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
    x = MYSTERY2(n^4)
    return x+MYSTERY1(n/2)+MYSTERY1(n/2)+MYSTERY1(n/2)+MYSTERY1(n/2)

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

Soluzione.

- Analizziamo prima la complessità di MYSTERY2, che esegue due chiamate 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
$$T'(n) = \begin{cases} 1 & n \leq 0 \\ 2T'(n/4) + 1 & n > 0 \end{cases}$$
e può essere risolta con il Master Theorem
$$\alpha = \log_4 2 = 1/2 > 0 = \beta \Rightarrow T'(n) = \Theta(n^\alpha) = \Theta(n^{1/2}) = \Theta(\sqrt{n})$$
- La funzione MYSTERY1 richiama MYSTERY2 una sola volta, passando come valore in input n^4. Dato che una chiamata a MYSTERY2 ha costo proporzionale alla radice quadrata del valore in input, abbiamo che tale chiamata a MYSTERY2 costa $\Theta(\sqrt{n^4}) = \Theta(n^2)$. La funzione MYSTERY1 esegue anche 4 chiamate ricorsive su 1/2 del valore in input. Tutte le altre operazioni hanno un costo costante. L'equazione di ricorrenza di MYSTERY1 è quindi
$$T(n) = \begin{cases} 1 & n \leq 0 \\ 4T(n/2) + n^2 & n > 0 \end{cases}$$
e può essere risolta con il Master Theorem
$$\alpha = \log_2 4 = 2 = \beta \Rightarrow T(n) = \Theta(n^\alpha \log n) = \Theta(n^2 \log n)$$

2. Data una Tabella Hash di dimensione $m = 11$, inizialmente vuota, in cui le collisioni sono risolte con indirizzamento aperto ed ispezione quadratica

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

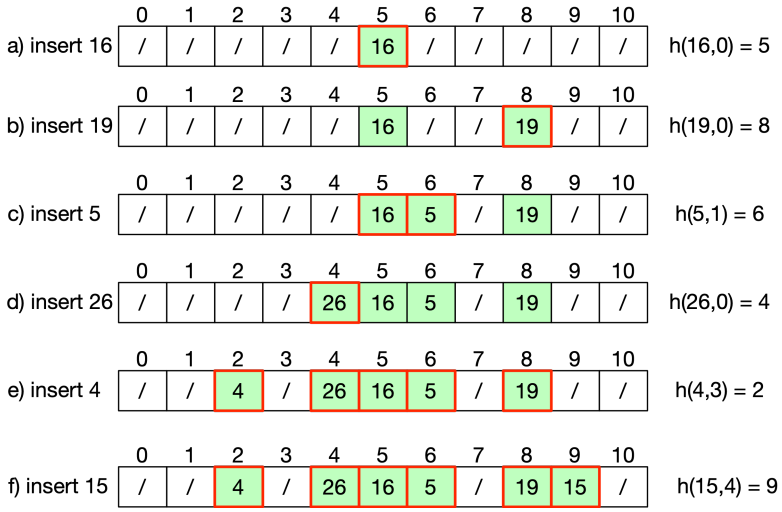
dove $h'(k)$ è la funzione ausiliaria

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

mostrare lo stato della tabella dopo ognuna delle seguenti operazioni:

a) insert 16 b) insert 19 c) insert 5
d) insert 26 e) insert 4 f) insert 15

Soluzione



3. Si consideri il seguente problema: a partire da n oggetti a disposizione, si devono impilare uno sopra l'altro la quantità massima di tali oggetti facendo attenzione che l'altezza ed il peso della pila creata non superino delle soglie date. Più precisamente bisogna progettare un algoritmo che dati in input due array di interi $h[1..n]$ (che indica le altezze degli n oggetti), $p[1..n]$ (che indica i pesi degli n oggetti), e due numeri interi H e P, restituisce il numero massimo di oggetti, presi fra gli n disponibili, che possono essere impilati creando una pila di altezza minore o uguale a H e peso minore o uguale a P.

Soluzione. È possibile utilizzare la programmazione dinamica considerando i problemi $T(i, j, k)$, con $i \in \{1, \ldots, n\}$, $j \in \{0, \ldots, H\}$ e $k \in \{0, \ldots, P\}$, tale che

> $T(i, j, k)$ = massimo numero di oggetti impilabili considerando solo i primi i oggetti, massima altezza complessiva delle pila j e massimo peso della pila k.

