Algoritmi e Strutture Dati – 9/1/2024

Esercizio A1 – Complessità – Punti ≥ 8

Per calcolare la complessità computazionale di fun(), definiamo l'equazione di ricorrenza associata a fr(). Innanzitutto, è facile vedere che la funzione fr() richiama se stessa su due sottevettori di dimensione $n/2$, 4 volte. La complessità della parte non ricorsiva è invece $\Theta(n \log n)$, dominata dalla chiamata di Mergesort.
L'equazione di ricorrenza è quindi:

$$T(n) = \begin{cases} 4T(n/2) + n \log n & n > 1 \\ 1 & n = 1 \end{cases}$$

Utilizzando il Master Theorem versione estesa, otteniamo che $\alpha = \log_2 4 = 2$, $f(n) = n \log n$. Ricadiamo nel caso (1), in quanto $f(n) = O(n^{2-\epsilon})$, per qualunque $\epsilon < 1$.
La complessità dell'algoritmo è quindi $\Theta(n^2)$.

Nota: abbiamo specificato che $n = 2^k$ perché il codice non è scritto benissimo, in quanto dimensioni che non siano potenze di 2 potrebbero disallineare la dimensione delle due metà calcolate all'inizio.

Esercizio A2 – Tree Repair Shop 2 – Punti ≥ 10

In questa versione, la rimozione di alcuni archi trasforma ciò che era inizialmente un albero non radicato in una foresta composta da più componenti connesse. Il nostro compito consiste nel "ricucire" questa foresta per riportarla a un singolo albero.
Per risolvere questo problema, adottiamo il seguente approccio:

- Identifichiamo le componenti connesse all'interno del grafo utilizzando la funzione cc(), discussa a lezione, che assegna un identificatore unico ad ogni nodo.
- Scegliamo un nodo "rappresentante" per ciascuna componente connessa. Questo viene fatto esaminando tutti i nodi e assegnando il nodo i-esimo come rappresentante della componente $id[i]$. In questo modo, registriamo l'ultimo nodo visitato di ogni componente. Questi rappresentanti sono memorizzati nel vettore *reps*, che ha una dimensione massima pari a n (nel caso estremo in cui tutti gli archi siano stati rimossi e il grafo si sia ridotto a un insieme di nodi isolati).
- Colleghiamo il nodo "rappresentante" di ogni componente (ad eccezione della prima) al nodo rappresentante la prima componente. Il ciclo **while** viene interrotto non appena tutte le componenti sono state connesse.

```
repairTree(GRAPH G)
int[] id = CC(G)
int[] reps = new int[1...G.n] = {0}
for i = 1 to G.n do
    reps[id[i]] = i
int i = 2
while i ≤ G.n and reps[i] ≠ 0 do
    G.insertEdge(reps[1], reps[i])
    i = i + 1
```

La complessità dell'algoritmo proposto è $\Theta(m + n)$, derivante dalla visita in profondità dell'algoritmo cc(n). Poichè il grafo originale era un albero e sono stati rimossi alcuni archi, sappiamo che $m = O(n)$ e quindi l'algoritmo è in realtà $\Theta(n)$.
Una versione alternativa, che ho trovato in diversi compiti, è la seguente: adatto la funzione per identificare le componenti connesse, tenendo la funzione ccdfs() così com'è. Richiamo la funzione ccdfs() sul nodo 1, ottenendo la prima componente connessa. Per ogni nodo, se trova un nodo u non ancora visitato ($id[u] = 0$), richiamo la ccdfs() su quel nodo. Si noti che abbiamo rimosso l'identificatore dela componente connessa e utilizziamo i valori 1 versus 0 per indicare un nodo visitato oppure no. La complessità è sempre $\Theta(n)$, come la precedente.

```
repairTree(GRAPH G)
    int id = new int[G.n] = {0}
    ccdfs(G, 1, 1, id)
    foreach u ∈ G.V() do
        if id[u] == 0 then
            ccdfs(G, 1, u, id)
            G.insertEdge(1, u)
```

Esercizio A3 – Un singolo – Punti ≥ 12

La prima cosa da fare è leggere bene il testo, come suggerito.

