Algoritmi e Strutture Dati - 19/07/2019

Esercizio A1

Andando per tentativi, proviamo con $\Theta(n^2)$. Proviamo quindi a dimostrare che $T(n) = O(n^2)$.

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

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + 2T(\lfloor n/4 \rfloor) + 3T(\lfloor n/6 \rfloor) + n^2 \\ &\leq c\lfloor n/2 \rfloor^2 + 2c\lfloor n/4 \rfloor^2 + 3c\lfloor n/6 \rfloor^2 + n^2 \\ &\leq c\frac{n^2}{4} + 2c\frac{n^2}{16} + 3c\frac{n^2}{36} + n^2 \\ &= cn^2\left(\frac{1}{4} + \frac{1}{8} + \frac{1}{12}\right) + n^2 \\ &= \frac{11}{24}cn^2 + n^2 \\ &\leq cn^2 \end{aligned}$$

  L'ultima disequazione è vera per $c \geq \frac{24}{13}$.

- Casi base:

$$\begin{aligned} T(1) = 1 \leq c \cdot 1^2 &\Rightarrow c \geq 1 \\ T(2) = 1 \leq c \cdot 2^2 &\Rightarrow c \geq 1/4 \\ T(3) = 1 \leq c \cdot 3^2 &\Rightarrow c \geq 1/9 \\ T(4) = 1 \leq c \cdot 4^2 &\Rightarrow c \geq 1/16 \\ T(5) = 1 \leq c \cdot 5^2 &\Rightarrow c \geq 1/25 \end{aligned}$$

Tutti questi casi sono soddisfatti da $c \geq 1$, e ovviamente da $c \geq \frac{24}{13}$.
Abbiamo quindi dimostrato che $T(n) = O(n^2)$, con $m = 1$ e $c \geq 24/13$.
È facile dimostrare che $T(n) = \Omega(n^2)$ per via della componente non ricorsiva n^2.
Abbiamo quindi dimostrato che $T(n) = \Theta(n^2)$.

Esercizio A2

Per questo esercizio esistono varie soluzioni di complessità diversa. Il suo scopo non è trovare chissà che soluzione, ma cercare di capire se gli studenti hanno compreso come gestire la complessità delle soluzioni. Dall'esito del compito, molti dei partecipanti hanno qualche difficoltà in questo senso.
Innanzitutto, dovreste essere più strategici. Il compito afferma che vengono accettate soluzioni $O(n^4)$, ma con punteggio 30% (ovvero 3 punti). La prima cosa da fare è assicurarsi 3 punti:

```
boolean four(int[] A, int n, int k)
for i1 = 1 to n do
    for i2 = i1 to n do
        for i3 = i2 to n do
            for i4 = i3 to n do
                if A[i1] + A[i2] + A[i3] + A[i4] == k then
                    return true
return false
```

Questa banalissima soluzione ha come complessità $O(n^4)$ e prende 30%, per l'appunto.
Ho visto altre soluzioni basate su un approccio basato su programmazione dinamica, in cui viene calcolato il valore $DP[i][j][\ell]$, che contiene **true** se è possibile ottenere il valore j come somma di ℓ valori presi dai primi i valori del vettore, **false** altrimenti. Il vettore DP può essere calcolato nel modo seguente:

$$DP[i][j][\ell] = \begin{cases} \textbf{true} & j = 0 \textbf{ and } \ell = 0 \\ DP[i-1][j][\ell] \textbf{ or } DP[i-1][j - A[i]][\ell - 1] & i > 0 \textbf{ and } j > 0 \textbf{ and } \ell > 0 \\ \textbf{false} & \text{altrimenti} \end{cases}$$

