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 MYSTERY nel caso ottimo e pessimo

Algorithm 1: MYSTERY

```
function MYSTERY(ARRAY A[1, ··· , n], INT a, INT c)
if a < c then
    /* Divide l'intervallo [a,c] in tre intervalli di simile ampiezza, [a,b],[b+1,c],[a',c'] */
    b = (a + c)/2 /* Punto medio in [a,c]                                                      */
    a' = (a + b)/2 /* Punto medio in [a,b]                                                     */
    c' = (b + 1 + c)/2 /* Punto medio in [b+1,c]                                               */
    /* Ordina il sotto-array centrale                                                          */
    INSERTIONSORT(A, a', c')
    /* Chiamate ricorsive sui tre sotto-array                                                  */
    MYSTERY(A, a, b)
    MYSTERY(A, b + 1, c)
    MYSTERY(A, a', c')
```

Soluzione.

- La complessità di MYSTERY dipende dalla complessità di INSERTIONSORT, che per il momento indichiamo con $T'(n)$. La funzione MYSTERY esegue una chiamata ad INSERTIONSORT e tre chiamate ricorsive, tutte su input di dimensione $n/2$, dove $n = c - a + 1$. L'equazione di ricorrenza di MYSTERY è quindi
$$T(n) = \begin{cases} 1 & n < 2 \\ 3T(n/2) + T'(n/2) & n \geq 2 \end{cases}$$
Il costo ottimo e pessimo dipendono dal costo ottimo e pessimo di INSERTIONSORT.
- Nel caso ottimo, INSERTIONSORT ha un costo lineare ogni volta che viene mandato in esecuzione. In questo caso, $T'(n/2) = O(n)$ e l'equazione di ricorrenza di MYSTERY diventa
$$T(n) = \begin{cases} 1 & n < 2 \\ 3T(n/2) + n & n \geq 2 \end{cases}$$
Possiamo risolvere tale equazione di ricorrenza con il Master Theorem
$$\alpha = \log_2 3 > 1 = \beta \Rightarrow T(n) = \Theta(n^\alpha) = \Theta(n^{\log_2 3})$$
Come ulteriore precisazione, notiamo che l'algoritmo termina in tempo costante quando $a \geq c$. Questo caso non può essere identificato con il caso ottimo in quanto se $a \geq c$ allora la dimensione del problema è $n = O(1)$ (cioè, l'algoritmo termina in tempo costante su input di dimensione costante).

- Nel caso pessimo, INSERTIONSORT ha un costo quadratico ogni volta che viene mandato in esecuzione. In questo caso, $T'(n/2) = O(n^2)$ e l'equazione di ricorrenza di MYSTERY diventa

$$T(n) = \begin{cases} 1 & n < 2 \\ 3T(n/2) + n^2 & n \geq 2 \end{cases}$$

  Possiamo risolvere tale equazione di ricorrenza con il Master Theorem

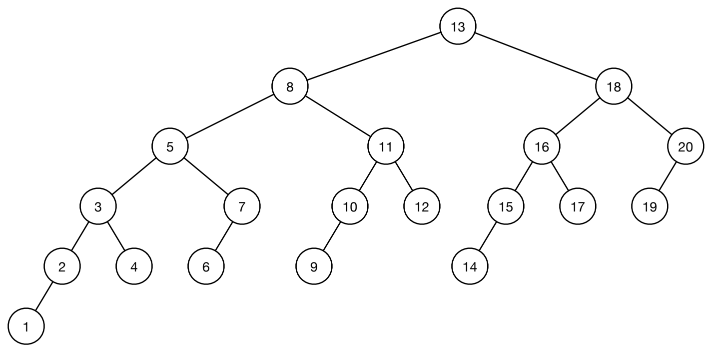
$$\alpha = \log_2 3 < 2 = \beta \Rightarrow T(n) = \Theta(n^\beta) = \Theta(n^2)$$

- In conclusione, MYSTERY ha costo $\Theta(n^{\log_2 3})$ nel caso ottimo e $\Theta(n^2)$ nel caso pessimo.

2. Dato il seguente albero AVL, effettuare le seguenti operazioni in ordine e mostrare lo stato dell'albero dopo ogni operazione

