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 ≤ 1 then
    return 1
else
    x = 0
    for i = 1, ··· , n do
        x = x + i
    return MYSTERY1(n/2) + MYSTERY1(n/2) + MYSTERY2(x)

function MYSTERY2(INT n) → INT
if n ≤ 1 then
    return 1
else
    x = 1
    for i = 1, ··· , n/2 do
        x = x × x
    return MYSTERY2(n/3) + MYSTERY2(n/3) + x
```

Soluzione. Analizziamo prima il costo di MYSTERY2. Il ciclo while in MYSTERY2 viene eseguito $\Theta(n)$ volte e la funzione richiama se stessa due volte con in input $n/3$. L'equazione di ricorrenza di MYSTERY2

$$T'(n) = \begin{cases} 1 & n \leq 1 \\ 2T'(n/3) + n & n > 1 \end{cases}$$

può essere risolta con il Master Theorem

$$\alpha = \log_3 2 < 1 = \beta \Rightarrow T'(n) = \Theta(n^\beta) = \Theta(n)$$

Il ciclo while in MYSTERY1 viene eseguito $\Theta(n)$ volte e calcola il numero

$$x = 1 + 2 + \cdots + n = \tfrac{n(n+1)}{2}$$

Il costo di una chiamata a MYSTERY2 con input x è quindi $\Theta(n(n+1)/2) = \Theta(n^2)$. Inoltre, MYSTERY1 richiama se stessa due volte su input $n/2$. L'equazione di ricorrenza di MYSTERY1 è quindi

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T'(n/2) + n^2 & n > 1 \end{cases}$$

e può essere risolta con il Master Theorem

$$\alpha = \log_2 2 = 1 < 2 = \beta \Rightarrow T'(n) = \Theta(n^\beta) = \Theta(n^2)$$

2. Consideriamo la seguente implementazione iterativa di una visita in profondità su un albero binario

Algorithm 2: DFS(TREE T)

```
S = STACK()
if T.root ≠ NIL then
    PUSH(S, T.root)
while S.size ≠ 0 do
    x = POP(S)
    VISIT(x)
    if x.right ≠ NIL then
        PUSH(S, x.right)
    if x.left ≠ NIL then
        PUSH(S, x.left)
```

a) Che tipo di visita in profondità (pre-ordine, post-ordine o in-ordine) implementa tale l'algoritmo?
b) Analizzare la complessità computazionale in termini di tempo e memoria dell'algoritmo
c) Tale implementazione è più o meno efficiente (in termini di utilizzo di memoria e tempo di calcolo) della sua versione ricorsiva?

Soluzione.

a) L'algoritmo DFS implementa una visita in pre-ordine. In maggiore dettaglio, possiamo notare che il nodo x in cima alla pila è prima visitato e successivamente prima il figlio destro e poi sinistro (se esistono) sono posizionati sulla pila. Questo implica che il figlio sinistro di x sarà in cima alla pila e, di conseguenza, questo implica che il sottoalbero sinistro di x sarà interamente visitato prima del suo sottoalbero destro.
b) Il costo dell'algoritmo DFS in termini di tempo è $\Theta(n)$, dove n è il numero di nodi nell'albero. Infatti, notiamo che l'algoritmo visita ogni nodo esattamente una volta. Il costo in termini di memoria dipende dalla topologia dell'albero. Ad esempio, se l'albero è una lista, la pila conterrà al massimo un nodo per volta. In generale, se un nodo ha due figli, solo il figlio destro di un nodo è mantenuto nella pila per un tempo più lungo mentre il figlio sinistro viene estratto nell'iterazione successiva. In conclusione il numero massimo di nodi nella pila dipende dalla lunghezza del percorso più lungo contenente nodi che abbiano sia un figlio destro che un figlio sinistro. La lunghezza di tale percorso è limitata superiormente dall'altezza h dell'albero binario, quindi l'utilizzo di memoria è $O(h)$
c) La versione ricorsiva della visita in pre-ordine ha sempre costo pessimo/medio/ottimo $\Theta(n)$ in termini di tempo di calcolo e $\Theta(h)$ in termini di utilizzo di memoria. Quindi sia la versione iterativa che ricorsiva hanno lo stesso costo pessimo per quanto la versione iterativa è generalmente più efficiente in termini di utilizzo di memoria. Ad esempio, se l'albero è una lista la versione ricorsiva utilizzerà $\Theta(n)$ di memoria mentre quella iterativa solo $O(1)$.

3. Gli organizzatori di un tour di concerti rock devono progettare l'impianto musicale, in particolare, devono decidere quali casse acustiche utilizzare. Hanno a disposizione n possibili casse, ognuna con una propria altezza $h[i]$ ed una propria potenza $p[i]$. Le casse selezionate dovranno essere impilate una sopra l'altra, creando quindi una pila di casse di altezza complessiva uguale alla somma delle altezze delle singole casse impilate. Tale altezza complessiva deve essere inferiore rispetto ad un valore H indicante l'altezza massima della pila di casse. L'obiettivo è quello di scegliere la combinazione di casse da impilare che massimizza la potenza complessiva, ovvero la somma delle potenze delle casse selezionate deve essere la massima possibile. Per risolvere tale problema, dovete

Cont.

quindi progettare un algoritmo che dati in input un numero intero H, che indica l'altezza massima per la pila di casse, e due array di interi $h[1..n]$ e $p[1..n]$ (che contengono rispettivamente le altezze $h[i]$ e le potenze $p[i]$ delle n casse) restituisce la potenza complessiva massima per una pila di casse di altezza complessiva minore o uguale a H.

Soluzione.
È possibile procedere utilizzando programmazione dinamica considerando i seguenti sottoproblemi:

> $P(i,j)$, con $i \in [1,n]$ e $j \in [0,H]$: potenza complessiva massima di una pila di casse di altezza minore o uguale a j realizzata avendo a disposizione le prime i casse.

Tali problemi possono essere risolti induttivamente rispetto a i nel seguente modo:

$$P(i,j) = \begin{cases} 0 & \text{se } i = 1 \text{ e } j < h[1] \\ p[1] & \text{se } i = 1 \text{ e } j \geq h[1] \\ P(i-1,j) & \text{se } i > 1 \text{ e } j < h[i] \\ max\{P(i-1,j),\ p[i] + P(i-1, j-h[i])\} & \text{altrimenti} \end{cases}$$

Si noti che la soluzione al problema richiesto coincide con il sottoproblema $P(n,H)$ in cui l'altezza massima risulta essere H e sono disponibili tutti le n casse.

Procediamo quindi a progettare un algoritmo (si veda Algoritmo 3) che risolve i problemi $P(i,j)$ memorizzando le relative soluzioni in una tabella $T[1..n, 0..D]$. L'algoritmo proposto ha costo

Algorithm 3: PotenzaMassima(Int H, Int $h[1..n]$, Int $p[1..n]$) → Int

