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 = 0
res = 1
while n ≥ 1 do
    n = n/3
    x = x + 1
    res = res × 2 × MYSTERY2(2x)
return res

function MYSTERY2(INT n) → INT
if n ≤ 1 then
    return 1
else
    x = 1
    for i = 1, ··· , n do
        x = 2 × x
    return MYSTERY2(2n/3) + x
```

Soluzione. Analizziamo prima il costo di MYSTERY2, la cui equazione di ricorrenza

$$T'(n) = \begin{cases} 1 & n \leq 1 \\ T'(2n/3) + n & n > 1 \end{cases}$$

può essere risolta con il Master Theorem

$$\alpha = \log_{3/2} 1 = 0 \text{ and } \beta = 1 \Rightarrow T'(n) = \Theta(n^\beta) = \Theta(n)$$

La funzione MYSTERY1 richiama la funzione MYSTERY2 con input $2x$, dove x è incrementato di 1 in ogni iterazione. Il costo totale di MYSTERY2 è quindi

$$T(n) = T'(2*1) + T'(2*2) + \cdots + T'(2*k) = \Theta(2*1 + 2*2 + \cdots + 2*k) = \Theta\left(2\sum_{x=1}^{k} x\right) = \Theta(k^2)$$

Per completare l'analisi è sufficiente calcolare il valore massimo x, che corrisponde al numero di iterazioni k del ciclo while in MYSTERY1.

Iterazione	1	2	$\cdots$	k
valore di n	n/3	$n/3^2$	$\cdots$	$n/3^k$
valore di x	1	2	$\cdots$	k

Il ciclo while termina quando $n/3^k < 1$, da cui possiamo ricavare che il ciclo while termina quando $k > \log_3 n$. Fissiamo $k = \log_3 n + c$, dove $c \geq 0$ è un valore costante. Il costo totale $T(n)$ di MYSTERY1 è definito da

$$T(n) == \Theta\left((\log_3 n + c)^2\right) = \Theta(\log^2 n)$$

2. Consideriamo un array A contenente n interi compresi nell'intervallo $[1, k]$

   a) Indicare il costo nel caso pessimo per ordinare A con
   - InsertionSort
   - MergeSort
   - CountingSort

   b) Indicare qual è tra gli algoritmi di ordinamento al punto a) quello maggiormente efficiente (in termini di costo pessimo) assumendo di sapere che
   - $k = \Theta(n)$
   - $k = \Theta(n \log n)$
   - $k = \Theta(n^2)$

   Soluzione.

   a) Il costo di InsertionSort e MergeSort è indipendente da k, mentre il costo di CountingSort dipende dall'ampiezza del range $[1, k]$
   - InsertionSort: caso pessimo $O(n^2)$
   - MergeSort: caso pessimo $\Theta(n \log n)$
   - CountingSort: caso pessimo $\Theta(n + k)$

   b) Come indicato al punto precedente, soltanto il caso pessimo di CountingSort è influenzato dal valore k
   - Se $k = \Theta(n)$, CountingSort ha costo pessimo $\Theta(n)$ e risulta essere la scelta migliore tra i tre algoritmi
   - $k = \Theta(n \log n)$, CountingSort ha costo pessimo $\Theta(n \log n)$ ed, insieme al MergeSort, è l'algoritmo più efficiente
   - $k = \Theta(n^2)$, CountinSort ha costo pessimo $\Theta(n^2)$, quindi MergeSort è l'algoritmo più efficiente

3. Su una base spaziale si trovano n astronauti che devono fuggire dalla base per un inconveniente tecnico. Per fuggire dalla base, sono disponibili k diverse navicelle di salvataggio. Ogni navicella di salvataggio deve contenere esattamente un predeterminato numero di astronauti; in altri termini, data la navicella i-esima, con $i \in \{1, \ldots, k\}$, tale navicella potrà contenere esattamente $N[i]$ astronauti. Progettare un algoritmo che dato il numero di astronauti n e l'array $N[1..k]$ che indica il numero di astronauti previsto su ognuna delle k navicelle, restituisce un booleano: *true* se esiste una combinazione di navicelle che permette di far fuggire esattamente tutti gli n astronauti, *false* altrimenti.

   Soluzione. Il problema corrisponde al cosiddetto *subset sum problem*, ovvero dato un array di interi capire se esiste un sottoinsieme dei valori contenuti nell'array la cui somma sia uguale ad un dato valore in input. Tale problema può essere risolto tramite programmazione dinamica.

   In particolare, dato il numero n di astronauti e l'array $N[1..k]$ con le capienze della k navicelle, si possono considerare i seguenti problemi $P(i, j)$, con $i \in \{1, \ldots, k\}$ e $j \in \{0, \ldots, n\}$, così definiti:

   $P(i, j) = true$ se esiste un sottoinsieme delle prime i navicelle che possa portare in salvo esattamente j astronauti, altrimenti $P(i, j) = false$.

Cont.

I problemi $P(i,j)$ possono essere risolti in modo iterativo rispetto all'indice i tenendo in considerazione che:

$$P(i,j) = \begin{cases} true & \text{se } j = 0 \\ true & \text{se } i = 1,\ j > 0 \text{ e } N[1] = j \\ false & \text{se } i = 1,\ j > 0 \text{ e } N[1] \neq j \\ P(i-1,j) & \text{se } i > 1,\ j > 0 \text{ e } N[i] > j \\ P(i-1,j)\ or\ P(i-1,j-N[i]) & \text{se } i > 1,\ j > 0 \text{ e } N[i] \leq j \end{cases}$$

La soluzione al problema iniziale coincide con $P(k,n)$, ovvero il sottoproblema che considera tutti gli n astronauti e le k navicelle.

L'Algoritmo 2 risolve tutti i problemi $P(i,j)$ salvando le relative soluzioni in una matrice booleana B, e alla fine restituisce $B[k,n]$. Il costo computazionale risulta essere $\Theta(k \times n)$ in quanto vengono eseguite una quantità prefissata di operazioni di costo costante per ogni cella della matrice B, avente dimensione $k \times (n+1)$, ma $\Theta(k \times (n+1)) = \Theta(k \times n)$.

Algorithm 2: NAVICELLE(INT n, INT $N[1..k]$) → BOOLEAN

