Algoritmi e Strutture Dati – 08/02/2021

Esercizio A1 – Punti ≥ 8

È facile vedere che la funzione è $\Omega(n^2 \log n)$, per via della parte non ricorsiva. Proviamo quindi a vedere se la funzione è anche $O(n^2 \log n)$.

- **Caso base**: per $n = 1$, si ottiene $T(1) = 1 \leq cn^2 \log n = c1^2 \log 1 = 0$, il che è ovviamente falso. Partiamo quindi da $n = 2$; dobbiamo provare tutti i casi compresi fra 2 e 7, perché solo a partire da $n = 8$ il valore $\lfloor n/4 \rfloor$ è maggiore uguale a 2 (primo caso base). Per $i = 2, \ldots, 7$, abbiamo $T(i) = 1 \leq ci^2 \log i$, il che significa che $c \geq \frac{1}{i^2 \log i}$. Tutte queste disequazioni con segno $\geq$ sono rispettate dal valore che si ottiene per $i = 2$, ovvero sia ha che $c \geq \frac{1}{2^2 \log 2} = 1/4$.
- **Ipotesi induttiva**: ipotizziamo che per ogni valore k, $2 \leq k < n$: $T(k) \leq k^2 \log k$
- **Passo induttivo**: consideriamo tutti i casi con $n \geq 8$, tali per cui $\lfloor n/2 \rfloor$ e $\lfloor n/4 \rfloor$ sono maggiori o uguali a 2 (il primo caso base); applichiamo la sostituzione e otteniamo:

$$
\begin{aligned}
T(n) &= 3T(\lfloor n/2 \rfloor) + 3T(\lfloor n/4 \rfloor) + n^2 \log n \\
&\leq 3c\lfloor n/2 \rfloor^2 \log \lfloor n/2 \rfloor + 3c\lfloor n/4 \rfloor^2 + \log \lfloor n/4 \rfloor + n^2 \log n && \text{Sostituzione} \\
&\leq 3/4\, cn^2 \log n/2 + 3/16\, cn^2 \log n/4 + n^2 \log n && \text{Eliminazione } \lfloor \rfloor \\
&\leq 12/16\, cn^2 \log n + 3/16\, cn^3 \log n + n^2 \log n && \log n/k < \log n, \text{con } k = 2, 4 \\
&= 15/16\, cn^2 \log n + n^2 \log n \overset{?}{\leq} cn^2 \log n && \text{Passaggio algebrico}
\end{aligned}
$$

L'ultima disequazione è vera per $c \geq 16$. Abbiamo quindi dimostrato che $T(n) = O(n^2 \log n)$ per $m = 2$, $c = 16$.

Esercizio A2 - Closest - Punti ≥ 10

Ovviamente, vista la richiesta di evitare algoritmi lineari, dovremo utilizzare la ricerca dicotomica e otterremo un algoritmo con costo $O(\log n)$. Il caso base è semplice: se ho un solo valore, questo è il valore da restituire. Nel caso generale, calcoliamo m come $\lfloor (i + j)/2 \rfloor$; utilizzando $\lfloor \rfloor$ e essendo $n > 1$, $m + 1$ esiste ed è incluso nel vettore. A questo punto la decisione è semplice:

- se $|A[m] - x| < |A[m + 1] - x|$, allora x è più vicino a m e si va a controllare nel sottovettore $A[i \ldots m]$, in quanto tutti i valori da $m + 1$ in poi saranno più distanti
- altrimenti si controlla nel sottovettore $A[m + 1 \ldots j]$, in quanto tutti i valori prima di $m + 1$ saranno più distanti (o avranno la stessa distanza)