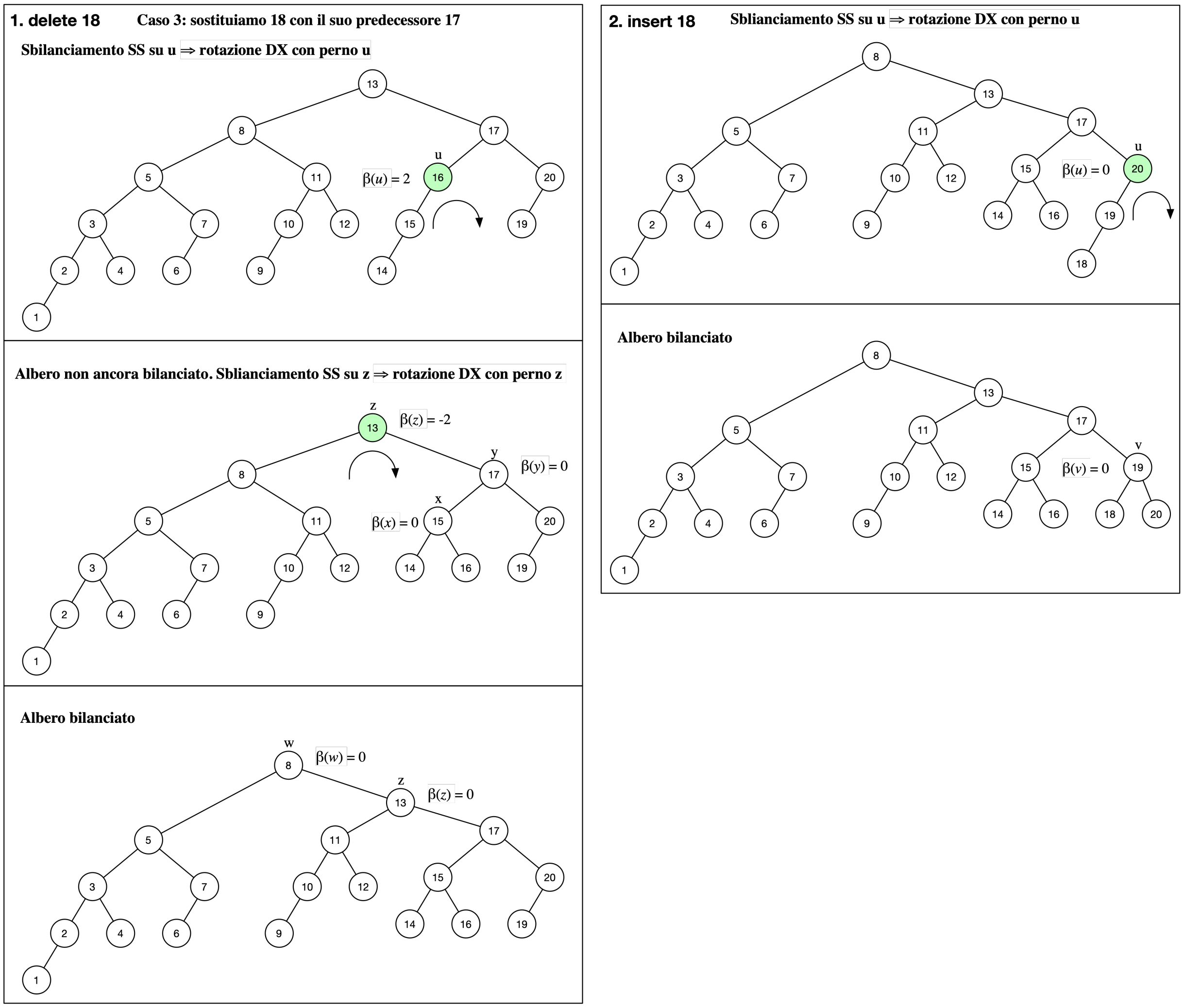
   (a) Delete 18

   (b) Insert 18



Indicare chiaramente quali operazioni sono eseguite in seguito ad inserimenti e/o rimozioni.

Soluzione. Nota. L'operazione di rimozione del nodo con chiave 18 comporta rotazioni a cascata.



3. Dato un array di numeri naturali $A[1..n]$ ed un numero naturale K, bisogna calcolare la quantità massima di numeri presi dall'array la cui somma sia esattamente K. Più precisamante, bisogna restituire quel numero intero m, più grande possibile, tale che esistono m indici diversi i_1, i_2, ..., i_m per cui $\sum_{j=1}^{m} A[i_j] = K$. Se non esiste una combinazione di numeri presi dall'array A con somma K, l'algoritmo deve restituire $-\infty$.

   Soluzione. Il problema richiede di trovare il sottoinsieme di cardinalità massima di valori presi da un array di interi $A[1..n]$ che sommati restituiscono un dato valore intero K. Tale problema può essere risolto utilizzando la programmazione dinamica, considerando i seguenti problemi $P(i,j)$, con $i \in \{1,\ldots,n\}$ e $j \in \{0,\ldots,K\}$, così definiti:

   > $P(i,j) = m$ se m è la cardinalità massima di un sottoinsieme di elementi presi tra i primi i elementi dell'array A che sommati danno j, altrimenti $-\infty$ se tale sottoinsieme non esiste.