Il valore $DP[n][k][4]$ contiene il valore da restituire.
Se viene correttamente scritta una soluzione basata su programmazione dinamica o memoization, la complessità risultante è $O(nk)$, pseudopolinomiale, che quindi non è $O(n^4)$. Notate che k può essere molto più grande di n^3, quindi questa soluzione è potenzialmente più lenta di quella vista sopra.
Alcuni poi hanno scritto soluzioni basata sulla formula ricorsiva di cui sopra, ma senza utilizzare programmazione dinamica o memoization, ottenendo una complessità pari a $O(2^n)$. In coerenza con l'affermazione che soluzioni $O(n^4)$ o superiore prendono il 30%, ho assegnato 30% anche a queste; ma non lo meriterebbero.
Notate inoltre che in un compito sulla prima parte non richiede conoscenze sulla programmazione dinamica; qui l'utilizzo della programmazione dinamica è dannoso.
Per iniziare ad attaccare il limite di $O(n^4)$, è sufficiente utilizzare una ricerca dicotomica molto semplice. Si ordina il vettore (costo $O(n \log n)$), si cicla su tre indici i_1, i_2, i_3 e si cerca sul vettore il valore $k - A[i_1] - A[i_2] - A[i_3]$ tramite ricerca dicotomica. L'algoritmo risultante ha costo $O(n^3 \log n)$. Alcuni studenti hanno proposto questa soluzione e hanno preso 90%.

```
boolean four(int[] A, int n, int k)
    sort(A)
    for i1 = 1 to n do
        for i2 = i1 to n do
            for i3 = i2 to n do
                if binarySearch(A, 1, n, k − A[i1] − A[i2] − A[i3]) then
                    return true
    return false
```

Sfruttando l'ampia disponibilità di memoria, alcuni studenti hanno utilizzato invece un dizionario basato su tabella hash e hanno memorizzato tutti i valori presenti in tale dizionario. Invece di utilizzare la ricerca dicotomica in tempo $O(\log n)$, hanno fatto la ricerca in tempo $O(1)$ nella tabella, riducendo il costo a $O(n^3)$. Questa soluzione ha preso 100%, essendo stata la migliore proposta.
Riporto questa soluzione qui; invece di un dizionario, propongo l'utilizzo diun insieme implementato tramite tabelle hash, con la stessa complessità.

```
boolean four(int[] A, int n, int k)
    SET S = Set()
    for i = 1 to n do
        S.insert(A[i])
    for i1 = 1 to n do
        for i2 = i1 to n do
            for i3 = i2 to n do
                if S.contains(k − A[i1] − A[i2] − A[i3]) then
                    return true
    return false
```

Ora, una soluzione che non avevo considerato è quella di inserire nell'insieme i valori ottenuti sommando due elementi del vettore A. A

questo punto,si cicla su due indici i_1, i_2 e si cerca sul vettore il valore $k - A[i_1] - A[i_2]$ tramite l'insieme, in tempo $O(1)$.

```
boolean four(int[] A, int n, int k)
    SET S = Set()
    for i1 = 1 to n do
        for i2 = i1 to n do
            S.insert(A[i1] + A[i2])
    for i1 = 1 to n do
        for i2 = i1 to n do
            if S.contains(k - A[i1] - A[i2]) then
                return true
    return false
```

Ora, questa soluzione ha costo $O(n^2)$. Bisogna fare attenzione però: potrebbe essere difficile inserire $10.000 \cdot (10.000 - 1)/2 \approx 50$ milioni di valori derivanti da coppie in un 1GB di memoria. Ad esempio, un **set** in Python con 50 milioni di interi richiede > 2.1GB
Uno studente, preoccupato per il consumo di memoria, ha considerato questa possibilità per poi scartarla, prendendo comunque 100% perché ha proposto la soluzione $O(n^3)$.
Un'ulteriore soluzione è quella di costruire un vettore di appoggio B di dimensione $n(n+1)/2 = O(n^2)$, contenente i valori ottenuti sommando due elementi del vettore A.
Si ordina il vettore con una complessità $O(n^2 \log n^2) = O(n^2 \log n)$.
A questo punto si hanno due possibilità:

- Per ogni valore $B[i]$ del vettore B, si cerca tramite ricerca binaria il valore $k - B[i]$, con costo $O(n^2 \log n^2) = O(n^2 \log n)$.
- Si utilizza l'algoritmo con costo $O(n^2)$ discusso nella prima esercitazione.

