Tutoraggio Progettazione di Algoritmi
Eserciziario completo
a.a. 2024/25

S. Bianco

26 maggio 2025

Indice

1 Fondamenti di Teoria dei Grafi

1.1 Esercizi

Esercizio 1.1 (Nodi equidistanti)**.** *Sia G un grafo diretto. Fissati due nodi speciali s, t in G, definiamo un nodo x in G come equidistante se* $\text{dist}(x, s) = \text{dist}(x, t)$*. In caso contrario, diciamo che il nodo x è nella sfera di influenza del nodo s se* $\text{dist}(x, t) < \text{dist}(t, x)$ *e del nodo t se* $\text{dist}(x, s) > \text{dist}(x, t)$*.*

Progettare un algoritmo di tempo $O(n + m)$ che, dati in input G, s e t, restituisca il vettore D di n elementi tale che $D[i] = a$ se il nodo i è nella sfera di influenza di s, $D[i] = t$ se è in quella di t e $D[i] = *$ se è equidistante.

(Soluzione)

Esercizio 1.2 (Domande su grafi)**.** *Rispondere alle seguenti domande, giustificando la risposta:*

1. *Sia T un albero di copertura minimo su un grafo non diretto connesso G. Sia G' il grafo ottenuto da G incrementando dello stesso valore $k > 0$ il peso di ognuno dei suoi archi. L'albero T è un albero di copertura minimo anche per G'?*

2. *Sia G un grafo diretto fortemente connesso e si consideri l'esecuzione di una visita in profondità su G. Tra archi all'indietro, in avanti e di attraversamento, quali verranno di sicuramente incontrati e quali non?*

3. *Si consideri l'albero T ottenuto a seguito di una visita in ampiezza su un grafo non diretto e connesso G. Sia $h(x)$ l'altezza del generico nodo x in T. Dato un arco (a, b) in $G - T$, può aversi verificarsi che $h(a) - h(b) > 2$? La risposta cambia se il grafo G è diretto?*

4. *Quanti ordinamenti topologici ci possono essere per un grafo diretto con un solo arco?*

5. *Sia G un grafo non diretto avente solo nodi con grado massimo 3 e sia x un nodo qualsiasi del grafo. Dato un intero $d \in \mathbb{N}$, qual è il numero di nodi che possono trovarsi a distanza esattamente d da x? Quanti a distanza massimo d?*

(Soluzione)

Esercizio 1.3 (Arcipelago)**.** *Un arcipelago è rappresentato da una matrice $n \times m$, dove ogni cella è marcata da uno 0, rappresentante il mare, o da un 1, rappresentante il terreno. Due celle appartengono alla stessa isola se e solo se sono marcate entrambe con 1 e sono adiacenti. Data in input la matrice M, progettare un algoritmo che in tempo $O(nm)$ restituisca il numero di isole nell'arcipelago.*

(Soluzione)

Esercizio 1.4 (Quadrati e colori)**.** *Dato un grafo G, definiamo G^2 come il grafo tale che $V(G^2) = V(G)$ e $E(G^2) = \{(x, y) \mid \exists z \ (x, z), (z, y) \in E(G)\}$. Definiamo invece un grafo come 2-colorabile quando è possibile assegnare un colore, scelto tra 2 colori, ad ogni nodo in modo che non vi siano nodi adiacenti con lo stesso colore.*

Supponendo che G sia 2-colorabile, è sempre vero che G^2 è 2-colorabile? Vi è differenza tra il caso diretto e indiretto? Se vero, fornire una dimostrazione. Se falso, fornire un controesempio.

(Soluzione)

Esercizio 1.5 (Grafo complementare)**.** *Dato un grafo G, definiamo $\overline{G}$ come grafo complementare di G se per ogni arco (u, v) si verifica che $(u, v) \in E(G)$ se e solo se $(u, v) \notin E(\overline{G})$. Dimostrare che almeno uno tra G e $\overline{G}$ è connesso.*

(Soluzione)

Esercizio 1.6 (Equivalenza tra DFS e BFS)**.** *Supponiamo che G sia un grafo non diretto per cui ogni DFS e ogni BFS produce lo stesso albero di visita, indipendentemente dal punto di partenza. Che proprietà deve necessariamente avere G?*

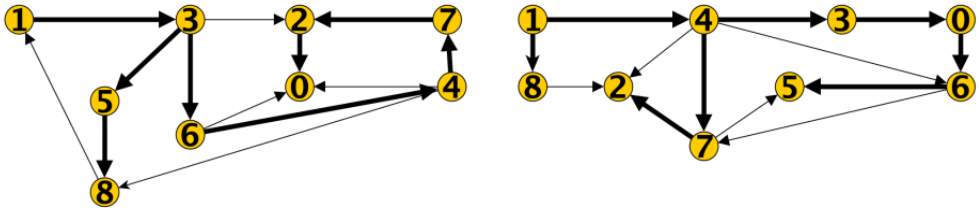
(Soluzione)

Esercizio 1.7 (Minimo antenato comune)**.** *Dato un albero T radicato in r e tre nodi $x, y, z \in V(T)$, definiamo il Lowest Common Ancestor (Minimo Antenato Comune) come il nodo che si trova sui tre cammini da r ai tre nodi e massimizzante la distanza tra esso e r.*

Progettare un algoritmo che dati in input il vettore dei padri P rappresentante un albero T e i tre nodi x, y, z restituisca $\mathrm{LCA}(x, y, z)$ in tempo $O(n + m)$.

(Soluzione)

Esercizio 1.8 (Visite in ampiezza)**.** *Si considerino i grafi diretti nelle due figure qui sotto e le due arborescenze di visita formate dagli archi marcati.*



Determinare per ciascuna delle due figure se l'arborescenza riportata può essere prodotta da una visita in profondità del grafo:

- *In caso di risposta negativa, motivare la risposta.*
- *In caso di risposta positiva, riportare una rappresentazione del grafo in forma di liste di adiacenza, ricordando che i nodi nelle liste di adiacenza devono essere considerati in sequenza da sinistra a destra. Inoltre, per gli archi non appartenenti alla visita, determinare quali sono archi all'indietro, in avanti e di attraversamento.*

(Soluzione)

Esercizio 1.9 (Somma dei gradi)**.** *Dimostrare o confutare che la somma dei gradi dei nodi di un grafo G non orientato è sempre un numero pari.*

(Soluzione)

Esercizio 1.10 (Connettività)**.** *Sia G un grafo non orientato con $2n$ nodi, ciascuno avente grado almeno n. Dimostrare o confutare che il grafo G è connesso.*

(Soluzione)

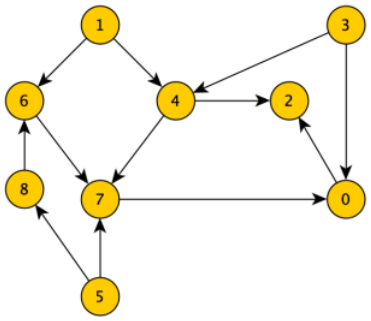
Esercizio 1.11 (Ordinamenti topologici)**.** *Sia T un albero radicato con n nodi, in cui gli archi sono stati orientati seconda la direzione padre-figlio. Determinare il numero minimo e il numero massimo di ordinamenti topologici che T può avere.*

(Soluzione)

Esercizio 1.12 (Sorgente e pozzo)**.** *Dimostrare o confutare che un DAG ha sempre almeno un nodo sorgente, ossia ha grado entrante 0, ed almeno un nodo pozzo, ossia ha grado uscente 0.*

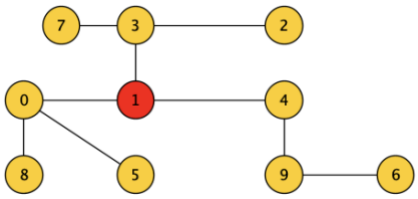
(Soluzione)

Esercizio 1.13 (Sorgenti verso un nodo)**.** *Progettare un algoritmo di tempo $O(n+m)$ che, dato in input un DAG G rappresentato tramite liste di adiacenza ed un suo nodo x, restituisca la lista dei nodi sorgente in G che possono raggiungere x. Ad esempio, per il seguente grafo l'output per $x = 7$ deve essere $[1, 5]$, mentre per $x = 4$ deve essere $[1, 3]$.*



(Soluzione)

Esercizio 1.14 (Tripartizione)**.** *Dato un grafo connesso aciclico e non diretto G, definiamo G come tripartibile se esiste un nodo x la cui rimozione sconnette il grafo in tre componenenti aventi lo stesso numero di nodi. Progettare un algoritmo di tempo $O(n)$ che, dato in input il grafo G rappresentato tramite liste di adiacenza, restituisca True se G è tripartibile, False altrimenti. Ad esempio, per il seguente grafo la rimozione del nodo $x = 1$ crea una tripartizione.*



(Soluzione)

1.2 Soluzioni

Soluzione 1.1 (Nodi equidistanti)**.** *(Traccia)*

Osserviamo che, trattandosi di un grafo diretto, la distanza tra x e s (o x e t) non è detto sia uguale alla distanza tra s ed x (o t e x). Tuttavia, osserviamo che $\text{dist}_G(x, s) = \text{dist}_{G^T}(s, x)$, dove G^T è il grafo trasposto di G, ossia il grafo la cui direzione degli è invertita. Per tanto, effettuando una BFS partendo da s e una BFS partendo da t è possibile calcolare due vettori delle distanze, per poi confrontarli indice per indice.

function EQUIDISTANT(G, s, t)
  $G^T = \text{TRASPOSE}(G^T)$
  $S = \text{BFS\_DIST}(G^T, s)$
  $T = \text{BFS\_DIST}(G^T, t)$
  $D = [0, \ldots, 0]$
  for $i = 1, \ldots, n$ **do**
    if $S[i] < T[i]$ **then**
      $D = s$
    else if $S[i] > T[i]$ **then**
      $D = t$
    else
      $D = *$
    end if
  end for
  Return D
end function

Soluzione 1.2 (Domande su grafi)**.** *(Traccia)*

1. Sia $w : E(G) \to \mathbb{R}^+$ la funzione di peso di G e sia $w' : E(G) \to \mathbb{R}^+$ quella di G', dove $w'(e) = w(e) + k$. Supponiamo per assurdo che T non sia un albero di copertura minimo per G'. Sia T' un albero di copertura minimo su G'. Poiché T' è minimo per G ma G' no, ne segue che:

$$\sum_{e \in E(T')} w'(e) < \sum_{e \in E(T)} w'(e) \implies \sum_{e \in E(T')} k + w(e) < \sum_{e \in E(T)} k + w(e)$$

Poiché sia T che T' hanno $n-1$ archi, otteniamo che:

$$k(n-1)+\sum_{e\in E(T')} w(e) < k(n-1)+\sum_{e\in E(T)} w(e) \implies \sum_{e\in E(T')} w(e) < \sum_{e\in E(T)} w(e)$$

dunque T' è un albero di copertura su G con peso minore di T, dando una contraddizione.

2. Se G è fortemente connesso allora esiste sicuramente un ciclo in esso (assumendo $n \geq 2$). Per tanto, è garantita l'esistenza di un arco all'indietro. Inoltre, ogni DFS su G produrrà un albero diretto con esattamente un ramo, dunque non possono esserci archi di attraversamento. Per quanto riguarda gli archi di in avanti, essi possono sia esserci che non esserci.

3. Supponiamo per assurdo che $h(a)-h(b) > 2$. Sia x l'antenato in comune di a e b che sia più lontano dalla radice. Siano P, Q rispettivamente il cammino $x \to b$ e $x \to a$ in T. Siccome $h(b) < h(a) - 2$, devono esistere almeno altri due nodi c e d sul cammino P, dunque Q ha lunghezza almeno 3. Per tanto, il cammino $P \cup \{(b, a)\}$ è più breve del cammino Q, implicando che la BFS abbia sbagliato la propria esecuzione, il che è assurdo. Per tanto, non può verificarsi che $h(a) - h(b) > 2$. La soluzione del caso diretto viene omessa.

4. Sia (x, y) l'unico arco del grafo. Senza la presenza di tale arco, avremmo $n!$ ordinamenti topologici. Considerando la dipendenza imposta dall'arco, invece, esattamente metà di tali ordinamenti posiziona il nodo x prima del nodo y, mentre la restante metà effettua il contrario. Per tanto, gli ordinamenti validi sono $\frac{n!}{2}$.

5. Soluzione omessa. *Nota*: ci sono più modi per trovare le risposte. Confrontatevi con altri studenti per cercare di svolgere i vari ragionamenti possibili.

Soluzione 1.3 (Arcipelago)**.** *(Traccia)*

È possibile risolvere l'esercizio in due modi. Il primo modo, prevede la creazione di un grafo che modelli l'arcipelago: ogni cella $M[i, j]$ della matrice marcata con 1 viene mappata ad un nodo $v_{i,j}$, creando un arco tra due celle

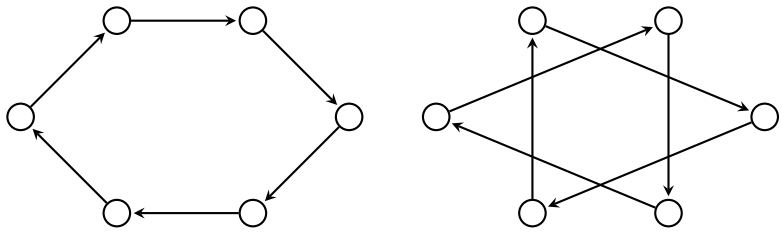
se e solo se sono adiacenti ed entrambe sono marcate con 1. Il numero di isole sarà dato dal numero di componenti connessi del grafo. La creazione del grafo richiede tempo $O(nm)$, mentre il conteggio dei componenti può essere effettuato con una DFS, richiedente anch'essa tempo $O(nm)$.

