgente risolto da Dijkstra, è che in questo caso ci possono essere più possibili sorgenti. Si rende quindi necessario tener conto che tutte queste sorgenti sono a distanza 0 e che tutte devono essere prese in considerazione come possibili vertici di partenza. L'algoritmo MultiSorgente è una versione dell'algoritmo di Dijkstra che semplicemente inserisce inizialmente tutte le possibili sorgenti nella coda con priorità associando ad esse la distanza 0. La visita proseguirà quindi in ordine di distanza non decrescente rispetto ad una qualche possibile sorgente. Appena si arriva ad un vertice a distanza superiore a K, oppure si raggiunge il vertice di destinazione t, l'algoritmo termina. Se invece la visita viene completata, abbiamo la garanzia che la destinazione non risulta essere raggiungibile da alcun nodo sorgente. Il costo computazionale dell'algoritmo MultiSorgenti nel caso pessimo coincide con il costo dell'algoritmo di Dijkstra, ovvero $O(m \log n)$ dove $m = |E|$ e $n = |V|$.

Algorithm 4: MultiSorgente(Grafo $G = (V, E, w)$, Set[Vertex] S, Vertex t, Number K) $\rightarrow$ Bool

```
/* inizializzazione strutture dati                                          */
n ← G.numNodi()
Number D[1..n]
for each vertex v ∈ V \ S do
    D[v] ← ∞
MinPriorityQueue[Int, Number] Q ← new MinPriorityQueue[Int, Number]()
for each vertex v ∈ S do
    D[v] ← 0
    Q.insert(v, D[v])
/* esecuzione algoritmo di Dijkstra                                         */
while not Q.isEmpty() do
    u ← Q.findMin()
    if D[u] > K then
        return false
    else if u == t then
        return true
    Q.deleteMin()
    for v ∈ u.adjacent() do
        if D[v] = ∞ then
            /* prima volta che si incontra v                                */
            D[v] ← D[u] + w(u, v)
            Q.insert(v, D[v])
        else if D[u] + w(u, v) < D[v] then
            /* scoperta di un cammino migliore per raggiungere v            */
            Q.decreaseKey(v, D[v] − D[u] − w(u, v))
            D[v] = D[u] + w(u, v)
return false
```

The End.