   I problemi $P(i,j)$ possono essere risolti in modo iterativo rispetto all'indice i tenendo in considerazione che:

$$P(i,j) \;=\; \begin{cases} 0 & \text{se } j = 0 \\ 1 & \text{se } j > 0,\ i = 1 \text{ e } j = A[i] \\ -\infty & \text{se } j > 0,\ i = 1 \text{ e } j \neq A[i] \\ P(i-1,j) & \text{se } j > 0,\ i > 1 \text{ e } j < A[i] \\ max(P(i-1,j), 1 + P(i-1, j - A[i])) & \text{se } j > 0,\ i > 1 \text{ e } j >= A[i] \end{cases}$$

Cont.

La soluzione al problema iniziale coincide con $P(n, K)$, ovvero il sottoproblema che considera tutti i valori in ingresso e la somma richiesta risulta essere K.

L'Algoritmo 2 risolve tutti i problemi $P(i, j)$ salvando le relative soluzioni in una matrice P, e alla fine restituisce $P[n, K]$. Il costo computazionale risulta essere $\Theta(n \times K)$ in quanto vengono eseguite alcune operazioni di costo costante per ogni cella della matrice P, avente dimensione $n \times (K + 1)$, ma $\Theta(n \times (K + 1)) = \Theta(n \times K) + \Theta(n) = \Theta(n \times K)$.

Algorithm 2: MassimoSottoinsieme(Int $A[1..n]$, Int K) $\to$ Int

```
Int P[1..n, 0..K]
for i ← 1 to n do
    P[i, 0] ← 0
for j ← 1 to K do
    if A[1] = j then
        P[1, j] ← 1
    else
        P[1, j] ← −∞
for i ← 2 to n do
    for j ← 1 to K do
        if j < A[i] then
            P[i, j] ← P[i − 1, j]
        else
            P[i, j] ← max(P[i − 1, j], 1 + P[i − 1, j − A[i]])
return P[n, K]
```

4. Si decide di collegare le n isole di un arcipelago tramite $n - 1$ ponti, in modo tale che tutte le isole siano fra loro collegate in modo diretto oppure in modo indiretto tramite una sequenza di ponti. Assumiamo di numerare le n isole con i numeri interi compresi fra 1 e n e di usare una matrice C, di tipo $Real[1..n, 1..n]$, per indicare il costo per costruire i possibili ponti di collegamento: in particolare, $C[i, j]$ indica il costo per costruire il ponte di collegamento diretto tra la isola i e la isola j (ovviamente, assumiamo $C[i, j] = C[j, i]$ in quanto i ponti collegano le isole in entrambe le direzioni). Bisogna progettare un algoritmo che riceve in input la matrice dei costi C e che restituisce in output il costo complessivo minimo per costruire $n - 1$ ponti che colleghino fra di loro (in modo diretto o indiretto) tutte le n isole dell'arcipelago.

   Soluzione. Il problema risulta essere una istanza del problema del calcolo del minimo albero di copertura (minimum spanning tree – MST). Si noti che in questo caso, il grafo in input è un grafo non orientato pesato completo con n vertici numerati da 1 a n (un vertice per ogni isola) e con il peso dell'arco (i, j) coincidente con $C[i, j]$.

   L'Algoritmo 3 utilizza l'algoritmo di Prim per calcolare il MST iniziando la costruzione del MST dal vertice 1. Durante la costruzione del MST, si utilizza il contatore *tot* per sommare i costi degli $n - 1$ archi selezionati. Al termine dell'algoritmo, viene restituita tale sommatoria. Il costo computazionale dell'algoritmo risulta essere, nel caso pessimo, $O(n^2 \log n)$ in quanto il grafo è completo, e quindi contiene $O(n^2)$ archi, e per ogni arco (nel caso pessimo) potrebbe essere necessario eseguire una *insert* o una *decreaseKey* sulla coda con priorità (entrambe operazioni di costo logaritmico nella dimensione della coda).

Algorithm 3: CoperturaIsole(Real[1..n,1..n] C $\rightarrow$ Real)

```
// inizializzazione strutture dati
NUMBER D[1..n], tot = 0
INT u, v
BOOL covered[1..n]
for i = 1 to n do
    D[i] = ∞
    covered[i] = false
D[1] = 0
MINPRIORITYQUEUE[INT, NUMBER] Q = new MINPRIORITYQUEUE[INT, NUMBER]()
Q.insert(1, D[1])
// esecuzione algoritmo di Prim
while not Q.isEmpty() do
    u = Q.findMin()
    Q.deleteMin()
    covered[u] = true
    tot = tot + D[u]
    for v ∈ [1..n] s.t. not covered[v] do
        if D[v] == ∞ then
            // prima volta che si incontra v
            Q.insert(v, C[u, v])
            D[v] = C[u, v]
        else if C[u, v] < D[v] then
            // scoperta di un arco migliore per raggiungere v
            Q.decreaseKey(v, D[v] − C[u, v])
            D[v] = C[u, v]
return tot
```

The End.