Algoritmi e Strutture Dati – 18/01/2021

Esercizio A1 – Punti ≥ 8

È facile vedere che la funzione è $\Omega(n^3 \log n)$, per via della parte non ricorsiva. Proviamo quindi a vedere se la funzione è anche $O(n^3 \log n)$.

- **Caso base**: per $n = 1$, si ottiene $T(1) = 1 \leq cn^3 \log n = 0$, 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 $T(i) = 1 \leq ci^3 \log i$, il che significa che $c \geq \frac{1}{i^3 \log i}$. Tutte queste disequazioni con segno $\geq$ sono rispettate dal valore che si ottiene per $i = 2$, ovvero sia ha che $c \geq \frac{1}{2^3 \log 2} = \frac{1}{8}$.
- **Ipotesi induttiva**: ipotizziamo che per ogni valore k, $2 \leq k < n$: $T(k) \leq k^3 \log k$
- **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:

$$
\begin{aligned}
T(n) &= 7T(\lfloor n/2 \rfloor) + 4T(\lfloor n/4 \rfloor) + n^3 \log n \\
&\leq 7c\lfloor n/2 \rfloor^3 \log \lfloor n/2 \rfloor + 4c\lfloor n/4 \rfloor^3 + \log \lfloor n/4 \rfloor + n^3 \log n && \text{Sostituzione} \\
&\leq 7cn^3/8 \log n/2 + 4cn^3/64 \log n/4 + n^3 \log n && \text{Eliminazione } \lfloor \rfloor \\
&\leq \frac{7}{8} cn^3 \log n + \frac{1}{16} cn^3 \log n + n^3 \log n && \log n/k < \log n, \text{con } k = 2, 4 \\
&= \frac{15}{16} cn^3 \log n + n^3 \log n \overset{?}{\leq} cn^3 \log n && \text{Passaggio algebrico}
\end{aligned}
$$

L'ultima disequazione è vera per $c \geq 16$. Abbiamo quindi dimostrato che $T(n) = O(n^3 \log n)$, per $m = 2$ e $c = 16$.

Abbiamo quindi dimostrato che $T(n) = O(n^3 \log n)$, per $c = 16$ e $m = 2$.

Esercizio A2 – Profondità minima – Punti ≥ 10

L'algoritmo proposto è una semplice postvisita. L'idea è di calcolare, per ogni nodo t, la distanza minima fra t e un nodo foglia nel sottoalbero radicato in t. Per questo motivo, se il nodo è una foglia, restituiamo 0. Altrimenti, restituiamo la distanza minima fra i figli sinistro/destro e le foglie radicate nei rispettivi sottoalberi, a cui aggiungiamo 1 per contare t. Per gestire la possibilità che un nodo abbia un solo figlio, utilizziamo il valore $+\infty$ per i nodi **nil**, che così non verrà mai selezionato da min(). Il valore restituito nella radice è il valore che cerchiamo.
La complessità dell'algoritmo è pari a quella di una visita di un albero, ovvero $\Theta(n)$.

```
int minDepth(TREE T)
if T == nil then
    return +∞
else if T.left == T.right == nil then
    return 0
else
    return 1 + min(minDepthRec(T.left), minDepthRec(T.right));
```

Questo algoritmo visita tutti i nodi dell'albero. È possibile migliorare l'algoritmo scrivendo una visita in ampiezza che si ferma alla prima foglia. In alcuni casi, questo può comportare un notevole miglioramento, a scapito di un maggior uso della memoria. La complessità nel caso pessimo, tuttavia, non cambia: in un albero perfetto, la profondità minima è pari all'altezza dell'albero, quindi bisogna comunque visitare metà dell'albero prima di fermarsi, con costo $O(n)$.

```
int minDepth(Tree t)
    int level = 0
    Queue Q = Queue()
    Q.enqueue(t)
    Q.enqueue(nil)
    while not Q.isEmpty() do
        Tree u = Q.dequeue()
        if u == nil then
            level = level + 1
            Q.enqueue(nil)
        else
            if u.left == u.right == nil then
                return level
            else
                if u.left ≠ nil then
                    Q.enqueue(u.left)
                if u.right ≠ nil then
                    Q.enqueue(u.right)
```

In questa soluzione, ho adottato il trucco di inserire **nil** per terminare il livello, che ho trovato nel compito di Dario Maddaloni.

