Algoritmi e Strutture Dati – Soluzione 21/06/2022

Esercizio A1 – Complessità – Punti ≥ 8

È possibile osservare che $T(n) = \Omega(n^2)$, visto il termine n^2 che compare nella parte non ricorsiva dell'equazione. Verifichiamo se è anche $O(n^2)$, nel qual caso il limite è stretto e abbiamo terminato.

- Caso base:
  - $T(1) = 1 \leq c \cdot 1^2 \Rightarrow c \geq 1$
  - $T(2) = 1 \leq c \cdot 2^2 \Rightarrow c \geq 1/4$
  - $T(3) = T\left(\left\lfloor \frac{3}{\sqrt{2}} \right\rfloor\right) + T\left(\left\lfloor \frac{3}{\sqrt{4}} \right\rfloor\right) + T\left(\left\lfloor \frac{3}{\sqrt{5}} \right\rfloor\right) + 3^2 = T(2) + T(1) + T(1) + 9 = 12$

  In entrambi i casi, il limite è soddisfatto da $c \geq 1$; non è necessario andare oltre $n = 2$, in quanto già per $n = 3$ la ricorsione è definita sui casi $T(1)$ e $T(2)$, la cui correttezza è stata dimostrata.
- Ipotesi induttiva: $\forall k < n : T(k) \leq ck^2$
- Passo induttivo:

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{\sqrt{2}} \right\rfloor\right) + T\left(\left\lfloor \frac{n}{\sqrt{4}} \right\rfloor\right) + T\left(\left\lfloor \frac{n}{\sqrt{5}} \right\rfloor\right) + n^2 \\ &\leq T\left(\frac{n}{\sqrt{2}}\right) + T\left(\frac{n}{\sqrt{4}}\right) + T\left(\frac{n}{\sqrt{5}}\right) + n^2 \\ &\leq cn^2/2 + cn^2/4 + cn^2/5 + n^2 \\ &= \frac{19}{20}cn^2 + n^2 \leq cn^2 \end{aligned}$$

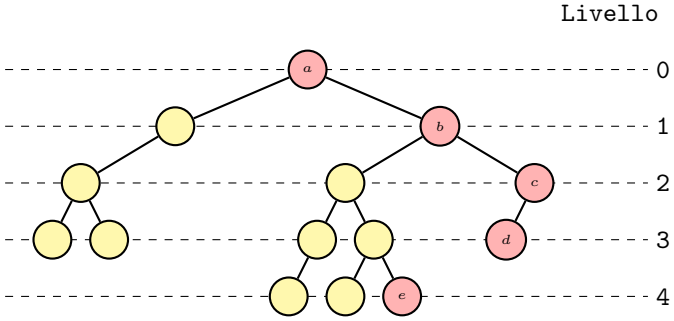
L'ultima disequazione è vera per se $19/20c + 1 \leq c$, ovvero se $c \geq 20$. Abbiamo quindi trovato due valori $c = 20$ e $m = 1$ per cui il limite asintotico superiore è soddisfatto. L'equazione di ricorrenza ha quindi complessità $\Theta(n^2)$.

Esercizio A2 – Lato destro – Punti ≥ 10

La soluzione proposta si basa sulla soluzione del problema larghezza() descritto nel libro e risolto negli esercizi dedicati agli alberi. Quando l'ultimo dei nodi del livello k viene estratto, la coda contiene tutti e soli i nodi del livello $k + 1$. Quindi, l'ultimo nodo inserito in coda al momento del cambio di livello deve essere stampato.
Essendo una semplice vista in ampiezza, il costo computazionale è pari a $\Theta(n)$.

```
rightView(Tree T)
int count = 1                          % # of nodes to be visited of current level; at the beginning, just the root
Queue Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    Tree t = Q.dequeue()
    if t.left ≠ nil then
        Q.enqueue(t.left)
    if t.right ≠ nil then
        Q.enqueue(t.right)
    count = count − 1
    if count == 0 then                 % Last node of the old level; we print it
        print t
        count = Q.size()               % All the nodes in the queue belong to the next level
```