```
BOOLEAN B[1..k, 0..n]
for i ← 1 to k do
    B[i,0] ← true
for j ← 1 to n do
    if N[1] = j then
        B[1,j] ← true
    else
        B[1,j] ← false
for i ← 2 to k do
    for j ← 1 to n do
        if N[i] > j then
            B[i,j] ← B[i-1,j]
        else
            B[i,j] ← B[i-1,j] or B[i-1,j-N[i]]
return B[k,n]
```

4. Dato un grafo orientato $G = (V,E)$ ed un vertice $v \in V$ bisogna calcolare il numero di vertici fortemente connessi con v. In altri termini, progetttare un algoritmo che dati in input il grafo orientato $G = (V,E)$ e un vertice $v \in V$ restituisce in output la cardinalità del seguente insieme: $\{w \in V \mid w \text{ raggiungibile da } v \text{ e } v \text{ raggiungibile da } w\}$.

   Soluzione. L'insieme dei vertici fortemente connessi con un vertice dato v può essere calcolato facendo l'intersezione fra i vertici raggiungibili da v e quelli da cui v è raggiungibile. Il primo insieme si calcola con una visita eseguita partendo da v, mentre il secondo si può calcolare sul grafo trasposto (ottenuto invertendo la direzione degli archi) facendo sempre una visita eseguita partendo da v.

   L'Algoritmo 3 implementa questa idea utilizzando due marcature per i vertici: $mark1$ usata durante la prima visita e $mark2$ usata durante la seconda visita. Si utilizza una visita in ampiezza BFS. Si noti che tra la prima visita e la seconda si cambia l'orientamento degli archi utilizzando una funzione ausiliaria CHANGEEDGEORIENTATION; non si fornisce una implementazione di tale operazione in quanto banale, ma si osserva che assumendo l'utilizzo di liste di adiacenza tale operazione ha un costo $O(n+m)$ (con n numero dei vertici e m numero degli archi) in quanto richiede di modificare tutte le liste di adiacenza di tutti gli n vertici, costruendo nuove liste che vengono popolate mentre si leggono le liste di adiacenza originarie che contengono la descrizione degli m archi. L'algoritmo usa un contatore *counter* per calcolare il numero di vertici che vengono marcati da entrambe le visite. Il costo computazionale dell'intero algoritmo risulta essere

Cont.

$O\big(3(n+m)\big) = O\big(n+m\big)$ in quanto deve eseguire 2 visite (ognuna con costo computazionale $O(n+m)$ assumendo l'utilizzo di liste di adiacenza) più l'operazione di cambio di orientamento degli archi che ha anch'essa costo computazionale $O(n+m)$, come già commentato.

Algorithm 3: ContaFortementeConnessi(Graph $G = (V, E)$, Vertex v) → Int

```
Queue q ← new Queue()                                   // Inizializzazione strutture dati
for x ∈ V do
    x.mark1 ← false
    x.mark2 ← false
q.enqueue(v)                                            // Esecuzione prima BFS
v.mark1 ← true
while not q.isEmpty() do
    u ← q.dequeue()
    for w ∈ u.adjacents() do
        if w.mark1 == false then
            q.enqueue(w)
            w.mark1 ← true
ChangeEdgeOrientation( G )                  // Cambio dell'orientamento di tutti gli archi
Int counter ← 0
q.enqueue(v)          // Seconda BFS con conteggio dei vertici già marcati dalla prima visita
v.mark2 ← true
counter ← counter + 1
while not q.isEmpty() do
    u ← q.dequeue()
    for w ∈ u.adjacents() do
        if w.mark2 == false then
            q.enqueue(w)
            w.mark2 ← true
            if w.mark1 == true then
                counter ← counter + 1
return counter             // Restituisce il numero di vertici visitati da entrambe le visite
```