Esercizio A3 – Individua il singolo – Punti ≥ 12

Il problema è risolvibile tramite un algoritmo divide-et-impera che considera il sottovettore $A[i \dots j]$, inizialmente $A[1 \dots n]$. Il caso base occorre quando $i = j$, ovvero c'è un solo elemento, che è quello da restituire.
Nel caso generale, si considera l'elemento centrale m del sottovettore considerato se dispari, o l'elemento precedente se pari. Se l'elemento m ed $m + 1$ sono uguali, significa che prima di m si trovano tutte coppie e quindi il valore cercato si trova a destra, dalla posizione $m + 2$ in poi. Se sono diversi, si trova a sinistra, entro la posizione m. E' possibile verificare che questo approccio funziona correttamente anche con tre elementi.
Volendo fare una dimostrazione di correttezza più approfondita, l'invariante è che il sottovettore che viene passato in input all'algoritmo inizia in una posizione dispari, ha dimensione dispari, e contiene l'elemento singolo. All'inizio, questo è vero perché $i = 1$, n dispari, e per assunzione contiene l'elemento singolo. Ad ogni passo, le tre proprietà vengono rispettate; al termine, essendoci un solo elemento, questo è l'elemento singolo.

```
findSingle(int[] A, int n)
    return fsRec(A, 1, n)
```

```
fsRec(int[] A, int n)
    if i == j then
        return A[i]
    else
        int m = ⌊(i + j)/2⌋
        if m mod 2 == 0 then
            m = m − 1
        if A[m] == A[m + 1] then
            return fsRec(A, m + 2, j)
        else
            return fsRec(A, i, m)
```

Essendo una ricerca dicotomica, la complessità è ovviamente $O(\log n)$.
Alcuni commenti:

- A differenza di altri problemi, questo problema dipende molto dalla numerazione dei vettori. Come al solito, la versione presentata qui assume che i vettori inizino dalla posizione 1.
- Esisteva una versione precedente di questo esercizio, che assumeva che i valori fossero ordinati. Come è possibile vedere, il problema è risolubile anche se i valori non sono ordinati. Ma la soluzione data nell'esercizio del gennaio 2018 vale anche per questo esercizio, come molti hanno notato.

Esercizio B1 – Parentesizzazioni – Punti ≥ 8

Il problema si risolve tramite la tecnica del backtrack, con printPar() procedura wrapper che crea un vettore S dove memorizzare le possibili parentesizzazioni e chiama la procedura printParRec() che prende in input cinque variabili:

- S è il vettore dove vengono memorizzate le parentesizzazioni
- *open* è il numero di parentesi attualmente aperte nel vettore S, inizialmente 0
- *toOpen* è il numero di parentesi aperte ancora da inserire, inizialmente n
- *toClose* è il numero di parentesi chiuse ancora da inserire, inizialmente n
- i è la posizione dove inserire la prossima parentesi, inizialmente 1

La procedura ricorsiva stamperà la parentesizzazione ogni volta che non esistono più parentesi aperte e chiuse da piazzare. In caso contrario, proverà a piazzare una parentesi aperta (possibile solo se il numero di parentesi aperte da piazzare è maggiore di 0) e/o chiusa (possibile se il numero di parentesi chiuse da piazzare è maggiore di 0 e il numero di parentesi attualmente aperte, da chiudere, è maggiore di 0).
Il numero di parentesizzazioni, come abbiamo visto nel problema del prodotto di catena di matrici, è $\Omega(2^n)$ (più precisamente, $\Omega\left(\frac{4^n}{n^{3/2}}\right)$, il costo della stampa é $\Theta(n)$, quindi il costo di questo algoritmo è esponenziale: $\Omega\left(n \cdot \frac{4^n}{n^{3/2}}\right) = \Omega\left(\frac{4^n}{\sqrt{n}}\right)$.

```
printPar(int n)
    ITEM[] S = new ITEM[2 · n]
    printParRec(S, 0, n, n, 1)
```

```
printParRec(ITEM[] S, int open, int toOpen, int toClose, int i)
    if toOpen == 0 and toClose == 0 then
        print S
    else
        if toOpen > 0 then
            S[i] = "("
            printParRec(S, open + 1, toOpen − 1, toClose, i + 1)
        if toClose > 0 and open > 0 then
            S[i] = ")"
            printParRec(S, open − 1, toOpen, toClose − 1, i + 1)
```