La complessità finale è comunque $O(n^2 \log n)$.
La memoria è comunque sufficiente, perchè bisogna registrare al massimo $8 \cdot 10^4 * (10^4 + 1)/2$ byte, meno di 400MB. L'ordinamento può essere effettuato sul posto tramite Quicksort.

```
boolean four(int[] A, int n, int k)
    int[] B = new int[1...n(n+1)/2]
    int t = 0
    for i1 = 1 to n do
        for i2 = i1 to n do
            t = t + 1
            B[t] = A[i1] + A[i2]
    sort(B)
    for i = 1 to t do
        if binarySearch(B, 1, t, k - B[i]) then
            return true
```

Si può fare meglio di così, con minore quantità di memoria e minor complessità spaziale? Lascio la questione aperta ;-)

Esercizio A3

Come in altri esercizi, l'idea è interpretare la griglia come un grafo, dove esistono archi solo fra celle esistenti dello stesso colore; le stanze e i muri sono componenti connesse.
Si utilizza una matrice *visited* per memorizzare le celle già visitate, inizializzata a **false**.
Partendo da ogni cella (r, c), quindi, si effettuano i seguenti controlli:

- **Numero singolo**: Se (r, c) contiene un numero ed è già stata visitata, la colorazione non è corretta in quanto (r, c) fa già parte di una stanza e quindi esiste una stanza con due numeri.
- **Colorazione corretta**: Se (r, c) contiene un numero ed è colorata di nero, la colorazione non è corretta.
- **2x2**: Escludendo i bordi inferiore/destro, verifichiamo che quattro celle $(r, c), (r + 1, c), (r, c + 1), (r + 1, c + 1)$ non abbiano tutto il colore 1 (e quindi la somma dei loro colori sia diversa da quattro).

- **Dimensione stanza**: Se (r, c) non è già stata visitata, è di colore bianco ($S[r][c] = 0$) e contiene un numero nella griglia iniziale ($G[r][c] > 0$), facciamo partire una visita DFS che conta il numero di celle, e verifichiamo che sia uguale a $G[r][c]$. In caso contrario, la colorazione non è corretta.
- **Numero muri**: Se la cella non è già stata visitata e è di colore nero ($S[r][c] = 1$), inizia un nuovo muro che percorriamo interamente con una visita DFS; incrementiamo il contatore di muri, che verrà verificato solo alla fine.
- **Stanze senza numeri**: Se restano celle non visitate (con valore 0), vuole dire che esistono stanze senza numero. Si controlla quindi che la somma dei valori $visited$ sia pari a n^2.

Il tutto ha costo $O(n^2)$, perché ogni cella viene visitata al più una volta. Per rendere più comprensibile il testo, assiamo che **white**=0 e **black**=0.

```
boolean isSolution(int[][] G, int[][] S, int n)
    int[][] visited = new int[1...n][1...n] = {0}                          % Init visited to 0
    int walls = 0                                                          % Check rules
    for r = 1 to n do
        for c = 1 to n do
            if G[r][c] > 0 and visited[r][c] == 1 then                     % Check single number in room
                return false
            if G[r][c] > 0 and S[r][c] == black then                       % Check correct coloring
                return false
            if r < n and c < n and S[r][c] + S[r][c+1] + S[r+1][c] + S[r+1][c+1] == 4 then   % Check absence 2x2 walls
                return false
            if visited[r][c] == 0 and S[r][c] == white and G[r][c] > 0 then   % Check room size
                if dfsVisit(G, S, visited, r, c, 0) ≠ G[r][c] then
                    return false
            if visited[r][c] == 0 and S[r][c] == black then                % Count walls
                walls = walls + 1
                dfsVisit(G, S, visited, r, c, 1)
    return walls == 1 and sum(visited) == n^2
```

