Algoritmi e Strutture Dati - 21/08/2018

Esercizio A1

Andando per tentativi, proviamo con $\Theta(n^2)$. E' facile vedere che la ricorrenza è $\Omega(n^2)$, per via della sua componente non ricorsiva. Proviamo quindi a dimostrare che $T(n) = O(n^2)$.

- Caso base: $T(n) = 1 \leq cn^2$, per tutti i valori di n compresi fra 1 e 6, ovvero:

$$c \geq 1, c \geq \frac{1}{4}, c \geq \frac{1}{9}, c \geq \frac{1}{16}, c \geq \frac{1}{25}, c \geq \frac{1}{36}$$

  Tutte queste disequazioni sono soddisfatte da $c \geq 1$.

- Ipotesi induttiva: $T(k) \leq ck^2$, per $k < n$

- Passo induttivo:

$$\begin{aligned} T(n) &= 3T(\lfloor n/3 \rfloor) + 4T(\lfloor n/4 \rfloor) + 12T(\lfloor n/6 \rfloor) + n^2 \\ &\leq 3c\lfloor n/3 \rfloor^2 + 4c\lfloor n/4 \rfloor^2 + 12\lfloor n/6 \rfloor^2 + n^2 \\ &\leq 3cn^2/9 + 4cn^2/16 + 12cn^2/36 + n^2 \\ &\leq 11/12cn^2 + n^2 \leq cn^2 \end{aligned}$$

  L'ultima disequazione è rispettata per $c \geq 12$.

Abbiamo quindi dimostrato che $T(n) = \Theta(n^2)$, con $m = 1$ e $c \geq 12$.

Esercizio A2

L'esercizio è simile all'esercizio del compito scorso, che avevamo risolto interpretando la matrice come un grafo e facendo una vista BFS visto che erano coinvolte le distanze. In questo caso, risolviamo il problema tramite una DFS - ne risulterà una soluzione più compatta.
Utilizziamo la matrice M di input come vettore *visited*; in altre parole, una cella è visitabile se il suo valore è maggiore di zero. Una volta scoperta, il valore viene posto a zero. Se la matrice deve essere conservata, sarà necessario farne una copia prima (con costo $O(n^2)$).
L'algoritmo scritto qui simula una ricerca delle componenti connesse nel grafo. Si parte da ogni cella non visitata, e si lancia una visita DFS. Per ogni nodo visitato, si aggiunge l'altezza ad un accumulatore di altezze $height$, si aggiunge 1 ad un contatore $count$ e si marca il nodo come visitato. Si continua quindi la visita affrontando le quattro caselle vicine. Al termine della visita ricorsiva, si calcola l'altezza media e la si confronta con il massimo.
Visto che ogni cella può essere visita al più una volta, la complessità è $O(n^2)$.

```
int topIsland(int[][] M, int n)
    float max = 0
    for r = 1 to n do
        for c = 1 to n do
            if M[r][c] > 0 then
                int height = 0
                int count = 0
                dfsRec(M, n, r, c, &height, &count)
                max = max(max, height/count)
    return max
```

```
dfsRec(int[][] M, int n, int r, int c, int count, int height)
    if 1 ≤ r < n and 1 ≤ c < n and M[r][c] > 0 then
        height = height + M[r][c]
        count = count + 1
        M[r][c] = 0                                  % Visited
        dfsRec(M, n, r − 1, c)
        dfsRec(M, n, r + 1, c)
        dfsRec(M, n, r, c − 1)
        dfsRec(M, n, r, c + 1)
```

Esercizio A3

Il problema può essere risolto modificando opportunamente la ricerca dicotomica, in cui cerchiamo di inviduare la posizione k del minimo assumendo che il vettore sia stato traslato di un fattore k.
L'idea è la seguente: si considera un sottovettore compreso fra gli indici i e j, e si considera l'elemento centrale $m = \lfloor (i+j)/2 \rfloor$.

- Se $A[i] > A[m]$, allora il minimo si troverà fra i ed m, in quanto il sottovettore di sinistra non è ordinato e quindi il minimo si troverà in quel sottovettore; si noti che necessariamente si ha che $A[m] < A[j]$, in quanto se fosse $A[m] > A[j]$ il vettore sarebbe ordinato in senso opposto.
- Se $A[m] > A[j]$, allora il minimo si troverà fra m e j, in quanto il sottovettore di destra non è ordinato e quindi il minimo si troverà in quel sottovettore; si noti che necessariamente si ha che $A[i] < A[m]$, in quanto se fosse $A[i] > A[m]$ il vettore sarebbe ordinato in senso opposto.
- Se entrambe le condizioni non sono vere, allora $A[i] < A[m] < A[j]$, e il vettore è ordinato. In questo caso bisogna restituire 0.