Il secondo metodo, invece, prevede di saltare la costruzione del grafo, trattando direttamente la matrice stessa come se fosse un grafo. Osserviamo che questa soluzione sia molto più efficiente della prima. Riportiamo il codice di quest'ultima soluzione.

```
function FIND-TOTAL-ISLANDS(M)
    c = 0
    for i = 1, ..., n do
        for j = 1, ..., n do
            if M[i, j] == 1 then
                DFS-ISLAND(M, i, j)
                c++
            end if
        end for
    end for
    Return c
end function
function DFS-ISLAND(M, i, j)
    if M[i, j] == 1 and 1 ≤ i, j ≤ n then
        M[i, j] = *
        DFS-ISLAND(M, i − 1, j)
        DFS-ISLAND(M, i + 1, j)
        DFS-ISLAND(M, i, j − 1)
        DFS-ISLAND(M, i, j + 1)
    end if
end function
```

Soluzione 1.4 (Quadrati e colori)**.** *(Traccia)*

L'affermazione è falsa: sia per il caso diretto che il caso indiretto, è sufficiente considerare il grafo formato da un ciclo di lunghezza 6.



Soluzione 1.5 (Grafo complementare)**.** *(Traccia)*

Quando $n = 1$, sia G che $\overline{G}$ sono banalmente connessi. Assumiamo quindi che $n \geq 2$. Se G è connesso, allora non vi è nulla da dimostrare. Supponiamo quindi che G non sia connesso. Per ogni coppia $u, v \in V(G)$, abbiamo due possibilità:

- Se $\text{comp}(u) \neq \text{comp}(v)$ allora almeno uno tra gli archi (u, v) e (v, u) non esiste in G, poiché altrimenti u e v sarebbero nello stesso componente. Dunque, ne segue che almeno uno tra (u, v) o (v, u) si trovi in $\overline{G}$, implicando che esista un cammino $u \to v$ oppure un cammino $v \to u$.
- Se $\text{comp}(u) = \text{comp}(v)$ allora esiste un cammino $u \to v$ ed un cammino $v \to u$. Poiché G non è connesso, deve esserci almeno un nodo z tale che nessuno degli archi $(u, z), (v, z), (z, u)$ e (z, v) esista in G. Di conseguenza, tali archi saranno in $\overline{G}$ e tramite essi è possibile formare un cammino $u \to z \to v$ (e anche un cammino $v \to z \to u$).

Poiché in entrambi i casi per ogni coppia di vertici u, v esiste il cammino $u \to v$ o $v \to u$, concludiamo che $\overline{G}$ sia connesso.

Soluzione 1.6 (Equivalenza tra DFS e BFS)**.** *(Traccia)*

Osserviamo che G debba essere necessariamente una *foresta*, ossia un grafo non diretto aciclico – il nome deriva dal fatto che i componenti sono tutti alberi. Difatti, per contronominale possiamo dimostrare che se G contiene un ciclo allora effettuando una DFS e una BFS partendo da un nodo del ciclo verranno percorsi archi differenti, generando quindi un albero di visita diverso.

Soluzione 1.7 (Minimo antenato comune)**.** *(Traccia)*

L'idea è quella di percorrere i tre cammini partendo dai nodi fino alla radice, creando tre stack contenenti gli antenati dei tre nodi. Osserviamo che la radice sarà sempre il top dei tre stack. Successivamente, finché il top è uguale per tutti e tre gli stack, esso viene rimosso da tutti e tre, restituendo l'ultimo top uguale nel caso contrario.

```
function FIND-ANCESTORS(P, v)
    S = new(Stack)
    S.push(v)
    while v ≠ P[v] do          ▷ La radice è l'unico nodo per cui v = P[v]
        v = P[v]
        S.push(v)
    end while
end function
function LCA(P, x, y, z)
    S_x = FIND-ANCESTORS(P, x)
    S_y = FIND-ANCESTORS(P, x)
    S_z = FIND-ANCESTORS(P, x)
    a = ∅
    while S_x.top() == S_y.top() == S_z.top() do
        a = S_x.pop()
        S_y.pop()
        S_z.pop()
    end while
    Return a
end function
```

Soluzione 1.8 (Visite in ampiezza)**.** *(Traccia)*

L'arborescenza T_2 descritta dalla seconda figura è impossibile che sia prodotta da una DFS (ad esempio, il ciclo $7, 5, 6$ è impossibile che sia percorso da una DFS nel modo descritto).

L'arborescenza T_1 descritta nella prima figura, invece, può essere ottenuta tramite una DFS avviata sul nodo 1. La rappresentazione tramite liste di adiacenza è data da:

$$T_1 = [\ [\ ], [3], [0], [5, 6, 2], [7, 8, 0], [8], [4, 0], [2], [1]\ ]$$

Archi all'indietro: $(8, 1)$

Archi in avanti: $(3, 2), (6, 0), (4, 0)$

Archi di attraversamento: $(4, 8)$

Soluzione 1.9 (Somma dei gradi)**.** *(Traccia)*

L'affermazione è vera. All'interno della somma $\sum_{v \in V(G)} \deg(v)$, ogni arco (u, v) viene contato due volte: una volta per il nodo u ed una volta per il nodo v. Per tanto, abbiamo che $\sum_{v \in V(G)} \deg(v) = 2\,|E(G)|$. Questo risultato è noto in letteratura come *Handshaking Lemma*.

Soluzione 1.10 (Connettività)**.** *(Traccia)*

L'affermazione è vera. Per dimostrarlo, supponiamo per assurdo che G ogni nodo abbia grado almeno n ma che G non sia connesso, dunque che G abbia almeno due componenti. Ciò implica che il componente H più piccolo di G possa avere massimo $\frac{2n}{2} = n$ nodi. Dato $x \in V(H)$ sappiamo che $\{x\} \cup N(x) \subseteq V(H)$, dove $N(x)$ è il *neighborhood* di x, ossia l'insieme dei suoi vicini. Tuttavia, poiché $|N(x)| \geq n$, otteniamo che H debba avere almeno $n+1$ nodi, generando una contraddizione. Per tanto, G deve necessariamente essere connesso.

Soluzione 1.11 (Ordinamenti topologici)**.** *(Traccia)*

Il minimo numero di ordinamenti topologici per un albero è 1, dato dal caso in cui T sia un unico cammino di n nodi. Il numero massimo, invece, è dato dal caso in cui T sia un albero radicato dove gli altri $n - 1$ nodi sono figli della radice.

Soluzione 1.12 (Sorgente e pozzo)**.** *(Traccia)*

L'affermazione è vera. Per dimostrarlo, supponiamo per assurdo che G sia un DAG ma che non esista un pozzo. Sia P il cammino più lungo in G. Poiché non esistono pozzi, l'ultimo nodo deve avere un arco uscente. Tuttavia, tale arco deve necessariamente tornare indietro verso uno degli altri nodi di P: se così non fosse, potremmo ottenere un cammino più lungo di P. Questo conclude che G contenga un ciclo, il che è assurdo per ipotesi. Per tanto, deve necessariamente esistere un pozzo in G. Per la sorgente è possibile fare un ragionamento analogo.

Soluzione 1.13 (Sorgenti verso un nodo)**.** *(Traccia)*

L'algoritmo può essere progettato in due modi:

- Dopo aver calcolato il grafo trasposto, ossia il grafo in cui la direzione di tutti gli archi è invertita, è sufficiente effettuare una DFS per trovare i pozzi raggiungibili da x. Ogni pozzo del grafo trasposto corrisponde ad una sorgente del grafo iniziale.
- Effettuando una DFS spezzata, aggiungiamo le sorgenti incontrate nella DFS solo se uno dei suoi figli può raggiungere il nodo x.

Soluzione 1.14 (Tripartizione)**.** *(Soluzione)*

Prima di progettare l'algoritmo, è opportuno osservare alcune condizioni imposte dal problema. Prima di tutto, notiamo che un grafo aciclico connesso e non diretto è per definizione un albero. Dunque, abbiamo che $m = n - 1$, implicando che $O(n + m) = O(n)$ in questo caso. Ciò ci suggerisce che il costo imposto si riferisca in realtà ad una visita. Successivamente, osserviamo che se G è tripartibile allora necessariamente deve valere che $n = 3k + 1$ per qualche $k \in \mathbb{N}$ e che il grado del nodo x che tripartisce il grafo deve essere esattamente tre. Quando una di tali due condizioni è falsificata, possiamo immediatamente rifiutare.

Tuttavia, tali condizioni non sono invece sufficienti affinché il grafo sia tripartibile: potremmo avere un grafo con $n = 3k + 1$ nodi ed un nodo di grafo 3 la cui rimozione generebbe tre componenti aventi un numero diverso di nodi. Una possibile soluzione prevede quindi di:

1. Controlla se $n \equiv 1 \pmod 3$. Se falso, possiamo immediatamente rifiutare. Se vero, poni $t = \frac{n-1}{3}$.
2. Esegui una DFS che, dato il nodo attualmente visitato x, vada a contare in modo ricorsivo il numero di nodi nei sottoalberi $T_{y_1}, \ldots, T_{y_\ell}$ dei figli $y_1, \ldots, y_\ell$ di x. Se uno dei figli ritorna il valore speciale -1, ritorna -1. Se $\ell \neq 3$, allora viene restituito $1 + |T_{y_1}| + \ldots + |T_{y_\ell}|$. Se $\ell = 3$ e $|T_{y_1}| = |T_{y_1}| = t$, ritorna il valore speciale -1. Se $\ell = 3$ e $\neg(|T_{y_1}| = |T_{y_1}| = t)$, allora viene restituito $1 + |T_{y_1}| + |T_{y_2}|$.
3. Se l'intera DFS ritorna -1 alla fine allora ritorna True, altrimenti ritorna False.

Osserviamo che se $|T_{y_1}| = |T_{y_2}| = \frac{n-1}{3}$ allora l'insieme $S = V(G) - T_{y_1} - T_{y_2} - \{x\}$ contiene anch'esso $\frac{n-1}{3}$ nodi (grazie al controllo iniziale modulo tre).

```
function TRIPARTITE(G)
    if n ≢ 1 (mod 3) then
        Return False
    end if
    t ← (n−1)/3
    if DFS-TRIPARTITE(G, r) = −1 then      ▷ r è un nodo iniziale casuale
        Return True
    end if
    Return False
end function
function DFS-TRIPARTITE(G, x)
    if deg(x) = 3 then
        sx ← DFS-TRIPARTITE(G, y_1)          ▷ y_1 è il figlio sinistro di x
        dx ← DFS-TRIPARTITE(G, y_2)          ▷ y_2 è il figlio destro di x
        if sx = −1 ∨ dx = −1 then
            Return −1
        end if
        Return 1 + sx + dx
    else
        c ← 0
        for i ∈ deg(x) do
            tmp ← DFS-TRIPARTITE(G, y_i)     ▷ y_i è l'i-esimo figlio di x
            if tmp = −1 then
                Return −1
            end if
            c ← c + tmp
        end for
        Return c
    end if
end function
```

2 Algoritmi Greedy

2.1 Esercizi

Esercizio 2.1 (Independent set)**.** *Dato un grafo G, definiamo un sottoinsieme di vertici $X \subseteq V(G)$ come independent set se nessuna coppia di vertici $x, x' \in X$ è adiacente. Progettare un algoritmo che, dato un grafo non diretto aciclico G, trovi in tempo $O(n)$ un independent set di cardinalità massima. Si dimostri la correttezza della soluzione proposta.*

(Soluzione)

Esercizio 2.2 (Resto in monete)**.** *Una nazione utilizza i seguenti tagli di moneta $1, 2, 5, 10, 20, 50$. Progettare un algoritmo che, dati un valore n, restituisca un insieme di monete di cardinalità minima avente somma n. Si dimostri la correttezza della soluzione proposta.*

(Soluzione)

Esercizio 2.3 (Stringa bilanciata)**.** *Definiamo una stringa sull'alfabeto $\{'(', ')'\}$ come bilanciata se rispetta la seguente definizione ricorsiva:*

- *La stringa vuota è bilanciata*
- *Se A è una stringa bilanciata allora anche (A) è bilanciata*
- *Se A e B sono due stringhe bilanciate allora anche AB è bilanciata*

Progettare un algoritmo che, data una stringa, restituisca il minimo numero di caratteri da eliminare affinché la stringa sia bilanciata. Si dimostri la correttezza della soluzione proposta.

(Soluzione)

Esercizio 2.4 (Scaffali)**.** *Mariano si è da poco trasferito nella sua nuova casa e vuole riempire uno spazio vuoto di M centimetri presente su un muro del soggiorno. Una volta arrivato da Ikea, Mariano trova solo due tipologie di tavole acquistabili: tavole da m_1 centimetri e tavole da m_2 centimetri. Assumiamo che $m_1 < m_2$. Vogliamo minimizzare la quantità totale di tavole (indipendentemente dalla tipologia) necessarie a minimizzare lo spazio non riempito sul muro (dunque vogliamo prima minimizzare lo spazio non riempito e a parità di spazio preferire l'utilizzo di minor tavole possibili).*

Per risolvere il problema, vengono proposti i seguenti due approcci greedy:

1. *Acquista tavole della prima tipologia finché la somma di tutte le lunghezze delle tavole acquistate fin'ora non supera $M - m_1$, poi acquista tavole della seconda tipologia finché la somma di tutte le lunghezze delle tavole acquistate fin'ora.*

2. *Acquista tavole della prima tipologia finché la somma di tutte le lunghezze delle tavole acquistate fin'ora non supera $M - m_1$, poi acquista tavole della seconda tipologia finché la somma di tutte le lunghezze delle tavole acquistate fin'ora.*

Dimostrare o confutare la correttezza dei singoli approcci.