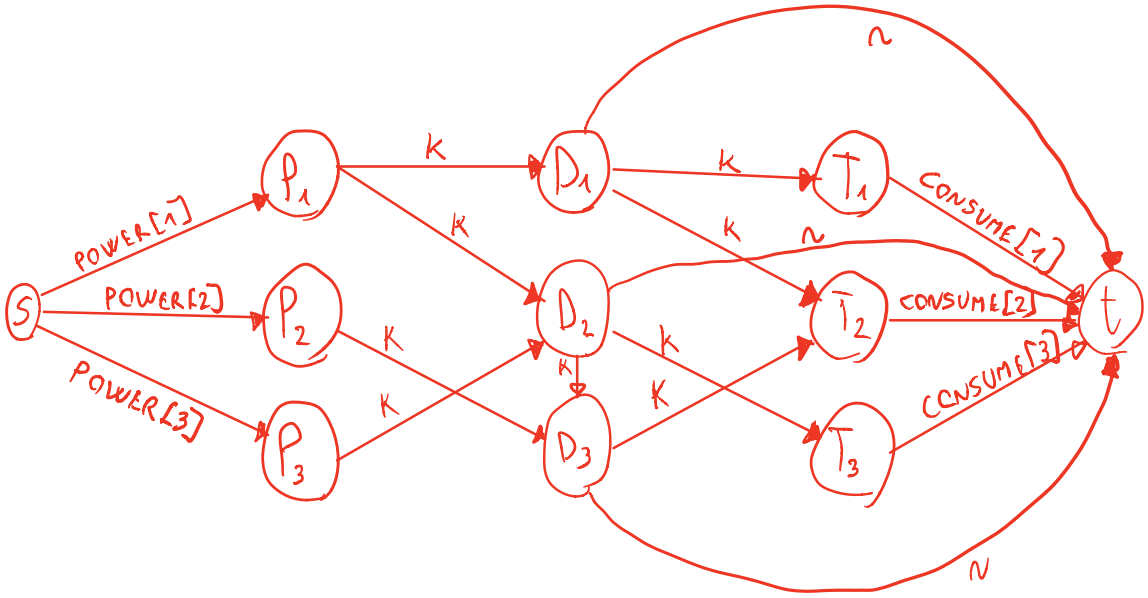
```
int dfsVisit(int[][] G, int[][] S, int[][] visited, int r, int c, int color)
    if r < 1 or c < 1 or r > n or c > n or visited[r][c] == 1 or S[r][c] ≠ color then
        return 0
    else
        visited[r][c] = 1
        return 1 + dfsVisit(G, S, visited, r + 1, c, color) + dfsVisit(G, S, visited, r − 1, c, color) +
                   dfsVisit(G, S, visited, r, c + 1, color) + dfsVisit(G, S, visited, r, c − 1, color)
```

Esercizio B1

Il problema può essere risolto trasformandolo in una rete di flusso.

- Si prende il grafo $G = (V, E)$; per ogni coppia $(u, v) \in V \times V$, si assegna la capacità $c(u, v) = k$ se $(u, v) \in E$, 0 altrimenti.
- Si aggiunge un nodo supersorgente s e un nodo superpozzo t.
- Si aggiunge un arco dalla sorgente ad ogni nodo $p \in P$, con peso $power[p]$.
- Si aggiunge un arco da ogni nodo di trasformazione $t \in T$ al superpozzo, con peso $consume[t]$.
- Sia aggiunge un arco da ogni nodo di distribuzione $d \in D$ al superpozzo, con peso r.

Si esegue Edmonds-Karp sul grafo, con complessità $O(VE^2)$. Si noti che il limite di Ford-Fulkerson non si applica qui, in quanto le capacità non sono a valori interi.
Se il flusso è pari a $\sum_{t \in T} consume[t] + r \cdot |D|$, è possibile rubare $r \cdot D$ MW senza essere scoperti.



