Algoritmi e Strutture Dati – 21/08/2018

Esercizio A1 – Complessità – Punti ≥ 6

Andando per tentativi, proviamo con $\Theta(n^2)$. È 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 – Minecraft – Punti ≥ 12

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 count = 0
                int height = 0
                dfsRec(M, n, r, c, &count, &height)
                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, &count, &height)
    dfsRec(M, n, r + 1, c, &count, &height)
    dfsRec(M, n, r, c − 1, &count, &height)
    dfsRec(M, n, r, c + 1, &count, &height)
```

Esercizio A3 – Vettori traslati – Punti ≥ 12

Il problema può essere risolto modificando opportunamente la ricerca dicotomica, in cui cerchiamo di individuare la posizione k del minimo, assumendo che il vettore sia stato traslato di un fattore k. La funzione wrapper shift(A) chiama la funzione findMin(), che restituisce la posizione del minimo, e sottrae uno per ottenere lo spostamento k.
La funzione findMin opera nel seguente modo: si assume che il minimo sia presente nel sottovettore di input e si chiama ricorsivamente la funzione sulla metà del sottovettore che sicuramente contiene il minimo, fino ad ottenere un vettore contenente un solo elemento, che è necessariamente il minimo.
Possono darsi due casi:

- Se $A[m] > A[j]$, questo indica che il minimo è compreso tra $m+1$ e j, perché $A[m]$ non può essere il minimo essendo più grande.
- Altrimenti, il minimo si trova nella metà del vettore compresa tra i e m.

Il codice, molto breve, è il seguente:

```
int shift(int[] A, int n)
return findMin(A, 1, n) − 1
```

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

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

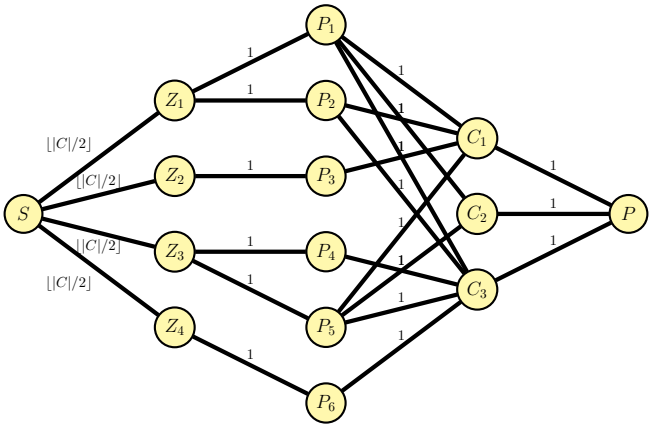
Nota Questa è una versione diversa da quella originariamente proposta, più semplice e lineare; è stata proposta nel Gennaio 2024 dallo studente "Degra" nel gruppo Telegram.

Esercizio B1 – Hateville Clubs – Punti ≥ 8

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.
È 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 – Più che doppio – Punti ≥ 10

Il problema è una semplice variazione del problema della massima sottosequenza crescente, discussa nel blocco di esercizi relativi alla programmazione dinamica.
Denotiamo con $DP[i]$ la lunghezza della massima sottosequenza in cui ogni elemento è più del doppio del precedente e che termina nella posizione i-esima. È possibile calcolare $DP[i]$ in maniera ricorsiva. Considerando l'indice i, si esaminano gli elementi $V[j]$ tali che $V[i] > 2V[j]$ nel sottovettore $V[1 \ldots i-1]$; $V[i]$ 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", considerando la sequenza composta dal singolo valore $V[i]$.
La formulazione ricorsiva può essere espressa come segue:

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

Traduciamo la formulazione ricorsiva nel codice seguente.

```
int moreThanDouble(int[] V, int n)
    int[] DP = new int[1...n]
    for i = 1 to n do
        DP[i] = 1                          % Inizializza ogni elemento con 1 come base
        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 lunghezza della massima sottosequenza più che doppia che termina in i, restituiamo il valore più alto trovato in DP.

Esercizio B3 – n-colorazione con gap – Punti ≥ 12

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.

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!

```
boolean 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] > 0 and |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 color[v] < 0 and gapColoringRec(G, C, color, v) then
          return true
      C.insert(c)
      color[u] = −1
  return false
```

```
boolean gapColoring(Graph G)
  Set C = Set()
  int[] color = new int[1 . . . G.n]
  for i = 1 to G.n do
    C.insert(i)
    color[i] = −1
  return gapColoringRec(G, C, color, 1)
```