Una volta risolti tutti tali sottoproblemi, la soluzione al problema iniziale coinciderà con $T(n, H, P)$. I problemi $T(i, j, k)$ possono essere risolti considerando che:

$$T(i,j,k) = \begin{cases} 0 & \text{se } i = 1 \text{ e } (h[1] > j \text{ o } p[1] > k) \\ 1 & \text{se } i = 1 \text{ e } h[1] \leq j \text{ e } p[1] \leq k \\ T(i-1,j,k) & \text{se } i > 1 \text{ e } (h[i] > j \text{ o } p[i] > k) \\ max(T(i-1,j,k), 1 + T(i-1, j-h[i], k-p[i])) & \text{se } i > 1 \text{ e } h[i] \leq j \text{ e } p[i] \leq k) \end{cases}$$

L'Algoritmo 2 risolve i problemi $T(i, j, k)$ inserendo le relative soluzioni nella tabella tridimensionale $T[1..n, 0..H, 0..P]$. Una volta riempita la tabella, restituisce il valore inserito in $T[n, H, P]$. Il costo computazionale di tale algoritmo risulta essere $T(n, H, P) = \Theta(n \times H \times P)$ in quanto le operazioni elementari hanno tutte costo costante, ed il blocco di maggiore costo è quello che contiene i tre **for** annidati le cui operazioni interne vengono eseguite $(n - 1) \times (H + 1) \times (P + 1)$ volte.

Cont.

Algorithm 2: IMPILARE(INT $h[1..n]$, INT $p[1..n]$, INT H, INT P) $\rightarrow$ INT

```
INT T[1..n, 0..H, 0..P]
// Inizializzazione delle celle con primo indice uguale a 1
for j ← 0 to H do
    for k ← 0 to P do
        if (h[1] > j o p[1] > k) then
            T[1, j, k] ← 0                                  // primo oggetto non impilabile
        else
            T[1, j, k] ← 1                                  // primo oggetto impilabile
// Riempimento restanti celle della tabella T
for i ← 2 to n do
    for j ← 0 to H do
        for k ← 0 to P do
            if (h[i] > j o p[i] > k) then
                T[i, j, k] ← T[i − 1, j, k]                 // i-esimo oggetto non impilabile
            else
                T[1, j, k] ← max{T[i − 1, j, k], 1 + T[i − 1, j − h[i], k − p[i]]}  // i-esimo ogg. impilabile
return T[n, H, P]
```

4. Progettare un algoritmo che prende in input un grafo non orientato e restituisce il numero di vertici presenti nella componente connessa che contiene il maggior numero di vertici. In altri termini, l'algoritmo deve controllare tutte le componenti connesse, per ogni componente connessa deve contare il numero di vertici che essa contiene, e poi deve restituire il massimo fra tutti questi numeri.

   Soluzione. Per visitare tutte le componenti connesse di un grafo non orientato è possibile utilizzare l'algoritmo di visita in profondità DFS. Per calcolare la dimensione della componente connessa di dimensione massima si possono applicare all'algoritmo classico delle minime modifiche: si utilizza una variabile *counter* per contare i vertici visitati durante una esecuzione della funzione $DFSvisit(v)$ invocata dalla funzione principale (questa esecuzione visita la componente connessa a cui appartiene v), si considera una variabile max che ricorda il valore massimo raggiunto da *counter*, si utilizza una semplice marcatura di tipo booleano (campo *visited*) ed infine si rimuovono i tempi di inizio e fine visita (che non vengono utilizzati). Al termine della visita, sarà sufficiente restituire il valore della variabile max.

   Tale algoritmo è descritto in pseudocodice come Algoritmo 3: il costo computazionale risulta il medesimo dell'algoritmo classico di visita in profondità, ovvero $O(n + m)$ con $n = |V|$ e $m = |E|$.

Algorithm 3: MassimaComponenteConnessa(Graph (E, V)) $\rightarrow$ Int

```
Int counter, max ← 0
// Inizializzazione marcatura dei vertici
for v ∈ V do
    v.visited ← false
// Esecuzione della DFS
for v ∈ V do
    if not v.visited then
        counter ← 0
        DFSVisit(v)
        if counter > max then
            max ← counter
return max

// visita DFS
DFSVisit(Vertex u)
u.visited ← true
counter ← counter + 1
for v ∈ u.adjacents do
    if not v.visited then
        DFSVisit(v)
```

The End.