Algoritmi e Strutture Dati – 12/06/2020

Esercizio A1

Limite asintotico inferiore Data la componente non ricorsiva, è facile vedere che $T(n) = \Omega(n^2)$.

Limite asintotico superiore Supponiamo che $T(n) = O(n^2)$. Dobbiamo dimostrare che

$$\exists c > 0, \exists m > 0 : T(n) \leq cn^2, \forall n \geq m$$

Nei casi base:

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

Ipotesi induttiva:

$$T(k) \leq ck^2, \forall k < n$$

Passo induttivo:

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2T\left(\left\lfloor \frac{n}{2\sqrt{2}} \right\rfloor\right) + 7T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + n^2 \\ &\leq cn^2/4 + 2cn^2/8 + 7cn^2/16 + n^2 \\ &= \frac{15}{16}cn^2 + n^2 \\ &\leq cn^2 \end{aligned}$$

L'ultima disequazione è vera per $c \geq 16$. Abbiamo quindi dimostrato che

$$\exists c > 0, \exists m > 0 : T(n) \leq cn^2, \forall n \geq m$$

per $c \geq 16$, $m = 1$.

Conclusioni Abbiamo quindi dimostrato che $T(n) = \Theta(n^2)$.

Esercizio A2

Se l'albero binario in input è completo, è sufficiente controllare che il valore di uno nodo sia superiore a quello del padre. La procedura ricorsiva mhRec() prende in input il nodo sotto controllo e il valore del padre, verifica che la condizione sia rispettata nel nodo e verifica ricorsivamente nei due figli, passando il proprio valore. Il caso base è dato da un nodo **nil**, che ritorna **true**. Potendo assumere che l'albero sia non vuoto, la funzione wrapper richiama mhRec() sui due figli della radice.

```
boolean isMinHeap(TREE T)
    return mhRec(T.left, T.value) and mhRec(T.right, T.value)
```

```
boolean mhRec(TREE t, int lb)
    if T == nil then
        return true
    else
        return t.value > lb and mhRec(t.left, t.value) and mhRec(t.right, t.value)
```

La complessità è quella di una visita in profondità dell'albero, ovvero $\Theta(n)$.

Esercizio A3

Innanzitutto, è necessario calcolare una stima superiore alla dimensione del vettore. Questo si può fare controllando la posizione a distanze esponenzialmente crescenti. Non appena si trova un valore $\perp$, si usa l'indice così trovato come stima superiore della dimensione.
Utilizziamo poi un algoritmo di ricerca binaria per individuare l'indice del primo valore positivo o nullo nel vettore.

```
int countNegatives(MisteryArray A)
    int n = 1
    while A[n] ≠ ⊥ do
        n = n · 2
    return cnRec(A, 1, n) − 1
```

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

Il ciclo while contenuto nel wrapper esegue al più $O(\log n)$ passi, in quanto la dimensione n cosi trovata è tale che $n < 2n$.
La procedura di ricerca ha costo $O(\log n)$, essendo una ricerca dicotomica.

Esercizio B1

È sufficiente costruire un vettore che contenga il massimo dei due elementi di ogni colonna e applicare la soluzione di Hateville al vettore così costruito. Infatti, se una casella viene selezionata su una colonna, non sarà possibile prendere l'altra casella della colonna stessa e nessun valore delle colonne precedente e successiva; ma in questo caso, tanto varrà prendere il massimo fra i due valori per ogni colonna e chiamare Hateville. La complessità dell'algoritmo proposto è $O(n)$.

```
int maxGain(int[][]A, int n)
    int[] D = new int[1 . . . n]
    for i = 1 to n do
        D[i] = max(A[1][i], A[2][i])
    return hateville(D, n)
```

Esercizio B2

Per raggiungere il lato in basso a destra, dobbiamo fare n passi a destra e n passi in basso, intervallati in qualunque modo.
Sia $DP[i][j]$ il numero di percorsi in una griglia di dimensione $i \times j$, potendo fare i passi lungo un asse e j passi lungo l'altro. È possibile esprimere ricorsivamente DP tramite la formula seguente:

$$DP[i][j] = \begin{cases} 1 & i = 0 \textbf{ or } j = 0 \\ DP[i-1][j] + DP[i][j-1] & \text{altrimenti} \end{cases}$$

Nel caso base, se $i = 0$ oppure $j = 0$ ho una griglia $0 \times n$ oppure $n \times 0$ (una retta), quindi il numero di percorsi è pari a 1. Altrimenti, posso fare due scelte: ridurre la griglia su un'asse o sull'altro. Il numero di percorsi espresso da queste due possibilità va sommato insieme.

```
paths(int n)
    int[][] DP = new int[0...n][0...n]
    for i = 0 to n do
        DP[i][0] = DP[0][i] = 1
    for i = 1 to n do
        for j = 1 to n do
            DP[i][j] = DP[i − 1][j] + DP[i][j − 1]
    return DP[n][n]
```

Questo algoritmo effettua $\Theta(n^2)$ somme.