Accorgersi della terza condizione è difficile, anche per l'ambiguità del testo - non viene specificato se k può essere uguale a zero oppure no. Per questo motivo, non ho valutato negativamente soluzioni che non gestiscono questo caso particolare.
Come caso base, quando si è rimasti con due elementi, quindi $j = i + 1$, allora il minimo si trova in j (se l'array fosse ordinato, avremmo già restituito zero).
Una dimostrazione più precisa dovrebbe essere basata su induzione sulla dimensione del sottovettore, con caso base 2.
Il codice, molto breve, è il seguente:

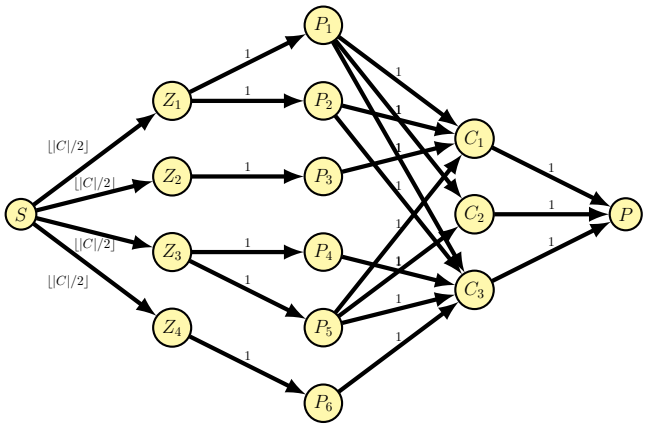
```
shift(int[] A, int i, int j)
    if i == j + 1 then
        return j
    int m = ⌊(i + j)/2⌋
    if A[i] > A[m] then
        return shift(A, i, m)
    if A[m] > A[j] then
        return shift(A, m, j)
    return 0
```

La complessità è ovviamente $O(\log n)$, derivando dalla ricerca dicotomica.

Esercizio B1

Il problema si risolve tramite una rete di flusso:

- Aggiungiamo una sorgente e un pozzo
- Aggiungiamo un nodo per ogni partito in Z
- Aggiungiamo un nodo per ogni persona in P
- Aggiungiamo un nodo per ogni club in C
- Colleghiamo la sorgente ad ogni partito, con peso $\lfloor C/2 \rfloor$, per evitare che un singolo partito abbia la maggioranza nel consiglio dei club
- Colleghiamo ogni partito ai suoi membri, con peso 1; si noti che ogni persona ha al più un collegamento con il partito, quindi riceverà al massimo un valore 1;
- Colleghiamo ogni persona con i club di cui fa parte, con peso 1;
- Colleghiamo ogni club con il pozzo, con peso 1 (per indicare che ogni club avrà al più un rappresentante)



In questo modo, ogni partito potrà avere al massimo un valore $\lfloor C/2 \rfloor$ in uscita; ogni persona al massimo un valore 1; ogni club al massimo un valore 1, come richiesto dal problema.
E' possibile costituire il consiglio se $|f^*| = |C|$, ovvero se ogni club è rappresentato; le associazioni persona-club con flusso maggiore di uno rappresentano gli eletti.
La dimensione del problema è

$$|V| = 2 + |Z| + |P| + |C|$$
$$|E| = |Z| + |P| + |C| + O(|P| \cdot |C|)$$

La complessità è quindi $O(|f^*|(|V| + |E|))$ secondo il limite di Ford-Fulkerson, quindi pari a $O(|P| \cdot |C|^2)$

Suggerimento Quando scrivo "descrivere" un algoritmo, è un buon segnale che il problema si risolve con una rete di flusso...