Esercizio B2

L'esercizio può essere risolto tramite backtrack, con costo $O(n \cdot n!)$. Si generano tutte le permutazioni dei valori in S (che sono $n!$) e poi si verifica, una ad una, se sono k-alternate, contandole in maniera ricorsiva. Tale verifica costa $O(n)$, da cui il costo $O(n \cdot n!)$.

```
int kAlternate(SET A, int k)
    int n = A.size()
    int[] S = new int[1...n]
    return kAlternateRec(A, k, S, 1, n)
```

```
int kAlternateRec(SET A, int k, int[] S, int i, int n)
    if i > n then
        return iif(isKAlternate(S, n, k), 1, 0)
    else
        int tot = 0
        SET choices = copy(A)
        foreach v ∈ choices do
            S[i] = v
            A.remove(v)
            tot = tot + kAlternateRec(A, k, S, i + 1, n)
            A.insert(v)
        return tot
```

La procedura che verifica se una sequenza è k-alternata utilizza due contatori, $countInc$ e $countDec$. Il primo elemento può fare parte sia di un sequenza crescente che descrescente, e quindi inizializziamo questa variabile a 1. Dal secondo elemento in poi, quando si incontra un valore crescente, si aumenta il contatore $countInc$ e si riporta il contatore $countDec$ a 1, per essere pronti a contare gli elementi decrescenti qualora la sequenza crescente si interrompesse. Quando si incontra un valore decrescente, si effettuano le stesse azioni scambiando $countInc$ e $countDec$.

```
boolean isKAlternate(int[] S, int n, int k)
    int countInc = 1
    int countDec = 1
    for i = 2 to n do
        if S[i − 1] < S[i] then
            countInc = countInc + 1
            if countInc > k then
                return false
            countDec = 1
        countDec = countDec + 1
        if countDec > k then
            return false
        countInc = 1
    return true
```

Si può ridurre la complessità di un fattore n e ridurre il tempo di calcolo tramite pruning, interrompendo la generazione di una permutazione quando questa non è più k-alternata. Notate che non dovendola verificare al termine, non è necessario salvare le scelte degli elementi nel vettore S. Questa ottimizzazione è stata aggiunta ad una versione precedente grazie al suggerimento di uno studente, Dumitru Damaschin.

```
int kAlternate(SET A, int k)
    int tot = 0
    SET choices = A
    foreach v ∈ choices do
        A.remove(v)
        tot = tot + kAlternateRec(A, k, v, A.size() − 1, 1, 1)
        A.insert(v)
    return tot
```

```
int kAlternateRec(SET A, int k, int prev, int i, int countInc, int countDec))
    if i == 0 then
        return 1
    int tot = 0
    SET choices = copy(A)
    foreach v ∈ choices do
        A.remove(v)
        if prev < v and countInc < k then
            tot = tot + kAlternateRec(A, k, v, i − 1, countInc + 1, 1)
        if prev > v and countDec < k then
            tot = tot + kAlternateRec(A, k, v, i − 1, 1, countDec + 1)
        A.insert(v)
    return tot
```

Si noti che il primo elemento viene scelto liberamente fra gli n elementi disponibili, e per questo viene scelto fuori dalla procedura ricorsiva. Dal secondo elemento in poi, un valore viene aggiunto solo se non crea sequenze lunghe $k + 1$ o più.

Esercizio B3

Il problema può essere risolto con programmazione dinamica. Sia $DP[i]$ il valore della sottosequenza crescente massimale contenuta nei primi i elementi di A e che termina nella posizione i-esima. $DP[i]$ può essere calcolata nel modo seguente:

$$DP[i] = \begin{cases} A[i] & \forall 1 \leq j < i : A[i] < A[j] \\ \max_{1 \leq j < i : A[j] < A[i]} \{DP[j] + A[i]\} & \text{altrimenti} \end{cases}$$

Questa equazione ricorsiva può essere tradotta in codice nel modo seguente, tramite programmazione dinamica. Il valore cercato è il massimo fra tutti gli elementi della tabella DP, in quanto la sottosequenza cercata può terminare in uno qualunque delle posizioni.

```
int maxIncreasing(int[] A, int n)
    int[] DP = new int[1 . . . n]
    for i = 1 to n do
        DP[i] = A[i]
        for j = 1 to i − 1 do
            if A[j] < A[i] then
                DP[i] = max(DP[i], DP[j] + A[i])
    return max(DP, n)
```

La complessità dell'algoritmo proposto è $\Theta(n^2)$.