(Soluzione)

Esercizio 2.5 (Alberi e colori)**.** *Sia dato un grafo non diretto e connesso avente vertici colorati di rosso o blu. Vogliamo costruire un albero di copertura con il minor numero di archi con estremi dello stesso colore (ossia archi rosso-rosso o archi blu-blu).*

Per risolvere il problema, vengono proposti i seguenti approcci greedy:

1. *Effettua una DFS modificata, dando precedenza ai vicini con colore diverso dal nodo attualmente visitato*

2. *Effettua una BFS modificata, dando precedenza ai vicini con colore diverso dal nodo attualmente visitato*

3. *Scorri l'insieme di tutti gli archi ed aggiungi tutti gli archi aventi estremi con colori diversi, saltando quelli che creerebbero cicli. Successivamente, scorri l'insieme di tutti gli archi con colori uguali, sempre saltando quelli che creerebbero cicli.*

Dimostrare o confutare la correttezza dei singoli approcci.

(Soluzione)

Esercizio 2.6 (Ciclo di costo minimo)**.** *Dato un grafo pesato non diretto G, vogliamo trovare all'interno di G un ciclo la cui somma dei costi degli archi è minima. Viene proposta la seguente strategia greedy:*

- *Partendo da un insieme vuoto S, ad ogni iterazione aggiungiamo all'insieme l'arco di peso minimo che non si trovi già in S. Se durante la procedura viene creato un ciclo, restituiamo tale ciclo. Altrimenti, se terminano gli archi e non viene mai creato un ciclo, restituiamo $\varnothing$.*

Dimostrare o confutare la correttezza dell'approccio.

(Soluzione)

Esercizio 2.7 (Guardie e ladri)**.** *Sia dato un array A di zeri ed uni e un intero positivo k. Ogni uno corrisponde ad una guardia, mentre ogni zero corrisponde ad un ladro. Vogliamo determinare il massimo numero di ladri che possono essere catturati, a patto che le seguenti condizioni vengano rispettate:*

- *Ogni guardia può catturare al massimo un ladro*
- *Ogni guardia può effettuare una cattura solo se il ladro bersagliato è a distanza al massimo k da essa*

Per risolvere il problema, viene proposto il seguente approccio greedy: scorrendo l'array, ogni guardia cattura il ladro più vicino non ancora catturato e distante massimo k da essa. Dimostrare o confutare la correttezza dell'approccio.

(Soluzione)

2.2 Soluzioni

Soluzione 2.1 (Independent set)**.** *(Soluzione)*

Prima di tutto, osserviamo che il problema è intrattabile per un grafo generico. Difatti, il problema *maximum independent set* è in realtà NP-hard per grafi generici – per gli interessati, ciò equivale a dire che non esiste un algoritmo che risolva il problema in tempo $O(n^k)$ per qualsiasi $k > 0$ (assumendo che $\mathsf{P} \neq \mathsf{NP}$).

Per tanto, risulta di fondamentale importanza la proprietà di aciclicità. In particolare, osserviamo che un grafo non diretto aciclico sia una foresta, ossia un grafo i cui componenti connessi sono tutti alberi. Inoltre, ogni albero con almeno due nodi possiede almeno una foglia.

Utilizziamo quindi il seguente approccio greedy: ad ogni iterazione, selezioniamo una foglia e la aggiungiamo all'output, per poi rimuovere tale foglia e il suo padre dal grafo.

```
function IND-SET-FOREST(G)
    X ← ∅
    while E(G) ≠ ∅ do
        for v ∈ V(G) do
            if deg(v) = 1 then
                X ← X ∪ {v}
                G ← G − {v, G[v][0]}          ▷ Rimuovi v e il suo padre
            end if
        end for
    end while
    Return X
end function
```

Le varie operazioni dell'algoritmo possono essere implementate in tempo $O(n)$ (come?). Dimostriamo quindi che il nostro approccio greedy è ottimale. Siano $X_0, \ldots, X_k$ le varie istanze dell'insieme X durante le iterazioni del while. Vogliamo dimostrare che ognuna di tali istanze è contenuta all'interno di una qualche soluzione ottimale.

Procediamo per induzione sul numero di iterazioni i del while. Per $i = 0$, abbiamo che $X_0 = \varnothing$, dunque X_0 è contenuto in ogni soluzione ottimale. Assumiamo quindi che esista una soluzione ottimale X^* tale $X_i \subseteq X^*$. Per

l'iterazione $i+1$, abbiamo che $X_{i+1} = X_i \cup \{v_{i+1}\}$, dove v_{i+1} è il vertice selezionato dall'iterazione. Se $v_{i+1} \in X^*$ allora banalmente $X_{i+1} \subseteq X^*$, concludendo il passo induttivo.

Consideriamo quindi il caso $v_{i+1} \notin X^*$. Per chiudere l'induzione, dimostreremo che esiste un'altra soluzione ottimale X' contenente X_{i+1}. Osserviamo che, per via della scelta greedy, nessuno dei vertici adiacenti a v_{i+1} si trova all'interno di X_i poiché altrimenti v_{i+1} sarebbe stato rimosso dal grafo. Inoltre, sempre per scelta greedy, sappiamo che al momento della selezione v_{i+1} ha un unico vertice adiacente u.

Supponiamo quindi per assurdo che $u \notin X^* - X_i$. Allora, potremmo aggiungere v_{i+1} ad X^* ed ottenere un independent set di cardinalità maggiore, il che è assurdo poiché X^* è ottimale. Dunque, necessariamente $u \in X^* - X_i$. In tal caso, è sufficiente scambiare u con v_{i+1} all'interno di X^*, ottenendo l'insieme $X' = (X^* - \{u\}) \cup \{v_{i+1}\}$. Tale insieme è ancora un independent set ed ha la stessa cardinalità di X^*, dunque è anch'esso una soluzione ottimale, con la differenza che $X_{i+1} \subseteq X'$. Questo conclude il passo induttivo anche in questo caso.

Consideriamo quindi la soluzione ottimale $X^\#$ contenente l'output X_k. Supponiamo per assurdo che esista un elemento $z \in X^\# - X_k$. Allora, per scelta greedy, l'unico modo in cui tale z non sia stato selezionato dall'algoritmo è se esso è stato rimosso in quanto è il padre di qualche vertice w selezionato dal nostro algoritmo. Tuttavia, ciò implica che anche w si trovi in $X^\#$ in quanto $X_k \subseteq X^\#$, contraddicendo l'ipotesi per cui $X^\#$ sia un independent set. Per tanto, non può esistere alcun z, concludendo che $X_k = X^\#$.

Soluzione 2.2 (Resto in monete)**.** *(Soluzione)*

Utilizziamo il seguente approccio greedy: ad ogni iterazione, selezioniamo la moneta più grande possibile, sottraendola al totale. Osserviamo che tale approccio greedy non funziona per tutti i tagli di monete, ma solo per quelli che soddisfano una specifica proprietà (quale?). Ad esempio, con i tagli $1, 5, 8, 10, 11$ per l'input $n = 13$ l'algoritmo selezionerebbe le monete $11, 1, 1$, mentre la soluzione ottimale è $5, 8$. La dimostrazione di correttezza è stata discussa in classe.

Soluzione 2.3 (Stringa bilanciata)**.** *(Soluzione)*

Utilizziamo il seguente approccio greedy: scorrendo la stringa da sinistra a destra, incrementiamo un counter inizializzato a 0 ogni volta che incontria-

mo un simbolo $'('$. Quando incontriamo un simbolo $')'$, invece, se il counter è maggiore di 0 allora lo decrementiamo, altrimenti al suo posto incrementiamo un altro counter contenente il numero di simboli da eliminare. La dimostrazione di correttezza è stata discussa in classe.

Soluzione 2.4 (Scaffali)**.** *(Traccia)*

Entrambi gli approcci proposti sono errati. Per confutarli, consideriamo il controesempio $M = 10, m_1 = 3, m_2 = 4$. Il primo approccio utilizzerebbe le tavole $3, 3, 3$ lasciando uno spazio rimanente pari ad 1, mentre la seconda utilizzerebbe le tavole $4, 4$ lasciando uno spazio rimanente pari a 2. Tuttavia, la soluzione ottimale è data da $4, 4, 3$.

Soluzione 2.5 (Alberi e colori)**.** *(Traccia)*

È facile notare che il primo approccio risulti essere del tutto inutile in quanto fissare un ordine di precedenza all'interno di una BFS non ha alcun effetto. Nel caso della DFS, invece, la precedenza può avere influenza, ma ciò risulta essere uno svantaggio.

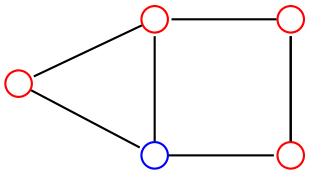


Figura 1: Controesempio al secondo approccio. Partendo dal nodo a sinistra, la DFS modificata prenderebbe solo 2 archi con colori distinti, mentre la soluzione ottimale ne ha 3.

Per quanto riguarda il terzo approccio, invece, esso risulta essere corretto. Dividiamo la dimostrazione della sua correttezza in tre fasi:

1. Dimostrare che la soluzione ritornata sia un albero di copertura
2. Dimostrare per induzione che ad ogni iterazione la soluzione parziale è sempre all'interno di una soluzione ottimale
3. Dimostrare che la soluzione ritornata coincida con la soluzione ottimale che la contiene.

Siano $S_0, \ldots, S_k$ le varie istanze dell'insieme S degli archi componenti durante le varie iterazioni. Sappiamo già che il grafo dato da S_k sarà aciclico per via del modo in cui vengono scelti gli archi. Inoltre, essendo il grafo iniziale G connesso, ogni nodo è incidente ad almeno un arco, dunque l'algoritmo potrà sempre selezionare almeno un arco adiacente ad esso, concludendo che $V(G) = V(S_k)$. Infinite, supponiamo per assurdo che S_k non sia connesso. Allora, esistono almeno due componenti in S_k collegate da almeno un arco in G poiché quest'ultimo è connesso. Tuttavia, ciò implica che l'algoritmo abbia sbagliato a non prendere tale arco in quanto esso non creerebbe cicli, il che è assurdo. Concludiamo quindi che S_k sia aciclico connesso e che copra tutti i vertici di G.

Procediamo quindi con la seconda fase. Per $i = 0$, abbiamo che $S_0 = \varnothing$, dunque S_0 è contenuto in ogni soluzione ottimale. Assumiamo quindi che esista una soluzione ottimale S^* tale $S_i \subseteq S^*$. Per l'iterazione $i+1$, abbiamo che $S_{i+1} = S_i \cup \{e_{i+1}\}$, dove $e_{i+1} = (x, y)$ è l'arco selezionato dall'iterazione. Se $e_{i+1} \in S^*$ allora banalmente $S_{i+1} \subseteq S^*$, concludendo il passo induttivo.

Consideriamo quindi il caso $e_{i+1} \notin S^*$. Per chiudere l'induzione, dimostreremo che esiste un'altra soluzione ottimale S' contenente S_{i+1}. Poiché S^* è un albero di copertura, in esso esisterà comunque un cammino P da x a y. Dunque, otteniamo che $C = P \cup e_{i+1}$ sia un ciclo in G (e solo in G). In particolare, in $C - S_i$ esisterà necessariamente un arco $e' = (y, z)$ tale che z sia connesso ad x tramite un cammino P'. Allora, ciò implica che $S' = (S^* - \{e_{i+1}\}) \cup \{e'\}$ è ancora aciclico e connesso. Inoltre, abbiamo che $|S'| = |S^*|$, concludendo che S' sia un albero di copertura contenente S_{i+1}.

Analizziamo quindi i colori degli archi appena scambiati. Per via del fatto che $e_{i+1} \in S_{i+1}$ ma $e' \notin S_{i+1}$, tramite la scelta greedy risulta impossibile che e_{i+1} sia un arco con colori uguali e che e' abbia colori diversi, poiché altrimenti l'algoritmo avrebbe dovuto selezionare prima e', il che è assurdo. Se invece e_{i+1} possiede terminali di colore diverso ma e' ha terminali di colore uguale, otteniamo che S' possieda un arco valido in più, contraddicendo l'ottimalità di S^*. Dunque, le uniche possibilità sono che entrambi gli archi abbiano terminali con colori uguali o che entrambi abbiamo terminali con colori diversi. Allora, scambiare i due archi preserva la stessa quantità di archi per le due tipologie, concludendo che S' sia una soluzione ottimale.

Arriviamo quindi alla terza fase. Consideriamo la soluzione ottimale $S^\#$ contenente l'output S_k. Poiché entrambi sono due alberi di copertura e $S_k \subseteq$

$S^{\#}$, deve necessariamente valere che $S_k = S^{\#}$.

Soluzione 2.6 (Ciclo di costo minimo)**.** *(Soluzione)*

La strategia proposta è errata in quanto essa potrebbe fallire. Ad esempio, nel controesempio riportato sotto, dopo aver selezionato gli archi $(d, c), (a, d), (b, c), (c, e)$ l'algoritmo dovrà scegliere un arco tra (a, b) e (d, e). Se viene scelto (a, b), verrà ritornato il ciclo sbagliato in quanto $C = \{b, c, e\}$ il ciclo minimo.

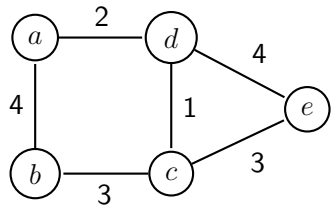


Figura 2: Controesempio all'approccio.

Soluzione 2.7 (Guardie e ladri)**.** *(Traccia)*

L'algoritmo è corretto. Per dimostrarne la correttezza, tuttavia, sviluppare un argomento basato sul singolo valore ritornato dalla soluzione risulta difficile. Risulta invece più semplice considerare l'insieme di ladri catturati dal nostro algoritmo e l'insieme di ladri catturati in una soluzione ottimale. Come sempre, la dimostrazione che vi riporto risulta lunga solo per questioni di chiarezza.