Esercizio B2 - Stiamo compatti - Punti ≥ 10

Questo problema è risolvibile in vari modi. Potrebbe essere risolto tramite backtrack, provando tutti i possibili sottoinsiemi e calcolando il minimo/massimo su di essi, con costo $\Theta(2^n)$ oppure $\Theta(n2^n)$ a seconda di come viene scritto; ma questa non è una soluzione efficiente, e non la scrivo perché non ne vale la pena.
Guardando l'esempio in cui valori sono ordinati, si può notare che i valori rimossi sono sui bordi; questo perché se la differenza fra il minimo e il massimo di un vettore non è inferiore o uguale a k, bisognerà considerare i sottoproblema in cui viene rimosso o il minimo o il massimo.
Quindi assumiamo che il vettore A sia ordinato, o ordiniamolo con costo $O(n \log n)$. Utilizziamo un approccio basato su programmazione dinamica. Sia $DP[i][j]$ il più piccolo insieme di valori da rimuovere da $A[i \dots j]$ per rispettare la condizione. La formula di programmazione dinamica è molto semplice:

$$DP[i][j] = \begin{cases} 0 & A[j] - A[i] \leq k \\ \min 1 + \{DP[i+1, j], DP[i, j-1]\} \end{cases}$$

Se $A[j] - A[i] \leq k$, non c'è nessun elemento da rimuovere e restituiamo 0. Altrimenti, l'intervallo è troppo grande, e dobbiamo provare a rimuovere l'elemento a sinistra oppure l'elemento a destra.
L'algoritmo può essere scritto in poche righe utilizzando memoization, oppure sfruttando la struttura dell'algoritmo per la catena di matrice che riempie i valori per diagonali:

```
int minRemove(int[] A, int n, int k)
sort(A, n)
int[][] DP = new int[1...n][1...n]
for i = 1 to n do                                          % Fill main diagonal
    DP[i][i] = 0
for h = 2 to n do                                          % h: diagonal index
    for i = 1 to n − h + 1 do                              % i: row
        int j = i + h − 1                                  % j: column
        if A[j] − A[i] ≤ k then
            DP[i][j] = 0
        else
            DP[i][j] = 1 + min(DP[i + 1, j], DP[i, j − 1])
return DP[1][n]
```

La complessità di questo algoritmo é $\Theta(n^2)$, perché dobbiamo riempire metà della matrice $n \times n$.
Infine, è possibile utilizzare un approccio "greedy", che sfrutta l'ordinamento nel modo seguente. Cerchiamo di calcolare il più grande sottoinsieme S che soddisfa la condizione su k, e da lì restituiamo $n - |S|$.
Si consideri un sottoinsieme dato dal sottovettore $A[i \dots j]$.

- Se $\max(A[i \dots j]) - \min(Ai \dots j) > k$, è inutile provare ad allargare il vettore alla sua destra - la condizione non sarà sicuramente rispettata, perché $A[j+1] > A[j]$. Quindi proviamo ad restringere il vettore alla sinistra ($i = i + 1$).
- Se $\max(A[i \dots j]) - \min(Ai \dots j) \leq k$, è inutile restringere il vettore alla sua sinistra - la condizione sarà rispettata, ma il vettore sarà più piccolo. Proviamo quindi ad allargare a destra ($j = j + 1$), aggiornando eventualmente il vettore massimo che abbiamo incontrato.

```
int minRemove(int[] A, int n, int k)
sort(A, n)
int i = 1
int j = 1
int maxSoFar = 0
while i ≤ n and j ≤ n do
    if A[j] − A[i] ≤ k then
        maxSoFar = max(maxSoFar, j − i + 1)
        j = j + 1
    else
        i = i + 1
return n − maxSoFar
```

