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 $n$) → INT

```
x = 0
while n ≥ 1 do
    n = n/2
    x = x + 1
return MYSTERY2(x)

function MYSTERY2(INT n) → INT
if n ≤ 0 then
    return 1
else
    x = 0
    for i = 1, ··· , n do
        for j = 1, ··· , n do
            x = x + n
    return MYSTERY2(n/2) + log n - x
```

**Soluzione**.

- Analizziamo prima la complessità di MYSTERY2, che esegue una chiamata ricorsiva su 1/2 del valore in input. Inoltre MYSTERY2 esegue un doppio ciclo for (da 1 a $n$) per calcolare il valore $x$. Tale ciclo ha un costo quadratico in $n$. L'equazione di ricorrenza di MYSTERY2 è quindi
$$T'(n) = \begin{cases} 1 & n \leq 0 \\ T'(n/2) + n^2 & n > 0 \end{cases}$$
e può essere risolta con il Master Theorem
$$\alpha = \log_2 1 = 0 < 2 = \beta \Rightarrow T'(n) = \Theta(n^\beta) = \Theta(n^2)$$
- Il costo della funzione MYSTERY1 dipende dal costo del ciclo while più il costo di una chiamata a MYSTERY2. Notiamo che la variabile $x$ viene incrementata ad ogni iterazione del ciclo while. Quindi, dato che il ciclo while viene eseguito $x$ volte e MYSTERY2 viene invocata con parametro $x$ il costo di MYSTERY1 è $\Theta(x) + \Theta(x^2)$. Dobbiamo quindi solo calcolare il valore di $x$ (in funzione di $n$), che corrisponde al numero massimo di iterazioni del ciclo while in MYSTERY1.

| Iterazione | 1 | 2 | $\cdots$ | k |
|---|---|---|---|---|
| valore di $n$ | n/2 | $n/4$ | $\cdots$ | $n/2^k$ |
| valore di $x$ | 1 | 2 | $\cdots$ | k |

Il ciclo while termina quando $n/2^k < 1$, da cui possiamo ricavare che il ciclo while termina quando $k > \log_2 n$. Fissiamo $x = k = \log_2 n + c$, dove $c \geq 0$ è un valore costante. Il costo totale $T(n)$ di MYSTERY1 è quindi definito dal costo del ciclo while più costo della chiamata a MYSTERY2

$$T(n) = \Theta(\log_2 n + c) + \Theta\left((\log_2 n + c)^2\right) = \Theta(\log^2 n)$$

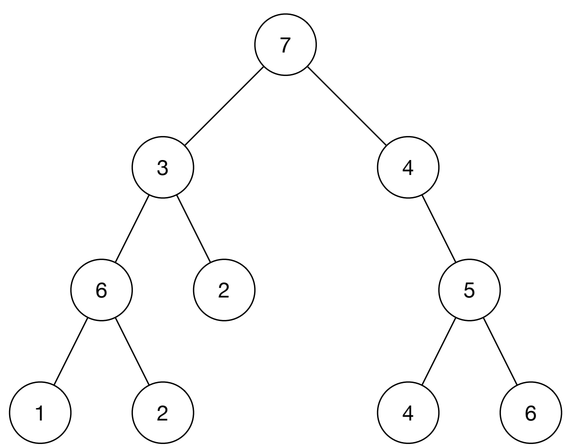
2. Considerare i seguenti valori chiave ottenuti visitando un albero binario (non di ricerca)

   - pre-ordine: 7 3 6 1 2 2 4 5 4 6
   - post-ordine: 1 2 6 2 3 4 6 5 4 7

   Disegnare un albero binario compatibile con tali visite (ce ne potrebbe essere più di uno). Spiegare come è stato individuato l'albero compatibile con le visite.

   **Soluzione**.

   - Da entrambe le visite possiamo individuare che la chiave del nodo radice è 7 (prima chiave nella visita pre-ordine ed ultima nella post-ordine)
   - Dalla visita pre-ordine sappiamo che 3 (che segue 7) è il figlio sinistro o destro (se non c'è un figlio sinistro) della radice 7
   - Dalla visita post-ordine sappiamo che 4 (che precede 7) è il figlio destro o sinistro (se non c'è un figlio destro) della radice 7
   - Dato che 3 e 4 sono chiavi distinte, l'unica possibilità è che 3 sia la chiave del figlio sinistro e 4 la chiave del figlio destro della radice
   - Possiamo quindi isolare le visite sul sottoalbero sinistro della radice
     - pre-ordine: 3 6 1 2 2
     - post-ordine: 1 2 6 2 3
   - In modo complementare le visite sul sottoalbero destro producono
     - pre-ordine: 4 5 4 6
     - post-ordine: 4 6 5 4
   - Ripetiamo il ragionamento fatto sopra su i due sottoalberi e alla fine otteniamo la seguente soluzione




   - L'unica ambiguità è sul nodo con chiave 5, che può essere indifferentemente figlio destro o sinistro del proprio padre