Procediamo con il solito metodo induttivo. Sia s il valore ritornato dal nostro algoritmo e sia s^* il valore di soluzione ottimale. Siano $X_1, \ldots, X_k$ le istanze dell'insieme X contenente i ladri catturati dal nostro algoritmo. Dimostriamo che per ogni $i \in [k]$ la soluzione parziale X_i è all'interno dell'insieme di ladri catturati in una qualche soluzione ottimale. Per $i = 0$, abbiamo che $X_0 = \varnothing$, dunque siamo all'interno di una qualsiasi soluzione (anche quelle ottimali quindi). Assumiamo quindi che esista un insieme X^* di ladri dato da una soluzione ottimale s^*, dunque tale che $|X^*| = s^*$, tale che $X_i \subseteq X^*$.

Per X_{i+1}, abbiamo due opzioni: $X_{i+1} = X_i$ poiché non abbiamo catturato nessun nuovo ladro oppure $X_{i+1} = X_i \cup \{\ell_{i+1}\}$ se abbiamo catturato un nuovo ladro. Nel primo caso, abbiamo che $X_{i+1} = X_i \subseteq X^*$, dunque X_{i+1} rimane all'interno di una qualche soluzione ottimale. Nel secondo caso, invece, osserviamo che se $\ell_{i+1} \in X^*$ allora $X_{i+1} \subseteq X^*$ è ancora vero.

Ci rimane quindi da considerare il caso in cui $X_{i+1} = X_i \cup \{\ell_{i+1}\}$ ma $\ell_{i+1} \notin X^*$. Osserviamo quindi che debba necessariamente esistere almeno un altro ladro $\ell_j \in X^* - X_{i+1}$ poiché altrimenti X_{i+1} sarebbe una soluzione con più ladri di quella ottimale, il che è impossibile. Ma allora, dato che $x_{i+1} \in X_{i+1} - X^*$ e $x_j \in X^* - X_{i+1}$, ciò può accadere solo se entrambi tali ladri vengono catturati dalla stessa guardia g_h, poiché altrimenti avremmo che o il nostro algoritmo abbia sbagliato a non catturare x_j oppure che la soluzione X^* non sia ottimale in quanto x_{i+1} potrebbe essere aggiunto in essa. Per via della scelta greedy, sappiamo che $\text{dist}(h, i+1) \leq \text{dist}(h, j)$. Di conseguenza, abbiamo che $X' = (X^* - \{x_j\}) \cup x_{i+1}$ è una soluzione ottimale, poiché $|X'| = |X^*| = s^*$, contenente X_{i+1}. Ciò conclude il passo induttivo.

Sia quindi $X^\#$ la soluzione ottimale tale che $X_k \subseteq X^\#$. Supponiamo per assurdo che $X_k \neq X^\#$, dunque che esista un ladro $\ell_t \in X^\# - X_k$. Per i vincoli imposti dal problema, tale ladro deve essere catturato da una guardia. Ma allora, esiste una guardia non occupata che avrebbe potuto catturare ℓ_t, implicando che il nostro algoritmo abbia sbagliato. Dunque, concludiamo necessariamente che $X_k = X^\#$.

3 Algoritmi Divide et Impera

3.1 Esercizi

Esercizio 3.1 (Potenza modulare)**.** *Progettare un algoritmo che dati tre interi a, n ed m calcoli il valore $a^n \pmod m$ in tempo $O(\log n)$.*

(Soluzione)

Esercizio 3.2 (Elemento non doppione)**.** *Sia A un array ordinato di n interi dove ogni valore in A occorre esattamente due volte tranne uno. Progettare un algoritmo di tempo $O(\log n)$ che trovi l'elemento non doppione in A.*

(Soluzione)

Esercizio 3.3 (Array con pozzo)**.** *Dato un array A di n elementi, un indice i al suo interno è detto "pozzo" se si verifica che $A[1] > \ldots > A[i] < \ldots < A[n]$. Progettare un algoritmo di complessità $O(\log n)$ che, assumendo la sua esistenza, trovi il pozzo di un array.*

(Soluzione)

Esercizio 3.4 (Radice quadrata)**.** *Progettare un algoritmo che dato un intero n calcoli il valore $\lfloor \sqrt{n} \rfloor$ in tempo $O(\log n)$*

(Soluzione)

Esercizio 3.5 (Somma complementare)**.** *Siano X e Y due array di n interi ordinati in senso crescente. Dato un intero z e i due array X, Y, progettare un algoritmo di complessità $O(n \log n)$ che trovi, se esiste, una coppia di indici (i, j) tali che $X[i] + Y[j] = z$.*

(Soluzione)

Esercizio 3.6 (Mediano di due array)**.** *Progettare un algoritmo di complessità $O(\log n)$ che dati in input due vettori A, B di n interi ordinati in senso non decrescente restituisca il mediano dei $2n$ elementi.*

Nota: essendo il numero totale di elementi pari, il mediano sarà la media tra i due valori centrali ottenuti disponendo tutti gli elementi in modo ordinato

(Soluzione)

Esercizio 3.7 (Sottostringhe binarie)**.** *Data una stringa binaria di lunghezza n, vogliamo trovare il numero delle sue sotto-stringhe che cominciano con 0 e finiscono con 1. Progettare un algoritmo divide et impera che risolva il problema in $\Theta(n \log n)$*

(Soluzione)

3.2 Soluzioni

Soluzione 3.1 (Potenza modulare)**.** *(Traccia)*

function POW_MOD_M(a, n, m)
  if $n = 0$ **then**
    return 1
  else if n è pari **then**
    val $\leftarrow$ pow_mod_n($a, \frac{n}{2}, m$)
    return val $\cdot$ val $\pmod m$
  else
    val $\leftarrow$ pow_mod_n($a, \frac{n-1}{2}, m$)
    return val $\cdot$ val $\cdot\, a \pmod m$
  end if
end function

Soluzione 3.2 (Elemento non doppione)**.** *(Traccia)*

Prima di tutto, osserviamo che un generico array B costruito come A contenga un elemento non doppione se e solo se $|B|$ è dispari. Questo ci permette di applicare un approccio divide et impera. Siano quindi a e b gli indici estremi del sotto-array di A attualmente considerato e sia m il punto medio tra a e b.

Prima di applicare la precedente osservazione, dobbiamo *aggiustare il tiro* dei due sotto-array indotti da m: se m ha un doppione, dobbiamo mantenere quest'ultimo all'interno dello stesso sotto-array di m. Per tanto, se $A[m] == A[m+1]$, utilizziamo $m+1$ come effettivo centro dell'array. A questo punto, ci basta controllare la cardinalità dei due sotto-array indotti dal centro per sapere quale dei due contenga l'elemento non doppione. Tuttavia, dobbiamo aggiungere un controllo speciale per il caso $|A| = 3$, altrimenti l'algoritmo andrà in loop infinito.

function NON-DOUBLE(A, a, b) ▷ a e b sono gli estremi di A
  if $a = b$ **then**
    return $A[a]$
  end if
  if $a = b - 2$ **then** ▷ $|A| = 3$
    return $(A[a] = A[a+1])\ ?\ A[b] : A[a]$
  end if

```
    m ← ⌊(a+b)/2⌋
    if A[m] = A[m + 1] then
        m ← m + 1
    end if
    if m − a ≡ 0 (mod 2) then                ▷ |A[a : m]| = m − a + 1 è dispari
        return NON-DOUBLE(A, a, m)
    else
        return NON-DOUBLE(A, m + 1, b)
    end if
end function
```

L'equazione di ricorrenza di tale algoritmo è chiaramente $T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$, la quale ha soluzione $T(n) = O(\log n)$, soddisfacendo il vincolo temporale richiesto. La correttezza dell'algoritmo può essere dimostrata per induzione su $|A|$ utilizzando le precedenti osservazioni.

Soluzione 3.3 (Array con pozzo)**.** *(Traccia)*

L'idea e la correttezza dell'algoritmo sono state discusse durante la lezione.

```
function FIND_SINK(A, a, b)
    if a = b then
        return a
    end if
    if a = b − 1 then
        return (A[a] < A[b]) ? a : b
    end if
    m ← ⌊(a+b)/2⌋
    if A[m − 1] < A[m] then
        return find_sink(A, a, m)
    else if A[m] > A[m + 1] then
        return find_sink(A, m + 1, b)
    else
        return m
    end if
end function
```

Soluzione 3.4 (Radice quadrata)**.** *(Traccia)*

```
function SQRT(n)
```

```
    a, b ← 1, n
    approx = 0
    while a ≤ b do
        m ← ⌊(a+b)/2⌋
        if m² = n then
            return m
        else if m² < n then
            approx ← m
            a ← m + 1
        else
            b ← m − 1
        end if
    end while
    return approx
end function
```

Soluzione 3.5 (Somma complementare)**.** *(Traccia)*

```
function ADDITIVE_SCOMPOSITION(X, Y, z)
    for i = 1, ..., n do
        j ← find_index(Y, z − X[i])
        if j ≠ ∅ then
            return (i, j)
        end if
    end for
    return ∅
end function
```

Soluzione 3.6 (Mediano di due array)**.** *(Traccia)*

```
function DOUBLE_MEDIAN(A, B, a, b)
    n = len(A)
    m = ⌊(a+b)/2⌋
    left_A = A[m]
    if m_A ≠ n − 1 then
        right_A = A[m]
        left_B = B[n − m − 2]
    else
        right_A = +∞
```

```
        left_B = −∞
    end if
    if m ≠ 0 then
        right_B = B[n − m − 1]
    else
        right_B = +∞
    end if
    if left_A ≤ right_B and left_B ≤ right_A then
        return (max(left_A, left_B) + min(right_A, right_B))/2
    else if left_A > left_B then
        return double_median(A, B, a, m − 1)
    else
        return double_median(A, B, m + 1, b)
    end if
end function
```

Soluzione 3.7 (Sottostringhe binarie)**.** *(Traccia)*

```
function FIND-SUBSTRINGS(S)
    n ← len(S)
    if n = 1 then
        return 0
    end if
    m ← ⌊n/2⌋
    c_sx ← FIND-SUBSTRINGS(S[1 : m])
    c_dx ← FIND-SUBSTRINGS(S[m : n])
    i, j ← 0
    for k = 1, . . . , m − 1 do
        if S[k] = 0 then
            i ← i + 1
        end if
    end for
    for k = m, . . . , n do
        if S[k] = 1 then
            j ← j + 1
        end if
    end for
    return c_sx + c_dx + i · j
end function
```

4 Programmazione Dinamica

4.1 Esercizi

Esercizio 4.1 (Sequenze di somma n con tre interi)**.** *Dati in input i tre numeri interi positivi x_1, x_2, x_3 e un numero n, progettare un algoritmo di costo $O(n)$ che trovi il numero di sequenze composte da x_1, x_2, x_3 (anche ripetuti) la cui somma degli elementi è n.*

Ad esempio, per $x_1 = 2, x_2 = 4, x_3 = 8$ e $n = 10$, l'output deve essere 10. Difatti, le sequenze possibili di somma 10 sono:

$$2,8 \qquad 8,2 \qquad 2,4,4 \qquad 4,2,4 \qquad 4,4,2 \qquad 2,2,2,4$$

$$2,2,4,2 \qquad 2,4,2,2 \qquad 4,2,2,2 \qquad 2,2,2,2,2$$

(Soluzione)

Esercizio 4.2 (Il Fantastico Mr. Fox)**.** *Il Fantastico Mr. Fox è un ladro professionista e sta progettando un grande furto in una nota strada in cui si trovano n case adiacenti tra loro, poste in linea retta. Ogni casa del quartiere ha una certa quantità m_i di soldi al suo interno. Inoltre, ciascuna di esse è dotata di un sistema di sicurezza che contatterà automaticamente la polizia se sia essa che una delle case ad essere adiacenti vengono derubate. Ad esempio, se la casa 4 e la casa 5 venissero derubate verrebbe allertato il sistema, mentre ciò non accadrebbe se venissero derubate le case 3 e 5.*

Progettare un algoritmo che dato in input l'insieme $m_1, \ldots, m_n$ di soldi all'interno delle case restituisca la quantità massima di soldi che Mr. Fox può rubare in una sola notte

(Soluzione)

Esercizio 4.3 (Numero di passeggiate)**.** *Progettare un algoritmo che dato in input un grafo indiretto G, un vertice $x \in V(G)$ e un valore $k \in \mathbb{N}$, restituisca il numero di passeggiate distinte lunghe k aventi x come origine. La complessità dell'algoritmo deve essere $O(n^2k)$*

Nota: ricordiamo che una passeggiata è un cammino su un grafo che può anche ripetere vertici ed archi già percorsi precedentemente.

(Soluzione)

Esercizio 4.4 (Bi-partizione di somma uguale)**.** *Dato un insieme S di n interi positivi, progettare un algoritmo che restituisca True se esiste una partizione di S in due sotto-insiemi la cui somma degli elementi di entrambi sia uguale e False altrimenti.*

(Soluzione)

Esercizio 4.5 (Cammino bicolore)**.** *Sia G un grafo diretto aciclico avente vertici colorati di rosso o blu. Progettare un algoritmo di tempo $O(n^2m)$ che dati il grafo G, due vertici $x, y \in V(G)$ e la lista C dei colori dei vertici di G, restituisca True se esiste un cammino $x \to y$ passante per lo stesso numero di vertici blu e rossi (es: un cammino passante per 8 vertici con 4 rossi e 4 blu), oppure False se tale cammino non esiste.*

(Soluzione)

Esercizio 4.6 (Din, Don, Jump!)**.** *La parrocchia della Croce di Santa Minerva sta organizzando un torneo per un nuovo gioco dato dalla combinazione di Campana e Salto triplo: Din, Don, Jump!*

Il gioco prevede n caselle poste in fila, ciascuna dotata di un punteggio (positivo o negativo). Partendo dalla prima casella, ogni giocatore può effettuare un numero arbitrario di salti in avanti. Per ogni casella toccata, il giocatore riceve il punteggio da essa indicato. Tuttavia, ogni salto ha un vincolo: dopo aver saltato k caselle, ogni salto successivo dovrà essere di massimo k caselle. In altre parole, il numero di caselle saltate non può crescere. In particolare, ogni giocatore è costretto a toccare la prima e l'ultima casella.