Note aggiuntive - non richieste nel compito Il valore restituito da questa funzione è pari a $\binom{2n}{n} = (2n)!/(n!)^2$, in quanto si tratta di formare una stringa di $2n$ simboli con n simboli D (down) e n simboli R (right). Al crescere di n, i valori restituiti formano la sequenza matematica https://oeis.org/A000984.
Sebbene ritengo perfettamente accettabile in un compito scrivere che l'algoritmo abbia un costo computazionale pari a $\Theta(n^2)$, nel caso di architetture a 64 bit questo è vero fino a $n = 33$. Dopo di che, i numeri coinvolti richiedono un numero crescente di bit, superiore a 64, e le somme non possono essere più considerate di costo costante.
È possibile dimostrare, utilizzando l'approssimazione di Stirling dei numeri fattoriali, che $\log(2n)!/(n!)^2 \leq 2n$:

$$\begin{aligned}
\log \frac{(2n)!}{(n!)^2} &\approx \log \frac{\sqrt{2\pi 2n}\left(\frac{2n}{e}\right)^{2n}}{\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}\right)^2} \\
&= \log \frac{2\sqrt{\pi n}\left(\frac{2n}{e}\right)^{2n}}{2\pi n\left(\frac{n}{e}\right)^{2n}} \\
&= \log \frac{2^{2n}}{\sqrt{\pi n}} \\
&= 2n - 1/2 \log(\pi n)
\end{aligned}$$

In altre parole, è possibile rappresentare il risultato con $O(n)$ bit (in particolare, con un valore leggermente inferiore a $2n$). Il costo delle somme è quindi lineare in n, e il costo complessivo dell'algoritmo è $O(n^3)$. Volendo migliorare la complessità, è possibile utilizzare direttamente la formula basata su numeri fattoriali per ottenere il risultato. In particolare, si può notare[1] che

$$\frac{2n!}{(n!)^2} = \frac{\prod_{i=n+1}^{2n} i}{\prod i = 1^n}$$

in altre parole, è possibile calcolare il risultato con $2n$ moltiplicazioni e una divisione. Omettiamo il codice per brevità. Poichè il risultato finale richiede $O(n)$ bit, l'algoritmo esegue in $O(n^3)$ se eseguito con l'algoritmo naive, $O(n^{1+\log_2 3})$ se eseguito con Karatsuba, e $\Theta(n^2 \log n \log\log n)$ se si utilizza Schönhage–Strassen, fino ad arrivare a $O(n^2 \log n)$ se si utilizza Harvey–van der Hoeven (che però ha costi moltiplicativi molto alti).

[1] Versione migliorata della spiegazione grazie a una mail di Amir Gheser

Esercizio B3

Iniziamo risolvendo prima il caso in cui si considerino solo somme palindrome con un numero pari di addendi. Come molti hanno notato, solo i valori pari di n possono dare origine a tali somme palindrome. Utilizziamo il backtracking per generare le possibili sommatorie. Memorizziamo in un vettore S metà della somma palindroma; l'altra metà verrà stampata in ordine inverso. Quindi, tutte le volte che viene scelto un valore, esso viene sottratto alla somma da generare due volte, una per la prima metà della stringa palindroma, una per la seconda metà.

```
genRec(int[] S, int i, int missing)
if missing == 0 then
    for k = 1 to i − 1 do
        print (S[k])
    for k = i − 1 downto 1 do
        print (S[k])
for j = 1 to missing/2 do
    S[i] = j
    genRec(S, i + 1, missing − 2 · j)
```

```
generate(int n)
if n mod 2 == 0 then
    int[] S = new int[1 . . . n/2]
    genRec(S, 1, n)
```

Si noti che tramite il controllo sul fatto che inizialmente n debba essere pari, e il fatto che nelle chiamate ricorsive viene sottratto $2j$, è possibile assumere che tutte le chiamate a $\mathsf{genrec}(S, i, missing)$ abbiano *missing* pari. Correttamente, questo algoritmo genera tutte le sommatorie di lunghezza pari.
Per costruire l'algoritmo completo, è possibile notare che tutte le volte che viene chiamata $\mathsf{genrec}(S, i, missing)$ con $missing > 0$, è possibile costruire una sommatoria palindroma composta dai primi $i - 1$ valori, poi *missing*, poi i primi $i - 1$ valori in ordine inverso. In questo modo, si ottiene la somma corretta $n = missing + 2 \cdot \sum_{k=1}^{i-1} S[k]$. L'algoritmo così modificato stampa una sommatoria ad ogni chiamata ricorsiva, con o senza l'elemento centrale a seconda che *missing* sia maggiore di zero oppure no.
Alcune note sulla correttezza dell'algoritmo:

- per come è costruito, *missing* non può mai assumere un valore negativo se n è non negativo;
- una chiamata ricorsiva termina quando $missing = 0$ oppure $missing = 1$, in quanto il ciclo **for** da 1 a $\lfloor missing/2 \rfloor$ non viene eseguito.

```
genRec(int[] S, int i, int missing)
for k = 1 to i − 1 do
    print (S[k])
if missing > 0 then
    print missing
for k = i − 1 downto 1 do
    print (S[k])
for j = 1 to ⌊missing/2⌋ do
    S[i] = j
    genRec(S, i + 1, missing − 2 · j)
```

```
generate(int n)
    int[] S = new int[1 ... ⌊n/2⌋]
    genRec(S, 1, n)
```

Questo algoritmo è ottimo, perché il numero di chiamate ricorsive corrisponde esattamente al numero di sommatorie palindrome da stampare. Ogni stampa ha costo $O(n)$ e il numero di sommatorie è superpolinomiale, in quanto ad ogni chiamata ricorsiva ho $missing/2$ scelte da compiere.

Note aggiuntive - non richieste nel compito Visto che ad ogni chiamata ricorsiva l'algoritmo genera sempre una delle sequenze palindrome richieste, per calcolare più precisamente la complessità è necessario conoscere quante sequenze vengono stampate.

Ho proceduto in maniera molto informatica: prima di lanciami in complesse valutazioni, ho scritto codice e le ho contate, sperimentalmente. Il risultato che è venuto fuori è interessante: $2^{\lfloor n/2 \rfloor}$. Per provare un risultato del genere in maniera elegante, tuttavia, è necessaria una buona dose di fantasia. La dimostrazione che segue è farina del sacco di Marta Fornasier, io ho solo scritto il testo descrittivo.

Partiamo calcolando $S(k)$, il numero di modi (permutazioni incluse) per cui è possibile esprimere k come sommatoria di valori positivi (non necessariamente palindromi). Ad esempio, $k = 4$ può essere espresso in 8 modi:

$$[4], [3, 1], [1, 3], [2, 2], [1, 1, 2], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]$$

È possibile dimostrare che $S(k) = 2^{k-1}$, per induzione:

- Caso base $k = 1$. Esiste un'unica sequenza, $[1]$, e $S(1) = 2^{1-1} = 1$. Il caso base è quindi dimostrato.
- Ipotesi induttiva: consideriamo il caso k, e assumiamo che $S(i) = 2^{i-1}$, per $1 \leq i < k$
- Passo induttivo: oltre alla sequenza $[k]$, è possibile esprimere k come somma di un valore i compreso fra 1 e $k - 1$, seguito da uno qualunque dei modi per esprimere $(k - i)$, che sappiamo essere 2^{k-i-1}. Quindi abbiamo

$$\begin{aligned} S(k) &= 1 + \sum_{i=1}^{k-1} 2^{k-i-1} \\ &= 1 + \sum_{i=0}^{k-2} 2^k && \text{per inversione della sommatoria} \\ &= 1 + 2^{k-1} - 1 = 2^{k-1} \end{aligned}$$

A questo punto, possiamo contare quanti modi ci sono per esprimere un valore n come sommatoria palindroma. Esprimiamo n come una sommatoria così composta

$$\overbrace{a_1, a_2, \ldots, a_t}^{k}, (n - 2k), \overbrace{a_t, \ldots, a_2, a_i}^{k}$$

dove k è la sommatoria dei termini $a_1, \ldots, a_t$ che poi vengono specchiati in $a_t, \ldots, a_1$.

È importante notare che k può assumere il valore 0 (corrispondente alla sommatoria composta dal solo n), oppure può essere un valore positivo compreso fra 1 e $\lfloor n/2 \rfloor$. Se n è pari e $k = n/2$, $(n - 2k)$ è pari a 0. Questo è il caso in cui la somma palindroma ha lunghezza pari, e lo zero "centrale" non viene considerato nella sommatoria.
Il numero totale di somme palindrome $SP(n)$ per il valore n sarà quindi 1 (sequenza composta dal solo n) più la sommatoria per esprimere un valore k compreso fra 1 e $\lfloor n/2 \rfloor$:

$$\begin{aligned} SP(n) &= 1 + \sum_{k=1}^{\lfloor n/2 \rfloor} S(k) \\ &= 1 + \sum_{k=1}^{\lfloor n/2 \rfloor} 2^{k-1} \\ &= 1 + \sum_{k=0}^{\lfloor n/2 \rfloor - 1} 2^k \\ &= 1 + 2^{\lfloor n/2 \rfloor} - 1 = 2^{\lfloor n/2 \rfloor} \end{aligned}$$

Quindi la complessità dell'algoritmo è $O(n \cdot 2^{n/2})$, dove $O(n)$ è il limite superiore per la stampa della sommatoria.