Molte soluzioni proposte non erano corrette perché assumevano che il nodo più a destra fosse un figlio destro; ma questo non è sempre vero, come dimostrato da questo esempio, dove d è sul lato destro ma non è un figlio sinistro:



Molte soluzioni erano basate su visite in profondità. A parte proposte basate su un vettore di di dimensione pari all'altezza (calcolata a parte) che registravano il nodo nella posizione del vettore pari al suo livello, molte soluzioni fallivano perché non erano in grado di comprendere il concetto di livello, solo quello di profondità.

Esercizio A3 – Mele marce – Punti ≥ 12

È possibile interpretare la matrice come un grafo, dove ogni posizione rappresenta un nodo e ogni nodo è connesso con le posizioni adiacenti. La soluzione proposta è basata su una visita in ampiezza che parte non da un singolo nodo, ma da tutti i nodi contenenti mele marce. Il problema è simile al problema "Grafi – Stessa distanza" visto durante le esercitazioni (Esercizio 3 del 6/6/11), all'Esercizio A3 del 25/7/2018 e all'Esercizio A3 del 7/2/2019.
L'idea è assegnare una distanza pari a zero a tutti i nodi che hanno una mela marcia, e calcolare la distanza minima (in numero di archi) a partire da uno qualunque di tali nodi. Per farlo, si utilizza un algoritmo basato su BFS, in cui tutte le mele marce iniziali sono inserite nella coda iniziale. Tutte le volte che si estrae un nodo, si controlla se fra i suoi vicini esiste una mela buona ($M[i][j] == 1$) non ancora visitata ($dist[i][j] < 0$) tramite la funzione addNode(); in caso positivo, si aggiunge il nodo alla coda e si aggiorna la distanza del nodo, calcolata come la sua distanza del nodo precedente più 1.

```
int badApple(int[][] M, int n)
    int[][] dist = new int[1 ≤ n][1 ≤ n] = {−1}
    QUEUE Q = Queue()
    for i = 1 to n do
        for j = 1 to n do
            if M[i][j] == −1 then
                dist[i][j] = 0
                Q.enqueue(⟨i, j⟩)
    while not Q.isEmpty() do
        i, j = Q.dequeue()
        addNode(Q, M, dist, i + 1, j, n, dist[i][j])
        addNode(Q, M, dist, i, j + 1, n, dist[i][j])
        addNode(Q, M, dist, i − 1, j, n, dist[i][j])
        addNode(Q, M, dist, i, j − 1, n, dist[i][j])
    return max(dist)
```

```
int addNode(QUEUE Q, int[][] M, int[][] dist,int i, int j, int n, int d)
    if 1 ≤ i ≤ n and 1 ≤ j ≤ n and M[i][j] == 1 and dist[i][j] < 0 then
        Q.enqueue(⟨i, j⟩)
        dist[i][j] = d + 1
```

La complessità è quella di una visita in ampiezza; essendoci n^2 nodi e $4n^2 - 4n$ archi, il costo computazionale è $\Theta(n^2)$.

Soluzioni errate Alcune soluzioni partivano dall'idea di fare un certo numero di passate (fino a $2n - 2$), durante le quali si scorre l'intera matrice e per ogni mela marcia trovata, si "infettano" i suoi vicini. Quando non ci sono più nodi da infettare, il processo termina. Il numero di passate misura il numero di giorni trascorsi. Un algoritmo del genere ha costo $O(n^3)$. Gran parte delle soluzioni, tuttavia, usavano un'unica matrice, facendo sì una mela marcita durante la passata k può far marcire un'altra mela nella stessa passata.

```
int badApple(int[][] M, int n)
    boolean infected = true
    int count = 0
    while infected do
        infected = false
        for i = 1 to n do
            for j = 1 to n do
                if M[i][j] == −1 then
                    infected = infected or infect(M, i − 1, j, n)
                    infected = infected or infect(M, i + 1, j, n)
                    infected = infected or infect(M, i, j − 1, n)
                    infected = infected or infect(M, i, j + 1, n)
        count = count + 1
    return count − 1
```

```
int infect(int[][] M, int i, int j, int n)
  if 1 ≤ i ≤ n and 1 ≤ j ≤ n then
    if M[i][j] == −1 then
      M[i][j] = −1
      return true
    else
      return false
```