1. *Progettare un algoritmo di complessità $O(n^3)$ che dati in input i punteggi $p_1, \ldots, p_n$ restituisca il punteggio massimo ottenibile da un giocatore*
2. *È possibile ridurre il costo di tale algoritmo a $O(n^2)$?*

(Soluzione)

Esercizio 4.7 (Sequenze lecite)**.** *Dati due interi positivi n ed m, definiamo come lecita una sequenza di interi che gode delle seguenti tre proprietà:*

1. *La sequenza è lunga n*
2. *Gli elementi della sequenza sono interi tra 1 e m*

3. *L'elemento in posizione i tale che $i \in (1, n]$ della sequenza è un divisore dell'elemento in posizione $i - 1$*

Dare un algoritmo che dati n ed m in input restituisca in $O(nm^2)$ il numero di sequenze lecite formabili con n ed m. Ad esempio per $n = 3$ e $m = 4$ l'algoritmo, l'output deve essere 13. Difatti, le uniche sequenze lecite sono:

444 442 441 422 421 411 333 331 311 222 221 211 111

(Soluzione)

Esercizio 4.8 (Sotto-sequenze lecite massime)**.** *Data una sequenza di interi X, definiamo una sotto-sequenza di X come lecita se essa non contiene elementi di X consecutivi. Definiamo inoltre come valore della sotto-sequenza la somma dei suoi elementi.*

Ad esempio, per $X = [5, -3, 4, 11, 2]$, le sotto-sequenze $[5, \_, 4, \_, 2]$ e $[\_, -3, \_, \_, 2]$ sono lecite, mentre $[5, \_, 4, 11, \_]$ non è lecita. Inoltre, il valore della sotto-sequenza $[5, \_, 4, \_, 2]$ è 11.

Dare lo pseudocodice di:

1. *Un algoritmo che, data una sequenza X in input, restituisca in $O(n)$ il massimo valore possibile per le sue sotto-sequenze lecite.*
2. *Un algoritmo che, data la sequenza X in input, restituisca in $O(n)$ una sotto-sequenza lecita di valore massimo*

Ad esempio, per $X = [5, -3, 4, 11, 2]$ una sotto-sequenza di valore massimo è $[5, \_, \_, 11, \_]$, avente valore pari a 16

(Soluzione)

Esercizio 4.9 (Cash Dance)**.** *Alice e Bob giocano a Cash Dance, un gioco a turni alternati dove vi sono n monete, ciascuna dotata di un valore, le quali sono poste in riga. Ad ogni turno il giocatore attuale può scegliere se prendere una moneta dall'inizio della riga o dalla fine della riga, rimuovendola. Il punteggio di ogni giocatore è dato dalla somma dei valori delle proprie monete rimosse. Inoltre, ogni giocatore è sempre a conoscenza di tutte le monete presenti nel gioco.*

Progettare un algoritmo di complessità $O(n^2)$ che dati i valori $v_1, \ldots, v_n$ delle monete (dove l'i-esima moneta viene prima dell'$(i+1)$-esima all'interno della riga) determini il punteggio massimo ottenibile da Alice assumendo che:

- *Alice muove per prima*
- *La strategia di Bob è quella di minimizzare il punteggio di Alice (infondo anche lui vuole vincere!)*

(Soluzione)

Esercizio 4.10 (Stringa codificata)**.** *Un messaggio (costituito solo dalle 26 lettere A-Z) è stato codificato in una stringa di cifre 0-9 utilizzato la seguente mappatura:*

$$'A' \to \text{"1"} \qquad 'B' \to \text{"2"} \qquad \dots \qquad 'Z' \to \text{"26"}$$

Per decodificare il messaggio, tutte le cifre devono essere raggruppate e mappate alle lettere originali che le hanno generate. Tuttavia, potrebbero esservi più decodifiche possibili per una data stringa, tra cui alcune potrebbero essere invalide.

Ad esempio, la stringa "11106" può essere decodificata come "AAJF" (raggruppandole come [1, 1, 10, 6]) oppure come "KJF" (raggruppandole come [11, 10, 6]), mentre il raggruppamento [1, 11, 06] è invalido poiché la codifica "06" non è generabile da alcuna lettera.

Progettare un algoritmo di complessità $O(n)$ che data in input la stringa S contenente n cifre calcoli il numero di modi in cui sia possibile decodificarla.

(Soluzione)

Esercizio 4.11 (La piccola Irene)**.** *Giocando con vostra figlia Irene trovate n scatole rettangolari, ciascuna descritta da una tripla di dimensioni (w_i, d_i, h_i), ossia larghezza, profondità e altezza. Irene vuole costruire una torre stabile impilando le n scatole a disposizione.*

Affinché una scatola i possa essere messa al di sopra di un'altra scatola j, è necessario che le dimensioni della base della scatola i siano minori o uguali a quelle della base della scatola j su cui poggia. Ad esempio, una scatola con base 4×3 non può essere poggiata su una scatola con base 2×3, ma può essere poggiata su una scatola con base 4×4.

1. *Assumendo che le scatole non possano essere ruotate, ossia che per ogni scatola solo la dimensione h_i possa essere rivolta verso l'alto mentre w_i*

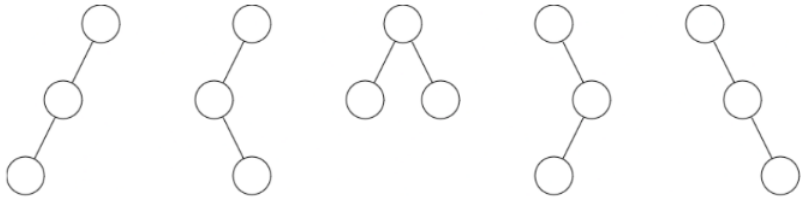
e d_i ne definiscono la base, progettare un algoritmo di costo $O(n^3)$ che date in input le dimensioni delle scatole restituisca l'altezza massima ottenibile per la torre

2. *Tramite la soluzione del punto precedente, progettare un algoritmo di tempo $O(n^3)$ che ricavi l'ordine delle scatole che compongono la torre*

(Soluzione)

Esercizio 4.12 (Numero di alberi binari)**.** *Progettare un algoritmo di complessità $O(n^2)$ che dato in input un intero positivo n restituisca il numero di diversi alberi binari aventi n nodi (indipendentemente dagli indici assegnati ad ogni nodo).*

Ad esempio, per $n = 1$ la risposta deve essere 1 in quanto esiste solo l'albero formato da una radice, mentre per $n = 3$ la risposta deve essere 5 in quanto i possibili alberi siano i seguenti:



(Soluzione)

Esercizio 4.13 (Tre operazioni)**.** *Progettare un algoritmo di tempo $O(n)$ che dato un intero n, con $n \geq 2$, vogliamo contare il numero di modi in cui è possibile ottenere n partendo dal numero 2 ed applicando una serie di operazioni scelte tra: incrementare di 1 il valore precedente, raddoppiare il valore precedente, triplicare il valore precedente.*

Ad esempio, per $n = 10$ l'output dovrà essere 9 in quanto abbiamo le seguenti serie di mosse:

$(2, 3, 4, 5, 6, 7, 8, 9, 10)$	$(2, 3, 4, 5, 10)$	$(2, 3, 4, 8, 9, 10)$
$(2, 3, 6, 7, 8, 9, 10)$	$(2, 3, 9, 10)$	$(2, 4, 5, 10)$
$(2, 4, 5, 6, 7, 8, 9, 10)$	$(2, 4, 8, 9, 10)$	$(2, 6, 7, 8, 9, 10)$

(Soluzione)

Esercizio 4.14 (Edit distance)**.** *Date due stringhe X, Y rispettivamente lunghe n e m caratteri, definiamo l'edit distance tra X e Y come il minimo numero di operazioni necessarie per convertire una delle due nell'altra. In particolare, le operazioni ammesse sono: aggiunta di un carattere in un punto qualsiasi della stringa, eliminazione di un carattere qualsiasi della stringa o rimpiazzo di un carattere qualsiasi della stringa.*

Progettare un algoritmo di costo $O(nm)$ che date in input due stringhe X, Y restituisca la loro edit distance.

(Soluzione)

Esercizio 4.15 (5-copertura di costo minimo)**.** *Data una sequenza X di n interi positivi, definiamo come 5-copertura di X una sotto-sequenza A di suoi elementi se, presi 5 elementi consecutivi qualsiasi della sequenza di X, almeno uno di questi è in A. Dare lo pseudocodice di un algoritmo di complessità $O(n)$ che data una sequenza X in input restituisca una 5-copertura di costo minimo e il suo costo.*

Ad esempio, per $X = [2, 4, 1, 2, 6, 4, 8, 3, 5, 1]$ una 5-copertura di costo minimo è $[\_, \_, 1, \_, \_, \_, \_, 3, \_, \_]$, avente costo pari a 4

(Soluzione)

Esercizio 4.16 (Kermit la rana)**.** *Kermit sta cercando di attraversare un fiume il cui percorso da una sponda all'altra è descritto da n quadranti posizionati uno accanto all'altro. Ogni quadrante può corrispondere ad una pietra, rappresentata da un 1, oppure una zona d'acqua, rappresentata da uno 0.*

Per poter attraversare, Kermit deve saltare da una pietra all'altra, senza mai toccare una zona d'acqua, rispettando i seguenti vincoli:

- *Quando Kermit effettua un salto di k quadranti, il salto successivo potrà essere solo di $k-1$, k o $k+1$ quadranti.*
- *Il primo salto dalla sponda iniziale al primo quadrante sarà sempre di ampiezza 1 (assumiamo quindi che il primo e l'ultimo quadrante contengano sempre una pietra).*
- *Non è possibile tornare su una pietra posizionata prima della pietra attuale*

Dato in input un array Q descrivente i quadranti del fiume:

1. *Progettare un algoritmo che in $O(n^2)$ restituisca True se Kermit possa raggiungere l'ultima pietra partendo dalla prima e False altrimenti.*

2. *Estendere la soluzione del punto precedente per ottenere una sequenza delle ampiezze dei salti che certifichi che Kermit possa raggiungere l'ultima pietra. Ritornare una lista vuota se non esiste tale sequenza.*

Ad esempio, data la lista $[1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1]$ il primo algoritmo dovrà ritornare True in quanto, una possibile sequenza di ampiezze dei salti è $[1, 2, 1, 2, 3, 4, 4]$, quale verrà ritornata dal secondo algoritmo. Per quanto riguarda la lista $[1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1]$, invece, il primo algoritmo ritornerà False poiché la distanza tra la quarta e l'ottava pietra è troppo ampia in ogni possibilità, dunque il secondo algoritmo ritornerà $[\,]$.

(Soluzione)

Esercizio 4.17 (Nelle terre di Renais)**.** *Lyon, principe dell'impero di Grado, ha catturato e imprigionato in una fortezza la principessa Eirika, del regno di Renais. La fortezza è rappresentata da una griglia di $n \times m$ stanze. Partendo dall'inizio della fortezza, la stanza $(0, 0)$, il principe Ephraim deve raggiungere la cella in cui è tenuta prigioniera sua sorella, la stanza (n, m). Durante la traversata, Ephraim può procedere solo nelle stanze direttamente a sud o ad est della precedente (dunque può solo spostarsi dalla casella (i, j) alla casella $(i + 1, j)$ o $(i, j + 1)$).*

Ephraim possiede dei punti salute iniziali, rappresentati da un numero intero. Ogni stanza del dungeon, incluse quella iniziale e quella finale, può contenere degli sgherri all'interno (la cui quantità è rappresentata da un intero negativo) o delle cure (la cui quantità è rappresentata da un intero positivo). Per ogni sgherro affrontato in una stanza, Ephraim perderà un punto salute, mentre ne guadagnerà uno per ogni cura consumata. Se in un qualsiasi momento la salute di Ephraim raggiunge un numero minore o uguale a 0, Ephraim morirà immediatamente.

Progettare un algoritmo che data in input la matrice M, i cui valori indicano gli interi rappresentanti gli elementi all'interno delle stanze del dungeon, trovi la minima quantità di salute che Ephraim deve avere per salvare sua sorella.

(Soluzione)

Esercizio 4.18 (Torneo)**.** *Durante un girone di un torneo, Fabio si ritrova assieme ad n sfidanti. Ciascuno sfidante i richiede uno sforzo s_i e fornisce p_i punti. Per qualificarsi al girone successivo, Fabio deve ottenere un punteggio pari ad almeno Q. Inoltre, per raggiungere tale punteggio Fabio può scegliere un qualsiasi sottoinsieme degli n sfidanti da affrontare.*

Progettare un algoritmo che dati in input $Q, s_1, \ldots, s_n, p_1, \ldots, p_n$, restituisca un sottoinsieme di sfidanti che permetta a Fabio di qualificarsi con lo sforzo minimo possibile.

(Soluzione)

4.2 Soluzioni

Soluzione 4.1 (Sequenze di somma n con tre interi)**.** *(Traccia)*

Definiamo il seguente array di dimensione $n + 1$:

$$T[k] = \text{(numero di sequenze di somma } k \text{ con valori } x_1, x_2, x_3)$$

Poiché $x_1, x_2, x_3 > 0$, per $k < 0$ non esisterà nessuna sequenza di somma k. Se invece $k = 0$, l'unica sequenza di somma k sarà la sequenza vuota. Infine, se $k > 0$ ogni sequenza di somma k corrisponderà alla somma tra il numero di sequenze di somma $k - x_1$, $k - x_2$ e $k - x_3$ (come se avessimo "accodato" il valore x_1, x_2 o x_3 a tutte le sequenze già calcolate).