Notate che è possibile aggiungere un controllo ulteriore se $|A[m] - x| == |A[m + 1] - x|$, allora x si trova esattamente a metà fra $A[m]$ e $A[m + 1]$ e possiamo restituire $A[m]$ o $A[m + 1]$, indifferentemente; oppure se $A[m] == x$, possiamo restituire direttamente $A[m]$. A mio parere, non vale la pena aggiungere queste ottimizzazioni.

```
int closest(int[] A, int n, int x)
    return closestRec(A, 1, n, x)
```

```
int closestRec(int[] A, int i, int j, int x)
    if i == j then
        return A[i]
    else
        m = ⌊(i + j)/2⌋
        else if |A[m] − x| < |A[m + 1] − x| then
            return closestRec(A, i, m, x)
        else
            return closestRec(A, m + 1, j, x)
```

Esercizio A3 – Conversione numeri – Punti ≥ 12

La soluzione parte dall'idea di considerare i valori numerici come nodi, le operazioni -1, $\cdot 2$ come archi, e realizzare la visita in ampiezza del grafo così ottenuto. In pratica, utilizziamo la distanza sul grafo (archi da attraversare) per restituire il numero minimo di operazioni per trasformare n in m.
Rispetto ad un grafo "vero", tuttavia, non conosciamo (a meno di fare un'analisi più approfondita) l'insieme dei nodi. Una prima soluzione quindi utilizza una tabella hash per memorizzare le distanze. Per il resto, la struttura ricalca quella di una visita in ampiezza, dove la tabella hash delle distanze serve anche come indicatore visitato / non visitato.

```
int minOps(int n, int m)
    Hash distance = Hash()
    distance.insert(n, 0)
    Queue queue = Queue()
    queue.enqueue(n)
    while distance.lookup(m) == nil do
        int u = queue.dequeue()
        int d = distance.lookup(u)
        foreach v ∈ {u · 2, u − 1} do
            if distance.lookup(v) == nil then
                queue.enqueue(v)
                distance.insert(v, d + 1)
    return distance.lookup(m)
```

Si noti che se $n = m$, l'algoritmo associerà la distanza 0 al valore $n = m$; quindi non entrerà mai nel ciclo **while** e restituirà 0.
Si noti che esiste sempre un cammino da n a m, quindi il ciclo **while** non controlla se la coda è vuota, ma si ferma quando il valore m è stato raggiunto.

Una soluzione del genere è corretta, tuttavia la sua complessità è elevata ed è difficile da calcolare; molti non l'hanno calcolata o l'hanno calcolata in maniera sbagliata. Prima di raggiungere il nodo m a distanza d, l'algoritmo dovrà visitare tutti i nodi a distanza inferiore a d. Poichè ad ogni estrazione del nodo vengono inseriti fino a due altri nodi, il numero di nodi da visitare cresce esponenzialmente con d.
Nel caso sia $n = m - 1$, la strategia per ottenere il percorso più breve è quella di continuare a sottrarre -1 fino ad ottenere $\lceil m/2 \rceil$, moltiplicare per 2, ed eventualmente sottrarre -1 se m è dispari. La distanza d è quindi pari a $\lceil m/2 \rceil$. Questo significa che nel caso in cui $n = m - 1$, l'algoritmo dovrà visitare $O(2^{m/2})$ nodi, un valore esponenziale.

Il problema è che tutte le volte che raddoppiamo per 2, ampliamo enormemente lo spazio di nodi da considerare. Facendo tuttavia un po' di considerazioni sulle caratteristiche di questo problema particolare, è possibile vedere che non ha senso generare valori non positivi e non ha senso considerare valori superiori a $2m$, perché una volta superato quel valore per raddoppio, possiamo solo "tornare indietro" con operazioni -1, e più ci allontaniamo da m e più operazioni -1 dovremo fare. È possibile essere ancora più precisi, notando che il caso peggiore avviene quando $n = m - 1$ e quindi il valore massimo che vale la pena considerare è $2m - 2$.
Quindi, la porzione del nostro grafo che vale la pena di esplorare avrà al massimo $2m - 2$ nodi e ogni nodo avrà al massimo 2 archi. Affinchè però il nostro algoritmo si limiti ad esplorare quella porzione, bisogna aggiungere una condizione nel codice:

```
int minConversion(int n, int m)
    Hash distance = Hash()
    distance.insert(n, 0)
    Queue queue = Queue()
    queue.enqueue(n)
    while distance.lookup(m) == nil do
        int u = queue.dequeue()
        int d = distance.lookup(u)
        foreach v ∈ {u · 2, u − 1} do
            if 1 ≤ v ≤ 2m − 2 and distance.lookup(v) == nil then
                queue.enqueue(v)
                distance.insert(v, d + 1)
    return distance.lookup(m)
```