3. Un bambino entra in un negozio e desidera comperare il numero massimo di caramelle, compatibilmente con la quantità limitata di denaro che ha a disposizione. Più precisamente, ha a disposizione una quantità di denaro quantificata da un numero intero $D$ e nel negozio sono presenti $n$ sacchetti di caramelle che potrebbe comperare. L'$i$-esimo sacchetto, con $i \in [1, n]$ (insieme degli interi fra 1 ed $n$, estremi inclusi), ha un costo rappresentato da un numero intero $c[i]$ e contiene una quantità di caramelle quantificato con il numero $q[i]$. Progettare un algoritmo che dati in input il numero

intero $D$ e due array di interi $c[1..n]$ e $q[1..n]$ (che contengono rispettivamente i costi $c[i]$ e le quantità di caramelle $q[i]$, per ognuno degli $n$ sacchetti) restituisce la quantità massima di caramelle che il bambino può comprare. Matematicamente, l'algoritmo deve restituire il valore massimo nell'insieme $\{ \sum_{j \in J} q[j] \mid \big(J \subseteq [1,n]\big)$ *and* $\big(\sum_{j \in J} c[j] \leq D\big) \}$.

**Soluzione.** È possibile procedere utilizzando programmazione dinamica considerando i seguenti sottoproblemi

> $P(i,j)$, con $i \in [1,n]$ e $j \in [0,D]$: numero massimo di caramelle che si possono acquistare con una quantità di denaro $j$ avendo a disposizione solo i primi $i$ sacchetti di caramelle.

Tali problemi possono essere risolti induttivamente rispetto a $i$ nel seguente modo:

$$P(i,j) = \begin{cases} 0 & \text{se } i = 1 \text{ e } j < c[1] \\ q[1] & \text{se } i = 1 \text{ e } j \geq c[1] \\ P(i-1,j) & \text{se } i > 1 \text{ e } j < c[i] \\ max\{P(i-1,j),\ q[i] + P(i-1, j-c[i])\} & \text{altrimenti} \end{cases}$$

Si noti che la soluzione al problema richiesto coincide con il sottoproblema $P(n,D)$ in cui la quantità di denaro risulta essere $D$ e sono disponibili tutti gli $n$ sacchetti di caramelle.

Procediamo quindi a progettare un algoritmo (si veda Algoritmo 2) che risolve i problemi $P(i,j)$ memorizzando le relative soluzioni in una tabella $T[1..n, 0..D]$. L'algoritmo proposto ha costo

**Algorithm 2:** CARAMELLE(INT $D$, INT $c[1..n]$, INT $q[1..n]$) → INT

```
INT T[1..n][0..D]
// Inizializzazione prima riga della tabella T
for j ← 0 to D do
    if j < c[1] then
        T[1, j] ← 0
    else
        T[1, j] ← q[1]
// Riempimento restanti righe della tabella T
for i ← 2 to n do
    for j ← 0 to D do
        if (j < c[i]) or (T[i − 1, j] > q[i] + T[i − 1, j − c[i]]) then
            T[i, j] ← T[i − 1, j]
        else
            T[i, j] ← q[i] + T[i − 1, j − c[i]]
// Restituisce T[n, D]
return T[n, D]
```

computazionale $\Theta(n \times D)$, in quanto al tempo di riempimento della tabella (ogni cella della tabella $T$ richiede tempo costante per essere riempita) si aggiunge la sola operazione di ritorno del valore $T[n,D]$ che ha costo costante.

4. Progettare un algoritmo che prende in input un grafo non orientato $G = (V,E)$, con $V$ insieme di vertici e $E$ insieme di archi, ed un vertice $s \in V$ e restituisce il numero di vertici raggiungibili da $s$ in $G$. Si ricorda che un vertice $t$ è raggiungibile da $s$ se esiste un cammino da $s$ a $t$.

   **Soluzione.** È possibile usare un qualsiasi algoritmo di visita partendo dal vertice $s$ e contando i vertici che vengono visitati. Ad esempio, l'Algoritmo 3 effettua una visita in ampiezza ed utilizza la variabile ausiliaria *counter* per contare il numero di vertici visitati.

   Il costo computazionale dell'algoritmo coincide con quello della visita in ampiezza, ovvero $O(m+n)$ (con $m$ numero di archi ed $n$ numero di vertici del grafo) assumendo l'utilizzo di liste di adiacenza per rappresentare il grafo.

**Algorithm 3:** ContaVertici(Graph $G = (V, E)$, Vertex $s$) $\rightarrow$ Int

```
Queue q ← new Queue()
Int counter
for x ∈ V do
    x.mark ← false
s.mark ← true
counter ← 1
q.enqueue(s)
while not q.isEmpty() do
    u ← q.dequeue()
    for w ∈ u.adjacents() do
        if not w.mark then
            w.mark ← true
            counter ← counter + 1
            q.enqueue(w)
return counter
```

The End.