In altre parole, abbiamo che:

$$T[i] = \begin{cases} 0 & \text{se } i < 0 \\ 1 & \text{se } i = 0 \\ T[i - x_1] + T[i - x_2] + T[i - x_3] & \text{se } i > 0 \end{cases}$$

Di conseguenza, l'algoritmo sarà:

```
function FIND_SEQUENCES(n, x1, x2, x3)
    T ← [0, ... 0]                                  ▷ n+1 elementi
    T[0] ← 1
    for i = 1, ..., n do
        for j = 1, 2, 3 do
            if i − xj ≥ 0 then
                T[i] ← T[i] + T[i − xj]
            end if
        end for
    end for
end function
```

Soluzione 4.2 (Il Fantastico Mr. Fox)**.** *(Traccia)*

Definiamo il seguente array di n elementi:

$$T[i] = \text{(massima quantità di soldi ottenibili con le prime } i \text{ case)}$$

Per ogni i-esima casa, abbiamo due possibili opzioni al fine di evitare di allertare il sistema: derubare l'i-esima casa e poi ragionare su quali derubare

tra le prime $i-2$ oppure non derubare l'i-esima casa e ragionare direttamente sulle prime $i-1$. In altre parole, abbiamo che:

$$T[i] = \begin{cases} m_1 & \text{se } i = 1 \\ \max(m_1, m_2) & \text{se } i = 2 \\ \max(T[i-1], T[i-2] + m_i) & \text{altrimenti} \end{cases}$$

Di conseguenza, l'algoritmo sarà:

function FIND_MAX_ROB($m_1, \ldots, m_n$)
  $T \leftarrow [0, \ldots, 0]$ ▷ n elementi
  $T[1] \leftarrow m_1$
  $T[2] \leftarrow \max(m_1, m_2)$
  for $i = 3, \ldots, n$ **do**
    $T[i] \leftarrow \max(T[i-1], T[i-2] + m_i)$
  end for
  return $T[n]$
end function

Soluzione 4.3 (Numero passeggiate)**.** *(Traccia)*

Consideriamo quindi la tabella $n \times k$, le cui celle sono definite come:

$$T[v, i] = (\text{numero di passeggiate diverse da } x \text{ a } v \text{ con } i \text{ archi})$$

Ognuna di tali celle può essere calcolata ricorsivamente come:

$$T[v, i] = \begin{cases} 0 & \text{se } v \neq x \text{ e } i = 0 \\ 1 & \text{se } v = x \text{ e } i = 0 \\ \sum_{(u,v) \in E(G)} T[u, i-1] & \text{altrimenti} \end{cases}$$

Una volta calcolata l'intera tabella, la soluzione sarà data da $\sum_{v \in V(G)} T[v, k]$.

function FIND_PATHS(G, x, y, k)
  $T \leftarrow$ Matrice $n \times k$
  for $v \in V(G)$ **do**
    $T[v, 0] \leftarrow 0$
  end for
  $T[x, 0] = 1$
  for $i = 1, \ldots, k$ **do**

```
        for v ∈ V(G) do
            for u ∈ N(v) do                ▷ N(v) è l'insieme degli adiacenti di v
                T[v, i] ← T[v, i] + T[u, i − 1]
            end for
        end for
    end for
    return Σ_{v∈V(G)} T[v, k]
end function
```

Soluzione 4.4 (Bi-partizione di somma uguale)**.** *(Traccia)*

Prima di tutto, è necessario osservare che se la somma s di tutti i valori presenti in S è dispari allora sarà impossibile trovare una soluzione, permettendoci di ritornare automaticamente *False*.

Assumiamo quindi che s sia pari. Poiché una partizione di S è costituita da due insiemi S_1, S_2 tali che $S_1 \cap S_2 = \varnothing$ e $S_1 \cup S_2 = S$, se la somma degli elementi di S_1 corrisponde a $\frac{s}{2}$ allora automaticamente la somma del sottoinsieme $S_2 = S - S_1$ corrisponderà anch'essa a $\frac{s}{2}$. Per tanto, possiamo ridurre il problema a trovare un sottoinsieme di S avente somma $\frac{s}{2}$.

Definiamo quindi la seguente tabella $(n+1) \times \left(\frac{s}{2}+1\right)$:

$$T[i,k] = (\text{esiste un sotto-insieme dei primi } i \text{ elementi con somma } k)$$

Denotiamo con $S_{1:i}$ la restrizione ai primi i elementi di S. Ovviamente, per ogni $i \in [1,n]$, l'insieme $S_{1:i}$ avrà sempre un sottoinsieme di somma $k = 0$, ossia l'insieme vuoto, fornendoci il caso base. Per una qualsiasi somma $k > 0$, invece, abbiamo due possibilità: esiste già un sottoinsieme di $S_{1:i-1}$ avente somma k, dunque possiamo ignorare l'aggiunta dell'i-esimo elemento, oppure esiste un sottoinsieme di $S_{1:i-1}$ avente somma $k - S[i]$ al quale possiamo aggiungere l'i-esimo elemento per ottenere un sottoinsieme di $S_{1:i}$ avente somma k.

Formalmente, dunque, avremo che:

$$T[i,k] = \begin{cases} \text{False} & \text{se } k < 0 \\ \text{True} & \text{se } k = 0 \\ T[i-1,k] \vee T[i-1,k-S[i]] & \text{se } k > 0 \end{cases}$$

Di conseguenza, l'algoritmo sarà:

```
function EQUAL_SUM_PARTITION(S)
    s ← Σ_{i=1}^{n} S[i]
    if s è dispari then
        return False
    end if
    T ← Matrice (n + 1) × (s/2 + 1)
    for i = 0, ... n do
        T[i, 0] = True
    end for
    for i = 0, ..., n do
        for k = 1, ..., s/2 do
            T[i, j] ← False
            if k − S[i] ≥ 0 then
                T[i, j] ← T[i − 1, k] ∨ T[i − 1, k − S[i]]
            else
                T[i, j] ← T[i − 1, k]
            end if
        end for
    end for
    return T[n, s/2]
end function
```

Soluzione 4.5 (Cammino bicolore)**.** *(Traccia)*

Definiamo la seguente tabella di dimensione $n \times n \times n$:

$$T[v, i, j] = \begin{pmatrix}\text{esistenza di un cammino } x \to v \text{ passante} \\ \text{per } i \text{ vertici rossi e } j \text{ vertici blu}\end{pmatrix}$$

Poiché un cammino passante per $i+j$ nodi ha necessariamente $i+j-1$ archi, e poiché un cammino può avere massimo n archi, dato un qualsiasi vertice $v \in V(G)$ si ha che il cammino $x \to v$ passante per i rossi e j blu dipenda da tutti i cammini degli entranti di v aventi $i-1$ rossi e j blu (se v è rosso) oppure i rossi e $j-1$ blue (se v è blu)

In altre parole, abbiamo che:

$$T[v,i,j] = \begin{cases} True & \text{se } v = x, i = 1, j = 0, C[x] = 'R' \\ True & \text{se } v = x, i = 0, j = 1, C[x] = 'B' \\ \bigvee_{(u,v)\in E(G)} T[u, i-1, j] & \text{se } v \neq x, C[x] = 'R' \\ \bigvee_{(u,v)\in E(G)} T[u, i, j-1] & \text{se } v \neq x, C[x] = 'B' \\ False & \text{altrimenti} \end{cases}$$

Dunque, l'algoritmo sarà:

function BIPARTITE_PATH(G, x, y, C)
    T = Tabella $n \times n \times n$ riempita con $False$
    if $C[x]$ == "R" **then**
        $T[x, 1, 0] \leftarrow$ True
    else
        $T[x, 0, 1] \leftarrow$ True
    end if
    for $i = 1, \ldots, n$ **do**
        for $j = 1, \ldots, n$ **do**
            for $v \in V(G) - \{x\}$ **do**
                for $u \in v$.entranti **do**
                    $T[v, i, j] \leftarrow$ False
                    if $C[v] =$ "R" **then**
                        if $i - 1 \geq 0$ **then**
                            $T[v, i, j] \leftarrow T[v, i, j] \vee T[v, i-1, j]$
                        end if
                    else
                        if $j - 1 \geq 0$ **then**
                            $T[v, i, j] \leftarrow T[v, i, j] \vee T[v, i, j-1]$
                        end if
                    end if
                end for
            end for
        end for
    end for
    for $k = 0, \ldots, n$ **do**
        if $T[y, k, k]$ **then**
            return $True$

```
        end if
    end for
    return False
end function
```

Soluzione 4.6 (Din, Don, Jump!)**.** *(Traccia)*

Al fine di poter modellare il problema tramite la programmazione dinamica, è necessario modellare i salti effettuati "all'indietro", partendo dalla casella su cui siamo atterrati alla fine del salto per poi considerare i salti precedenti. Definiamo quindi la seguente tabella $n \times (n-1)$:

$$T[i,k] = \begin{pmatrix} \text{massimi punti ottenibili effettuando un} \\ \text{salto di } k \text{ caselle atterrando sulla casella } i \end{pmatrix}$$

Se atterriamo sulla casella i effettuando un salto di k caselle, ciò significa che il salto è stato effettuato dalla casella $i-k$, dando vita a tre casi:

- Se $i < k+1$ allora saremmo dovuti saltare da una casella precedente alla prima casella, il che è impossibile
- Se $i = k+1$ allora siamo saltati direttamente dalla prima casella alla casella i, dunque il nostro punteggio sarà $p_i + p_1$
- Se $i > k+1$ allora siamo saltati da una casella successiva alla prima, dunque il nostro punteggio sarà dato dalla somma di p_i e il massimo punteggio ottenibile effettuando un salto di al minimo k caselle atterrando sulla casella $i-k$, ossia quella da cui abbiamo appena saltato

Formalmente, dunque, avremo che:

$$T[i,k] = \begin{cases} -\infty & \text{se } i < k+1 \\ p_i + p_1 & \text{se } i = k+1 \\ p_i + \max\limits_{k \le j \le n-1}(T[i-k,j]) & \text{se } i > k+1 \end{cases}$$

Dunque, l'algoritmo sarà:

```
function DIN_DON_JUMP(p_1, ..., p_n)
    T = Matrice n × (n − 1) riempita con −∞
    for i = 1, ..., n do
```

```
        for k = 1, ..., n − 1 do
            if i < k + 1 then
                T[i, k] ← −∞
            else if i == k + 1 then
                T[i, k] ← p_i + p_1
            else
                m ← −∞
                for j = k, ..., n − 1 do
                    m ← max(m, T[i − k, j])
                end for
                T[i, k] ← p_i + m
            end if
        end for
    end for
    return max(T[n, 1], ..., T[n, n])
end function
```

Visti i tre cicli annidati, risulta evidente che il costo di tale algoritmo sia $O(n^3)$. Tuttavia, possiamo ottimizzare il costo definendo in modo leggermente diverso la tabella:

$$T[i,k] = \begin{pmatrix}\text{massimi punti ottenibili effettuando un salto}\\ \text{di minimo } k \text{ caselle atterrando sulla casella } i\end{pmatrix}$$

Tale modifica, apparentemente innocua, cambia radicalmente il modo in cui la tabella risulta formalizzata: effettuare un salto di almeno k caselle equivale ad aver effettuato o un salto di esattamente k caselle oppure un salto di almeno $k+1$ caselle.

$$T[i,k] = \begin{cases} -\infty & \text{se } i < k+1 \\ p_i + p_1 & \text{se } i = k+1 \\ \max(p_i + T[i-k,k], T[i,k+1]) & \text{se } i > k+1 \end{cases}$$

Tuttavia, osserviamo che ora per calcolare $T[i,k]$ sia necessario il valore di $T[i,k+1]$. Questo non è un problema: ci basta percorrere il secondo ciclo for in ordine inverso, dunque con $k = n, \ldots, 1$. Inoltre, è sufficiente in realtà controllare solo i valori $k = i, \ldots, 1$, visto che sappiamo già che tutti i valori tra $k = n, \ldots, i+1$ ci daranno un salto con punteggio $-\infty$.

function DIN_DON_JUMP_OPT($p_1, \ldots, p_n$)

```
    T = Matrice n × (n − 1) riempita con −∞
    for i = 1, . . . , n do
        for k = i, . . . , 1 do
            T[i, k] ← −∞
            if i == k + 1 then
                T[i, k] ← p_i + p_1
            else
                T[i, k] ← max(p_i + T[i − k, k], T[i, k + 1])
            end if
        end for
    end for
    return max(T[n, 1], . . . , T[n, n])
end function
```

Soluzione 4.7 (Sequenze lecite)**.** *(Traccia)*

Impostiamo la seguente tabella di dimensione $n \times m$

$$T[i,j] = \begin{pmatrix} \text{sequenze lecite lunghe } i \\ \text{terminanti con } j \end{pmatrix}$$

Poiché ogni i-esimo elemento deve essere in grado di dividere l'$(i-1)$-esimo elemento di ogni sequenza lecita, otteniamo che:

$$T[i,j] = \begin{cases} 1 & \text{se } i = 1 \\ T[i-1,x_1] + \ldots + T[i-1,x_k] & \text{altrimenti} \end{cases}$$

dove $x_1, \ldots, x_k \in \{x \in [j,m] \mid x \equiv 0 (\text{mod } j)\}$, ossia gli elementi da 1 a m divisibili per j

L'algoritmo finale sarà il seguente, avente costo pari a $O(nm^2)$:

```
function VALID-SEQUENCES(n, m)
    T = tabella n × m
    for j ← 1, . . . , n do
        T[1, j] = 1
    end for
    for i = 1, . . . , n do
        for j = 1, . . . , m do
            for x = 1, . . . , m do
```

```
            if x ≡ 0 (mod j) then
                T[i, j] ← T[i, j] + T[i − 1, x]
            end if
        end for
    end for
end for
return ∑_{i=1}^{m} T[n, i]
end function
```

