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

```
if n ≤ 0 then
    return 1
else
    return MYSTERY1(n − 1)+MYSTERY2(n)

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

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
$$\alpha = \log_4 1 = 0 = \beta \Rightarrow T'(n) = \Theta(n^\alpha \log n) = \Theta(n^0 \log n) = \Theta(\log n)$$
- La funzione MYSTERY1 richiama MYSTERY2 una sola volta, passando come valore in input n. Dato che una chiamata a MYSTERY2 ha costo logaritmico, abbiamo che tale chiamata ha costo $\Theta(\log n)$. La funzione MYSTERY1 esegue anche una chiamata ricorsiva con valore in input $n-1$. Dato che tutte le altre operazioni hanno un costo costante, l'equazione di ricorrenza di MYSTERY1 è
$$T(n) = \begin{cases} 1 & n \leq 0 \\ T(n-1) + \log n & n > 0 \end{cases}$$

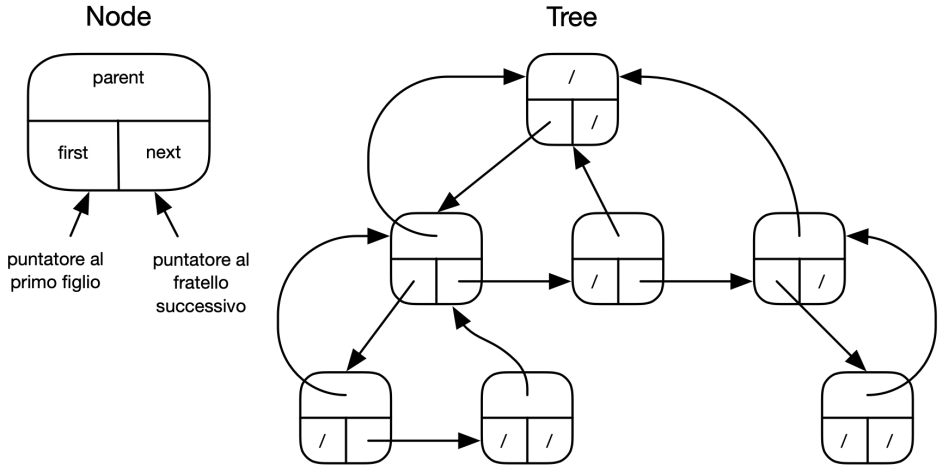
Tale equazione di ricorrenza può essere risolta con il metodo iterativo:

$$\begin{aligned} T(n) &= T(n-1) + \log n \\ &= T(n-2) + \log(n-1) + \log n \\ &= T(n-3) + \log(n-2) + \log(n-1) + \log n \\ &= ... \\ &= T(n-i) + \sum_{k=0}^{i-1} \log(n-k) \end{aligned}$$

La ricorsione termina quando $n - i = 0 \Rightarrow i = n$. Sostituendo $i = n$ nell'equazione sopra otteniamo (ricordiamo che per le proprietà dei logaritmi $\log a + \log b = \log ab$ e che $\log n! = \Theta(n \log n)$)

$$\begin{aligned} T(n) &= T(0) + \sum_{k=0}^{n-1} \log(n-k) \\ &= 1 + \sum_{k=1}^{n} \log k \\ &= 1 + \log\left(\prod_{k=1}^{n} k\right) = 1 + \log n! \\ &= \Theta(n \log n) \end{aligned}$$

2. Sia T una struttura dati albero non-binario come rappresentato in figura



a) Implementare un algoritmo ricorsivo che ritorna true se e solo se T è una lista, cioè se ogni nodo di T ha al massimo un figlio

b) Analizzare il costo dell'algoritmo proposto nel caso pessimo e ottimo

Nota. Su un albero vuoto l'algoritmo deve ritornare true (un albero vuoto può essere interpretato come una lista vuota)

Soluzione.

a) Per verificare se un nodo abbia più di un figlio, è sufficiente verificare che il campo next di ogni nodo sia NIL. Se questa condizione risulta essere vera allora ci sarà nell'albero un nodo con almeno due figli. In tal caso ritorniamo false.

b) Sia n in numero di nodi nell'albero
   - Caso pessimo: $\Theta(n)$ (ogni nodo in T ha un solo figlio, dobbiamo visitarli tutti per poterlo verificare)
   - Caso ottimo: $O(1)$ (il nodo radice ha due figli)

Cont.

Algorithm 2: ISLIST(NODE T) $\rightarrow$ BOOL