In questo caso, il nostro algoritmo avrà complessità $\Theta(m)$, in quanto i $2m - 2$ nodi vengono visitati al più una volta.

Un ulteriore possibile miglioramento (che non cambia la complessità) rimuove la tabella hash e utilizza direttamente un vettore di dimensione $2m - 2$.

```
int minConversion(int n, int m)
    int[] distance = new int[1...2m − 2] = {−1}
    distance[n] = 0
    Queue queue = Queue()
    queue.enqueue(n)
    while distance[m] < 0 do
        int u = queue.dequeue()
        foreach v ∈ {u · 2, u − 1} do
            if 1 ≤ v ≤ 2m − 2 and distance[v] < 0 then
                queue.enqueue(v)
                distance.insert(v, distance[u] + 1)
    return distance[m]
```

Alcune soluzioni hanno esplicitamente costruito un grafo, inserendo fino a $2m$ nodi e aggiungendo gli archi verso i nodi in posizione -1, $\cdot 2$ (se esistenti). Dopo di chè, hanno usato l'algoritmo per il calcolo delle distanze che abbiamo visto a lezione. La complessità in questo caso resta $O(m)$, perché il grafo risultante ha $O(m)$ nodi e $O(m)$ archi.

Addendum Filippo Garosi e Lorenzo Marogna mi hanno fatto notare che esiste una soluzione $O(\log m)$, senza però fornire una dimostrazione. Un grosso spunto nella dimostrazione viene da Cristian Consonni.
L'idea si basa sull'osservazione che è possibile invertire il problema, ovvero i due problemi seguenti sono equivalenti: È possibile invertire il problema, ovvero i due problemi seguenti sono equivalenti:

1. fare diventare n uguale a m con le operazioni:
   (a) moltiplicare per 2, ottenendo $2n$
   (b) sottrarre 1, ottenendo $n - 1$
2. fare diventare m uguale a n con le segenti possibilità:
   (a) se m è pari dividere per 2, ottenendo $\frac{m}{2}$ (una operazione)
   (b) se m è dispari, aggiungere 1 e dividere per 2 (due operazioni)

Per dimostrare che sono equivalenti, si consideri una sequenza minima del problema 1. Si supponga esista una sequenza del problema 2 più corta; invertendo le operazioni, si ottiene una sequenza per il problema 1 che è più corta di quella minima, assurdo.
L'algoritmo proposto è il seguente, qui mostrato in versione ricorsiva

```
int minOps(int n, int m)
    if m ≤ n then
        return n − m
    else
        if m mod 2 == 0 then
            return minOps(n, m/2) + 1                    % +1 per la la moltiplicazione
        else
            return minOps(n, m + 1/2) + 2                % +2 per la la moltiplicazione + sottrazione
```

Ragioniamo per induzione su m:

