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
y = 0
for i = 1, ··· , n do
    x = x * 2
    y = y + MYSTERY2(x)
return y

function MYSTERY2(INT n) → INT
if n ≤ 0 then
    return 1
else
    return MYSTERY2(n/4) + n
```

Soluzione.

- Analizziamo prima la complessità di MYSTERY2, che esegue una chiamata ricorsiva su 1/4 del valore in input. Tutte le altre operazioni in MYSTERY2 hanno un costo costante. L'equazione di ricorrenza di MYSTERY2 è quindi
$$T'(n) = \begin{cases} 1 & n \leq 0 \\ T'(n/4) + 1 & n > 0 \end{cases}$$
e può essere risolta con il Master Theorem
$$\alpha = \log_4 1 = 0 = \beta \Rightarrow T'(n) = \Theta(n^\alpha \log n) = \Theta(\log n)$$
- La funzione MYSTERY1 richiama iterativamente n volte MYSTERY2 passando come valore in input x potenze crescenti di 2: nella prima chiamata $x = 2$, nella seconda $x = 2^2 = 4$, nell'ultima $x = 2^n$. Poiché MYSTERY2 ha un costo logaritmico, eseguire n iterazione del ciclo for in MYSTERY1 costa

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

2. Progettare un algoritmo ricorsivo che, date in input due liste concatenate semplici L_1 e L_2, contenenti chiavi intere, rimuova da L_1 tutti gli interi che compaiono anche in L_2 (senza modificare L_2). Ad esempio, se $L_1 = [2, 5, 3, 1, 10, 1]$ e $L_2 = [3, 1]$ alla fine dell'esecuzione avremo $L_1 = [2, 5, 10]$ e $L_2 = [3, 1]$.

   Soluzione. Possiamo implementare tale algoritmo in diversi modi. In ogni possibile implementazione è necessario verificare che un numero in L_1 compaia in L_2 (o viceversa). Dato che L_1 ed L_2 sono liste concatenate non ordinate, non abbiamo molte possibilità per velocizzare tale ricerca nel caso pessimo. Forniamo quindi una possibile soluzione che fa uso di una ricerca lineare. La funzione ricorsiva principale, COMPLEMENT, ritorna il puntatore alla (eventualmente nuova) testa della ista L_1. Tale funzione deve essere chiamata nel seguente modo

$$L1.head = \text{COMPLEMENT}(L1.head, L2)$$

Algorithm 2: COMPLEMENT(NODE L_1, LIST L_2) $\rightarrow$ NODE

```
if L1 == NIL or L2.head == NIL then
    return L1
else if SEARCH(L2.head, L1.key) then
    return COMPLEMENT(L1.next, L2)
else
    L1.next = COMPLEMENT(L1.next, L2) return L1

function SEARCH(NODE L, INT x) → BOOL
while L ≠ NIL and L.key ≠ x do
    L = L.next
return L ≠ NIL
```

   Assumiamo $|L_1| = m$ e $|L_2| = n$. La funzione COMPLEMENT visita interamente la lista L_1 quindi una esecuzione ha costo $\Omega(m)$

   - Caso pessimo: $\Theta(mn)$ (intersezione vuota tra L_1 ed L_2, quindi la funzione SEARCH costa $\Theta(n)$)
   - Caso ottimo: $\Theta(m)$ (se L_1 contiene un solo numero ripetuto e tale numero è in testa a L_2, la funzione SEARCH ha costo $O(1)$)
   - Caso medio: $\Theta(mn)$ (il caso medio dipende dal tempo medio di ricerca su L_2, che è $\Theta(n/2) = \Theta(n)$ dato che si tratta di una ricerca lineare)

3. Progettare un algoritmo che riceve in input un array di numeri (che possono essere sia positivi che negativi) ordinati in modo non decrescente ed un numero positivo K, e che restituisce in output il numero di valori nell'array inclusi nell'intervallo limitato chiuso $[-K, K]$ (ovvero restituisce il numero di elementi x dell'array tali che $-K \leq x \leq K$).

   Soluzione. Si può procedere con un approccio divide-et-impera che evita di fare il passo ricorsivo se è possibile verificare immediatamente che tutti gli elementi del sottovettore da considerare ricadono tutti, oppure nessuno, nell'intervallo $[-K, K]$.

   L'Algorithm 3 invoca una funzione ricorsiva che ad ogni chiamata dimezza la distanza fra i parametri i e f indicanti inizio e fine del sottovettore da considerare. Si consideri ora un certo

Cont.

Algorithm 3: ContaElementiInIntervallo($Real\ A[1..n], Real\ K) \rightarrow Int$

```
return ContaDivideEtImpera(A, K, 1, n)

Function ContaDivideEtImpera(Real A[1..n], Real K, Int i, Int f) → Int
if i > f then
    return 0
else if A[i] ≥ −K and A[f] ≤ K then
    return f − i + 1
else if A[i] > K or A[f] < −K then
    return 0
else
    Int m = Floor((i + f)/2)
    return ContaDivideEtImpera(A, K, i, m) + ContaDivideEtImpera(A, K, m + 1, f)
```

livello di annidamento delle chiamate ricorsive: solo due istanze della funzione a tale livello (quelle relative a sottovettori $A[i..f]$ che contengono sia valori in $[-K, K]$ che fuori) effettueranno le 2 chiamate ricorsive. Quindi ad ogni livello al più 4 istanze vengono eseguite. Il numero massimo di livelli di annidamenti è logaritmico in quanto, come detto sopra, 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. Progettare un algoritmo che riceve in input un grafo orientato $G = (V, E)$, un vertice $v \in V$, ed un numero positivo D, e che restituisce in output il numero di vertici a distanza D da v. Si ricorda che la distanza di un vertice v_2 da un vertice v_1 è il numero minimo di archi da attraversare per spostarsi da v_1 a v_2.

   Soluzione. Si può effettuare una visita in ampiezza BFS a partire dal vertice v in quanto tale visita esplora i vertice in ordine di distanza dal vertice di partenza. È sufficiente contare i vertici a distanza D, evitando di inserire nella coda vertici a distanza superiore che non è necessario visitare. L'Algorithm 4 implementa questa idea; si noti che si utilizza la distanza ∞ per indicare vertici non ancora visitati.

   Per quanto riguarda il costo computazionale, il caso pessimo si verifica quando tutti i vertici del grafo sono a distanza da v minore o uguale a D. In questo caso l'algoritmo visita l'intero grafo e quindi il costo computazionale risulta coincidere con il costo di una intera visita BFS, ovvero il costo sarà in $O(n+m)$, con $n = |V|$ e $m = |E|$, assumendo implementazione del grafo tramite liste di adiacenza.

Cont.

Algorithm 4: ContaVerticiADistanza(Graph $G = (V, E)$, Vertex v, Nat D) $\rightarrow$ Nat

```
/* Inizializzazione strutture dati                                    */
Nat counter ← 0
Queue q ← new Queue()
for x ∈ V do
    x.distance ← ∞
v.distance ← 0
q.enqueue(v)
/* Esecuzione BFS                                                     */
while not q.isEmpty() do
    u ← q.dequeue()
    if u.distance == D then
        /* Vertice a distanza D                                       */
        counter ← counter + 1
    else
        /* Vertice a distanza inferiore a D                           */
        for w ∈ u.adjacents() do
            if w.distance == ∞ then
                /* Vertice non ancora visitato                        */
                w.distance ← u.distance + 1
                q.enqueue(w)
return counter
```

The End.