Si consideri una matrice composta da tutti 1, tranne il nodo in alto a sinistra $(1, 1)$:

-1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Nell'esecuzione dei cicli **for** annidati, quando $i = 1$ e $j = 1$ il nodo (1,1) infetta i nodi (2,1) e (1,2); quando $i = 1$ e $j = 2$ il nodo (1,2) infetta i nodi (2,2) e (1,3); e così via, riempiendo tutte le righe e tutte le colonne. Al completamento dei cicli **for** annidati, tutti i nodi vengono infettati e l'algoritmo restituisce 1.

Molti studenti hanno invece scelto la strada di una visita in profondità. Questo approccio è sbagliato, perché una visita in profondità non calcola la lunghezza del percorso minimo fra due nodi, ma potenzialmente può identificare percorsi molto lunghi.
Nella matrice seguente, per esempio, partendo dal nodo in posizione (1,1), una visita in profondità potrebbe percorrere l'intero bordo prima scendendo, poi andando a destra, poi salendo, poi andando a sinistra. La lunghezza di tale percorso è $4n - 4$, mentre la distanza fra l'angolo alto-sinistra e l'angolo basso-destra è pari a $2n - 2$. L'algoritmo restituirebbe quindi un valore sbagliato.

-1	1	1	1	1
1	0	0	0	1
1	0	0	0	1
1	0	0	0	1
1	1	1	1	1

Esercizio B1 – Math ways – Punti ≥ 8

Il problema può essere risolto tramite backtracking. Si crea un vettore S contenente i valori della sequenza, inizializzando il primo elemento a k. Dopo di che, si chiamerà la procedura ricorsiva che prova ad aggiungere gli elementi successivi, utilizzando una delle tre operazioni previste. Se il valore precedente nella sequenza è maggiore di n, si farà backtrack senza stampare nulla; se si è raggiunto n, viene stampata la sequenza e si termina; altrimenti si fa un ulteriore passo di backtrack.
Una stima molto grezza del costo computazionale è $O(n \cdot 3^n)$, perché ad ogni chiamata ricorsiva è possibile fare tre diverse scelte e il costo di stampa è $O(n)$. Tuttavia, il numero di sequenze è molto più basso, grazie al fatto che la moltiplicazione e l'elevamento al quadrato fanno crescere velocemente i valori.

```
mathways(int n, int k)
  int[] S = new int[1 . . . n]
  mwRec(S, 1, k, n)
```

```
mwRec(int[] S, int i, int n)
  S[i] = k
  if S[i] == n then
    print S[1 . . . i]
  else if S[i] < n then
    mwRec(S, i + 1, S[i] + 1, n)
    mwRec(S, i + 1, S[i] · 2, n)
    mwRec(S, i + 1, (S[i])^2, n)
```

Si noti che il valore iniziale $k = 3$ è stato scelto per evitare il caso particolare in cui $k = 2$ dà origine allo stesso valore quando si calcola $2k$ e k^2. L'avevo inserito nel compito di Febbraio 22, l'avevo tolto nel compito di Giugno 22 perché mi ero completamente dimenticato, ma fortunatamente Cristian se ne è accorto.