- **Caso base:** se $m = 1$, n è necessariamente maggiore o uguale a 1. Il numero minimo di mosse per portare m ad n è pari a $n - m$.
- **Ipotesi induttiva:** supponiamo che l'algoritmo sia corretto per tutti i valori $m' < m$.
- **Passo induttivo:** consideriamo tre casi possibili:
  - $m \leq n$: poiché l'operazione di divisione ci allontanerebbe dall'obiettivo n, l'unica cosa che possiamo fare è sommare $+1$ un numero di volte pari a $n - m$.
  - $m > n$ e m pari: sia $S = o_1, o_2, \ldots, o_t$ la sequenza di operazioni di lunghezza minima che porta m a raggiungere il valore n. È facile vedere che o_1 deve essere una divisione per due. Se così non fosse, le prime $2k$ operazioni $S = o_1, o_2, \ldots, o_{2k}$ sarebbero operazioni di incremento e l'operazione o_{2k+1} sarebbe una divisione. Dopo queste operazioni, si ottiene il valore $(m + 2k)/2 = m/2 + k$, con $2k + 1$ operazioni. Tuttavia, potremmo sostituire questa prima parte della sequenza con una divisione per 2 e k operazioni di incremento, ottenendo ugualmente $m/2 + k$ ma con $k + 1$ operazioni, per poi proseguire con le operazioni $o_{2k+2}, o_{2k+3}, \ldots, o_t$. Questo sarebbe assurdo, perché questa sequenza sarebbe più breve della sequenza di lunghezza minima.
  - $m > n$ e m dispari: l'unica operazione possibile è un incremento, ottenendo un valore $m' = m + 1$ pari che ricade nel caso precedente.

Esercizio B1 – Somma di primi – Punti ≥ 8

Il problema è simile ad esercizi visti in passato, va solo riadattato per tenere conto dell'insieme ordinato di primi. La soluzione è ovviamente basata su backtrack.

```
primesRec(int[] P, int [] S, int p, int k, int n)
  if n==0 and k == 0 then
  |  print(S)
  else if p > 0 and k > 0 and n > 0 then
  |  primesRec(P, S, p − 1, k, n)
  |  S[k] = P[p]
  |  primesRec(P, S, p − 1, k − 1, n − P[p])
```

```
primes(int[] P, int p, int k, int n)
  int[] S = new int[1 . . . k]
  primesRec(P, S, p, k, n)
```

La procedura wrapper crea un vettore S di dimensione k, dove verranno memorizzati i valori scelti per comporre la somma. Ad ogni chiamata ricorsiva, vengono passati i tre parametri p, k, n, intesi come primi rimanenti, caselle in S rimanenti, valore da comporre.
Per l'elemento in posizione p, ci sono due possibilità - il valore viene utilizzato per la somma oppure no.

- Se non viene utilizzato, non è necessario scrivere su S e basta fare arretrare solo il valore p (l'elemento non viene più considerato), chiamando ricorsivamente primesRec().
- Se viene utilizzato, il valore viene scritto su $S[k]$ e si fa arretrare p (l'elemento non viene più considerato), k (una casella è stata utilizzata) e n (si cerca di ottenere $p - P[n]$ con i restanti primi e le restanti posizioni), chiamando ricorsivamente primesRec().

Questo è possibile solo se $p > 0$ (ci sono ancora caselle da scegliere), $k > 0$ (c'è ancora posto nel vettore S), $n > 0$ (dobbiamo ancora raggiungere la somma originale).
Quando invece $n = 0$ (abbiamo ottenuto la somma cercata) e $k = 0$ (con il numero richiesto di termini), possiamo stampare S che contiene i valori richiesti.
In tutti gli altri casi la ricorsione termina senza eseguire nulla.
La complessità è limitata superiormente da $\binom{p}{k} = O(p^k)$.