Soluzione 4.8 (Sotto-sequenze lecite massime)**.** *(Traccia)*

Impostiamo il seguente array di dimensione $n + 1$

$$T[k] = \begin{pmatrix} \text{sotto-sequenza lecita di valore} \\ \text{massimo con valori tra } x_1, \ldots, x_k \end{pmatrix}$$

Poiché ogni k-esimo elemento può essere aggiunto o no alla sotto-sequenza (creando o espandendo un gap in essa), otteniamo che:

$$T[k] = \begin{cases} \max(0, x_1) & \text{se } k = 1 \\ \max(T[1], x_2) & \text{se } k = 2 \\ \max(T[k-2] + x_k, T[k-1]) & \text{altrimenti} \end{cases}$$

Il primo algoritmo richiesto avente costo $O(n)$ sarà il seguente:

```
function FILL-TABLE(X)
    n ← len(X)
    T ← [0, . . . , 0]
    T[1] ← max(0, X[1])
    T[2] ← max(T[1], X[2])
    for k = 3, . . . , n do
        T[k] ← max(T[k − 2] + X[k], T[k − 1])
    end for
    return T
end function
function MAX-SUBSEQUENCE-VALUE(X)
    n ← len(X)
    T ← fill − table(X)
    return T[n]
```

end function

Una volta risolto il primo algoritmo, possiamo utilizzare la tabella calcolata per risolvere anche la seconda richiesta:

1. Calcoliamo la stessa tabella T
2. Partendo da $k = n$, decrementiamo k ad ogni iterazione fino a raggiungere $k = 2$.
3. Se $T[k] = T[k-2] + X[k]$, allora l'elemento $X[k]$ era stato considerato per il valore della sotto-sequenza massima, venendo aggiunto all'output. In caso contrario, l'elemento verrà scartato.
4. Se $T[2] = T[0] + X[k]$, allora l'elemento $X[2]$ è stato aggiunto alla sequenza, implicando che non sia possibile aggiungere l'elemento $X[1]$. In caso contrario, l'elemento $X[1]$ verrà aggiunto se e solo se esso è positivo, poiché altrimenti il valore massimo della sotto-sequenza finale decrementerebbe.

```
function MAX-SUBSEQUENCE(X)
    n ← len(X)
    T ← fill − table(X)
    Sol ← [_, . . . , _]
    last ← n
    for k = n, . . . , 2 do
        if T[k] = T[k − 2] + X[k] then
            Sol[k] ← X[k]
            last ← k − 2
        else
            last ← k − 1
        end if
    end for
    if last = 1andX[1] > 0 then
        Sol[1] ← X[1]
    end if
    return Sol
end function
```

Soluzione 4.9 (Cash Dance)**.** *(Traccia)*

Supponiamo di essere ad un turno generico della partita, dove la riga è composta dalle monete $m_i, m_{i+1}, \ldots, m_{j-1}, m_j$. Definiamo quindi la seguente tabella $n \times n$ come:

$$T[i,j] = \begin{pmatrix} \text{massimo punteggio di Alice quando} \\ m_i \text{ è la prima moneta e } m_j \text{ è l'ultima} \end{pmatrix}$$

Poiché Bob gioca in modo ottimale, ad ogni suo turno sceglierà la moneta di valore maggiore tra le due disponibili in modo che il punteggio di Alice sia il minore possibile. Abbiamo quindi due situazioni:

- Se Alice scegliesse la moneta m_i, allora Bob sceglierà la moneta di valore maggiore tra m_{i+1} e m_j. Di conseguenza, Alice si ritroverà con le monete $m_{i+2}, \ldots, m_j$ oppure $m_{i+1}, \ldots, m_{j-1}$
- Se Alice scegliesse la moneta m_j, allora Bob sceglierà la moneta di valore maggiore tra m_i e m_{j-1}. Di conseguenza, Alice si ritroverà con le monete $m_{i+1}, \ldots, m_{j-1}$ oppure $m_i, \ldots, m_{j-2}$

Inoltre, notiamo che ogni caso in cui $i > j$ sia impossibile, poiché la prima moneta non può venire dopo l'ultima. La scelta di Alice sarà quindi quella che la porterà nella situazione migliore, ossia quella in grado di massimizzare la somma tra il valore della moneta scelta e il valore ottenibile con le monete rimanenti. In altre parole, abbiamo che:

$$T[i,j] = \begin{cases} 0 & \text{se } i > j \\ v_i & \text{se } i = j \\ \max(v_i, v_j) & \text{se } i = j-1 \\ \max \begin{pmatrix} v_i + \min(T[i+2,j],\ T[i+1,j-1]), \\ v_j + \min(T[i+1,j-1],\ T[i,j-2]) \end{pmatrix} & \text{altrimenti} \end{cases}$$

A questo punto, notiamo che sia necessario fare attenzione al calcolo di $T[i,j]$: è necessario riempire la tabella in "anti-diagonale", in modo che i valori necessari siano sempre stati già calcolati correttamente. In particolare, sarà necessario decrementare il valore di i partendo da n, mentre j verrà incrementato da 1 fino ad n. Tuttavia, siccome dobbiamo ignorare i casi in cui $i > j$, possiamo far partire j da i fino ad n in ogni sotto-ciclo.

function FIND_OPTIMAL_STRATEGY($v_1, \ldots, v_n$)

```
    T = Matrice n × n
    for i = n, ..., 1 do
        for j = i, ..., n do
            if i > j then
                T[i, j] = 0
            else if i == j then
                T[i, j] = v_i
            else if i == j − 1 then
                T[i, j] = max(v_1, v_j)
            else
                x = T[i + 2][j]
                y = T[i + 1][j − 1]
                z = T[i][j − 2]
                T[i, j] = max(v_1 + min(x, y), v_j + min(y, z))
            end if
        end for
    end for
    return T[1, n]
end function
```

Soluzione 4.10 (Stringa codificata)**.** *(Traccia)*

Per via della presenza di alcune codifiche invalide nel caso in cui vi sia da zero, per risolvere il problema risulta più comodo considerare i raggruppamenti possibili a partire dalla fine della stringa.

Definiamo quindi il seguente array $n + 1$ elementi:

$$T[i] = \text{(numero di decodifiche possibili con le ultime } i \text{ cifre della stringa)}$$

Tramite tale definizione, se l'$(n - i)$-esima cifra risulta essere uno 0 allora non vi sarà alcuna decodifica possibile ottenibile tramite l'aggiunta di tale cifra all'inizio dell'insieme di quelle considerate.

Se $(n-i)$-esima cifra è diversa da 0, invece, il numero di decodifiche ottenibili tramite la sua aggiunta corrisponde al numero di decodifiche possibili considerandola come a se stante oppure come estensione della cifra $(n-i-1)$, ossia come se la nuova cifra rappresentasse la parte decimale della cifra considerata

precedentemente. In altre parole, abbiamo che:

$$T[i] = \begin{cases} 1 & \text{se } i = 0 \\ T[i-1] & \text{se } i = 1 \land S_{n-i} \neq 0 \\ T[i-1] & \text{se } i > 1 \land S_{n-i} \neq 0 \land 10 \cdot S_{n-i} + S_{n-(i-1)} > 26 \\ T[i-1] + T[i-2] & \text{se } i > 1 \land S_{n-i} \neq 0 \land 10 \cdot S_{n-i} + S_{n-(i-1)} \leq 26 \\ 0 & \text{altrimenti} \end{cases}$$

Dunque, l'algoritmo sarà:

```
function WAYS_TO_DECODE(S)
    T = [0, ..., 0]                                   ▷ n+1 elementi
    T[0] = 1
    for i = 1, ..., n do
        if S[n − i] == '0' then
            T[i] = 0
        else
            T[i] = T[i − 1]
            if i > 1 and 10 · S[n − i] + S[n − (i − 1)] ≤ 26 then
                T[i] += T[i − 2]
            end if
        end if
    end for
    return T[n]
end function
```

Soluzione 4.11 (La piccola Irene)**.** *(Traccia)*

Definiamo la seguente tabella di dimensione $n \times n$:

$$T[i,k] = \begin{pmatrix} \text{massima altezza ottenibile con una torre di} \\ k \text{ scatole totali e dove la scatola } i \text{ è in cima} \end{pmatrix}$$

Per ogni scatola i, definiamo l'insieme $V(i)$ come l'insieme delle scatole su cui possa essere poggiata, ossia $V(i) = \{1 \leq j \leq n \mid w_i \leq w_j, d_i \leq d_j\}$. A questo punto il problema risulta semplice: per ogni scatola i sarà sufficiente considerare la scelta migliore tra tutte le sotto-torri su sia possibile poggiarla.

In altre parole, abbiamo che:

$$T[i,k] = \begin{cases} h_i & \text{se } k = 1 \\ 0 & \text{se } k > 1 \text{ e } V(i) = \varnothing \\ h_i + \max\limits_{j \in V(i)} (T[j, k-1]) & \text{se } k > 1 \text{ e } V(i) \neq \varnothing \end{cases}$$

Una volta calcolata l'intera tabella, l'altezza massima raggiungibile sarà data dalla cella all'interno con il valore massimo. Difatti, non è detto che la torre più alta utilizzi tutte le scatole a disposizione.

Successivamente, tramite tale valore massimo sarà possibile ripercorrere la tabella al contrario sottraendo ripetutamente l'altezza della scatola attualmente in cima, ottenendo l'intera sequenza.

Dunque, l'algoritmo sarà:

```
function LITTLE_IRENE(D)
    T = Tabella n × n
    for i = 1, ..., n do
        T[i, 1] = h_i
    end for
    for i = 1, ..., n do
        for k = 2, ..., n do
            for j = 1, ..., n do
                if i == j then
                    continue
                end if
                T[i, k] = 0
                if w_i ≤ w_j and d_i ≤ d_j then
                    if T[i, k] < h_i + T[j, k − 1] then
                        T[i, k] = h_i + T[j, k − 1]
                    end if
                end if
            end for
        end for
    end for
    s = 1
    t = 1
    for i = 1, ..., n do
        for k = 1, ..., n do
            if T[s, t] < T[i, k] then
                s = i
                t = k
            end if
        end for
    end for
```

```
    List L = ∅
    L.head_insert(s)
    for k = t, ..., 1 do
        for j = 1, ..., n do
            if T[s,k] == h_s + T[j,k-1] and w_s ≤ w_j and d_s ≤ d_j then
                L.head_insert(j)
                s = j
                break
            end if
        end for
    end for
    return L
end function
```

Soluzione 4.12 (Numero di alberi binari)**.** *(Traccia)*

Impostiamo il seguente array T:

$$T[i] = \text{(numero di alberi binari distinti con } i \text{ nodi)}$$

Per calcolare ogni cella di T, sfruttiamo il fatto che all'interno di un albero binario di i nodi si abbia che $i = 1+sx+dx$, dove sx e dx sono rispettivamente il numero di nodi nel sotto-albero sinistro e destro della radice. Sapendo che il sotto-albero sinistro abbia sx nodi e che quello destro ne abbia dx, moltiplichiamo tutti i modi per ottenere il sotto-albero sinistro per tutti i modi per ottenere il sotto-albero destro. Dunque, abbiamo che:

$$T[i] = \begin{cases} 1 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ \sum_{k=0}^{i-1} T[k] \cdot T[i-k-1] & \text{altrimenti} \end{cases}$$

Il codice finale segue dalla definizione della tabella.

Soluzione 4.13 (Tre operazioni)**.** *(Soluzione)*

Impostiamo la seguente tabella:

$$T[i] = \text{(numero di serie possibili per raggiungere } i \text{ partendo da 2)}$$

Notiamo che sia sempre possibile raggiungere il valore i tramite un incremento, mentre è possibile raggiungere i tramite un raddoppiamento o un triplicamento rispettivamente solo se i è pari o un multiplo di 3. Abbiamo quindi che:

$$T[i] = \begin{cases} 1 & \text{se } i = 2 \\ T[i-1] & \text{se } i \not\equiv 0 \pmod 2 \text{ e } i \not\equiv 0 \pmod 3 \\ T[i-1] + T[i/2] & \text{se } i \equiv 0 \pmod 2 \text{ e } i \not\equiv 0 \pmod 3 \\ T[i-1] + T[i/3] & \text{se } i \not\equiv 0 \pmod 2 \text{ e } i \equiv 0 \pmod 3 \\ T[i-1] + T[i/2] + T[i/3] & \text{se } i \equiv 0 \pmod 2 \text{ e } i \equiv 0 \pmod 3 \end{cases}$$

Il codice segue dalla compilazione della tabella.

Soluzione 4.14 (Edit distance)**.** *(Traccia)*

Definiamo la seguente tabella di dimensioni $(n+1) \times (m+1)$:

$$T[i,j] = \begin{pmatrix} \text{numero minimo di operazioni per convertire i} \\ \text{primi } i \text{ caratteri di } X \text{ nei primi } j \text{ caratteri di } Y \end{pmatrix}$$

Per ogni i-esimo carattere di X considerato e ogni j-esimo carattere di Y, abbiamo due opzioni: tali caratteri sono uguali oppure differenti. Nel primo caso, sarà sufficiente considerare solo il numero di operazioni necessarie per convertire i primi $i-1$ caratteri di X nei primi $j-1$ caratteri di Y. Nel secondo caso, invece, sarà necessario considerare tutte e tre le opzioni disponibili:

- Il carattere di X viene sostituito con il carattere di Y, dunque sarà necessario considerare il numero di operazioni necessarie per convertire i primi $i-1$ caratteri di X nei primi $j-1$ caratteri di Y, come nel caso precedente.
- Il carattere di X viene eliminato, dunque sarà necessario considerare il numero di operazioni necessarie per convertire i primi $i-1$ caratteri di X nei primi j caratteri di Y
- Il carattere di Y viene eliminato, dunque sarà necessario considerare il numero di operazioni necessarie per convertire i primi i caratteri di X nei primi $j-1$ caratteri di Y

In particolare, notiamo che non sia necessario considerare i casi in cui venga aggiunto un carattere poiché eliminarne uno dalla stringa più lunga delle due è equivalente ad aggiungerne uno dalla stringa più corta delle due.