Esercizio B1 – Math ways – Punti ≥ 8

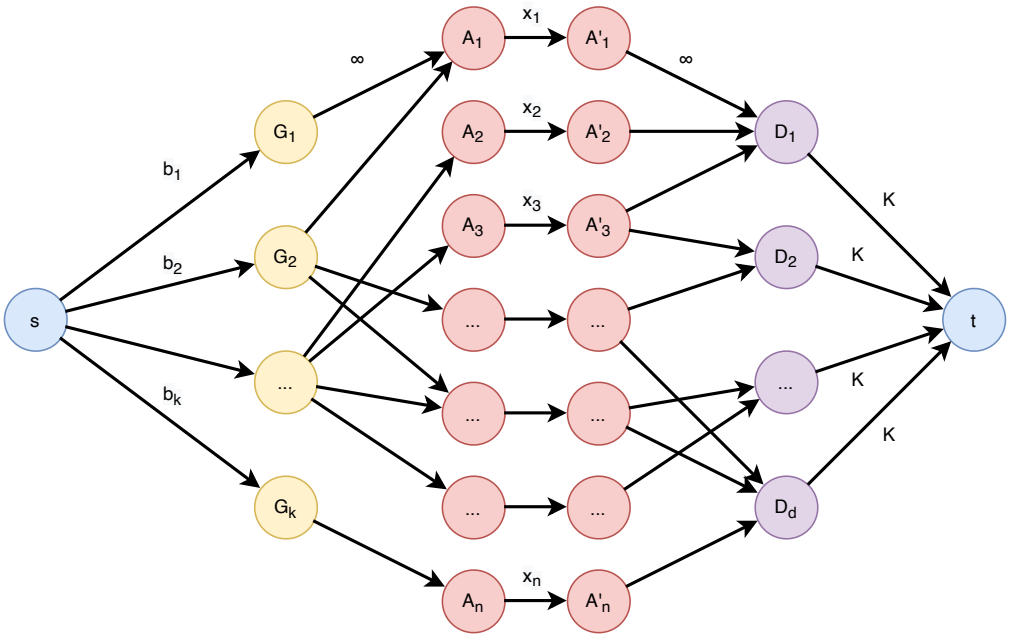
Il problema può essere risolto tramite una rete di flusso.
Costruiamo una rete $G = (V, E, s, t, c)$. L'insieme V dei nodi è costituito nel modo seguente:

- un nodo s (sorgente);
- un nodo t (pozzo);
- un nodo G_j per ognuno dei gruppi;
- una coppia di nodi A_i e A'_i per ognuna delle azioni;
- un nodo D_l per ognuno dei delegati

L'insieme degli archi è costituito nel modo seguente:

- un arco (s, G_j) per ogni gruppo g_j, con capacità b_j (il budget del gruppo); in questo modo ogni gruppo riceverà al più b_j budget;
- un arco (G_j, A_i) per ogni gruppo $j \in grp_i$, che associa un'azione a un gruppo; possiamo utilizzare $+\infty$ per la capacità, oppure b_j; in entrambi i casi, il valore sarà limitato da b_j;
- un arco (A_i, A'_i) per ogni azione, con capacità x_i; questo serve ad assicurarsi che l'azione riceverà esattamente un budget x_i;
- un arco (A'_i, D_l) per ogni delegato $l \in del_i$, che associa un'azione a un delegato; possiamo utilizzare $+\infty$ per la capacità, oppure K; in entrambi i casi, il valore sarà limitato da K;
- infine, un arco (D_l, t) da ogni delegato al pozzo, con capacità K.

Il grafo risultante è il seguente:



Eseguendo uno degli algoritmi visti a lezione su questa rete, si otterrà una divisione che rispetta tutti i vincoli se e solo se ognuno degli archi (A_i, A'_i) avrà esattamente valore x_i.
Per il calcolo della complessità, è possibile osservare che:

$$|V| = 2 + k + 2n + d = O(k + n + d)$$
$$[E] = k + O(kn) + O(dn) + d = O((k + d) \cdot n)$$

Il valore del flusso massimo atteso è pari a $X = \sum_{i=1}^{n} x_i$
Utilizzando il limite di Ford-Fulkerson, si ottiene $O((k + d) \cdot n \cdot X)$.