Esercizio B2 – Zaino leggero – Punti ≥ 10

Utilizziamo la programmazione dinamica. Dati i primi i valori e una capacità residua c, il numero minimo di valori $DP[i][c]$ può essere calcolato nel modo seguente:

$$DP[i][c] = \begin{cases} \infty & c < 0 \\ 0 & c = 0 \\ \infty & c > 0 \wedge i = 0 \\ \min(DP[i-1][c], DP[i-1][c - W[i]] + 1) & c > 0 \wedge i > 0 \end{cases}$$

La logica è la seguente:

- restituisco ∞ se la capacità c è minore di zero (il che significa che in precedenza ho usato un valore troppo grande);
- restituisco 0 se la capacità c è uguale a 0 (mi bastano 0 valori, caso base)
- restituisco ∞ se ho ancora resta da dare ($c > 0$), ma non ho più valori a disposizione ($i > 0$);
- restituisco il massimo fra due possibilità: non utilizzare la moneta i-esima oppure la utilizza, aumentando di 1 il numero di valori usati.

Questa formula può essere trasformata in un algoritmo tramite memoization.

```
int ssRec(int[] W, int i, int c, int[][] DP)
if c < 0 then
    return ∞
else if c == 0 then
    return 0
else if i == 0 then
    return ∞
else
    if DP[i][c] < 0 then
        DP[i][c] = min(ssRec(W, i − 1, c, DP), ssRec(W, i − 1, c − W[i], DP) + 1)
    return DP[i][c]
```

```
int smallSum(int[] W, int n, int C)
int[][] DP = new int[1 . . . n][1 . . . C]
return ssRec(W, n, C)
```

Il costo è $O(nC)$ per inizializzazione della tabella e nel caso pessimo in cui si debba riempirla tutta; l'algoritmo può essere migliorato con l'uso di una tabella hash.

Esercizio B3 – Trova l'ordine – Punti ≥ 12

A prima vista, parlando di permutazioni, si potrebbe essere tentati di risolvere il problema tramite backtrack. Una soluzione rapida da scrivere consiste nel generare tutte le permutazioni dei valori presenti con l'algoritmo visto a lezione e verificare che la permutazione rispetti le disequazioni, nel qual caso si stampa e si termina. Il costo è $O(n \cdot n!)$ nel caso pessimo in cui la permutazione da scegliere sia l'ultima visitata. È ovviamente migliorare tale soluzione facendo pruning qualora l'insieme parziale di valori scelti non rispettino le disequazioni.

Tuttavia, questo particolare problema può essere risolto con un approccio greedy.
Innanzitutto, partiamo dalla dimostrazione che una soluzione esiste sempre (data per scontata nel compito, ma dimostrata qui per completezza). È molto semplice dimostrare l'esistenza per induzione su n:

- Il caso base $n = 1$ corrisponde ad un unico valore e a nessuna disequazione. Banalmente, la sequenza rispetta l'ordinamento richiesto.
- Come ipotesi induttiva, assumiamo che esista sempre una soluzione per $n - 1$ valori e $n - 2$ segni di disequazione.
- Nel passo induttivo con n valori, si consideri il primo segno di disequazione e supponiamo sia $<$. Scegliamo il valore minimo fra gli n valori, e riduciamoci al problema con $n - 1$ valori e $n - 2$ segni di disequazione. Per induzione, una soluzione esiste di sicuro. Poichè n è minimo, è sicuramente più piccolo del primo valore della soluzione per $n - 1$ valori e $n - 2$ segni di disequazione. Quindi concatenando il minimo e la soluzione per $n - 1$, ottengo una soluzione per n.
- Ragionamento analogo se il primo segno è un $>$, nel qual caso si deve scegliere il valore massimo.

L'algoritmo ispirato da questa dimostrazione è il seguente: si ordinano i valori e poi si procede scegliendo il minimo o il massimo dei valori rimanenti, a seconda del segno che si incontra. Il costo totale è $O(n \log n)$, dominato dall'ordinamento.

```
int[] findOrder(int[] A, Item[] D, int n)
int[] S = new int[1 . . . n]
sort(A)
int i = 1
int j = n
for k = 1 to n − 1 do
    if D[i] == '<' then
        S[k] = A[i]
        i = i + 1
    else
        S[k] = A[j]
        j = j − 1
[n] = A[i]
return S
```