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)

```
if n ≤ 0 then
    return 1
else
    x = 0
    while x^2 < n do
        x = x + 1
    return MYSTERY1(n/4) + MYSTERY1(n/4) + MYSTERY2(x) + x^2

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

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 funzione MYSTERY1 esegue due chiamate ricorsive su input $n/4$ e una chiamata alla funzione MYSTERY2 con input x, dove il valore x rappresenta il numero di volte che il ciclo while in MYSTERY1 viene eseguito. Tale ciclo while termina quando $x^2 \geq n$, e dato che x viene incrementato di 1 ad ogni iterazione concludiamo che al termine del ciclo $x \geq \sqrt{n}$ e che il ciclo stesso viene eseguito $\Theta(\sqrt{n})$ volte. Dato che una chiamata a MYSTERY2 ha costo logaritmico, l'equazione di ricorrenza che descrive MYSTERY1 è la seguente:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/4) + \sqrt{n} + \log\sqrt{n} & n > 1 \end{cases} = \begin{cases} 1 & n \leq 1 \\ 2T(n/4) + \sqrt{n} & n > 1 \end{cases}$$

Tale equazione di ricorrenza può essere risolta con il Master Theorem

$$\alpha = \log_4 2 = 1/2 = \beta \Rightarrow T(n) = \Theta(n^\alpha \log n) = \Theta(\sqrt{n} \log n)$$

2. Consideriamo un array di interi $A[1, \cdots, n]$ ed un intero k. Il problema noto come **Two-Sum** chiede di verificare se in A esistano due elementi la cui somma sia esattamente k. Il seguente algoritmo *brute-force* risolve il problema *Two-Sum*.

Algorithm 2: TWOSUM(INT $A[1, \cdots, n]$, INT k) → BOOL

```
for i = 1, ··· , n − 1 do
    for j = i + 1, ··· , n do
        if A[i] + A[j] == k then
            return true
return false
```

a) Analizzare la complessità computazionale dell'algoritmo brute-force nel caso pessimo e medio

b) Progettare un algoritmo più efficiente dell'algoritmo brute-force nel caso pessimo o medio

Soluzione.

a) L'algoritmo brute-force valuta nel caso pessimo $n-1+n-2+\cdots+1 = n(n-1)/2 = \Theta(n^2)$ coppie di interi in A. Assumendo probabilità uniforme, in media l'algoritmo brute-force analizzerà metà di tali coppie di interi prima di trovare una soluzione, quindi $n(n-1)/4 = \Theta(n^2)$ coppie. Il costo nel caso pessimo e medio è quindi quadratico sul numero di nodi $\Theta(n^2)$.

b) Un algoritmo più efficiente dell'algoritmo brute-force può essere ottenuto ordinando prima A e poi verificando con una sola scansione lineare coppie di numeri in A, partendo dalla coppia formata dal più piccolo e dal più grande numero in A.

Algorithm 3: TWOSUM2(INT $A[1, \cdots, n]$, INT k) → BOOL

```
MERGESORT(A)
i = 1
j = n
while i < j do
    if A[i] + A[j] == k then
        return true
    else if A[i] + A[j] < k then
        i = i + 1
    else
        j = j − 1
return false
```

Ordinare A col Mergesort ha costo $\Theta(n \log n)$ nel caso medio e pessimo. Il ciclo while è eseguito n volte nel caso pessimo (nessuna coppa di numeri ha somma uguale a k). Il Mergesort domina comunque i costi, quindi TWOSUM2 ha costo pessimo e medio $\Theta(n \log n)$, più efficiente del costo medio e pessimo di TWOSUM.

Una possibile alternativa è quella di usare una Tabella Hash per mantenere traccia dei numeri già visti.
Le operazioni di inserimento e ricerca su una Tabella Hash hanno costo medio costante $O(1)$, quindi nel caso medio TWOSUM3 ha costo medio lineare $O(n)$ e risulta essere più efficiente di TWOSUM e TWOSUM2. D'altra parte, nel caso pessimo le operazioni di inserimento e ricerca su una Tabella Hash hanno un costo lineare sul numero di elementi della tabella (assumendo che la Tabella Hash gestisca le collisioni con concatenazione). In questo caso, il costo pessimo è dato da $2\sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$. Concludiamo che, nel caso pessimo TWOSUM3 è asintoticamente equivalente a TWOSUM.

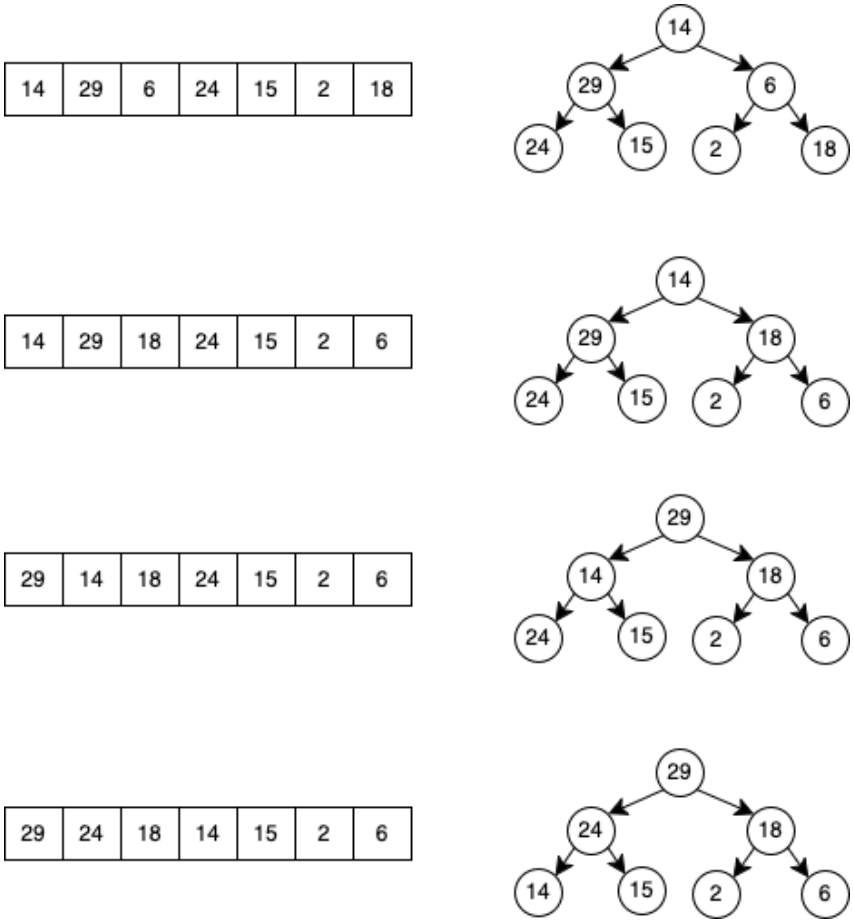
Cont.

Algorithm 4: TWOSUM3(INT $A[1, \cdots, n]$, INT k) $\rightarrow$ BOOL