Esercizio B3 – Double sequence – Punti ≥ 12

Esistono diversi metodi per risolvere il problema tramite programmazione dinamica.
Sia $DP[i][j]$ il numero di sequenze lunghe i, composte da numeri tutti minori di j e che rispettano i vincoli sul raddoppio. Questo valore può essere calcolato nel modo seguente:

$$DP[i][j] = \begin{cases} 0 & j = 0 \\ j & j > 0 \wedge i = 1 \\ DP[i-1][\lfloor j/2 \rfloor] + DP[i][j-1] & j > 0 \wedge i > 1 \end{cases}$$

- Se $j = 0$, non è possibile comporre sequenze di interi positivi.
- Se $j > 0$ e $i = 1$, è possibile comporre una sequenza di un elemento composto da tutti i valori $1, 2, \ldots, j$ (che sono j).
- Altrimenti, ci sono due possibilità:
  - si termina la sequenza con il valore j e quindi si considerano tutte le sequenze di lunghezza $i - 1$ e valore massimo $\lfloor j/2 \rfloor$;
  - si scarta il valore j e si considerano tutte le sequenze di lunghezza i con valore massimo $j - 1$.

```
int countSequences(int n, int m)
    int[][] DP = new int[1...n][1...m] = {−1}
    return csRec(DP, n, m)
```

```
int csRec(int[][] DP, int i, int j)
    if j == 0 then
        return 0
    else if i == 1 then
        return j
    else
        if DP[i][j] < 0 then
            DP[i][j] = csRec(DP, i − 1, ⌊j/2⌋) + csRec(DP, i, j − 1)
        return DP[i][j]
```

La complessità è pari a $O(mn)$, il costo per inizializzare e riempire la tabella.

Una soluzione alternativa, più costosa, è la seguente. Sia $DP[i][j]$ il numero di sequenze composte da i elementi, che terminano esattamente con il valore j e che rispettano i vincoli sul raddoppio. $DP[n][m]$ può essere calcolato nel modo seguente:

$$DP[i][j] = \begin{cases} 0 & j = 0 \\ 1 & j > 0 \wedge i = 1 \\ \sum_{k=1}^{\lfloor j/2 \rfloor} DP[i-1][k] & j > 0 \wedge i > 1 \end{cases}$$

- Se $j = 0$, non è possibile comporre sequenze di interi positivi.
- Se $j > 0$ e $i = 1$, è possibile comporre una sequenza di un elemento composto dal solo valore j.
- Altrimenti, si termina la sequenza con il valore m e si sommano insieme tutte le sequenze di $i - 1$ elementi che terminano con valori compresi fra 1 e $\lfloor m/2 \rfloor$, che quindi rispettano la regola del raddoppio.

Una volta ottenuta la matrice DP, il risultato finale è data dalla somma dell'n-esima riga della matrice, ovvero la somma di tutte le sequenze che terminano con un valore compreso fra 1 e m:

$$\sum_{k=1}^{m} DP[n][k]$$

Una piccola ottimizzazione consiste nel notare che il valore iniziale della sommatoria può essere limitato superiormente da $\left\lfloor \frac{m}{2^{n-1}} \right\rfloor$, in quanto valori più grandi daranno origine a sequenze che contengono valori superiori a m (per via degli $n - 1$ raddoppi).
Il codice per risolvere il problema è il seguente:

```
int countSequences(int n, int m)
    int[][] DP = new int[1...n][1...m] = {−1}
    int ret = 0
    for k = 1 to ⌊m / 2^(n−1)⌋ do
        ret = ret + csRec(DP, n, k)
    return ret
```

```
int csRec(int[][] DP, int n, int m)
    if m == 0 then
        return 0
    else if n == 1 then
        return 1
    else
        if DP[n][m] < 0 then
            DP[n][m] = 0
            for i = 1 to ⌊m/2⌋ do
                DP[n][m] = DP[n][m] + csRec(DP, n − 1, i)
        return DP[n][m]
```

La complessità è pari a $O(nm^2)$, il costo per inizializzare e riempire la tabella, tenendo conto che riempire ogni casella costa $O(m)$.