Esercizio B2

Il problema è una semplice variazione del problema della massima sottosequenza crescente, discussa nel blocco di esercizi relativi alla programmazione dinamica.
Denotiamo con $D[i]$ la dimensione della massima sottosequenza *più che doppia* che termina nella posizione i-esima. È possibile calcolare $D[i]$ in maniera ricorsiva. Si consideri l'indice i e si considerino gli elementi $V[j]$ tali che $V[i] > 2V[j]$ nel sottovettore $V[1 \dots i-1]$; $V[i]$ può essere utilizzato per estendere la più lunga sottosequenza che termina in uno di questi elementi. Se non esistono elementi con queste caratteristiche, allora dobbiamo "ricominciare da capo", ovvero considerare la sequenza composta dal singolo valore $V[i]$.
Un modo per esprimerlo è il seguente:

$$D[i] = \begin{cases} 1 & \forall j, 1 \leq j < i : V[i] \leq 2V[j] \\ \max_{1 \leq j \leq i-1 \wedge V[i] > 2V[j]} \{D[j]\} + 1 & \exists j, 1 \leq j < i : V[i] > 2V[j] \end{cases}$$

Traduciamo la formulazione ricorsiva nel codice seguente.

```
moreThanDouble(int[] V, int n)
    int[] DP = new int[1...n]
    for i = 1 to n do
        DP[i] = 1                          % Se non troviamo valori che rispettino la condizione, si "riparte" da questo elemento
        for j = 1 to i − 1 do
            if V[i] > 2V[j] then
                DP[i] = max(DP[i], DP[j] + 1)
    return max(DP)
```

Siccome $DP[i]$ contiene la massima sottosequenza più che doppia che *termina in* i, dovremo poi restituire il valore più alto che si trova in tale vettore.
La complessità è $O(n^2)$, derivante dal doppio ciclo dell'implementazione.

Suggerimento Alcuni studenti hanno risolto il problema in dieci minuti, individuando la soluzione negli appunti del corso e copiandola/adattandola. Questo è un esame "a libri aperti": il vantaggio è che potete portarvi tutto quello che volete, lo svantaggio è che se non vi portate niente, il compito diventa molto più difficile. Ora, vedo troppi studenti presentarsi all'esame con solo la penna. So che molti di voi studiano sui tablet e la mia non vuole essere una spinta alla deforestazione, ma vi conviene stamparvi un po' di materiale...

Esercizio B3

Sebbene sia possibile che esistano delle condizioni necessarie e/o sufficienti per dire se un certo grafo è colorabile con una n-colorazione con gap (ad esempio, un ciclo di 5 nodi è 5-colorabile, un ciclo di 4 nodi non è 4-colorabile), non è ovviamente richiesto durante un compito analizzare tali situazioni - e poi richiederebbe una dimostrazione formale delle proprie affermazioni.
Visto che il problema generale della colorazione è NP-completo e viene affrontato tramite backtrack, faremo la stessa cosa qui.
L'idea generale è questa: si visita il grafo tramite un meccanismo di backtrack, avendo a disposizione un insieme C di colori ancora non utilizzati, rispettando le regole di colorazione e restituendo **true** se si riescono ad utilizzare tutti i colori.
La complessità è ovviamente superpolinomiale; nel caso limite di un grafo completo, è necessario provare tutte le permutazioni dei nodi a partire da 1, che sono $O(n!)$; ovviamente, il controllo sulla regole dei valori consecutivi causerà il pruning di tanti casi, ma la soluzione resta comunque superpolinomiale.

```
gapColoringRec(GRAPH G, SET C, int[] color, int u)
    foreach c ∈ C do
        boolean ok = true
        foreach v ∈ G.adj(u) do
            if |color[v] − c| < 2 then
                ok = false
        if ok then
            color[u] = c
            C.remove(c)
            if C.isEmpty() then
                return true
            for v ∈ G.adj(u) do
                if gapColoringRec(G, C, color, u) then
                    return true
            C.insert(c)
    color[u] = ∞
    return false
```

```
gapColoring(GRAPH G)
    SET C = Set()
    int[] color = new int[1...G.n]
    for i = 1 to G.n do
        C.insert(i)
        S[i] = ∞
    return gapColoringRec(G, C, color, 1)
```

Suggerimento Abbiamo affrontato la colorabilità in due occasioni:

- parlando di visite in profondità e introducendo il concetto di grafo bipartito, per cui ogni grafo bipartito è bi-colorabile
- parlando di NP-completezza, per cui il problema generale non è (non pare) risolvibile in tempo polinomiale.

Affrontando un problema di colorabilità, la domanda che ci si deve porre è: il grafo è bipartito? No? Allora vai di backtrack!