```
H = HashTable()
for i = 1 ··· , n do
    if search(H, k − A[i]) == true then
        return true // The number k − A[i] is in A
    else
        insert(H, A[i], true) // Insert the pair (A[i], true) in the Hash Table
return false
```

3. Si consideri un array che contiene i valori $[14, 29, 6, 24, 15, 2, 18]$. Descrivere in che modo viene modificato tale array se viene passato alla funzione *heapify* per trasformarlo in un max-heap.

   Soluzione. Per descrivere in che modo viene modificato l'array è preferibile riportare anche la rappresentazione ad albero binario dell'array. L'immagine sotto riporta a sinistra le modifiche apportate all'array e a destra il corrispondente albero heap. La prima modifica è lo scambio dei valori 6 e 18 per rendere max-heap il sottoalbero destro della radice (si noti che il sottoalbero sinistro è già un max-heap). Successivamente, deve essere sistemata la radice che comporta uno scambio fra 14 e 29, e a seguire lo scambio fra 14 e 24.



4. Si considerino n variabili $x_1, x_2, \ldots, x_n$ e k disequazioni $x_{l[i]} < x_{m[i]}$ (con $i \in [1..k]$) dove $l[i]$ e $m[i]$ sono interi compresi fra 1 e n. Bisogna capire se esiste un assegnamento di valori reali alle variabili $x_1, x_2, \ldots, x_n$ che soddisfano tutte le k disequazioni. Ad esempio, se $n = 3$ e le disequazioni sono $x_1 < x_2$ e $x_2 < x_3$ allora esiste un assegnamento, mentre se $n = 2$ e le disequazioni sono $x_1 < x_2$ e $x_2 < x_1$ allora non esiste un assegnamento. Bisogna progettare un algoritmo che prende in input il numero naturale n e i due array $l[1..k]$ e $m[1..k]$, che rappresentano le k disequazioni $x_{l[i]} < x_{m[i]}$, e restituisce *true* se è possibile assegnare alle variabili $x_1, x_2, \ldots, x_n$ dei valori reali che soddisfano tutte le disequazioni, *false* altrimenti.

Cont.

Soluzione. Il sistema di disequazioni risulta non soddisfacibile se e solo se esiste una sequenza "circolare" di disequazioni che per transitività implica che una variabile debba essere strettamente minore di se stessa. Ad esempio $x_1 < x_2$ e $x_2 < x_1$ è una di tali sequenze in quanto implica l'assurdo $x_1 < x_1$. Un modo possibile per cercare questi tipi di sequenze di disequazioni consiste nel trasformare le disequazioni in un grafo orientato G, con i nodi $x_1, x_2, \ldots, x_n$ e gli archi $(x_{l[i]}, x_{m[i]})$ (corrispondenti alle disequazioni $x_{l[i]} < x_{m[i]}$) per poi verificare la presenza di cicli in G. Per cercare la presenza di un ciclo in un grafo orientato si può eseguire una DFS (vista in profondità) e controllare l'esistenza di almeno uno dei cosiddetti archi all'indietro.

Algorithm 5: DisequazioniSoddisfacibili(Natural n, Integer $l[1..k]$, Integer $m[1..k]$) → Bool

```
Graph G ← new Graph()
for i ∈ [1..n] do
    G.addVertex(x_i)
for j ∈ [1..k] do
    G.addEdge(x_{l[j]}, x_{m[j]})

 // Ricerca loop tramite DFS
global Bool loop = false
for each vertex v in G do
    v.mark ← white ;
for each vertex v in G do
    if (not loop) and (v.mark = white) then
        DFSvisit(v)
 // Si controlla se si è trovato il ciclo
if loop then
    return false
else
    return true

DFSvisit(Vertex u)
u.mark ← grey
for v ∈ u.adjacents do
    if v.mark = grey then
        loop ← true
    if (not loop) and (v.mark = white) then
        DFSvisit(v)
u.mark ← black
```

Tale soluzione è riportata come Algoritmo 5. Si noti l'utilizzo della variabile globale *loop* che viene settata a *true* appena si trova un ciclo. Quando *loop* viene settata a *true* non verranno più fatte invocazioni alla funzione DFSvisit. Il costo computazionale di tale algoritmo, nel caso pessimo, è il medesimo della visita in profondità, ovvero $O(n + k)$ assumendo implementazione del grafo tramite liste di adiacenza.

The End.