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) $\to$ INT

```
x = 0
for i = 2, ··· , n do
    x = x + MYSTERY2(2^100 n/i)
return x

function MYSTERY2(INT n) → INT
if n ≤ 0 then
    return 1
else
    return MYSTERY2(n/3) + MYSTERY2(n/3) + MYSTERY2(n/3) + n^3 + n^2 + n
```

Soluzione. Analizziamo prima il costo di MYSTERY2 che esegue tre chiamate ricorsive con input ridotto ad 1/3 del valore originale. Le rimanenti operazioni in MYSTERY2 hanno costo costante. L'equazione di ricorrenza di MYSTERY2:

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

può essere risolta con il Master Theorem

$$\alpha = \log_3 3 = 1 > 0 = \beta \Rightarrow T'(n) = \Theta(n^\alpha) = \Theta(n)$$

Il costo di MYSTERY1 dipende dal costo del suo unico ciclo for. Tale ciclo viene eseguito $n-1$ volte per i che va da 2 ad n. Ad ogni iterazione viene chiamata la funzione MYSTERY2 con input $2^100n/i$. Il costo totale del ciclo for è quindi:

$$\Theta(2^{100}n/2) + \Theta(2^{100}n/3) + \cdots + \Theta(2^{100}n/n) = 2^{100}\Theta\left(\sum_{i=2}^{n} \frac{n}{i}\right) = \Theta\left(n\sum_{i=2}^{n} \frac{1}{i}\right)$$

Poiché la serie armonica $1/i$, per i che va da 2 ad n, è limitata superiormente da $\ln n$

$$\sum_{i=2}^{n} \frac{1}{i} \leq \ln n \Rightarrow \Theta\left(\sum_{i=2}^{n} \frac{1}{i}\right) = O(\log n)$$

concludiamo che la complessità nel caso pessimo di MYSTERY1 è

$$T(n) = \Theta\left(n \sum_{i=2}^{n} \frac{1}{i}\right) = O(n \log n)$$

2. Considerare una Tabella Hash T di dimensione $m = 11$, inizialmente vuota. La funzione hash è definita sul metodo della divisione

$$h'(k) = k \mod m$$

e abbiamo la seguente sequenza di operazioni:

1) insert 39 2) insert 15 3) insert 18
4) insert 17 5) insert 28 6) insert 6

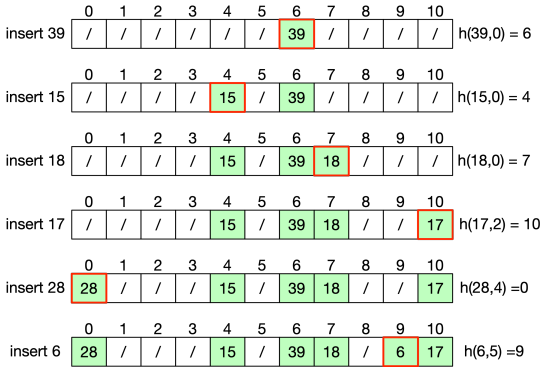
a) Mostrare lo stato di T dopo l'esecuzione delle operazioni precedenti, assumendo che le collisioni in T siano gestire con indirizzamento aperto e ispezione quadratica

$$h(k, i) = (h'(k) + i^2) \mod m$$

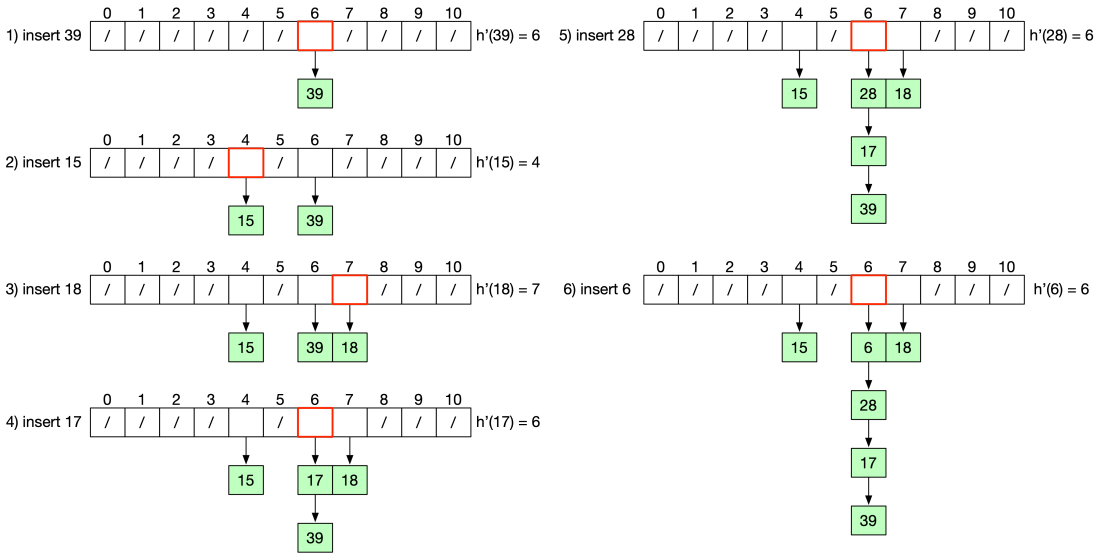
b) Mostrare lo stato di T dopo l'esecuzione delle operazioni precedenti, assumendo che le collisioni in T siano gestire con concatenamento