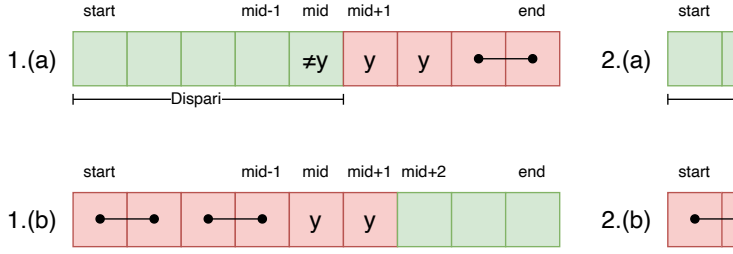
- il valore che stiamo cercando compare una e una sola volta;
- tutti gli altri valori compaiono un numero pari di volte;
- i valori diversi da y compaiono a coppie consecutive, e non è possibile che esistano 3 o più valori consecutivi uguali;
- poiché soluzioni lineari non vengono considerate, questo suggerisce un approccio basato su ricerca dicotomica.

A parte la possibilità che le coppie si ripetano (ma non consecutivamente), il problema può essere risolto tramite la soluzione del problema A3 del 18/1/2021. Proponiamo qui una soluzione alternativa.

Ad ogni chiamata ricorsiva di osRec$(A, start, end)$, vogliamo assicurarci che valga il seguente invariante: (i) il valore x è contenuto in $A[start \dots end]$ e (ii) le proprietà del vettore sono comunque rispettate. Questo comporta fra l'altro che (iii) la dimensione del vettore $A[start \dots end]$ è sempre dispari.

Consideriamo l'elemento centrale m; sono dati due casi:

1. Il numero di elementi $A[start \dots mid]$ è dispari. Possono darsi due sotto-casi:
   (a) l'elemento $A[mid]$ è diverso da $A[mid+1]$. Allora l'elemento che cerchiamo è compreso nel sottovettore $A[start \dots mid]$ (di dimensione dispari), in quanto il sottovettore $A[mid+1 \dots end]$ ha dimensione pari e non può contenere il valore x.
   (b) l'elemento $A[mid]$ è uguale da $A[mid+1]$. Allora l'elemento che cerchiamo è compreso in $A[mid+2 \dots end]$ (di dimensione dispari), in quanto il sottovettore $A[start \dots mid+1]$ ha dimensione pari e non può contenere il valore x.
2. Il numero di elementi $A[start \dots mid]$ è pari. Possono darsi due sotto-casi:
   (a) l'elemento $A[mid]$ è uguale da $A[mid+1]$. Allora l'elemento che cerchiamo è compreso nel sottovettore $A[start \dots mid-1]$ (di dimensione dispari), in quanto il sottovettore $A[mid \dots end]$ ha dimensione pari e non può contenere il valore x.
   (b) l'elemento $A[mid]$ è diverso da $A[mid+1]$. Allora l'elemento che cerchiamo è compreso in $A[mid \dots end]$ (di dimensione dispari), in quanto il sottovettore $A[start \dots mid-1]$ ha dimensione pari e non può contenere il valore x.



Questi quattro casi sono rappresentati nella figura seguente: x

```
int oneSingle(int[] A, int n)
    return osRec(A, 1, n)
```

```
int osRec(int[] A, int start, int end)
if start == end then
    return A[start]
else
    mid = ⌊(start + end)/2⌋
    if (mid − start + 1) mod 2 == 1 then
        if A[mid] ≠ A[mid + 1] then
            return osRec(A, start, mid)                    % Caso 1.(a)
        else
            return osRec(A, mid + 1, end)                  % Caso 1.(b)
    else
        if A[mid] == A[mid + 1] then
            return osRec(A, start, mid − 1)                % Caso 2.(a)
        else
            return osRec(A, mid, end)                      % Caso 2.(b)
```

Si noti che anche quando il sottovettore considerato ha dimensione 3, il sottovettore $A[start \dots mid]$ ha dimensione 2 e quindi il caso 2.(b) (con un pericoloso $mid + 2$) non si applica. Il costo computazionale è ovviamente $\Theta(\log n)$.