Inoltre, notiamo che, considerando i primi 0 caratteri di Y, per i primi i caratteri di X sarà necessario effettuare i cancellature. Analogamente ciò si verifica se consideriamo i primi 0 caratteri di X e i primi j caratteri di Y.

In altre parole, abbiamo che:

$$T[i,j] = \begin{cases} i & \text{se } j = 0 \\ j & \text{se } i = 0 \\ T[i-1, j-1] & \text{se } X_i = Y_j \\ 1 + \min(T[i-1,j-1], T[i-1,j], T[i,j-1]) & \text{se } X_i \neq Y_j \end{cases}$$

Dunque, l'algoritmo sarà:

```
function EDIT-DISTANCE(X, Y)
    T ← Tabella (n + 1) × (m + 1)
    for i = 0, ..., n do
        T[i, 0] ← i
    end for
    for j = 0, ..., m do
        T[0, j] ← j
    end for
    for j = 1, ..., m do
        for i = 1, ..., n do
            if X[i] = Y[j] then
                T[i, j] ← T[i − 1, j − 1]
            else
                T[i, j] ← 1 + min(T[i − 1, j − 1], T[i − 1, j], T[i, j − 1])
            end if
        end for
    end for
    return T[n, m]
end function
```

Soluzione 4.15 (5-copertura di costo minimo)**.** *(Soluzione)*

Impostiamo il seguente array di dimensione n

$$T[k] = \begin{pmatrix} \text{costo della 5-copertura minima con} \\ \text{valori tra } x_1, \ldots, x_k \text{ e contenente } x_k \end{pmatrix}$$

Il primo elemento sarà sicuramente l'elemento ricoprente il sottoarray del sottoproblema con $k = 1$. Per i sottoproblemi con $k = 2, \ldots, 5$, invece, la copertura di valore minimo sarà data dal elemento $x_1, \ldots, x_5$ avente valore minimo, in quanto esso è in grado di coprire tutte le $k-1$ posizioni precedenti ad esso.

Per i sottoproblemi con $k > 5$, invece, il valore sarà dato dal minimo valore raggiunto nei cinque sottoproblemi precedenti, con l'aggiunta del nuovo elemento. Otteniamo quindi che:

$$T[k] = \begin{cases} x_1 & \text{se } k = 1 \\ \min(T[k-1], 0) + x_k & \text{se } k = 2 \\ \min(T[k-2], T[k-1], 0) + x_k & \text{se } k = 3 \\ \min(T[k-3], T[k-2], T[k-1], 0) + x_k & \text{se } k = 4 \\ \min(T[k-4], T[k-3], T[k-2], T[k-1], 0) + x_k & \text{se } k = 5 \\ \min(T[k-5], T[k-4], T[k-3], T[k-1], T[k-1]) + x_k & \text{altrimenti} \end{cases}$$

Il costo della 5-copertura minima sarà dato da

$$\min(T[n], T[n-1], T[n-2], T[n-3], T[n-4])$$

poiché uno degli ultimi 5 valori dovrà necessariamente essere presente nella 5-copertura minima, altrimenti x_n sarà scoperto. Inoltre, possiamo utilizzare la tabella calcolata per risolvere anche la seconda richiesta. Partendo da $k = n$, aggiungiamo alla soluzione l'elemento x_i tra $x_{k-4}, \ldots, x_k$ avente valore minimo in T poiché sappiamo che esso sia quello all'interno della copertura minima. Successivamente, poniamo $k = i - 1$ e ripetiamo l'intera procedura finché $k - 4 < 0$.

```
function FILL-TABLE(X)
    T ← Array di n elementi inizializzato a 0
    T[1] ← X[1]
    for k = 2, ..., n do
        for i = 1, ..., 5 do
            if k − i < 1 and 0 < T[k] then
                T[k] ← 0
            else if T[k − i] < T[k] then
                T[k] ← T[k − i]
            end if
        end for
```

```
        T[k] ← T[k] + X[k]
    end for
    return T
end function
function MIN-5-COVER(X)
    T ← FILL-TABLE(X)
    min_cost ← min(T[n], T[n − 1], T[n − 2], T[n − 3], T[n − 4])
    Sol ← [_, . . . , _]
    k ← n
    i ← k
    for j = 1, ..., 4 do
        if T[k − j] < T[i] then
            i ← j
        end if
    end for
    Sol[i] ← X[i]
    k ← i − 1
    while k − 4 ≥ 0 do
        i ← k
        for j = 1, ..., 4 do
            if T[k − j] < T[i] then
                i ← j
            end if
        end for
        Sol[i] ← X[i]
        k ← i − 1
    end while
    return min_cost, Sol
end function
```

Soluzione 4.16 (Kermit la rana)**.** *(Traccia)*

Per risolvere questo esercizio, procediamo in modo simile all'esercizio *Din, Don, Jump!*, modellando i salti effettuati "all'indietro", ossia partendo dal quadrante su cui siamo atterrati alla fine del salto per poi considerare i salti precedenti. Definiamo quindi la seguente tabella $n \times (n-1)$:

$$T[i,k] = \begin{pmatrix} \text{raggiungibilità del quadrante } i \\ \text{effettuando un salto di ampiezza } k \end{pmatrix}$$

Se atterriamo sul quadrante i effettuando un salto di k quadranti, ciò significa che il salto è stato effettuato dal quadrante $i-k$, sul quale possiamo essere atterrati precedentemente effettuando un salto di $k-1, k$ o $k+1$ quadranti.

In altre parole, abbiamo che:

$$T[i,k] = \begin{cases} False & \text{se } i < 1 \text{ o } i > n \\ False & \text{se } Q[i] = 0 \\ False & \text{se } i = 1, k \neq 1 \\ True & \text{se } i = 1, k = 1 \\ T[i-k, k-1] \vee T[i-k, k] \vee T[i-k, k+1] & \text{altrimenti} \end{cases}$$

Dunque, l'algoritmo sarà:

```
function KERMIT(Q)
    T = Tabella n × (n − 1)
    T[1, 1] = True
    for k = 2, . . . , n do
        T[1, k] = False
    end for
    for i = 2, . . . , n do
        if Q[i] ≠ 0 then
            for k = 1, . . . , i do
                T[i, k] = False
                for d ∈ {k − 1, k, k + 1} do
                    if 1 ≤ d ≤ n then
                        T[i, k] = T[i, k] ∨ T[i − k, d]
                    end if
                end for
            end for
        end if
    end for
    return ⋁_{k=1}^{n−1} T[n, k]
end function
```

Soluzione 4.17 (Nelle terre di Renais)**.** *(Traccia)*

La strategia migliore, diversamente dal solito, risulta essere quella di costruire la soluzione partendo dalla fine. In particolare, partendo da tale

cella, cerchiamo il minimo di vita necessario per raggiungere tale cella da un qualsiasi altro punto del dungeon. Ovviamente, la soluzione sarà data dal minimo necessario partendo dalla prima cella del dungeon. Definiamo quindi la seguente tabella $n \times m$ come:

$$T[i,j] = \begin{pmatrix} \text{minimi punti vita necessari per raggiungere} \\ \text{la cella } (n,m) \text{ partendo dalla cella } (i,j) \end{pmatrix}$$

Il caso base, dunque sarà dato dall'ultima cella stessa: se la stanza contiene k demoni (dunque $M[n,m] = -k$), allora sarà necessaria una quantità di vita pari a $k+1$. Altrimenti, se la stanza contiene una cura, sarà sufficiente avere un solo punto vita in quanto non subiremo danni. Successivamente, per ogni altra stanza (i,j), il valore sarà dato dalla la vita necessaria per rimanere vivi nella stanza (i,j) (ottenuta negando il valore $M[i,j]$) sommata al minimo tra la quantità di vita necessaria per proseguire a sud o ad est.

Tuttavia, come nel caso base, se tale vita necessaria fosse negativa (ad esempio se ci siamo "curati troppo" sostando nella stanza (i,j)), sarà sufficiente avere un solo punto vita. Inoltre, è necessario puntualizzare che per i due bordi in cui ci troviamo nell'ultima riga o l'ultima colonna del dungeon, sarà possibile proseguire solo in una delle due direzioni.

Formalmente, dunque, avremo che:

$$T[i,j] = \begin{cases} \max(1, -M[i,j] + 1) & \text{se } i = n \text{ e } j = m \\ \max(1, -M[i,j] + T[i+1,j]) & \text{se } i \neq n \text{ e } j = m \\ \max(1, -M[i,j] + T[i,j+1]) & \text{se } i = n \text{ e } j \neq m \\ \max(1, -M[i,j] + \min(T[i+1,j], T[i,j+1])) & \text{altrimenti} \end{cases}$$

Di conseguenza, l'algoritmo sarà:

function SOLVE_DUNGEON(M)
  T = Matrice $n \times m$
  $T[n,m] = \max(1, -M[n,m] + 1)$
  for $i = n-1, \ldots, 0$ **do**
    $T[i,m] = \max(1, -M[i,m] + T[i+1,m])$
  end for
  for $j = m-1, \ldots, 0$ **do**
    $T[n,j] = \max(1, -M[n,j] + T[n,j+1])$
  end for
  for $i = n-2, \ldots, 0$ **do**

```
        for j = m − 2, . . . , 0 do
            T[i, j] = max(1, −M[i, j] + min(T[i + 1, j], T[i, j + 1]))
        end for
    end for
    return T[0, 0]
end function
```

Soluzione 4.18 (Torneo)**.** *(Traccia)*

Il problema risulta essere simile al problema dello zaino, con la differenza che questa volta vogliamo che la capienza totale dello zaino venga raggiunta. Impostiamo quindi la seguente tabella.

$$T[k, q] := \begin{pmatrix} \text{minimo sforzo totale raggiungibile da un sotto-insieme} \\ \text{dei primi } k \text{ sfidanti con punteggio totale almeno a } q \end{pmatrix}$$

la quale viene compilata in modo analogo al problema dello zaino:

$$T[k, q] = \begin{cases} 0 & \text{se } k = 0 \vee q = 0 \\ \min(T[k-1, q], s_k) & \text{se } p_k \geq q \\ \min(T[k-1, q], T[k-1, q-p_k] + s_k) & \text{se } p_k < q \end{cases}$$

Il codice finale segue dalla definizione della tabella.

5 Backtracking

5.1 Esercizi

Esercizio 5.1 (Stampa le parentesi valide)**.** *Progettare un algoritmo che dato un intero positivo n in input stampi tutte le stringhe contenenti n coppie di parentesi, ossia per ogni parentesi aperta deve esservene una chiusa. La complessità dell'algoritmo deve essere $O(nS(n))$, dove $S(n)$ sono il numero di stringhe da stampare.*

Ad esempio, per $n = 3$ le stringhe da stampare sono:

"((()))" "(()())" "(())()" "()(())" "()()()"

(Soluzione)

Esercizio 5.2 (Stampa delle partizioni di n)**.** *Dato un numero intero positivo n, definiamo come partizioni di tutte le sequenze di elementi inferiori o uguali ad n la cui somma è esattamente n. In particolare, distinguiamo tra partizioni con ordine e senza ordine. Ad esempio e partizioni con ordine di $n = 4$ sono:*

$$1,1,1,1 \qquad 2,1,1 \qquad 1,2,1 \qquad 1,1,2 \qquad 2,2 \qquad 3,1 \qquad 1,3 \qquad 4$$

mentre quelle senza ordine di $n = 4$ sono:

$$1,1,1,1 \qquad 2,1,1 \qquad 2,2 \qquad 3,1 \qquad 4$$

In generale, indichiamo con $P(n)$ il numero di partizioni di n senza ordine.

Progettare un algoritmo di complessità $O(nP(n))$ che stampi tutte le partizioni di n senza ordine.

(Soluzione)

5.2 Soluzioni

Soluzione 5.1 (Stampa parentesi valide)**.** *(Traccia)*

Notiamo che per avere delle coppie ben formattate bisogna far si che il numero di parentesi aperte deve essere uguale al numero di parentesi chiuse in modo da poter associare una parentesi chiusa ad ogni parentesi aperta.

Osserviamo ora che n sia il numero massimo di parentesi aperte, mentre le parentesi chiuse le possiamo mettere solo se ci sono delle parentesi aperte che non sono assiociate a nulla. Quindi, ogni volta possiamo fare 2 decisioni

```
function PRINT_VALID_PARENTHESIS(n, opened = 0, Sol = [])
    if n == 0 then
        output = "".join(Sol) + ')' · opened
        print(output)
        return
    end if

    Sol.append('(')
    print_valid_parenthesis(n − 1, opened + 1, Sol)
    Sol.pop()

    if opened ≥ 1 then
        Sol.append(')')
        print_valid_parenthesis(n, opened − 1, Sol)
        Sol.pop()
    end if
    return
end function
```

Soluzione 5.2 (Stampa delle partizioni di n)**.** *(Traccia)*

Durante la generazione delle partizioni, ogni volta che inseriamo un numero possiamo inserire o un numero maggiore o il numero più grande attualmente che nel caso della lista vuota sarebbe 1.

```
function PRINT_PARTITIONS(n, last = 1, Sol = [])
    if n == 0 then
        print("".join(Sol))
        return
    end if
    if n ≤ −1 or last ≥ n + 1 then
        return
    end if
    Sol.append(last)
    print_partitions(n − last, last, Sol)
    Sol.pop()
    print_partitions(n, last + 1, Sol)
    return
end function
```