```
if T == NIL then
    return true
else if T.next ≠ NIL then
    return false
else
    return ISLIST(T.first)
```

3. Progettare un algoritmo che dati in input un array di numeri $A[1..n]$ ordinato in modo crescente ed un numero K, restituisce il più piccolo numero in A maggiore o uguale a K. Ad esempio, se A contiene i numeri $-10, 7, 15, 29, 33$ e K vale 13, l'algoritmo deve restituire 15. Se tutti i numeri in A sono minori di K, l'algoritmo deve restituire ∞.

   Soluzione. È possibile adottare una tecnica divide-et-impera utilizzando una funzione ausiliaria ricorsiva, simile alla ricerca binaria, che ricorsivamente controlla la posizione media in una data porzione dell'array. Se il valore in tale posizione intermedia risulta essere il più piccolo numero in A maggiore o uguale a K lo restituisce, altrimenti effettua una chiamata ricorsiva procedendo la ricerca sulla prima oppure sulla seconda metà della porzione dell'array. Tale algoritmo, si veda Algoritmo 3, deve essere inizialmente invocato con RICERCA$(A, K, 1, n)$ e risulta avere il medesimo costo computazionale della ricerca binaria, ovvero $O(\log n)$.

Algorithm 3: RICERCAMAGGIOREUGUALE(NUMBER $A[1..n]$, NUMBER K) $\rightarrow$ NUMBER

```
// chiamata alla funzione ausiliaria di ricerca
return RICERCA(A, K, 1, n)

// funzione ausiliaria di ricerca
function RICERCA (NUMBER A[1..n], NUMBER K, INT i, INT j) → NUMBER
if i > j then
    return ∞
else
    INT m ← Floor((i + j)/2)
    if A[m] ≥ K and (m = 1 or A[m − 1] < K) then
        return A[m]
    else if A[m] < K then
        return RICERCA(A, K, m+1, j)
    else
        return RICERCA(A, K, i, m-1)
```

4. Progettare un algoritmo che prende in input un grafo orientato $G = (V, E)$, con V insieme di vertici e E insieme di archi orientati, e due vertici $s, t \in V$ e stampa, se esiste, un cammino da s a t di lunghezza minima, ovvero che attraversa il numero minimo di archi.

   Soluzione. È sufficiente effettuare una visita in ampiezza del grafo a partire da s, e nel caso in cui t venga visitato stampare il cammino da s a t trovato dalla visita. L'algoritmo (si veda Algoritmo 4) ha il medesimo costo computazionale della visita in ampiezza che, assumendo implementazione tramite liste di adiacenza, risulta essere $O(|V| + |E|)$.

Cont.

Algorithm 4: CamminoLunghezzaMinima(Graph $G = (V, E)$, Vertex s, Vertex t)

```
Vertex u, v, w
Queue q ← new Queue()
for v ∈ V do
    v.mark ← false
s.mark ← true
q.enqueue(s)
while (not q.isEmpty()) AND (not t.mark) do
    u ← q.dequeue()
    for w ∈ u.adjacents() do
        if not w.mark then
            w.mark ← true
            w.parent ← u
            q.enqueue(w)
if t.mark then
    PrintPath(s, t)
else
    print "nodo non raggiungibile"

// funzione ausiliaria di stampa del cammino
procedure PrintPath (Vertex s, Vertex v)
if v ≠ s then
    PrintPath(s, v.parent)
print v
```

The End.