Esercizio B1 – Metà o meno – Punti ≥ 8

È possibile risolvere il problema utilizzando la tecnica del backtracking. Viene utilizzato uno stack che permette dinamicamente di aggiungere e togliere elementi. Inizialmente, inseriamo il valore n che sappiamo essere sempre presente nelle sequenza da stampare e chiamiamo ricorsivamente la funzione lthRec(), che ricorsivamente stampa tutte le sequenze.
Nelle chiamate di lthRec(), la prima cosa che facciamo è stampare lo stack[1]. La stampa avviene senza "aspettare" di raggiungere il valore 1, perché sia dalla definizione del problema che dall'esempio si comprende che dobbiamo stampare tutte le sequenze che rispettano la regola, anche quelle corte. Quindi, tutte le volte che viene inserito un valore (corrispondente all'inizio della chiamata), procediamo alla stampa.
Dopo di che, si utilizza come valore seguente un intero i compreso fra 1 e $\lfloor stack.\mathsf{top}()/2 \rfloor$, dove $stack.\mathsf{top}()$ è l'ultimo valore inserito nello stack. Si noti che se $last = 1$, il ciclo **for** va a vuoto e la ricorsione termina.

```
lessThanHalf(int n)
STACK stack = Stack()
stack.push(n)
return lthRec(stack)
```

```
lthRec(STACK stack)
print stack                          % Assumiamo che stampi nell'ordine corretto
for i = 1 to ⌊stack.top()/2⌋ do
    stack.push(i)
    lthRecstack
    stack.pop()
```

Complessità, considerazioni sulla valutazione Quando ho scritto l'esercizio, mi sono preoccupato di scrivere il codice per verificare la difficoltà. Ho visto che era molto semplice e, quindi, l'ho inserito come primo esercizio, senza preoccuparmi di calcolare la complessità, che invece si è rivelata molto complessa. Dal punto di vista della valutazione, non ho valutato l'analisi della complessità.

Complessità, analisi più approfondita La complessità computazionale di un algoritmo simile non è semplice da calcolare. Innanzitutto, partiamo dall'osservazione che la lunghezza massima di una sequenza è pari

[1] Come spesso accade in questi esercizi, assumiamo che lo stack venga stampato nell'ordine corretto; in altre parole, non perdiamo tempo a gestire il pretty printing e ci concentriamo sugli aspetti algoritmici più complessi

a $1 + \lfloor \log_2 n \rfloor$, in quanto ad ogni elemento successivo dimezziamo il valore. Possiamo quindi dire che l'operazione di stampa è $O(\log n)$, a differenza del solito $O(n)$.
Concentriamoci quindi sul numero $T(n)$ di sequenze possibili il cui valore iniziale sia n. Dobbiamo considerare la sequenza costituita dal solo valore n e poi considerare le sequenze date da n seguito da tutte le possibili sequenze che iniziano con un valore compreso fra 1 e $\lfloor n/2 \rfloor$:

$$T(n) = \begin{cases} 1 + \sum_{i=1}^{\lfloor n/2 \rfloor} T(i) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

Utilizzando il metodo della sostituzione, è possibile dimostrare che $T(n) = \Omega(n^k)$ per qualunque valore k; in altre parole, $T(n)$ è superpolinomiale. Proviamo infatti a dimostrare che $T(n) \geq cn^k$ e scriviamo:

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{\lfloor n/2 \rfloor} T(i) \\ &\geq 1 + \sum_{i=1}^{\lfloor n/2 \rfloor} ci^k \\ &= 1 + c \sum_{i=1}^{\lfloor n/2 \rfloor} i^k \end{aligned}$$

Solamente a titolo esemplificativo, prendiamo il caso $k = 2$; si ottiene quindi:

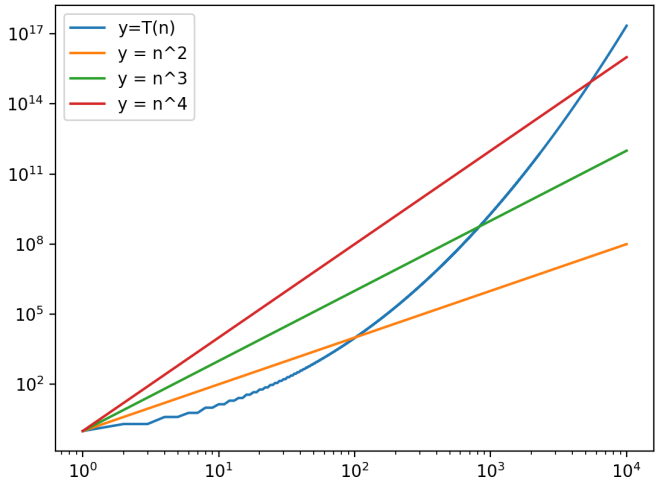
$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{\lfloor n/2 \rfloor} T(i) \\ &\geq 1 + \sum_{i=1}^{\lfloor n/2 \rfloor} ci^2 \\ &= 1 + c \sum_{i=1}^{\lfloor n/2 \rfloor} i^2 \\ &= 1 + c \frac{\lfloor n/2 \rfloor (\lfloor n/2 \rfloor + 1)(2\lfloor n/2 \rfloor + 1)}{6} \\ &\geq cn^2 \end{aligned}$$