```
Int T[1..n][0..D]
 // Inizializzazione prima riga della tabella T
for j ← 0 to D do
    if j < h[1] then
        T[1, j] ← 0
    else
        T[1, j] ← p[1]
 // Riempimento restanti righe della tabella T
for i ← 2 to n do
    for j ← 0 to D do
        if (j < h[i]) or (T[i − 1, j] > p[i] + T[i − 1, j − h[i]]) then
            T[i, j] ← T[i − 1, j]
        else
            T[i, j] ← p[i] + T[i − 1, j − h[i]]
 // Restituisce T[n, D]
return T[n, D]
```

computazionale $\Theta(n \times D)$, in quanto al tempo di riempimento della tabella (ogni cella della tabella T richiede tempo costante per essere riempita) si aggiunge la sola operazione di ritorno del valore $T[n,D]$ che ha costo costante.

4. Si consideri un grafo G non orientato con:
   - vertici $V = \{1,2,3,4,5,6,7,8\}$
   - archi $E = \{\{1,2\},\{1,4\},\{2,3\},\{3,4\},\{3,5\},\{4,5\},\{4,6\},\{5,8\},\{6,7\},\{7,8\}\}$

   Descrivere a parole quale algoritmo, tra quelli studiati a lezione, si può usare per calcolare il cammino di lunghezza minima (ovvero che attraversa il numero minimo di archi) dal vertice 1 al vertice 8. Descrivere poi l'esecuzione di tale algoritmo sul grafo G dell'esempio.

   Soluzione.
   Per calcolare il cammino di lunghezza minima fra due vertici i e j di un grafo (ovvero il numero minimo di archi di un cammino da i a j) è possibile effettuare una visita in ampiezza BFS a partire dal vertice i, terminando la visita al momento del raggiungimento del vertice j. Si ricorda

Cont.

che la visita in ampiezza utilizza una coda in cui viene inserito inizialmente il vertice di partenza della visita (indicato a distanza 0) ed un ciclo che ripetutamente estrae un vertice dalla coda ed inserisce nella coda eventuali nuovi vertici raggiungibili dal vertice appena estratto (a tali vertici viene assegnata distanza pari alla distanza del vertice appena estratto incrementata di 1).

Nel caso particolare del testo dell'esercizio, nella coda si inserisce inizialmente il vertice 1 (assegnando distanza 0). Si eseguono successivamente i seguenti cicli:

- si estrae 1 dalla coda e si inseriscono i vertici 2 e 4 (a cui viene assegnata distanza 1);
- si estrae 2 dalla coda e si inserisce il vertice 3 (a cui viene assegnata distanza 2);
- si estrae 4 dalla coda e si inseriscono i vertici 5 e 6 (a cui viene assegnata distanza 2);
- si estrae 3 dalla coda;
- si estrae 5 dalla coda e si raggiunge il vertice 8.

Al raggiungimento del vertice 8 la visita viene interrotta e si restituisce la relativa distanza 3.

The End.