Soluzione.

a) Indirizzamento aperto



b) Concatenamento



3. Progettare un algoritmo che, dato un array di numeri $A[1..n]$, restituisce un array $MAX[1..\lceil \log n \rceil]$ (dove $\lceil \log n \rceil$ è l'arrotondamento all'intero superiore di $\log n$) con i $\lceil \log n \rceil$ valori più grandi presenti nell'array in input. L'algoritmo deve possibilmente avere una complessità in tempo migliore di $\Theta(n \log n)$.

Soluzione.

Il problema potrebbe essere risolto utilizzando le operazione *heapify*, *findMax* e *deleteMax* utilizzate dall'algoritmo heapsort: *heapify* può essere usato per trasformare l'array A in un heap, e successivamente si eseguono per $\lceil \log n \rceil$ volte le operazioni *findMax* e *deleteMax* per estrarre il massimo valore rimasto in A ed inserirlo in un nuovo array che al termine viene restituito.

Algorithm 2: LogNmassimi(*Number* $A[1..n]$) $\rightarrow$ *Number*$[1..\lceil \log n \rceil]$

```
Number MAX[1..⌈log n⌉]
heapify(A)
for i = 1 to ⌈log n⌉ do
    MAX[i] = findMax(A)
    deleteMax(A)
return MAX
```

Il costo computazionale dell'algoritmo somma al costo $O(n)$ di *heapify* il costo di $\lceil \log n \rceil$ esecuzioni di *findMax* e *deleteMax*, che hanno rispettivamente costo $O(1)$ e $O(\log n)$. Quindi complessivamente avremo complessità in tempo pari a $O(n + \log n + \log n \log n) = O(n)$ (in quanto l'ordine di grandezza $O(n)$ è superiore sia dell'ordine di grandezza $O(\log n)$ sia dell'ordine di grandezza $O(\log n \log n)$).

4. Progettare un algoritmo che dato un grafo orientato $G = (V, E)$ e due vertici s e t restituisce la distanza di t da s (ovvero il numero mimimo di archi da attraversare per andare da s a t). Nel caso in cui t non sia raggiungibile da s l'algoritmo deve restituire ∞.

   Soluzione.

   È sufficiente effettuare una visita in ampiezza del grafo a partire da s e restituire la distanza di t. L'algoritmo (si veda Algoritmo 3) corrisponde con l'algoritmo BFS visto a lezione in cui si usa il

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

```
Queue q ← new Queue()
for x ∈ V do
    x.dist ← ∞
s.dist ← 0
q.enqueue(s)
while not q.isEmpty() do
    u ← q.dequeue()
    for w ∈ u.adjacents() do
        if w == t then
            return u.dist+1
        else if w.dist == ∞ then
            w.dist ← u.dist + 1
            q.enqueue(w)
return ∞
```

campo *dist* anche per tener traccia dei vertici già visitati (per capire se un vertice w non è ancora stato visitato basta controllare se $w.dist == \infty$) con l'aggiunta di un controllo che termina la visita appena si raggiunge il vertice t. Infine, se t non viene visitato, l'algoritmo restituisce ∞.

L'algoritmo, nel caso pessimo, ha il medesimo costo computazionale della visita in ampiezza che, assumendo implementazione del grafo tramite liste di adiacenza, risulta essere $O(n + m)$ dove $n = |V|$ e $m = |E|$.

The End.