Questo algoritmo ha costo $O(n \log n)$, dominato dall'ordinamento. Infatti ad ogni iterazione del ciclo, o l'indice i o l'indice j vengono incrementati di 1; non appena uno dei due indici supera n, l'algoritmo termina. Quindi il numero massimo di iterazioni per il ciclo **while** è pari $2n - 1$.
Una proposta alternativa di costo $O(n \log n)$ [1] consiste nell'ordinare il vettore (costo $O(n \log n)$). Si scorre poi ogni elemento $A[i]$ con $i = 1 \dots n$, preso come primo elemento di un intervallo di valori; si cerca, tramite ricerca dicotomica, l'indice j più grande tale che $A[j] \leq A[i] + k$. $A[i \dots j]$ è quindi l'intervallo più grande che rispetta il vincolo su k e inizia nella posizione i. Scorrendo tutti i possibili indici iniziali, si trova l'intervallo più grande in assoluto memorizzandolo in una variabile *maxSoFar* e come nell'esempio precedente, si restituisce $n - maxSoFar$. Il costo di n ricerche dicotomiche è $O(n \log n)$, e quindi la complessità finale è $O(n \log n)$.
Vista la complessità identica all'esercizio precedente, lasciamo la scrittura del codice come esercizio.

Esercizio B3 - Stessa visita - Punti ≥ 12

Ragionando sul problema, si può notare che qualunque sia la struttura dell'albero, una visita in prodondità simmetrica induce uno e un solo ordine di visita dei nodi, e che è sufficiente piazzare i numeri da 1 a n seguendo tale ordine. Questo principio è stato sfruttato nella soluzione dell'esercizio A2 del 3/7/2020. Quindi, la parte relativa alla numerazione da 1 a n non deve essere considerata, e ci basta calcolare il numero di alberi strutturalmente diversi contenenti n nodi.
Questo problema è già stato risolto nel fascicolo di esercizi sugli alberi; ripetiamo qui il testo per completezza. Per definire una ricorrenza, ci basiamo sulla seguente osservazione. Un albero di n nodi ha sicuramente una radice; i restanti $n - 1$ nodi possono essere distribuiti nei sottoalberi destro e sinistro della radice. Ad esempio, un albero di 4 nodi può avere 3 nodi nel

[1] Credits: Amir Ghesir, via email

sottoalbero destro e 0 nel sinistro; oppure 2 nodi nel destro e 1 nel sinistro; oppure 1 nel destro e 2 nel sinistro; oppure 0 nel destro e 3 nel sinistro.
Detto quindi k il numero di nodi nell'albero sinistro, una formula ricorsiva per calcolare il numero di alberi strutturalmente diversi $DP[n]$ è la seguente:

$$DP[n] = \begin{cases} 1 & n \leq 1 \\ \sum_{k=0}^{n-1} DP[k]DP[n-1-k] & n > 1 \end{cases}$$

Non abbiamo mai scritto una versione basata su programmazione dinamica, ma è presto fatto:

```
int sameVisit(int n)
    int[] DP = new int[0...n]
    DP[0] = DP[1] = 1
    for i = 2 to n do
        DP[i] = 0
        for k = 0 to i − 1 do
            DP[i] = DP[i] + DP[k] · DP[n − k − 1]
    return DP[n]
```

Per via del doppio ciclo, il costo è $\Theta(n^2)$, considerando il criterio di costo uniforme.
Una soluzione di questo tipo, con questa analisi di complessità prende 100%, perché utilizza in maniera intelligente e rapida quanto visto a lezione e negli esercizi. Quanto segue va oltre quanto richiesto durante un compito di durata limitata.
Volendo fare un passo in più, si può notare che il numero di alberi binari strutturalmente diversi con n nodi è pari all'n-esimo numero di Catalan, che è così definito:

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} = \Theta\left(\frac{4^n}{n^{3/2}}\right)$$

È possibile calcolare $n!$, $(n+1)!$, $(2n)!$ in un'unica passata di costo lineare, utilizzando quanto visto a lezione. È quindi possibile risolvere il problema in tempo lineare.
L'ultima osservazione da fare è che l'analisi di complessità basata sul costo di criterio uniforme non è corretta qui, perché i numeri coinvolti crescono esponenzialmente in n; quindi, il numero di bit necessari per memorizzarli cresce linearmente in n.

- Nel caso del primo algoritmo, sono eseguite $\Theta(n^2)$ moltiplicazioni fra numeri di n bit; utilizzando Karatsuba, abbiamo un costo per ogni moltiplicazione di $O(n^{\log_2 3})$, e quindi il costo totale è $O(n^{2+\log_2 3}) \approx O(n^{3.58})$ utilizzando il criterio di costo logaritmico.
- Nel caso del secondo algoritmo, i valori si ottengono tramite $2n$ somme di costo $O(n)$, e quindi il costo è $O(n^2)$ utilizzando il criterio di costo logaritmico.