Senza entrare nei dettagli del calcolo di c, l'ultima disequazione è vera perché il penultimo valore è $\Theta(n^3)$.
Per dimostrare il caso generale, si utilizza la formula di Faulhaber[2]:

$$\sum_{i=1}^{n} i^k = \frac{1}{k+1} \sum_{j=0}^{k} (-1)^j \binom{k+1}{j} B_j n^{k+1-j}$$

dove B_j rappresenta l'i-esimo numero di Bernoulli. Al di là della complessità di questa formula, quello che ci interessa è notare che la sommatoria di potenze di grado k ha come grado $k + 1$, e questo ci permette di dimostrare che la funzione $T(n)$ è superpolinomiale.
Per avere un'idea di cosa succede, si osservi il plot seguente, che mostra come il numero di sequenze così ottenute cresce più rapidamente dei polinomi di grado 2,3,4, in un grafico log-log:

[2] https://it.wikipedia.org/wiki/Somma_di_potenze_di_interi_successivi



Ovviamente, un'analisi di questo tipo va al di là di quanto richiesto in un compito, ma vi dà l'idea di come si possano utilizzare i metodi che conosciamo (e un po' di Wikipedia) per comprendere meglio la complessità di una funzione.

Esercizio B2 – Alberibelli – Punti ≥ 10

Sia $DP[n]$ il numero di alberibelli composti da n nodi. La formula ricorsiva per calcolare tale valore è la seguente:

$$DP[n] = \begin{cases} 1 & n = 1 \\ 0 & n = 2 \\ \sum_{j=1}^{n-2} DP[j] \cdot DP[(n-1)-j] & n > 2 \end{cases}$$

- Se $n = 1$, esiste un solo alberobello, composto da un nodo con zero figli.
- Se $n = 2$, non è possibile comporre un alberobello, perché l'unico modo di comporre un albero di 2 nodi è dato da una radice con un figlio, che non rispetta la definizione di alberobello.
- In tutti gli altri casi si considera una radice con due figli e si distribuiscono i restanti $n-1$ nodi nei due sottoalberi sinistro e destro della radice. Il numero di nodi a sinistra va da 1 a $n-2$; di conseguenza, il numero di nodi a destra è $(n-1)-i$ e va da $n-2$ a 1. Ogni sottoalbero a sinistra può essere abbinato con ogni sottoalbero a destra; quindi i numeri di sottoalberi nei due lati vanno moltiplicati fra di loro.

È possibile osservare che nel caso di alberi con un numero pari di nodi, la formula correttamente restituisce 0; questo perché $n-1$ è dispari, e quindi almeno uno dei due sottoalberi ha un numero pari di nodi, moltiplicando quindi per zero.
È sicuramente più efficiente sfruttare questo fatto, evitando di calcolare e utilizzare tutte le dimensioni pari nelle formule.

```
countAB(int n)
if n mod 2 == 0 then
    return 0
else
    int DP = new int[1...n]
    DP[1] = 1
    for i = 3 to n step 2 do
        DP[i] = 0
        for j = 1 to i − 2 step 2 do
            DP[i] = DP[i] + DP[j] · DP[(i − 1) − j]
    return DP[n]
```

La complessità è ovviamente $O(n^2)$ per via dei due cicli annidati.

Esercizio B3 – LCNCS – Punti ≥ 12

La soluzione può essere ottenuta adattando la formula per la LCS. Quando si trovano due caratteri uguali, è possibile:

- selezionarli, nel qual caso si devono saltare entrambi i caratteri precedenti nelle stringhe originali;
- non selezionarli, nel qual caso si sceglie di rimuovere un carattere dall'una o dall'altra stringa.

La formula ricorsiva è la seguente:

$$DP[i][j] = \begin{cases} 0 & i \leq 0 \ \vee j \leq 0 \\ max\{DP[i-1][j], DP[i][j-1], DP[i-2][j-2]+1\} & i > 0 \wedge j > 0 \wedge T[i] = U[j] \\ max\{DP[i-1][j], DP[i][j-1]\} & i > 0 \wedge j > 0 \wedge T[i] \neq U[j] \end{cases}$$

Si noti che nel caso base abbiamo gestito anche il caso in cui gli indici siano negativi; infatti, se i primi due caratteri sono uguali ($i = j = 1$), si ottiene $i - 2 = j - 2 = -1$.
La formula sopra è "prudente": permette di scegliere fra prendere e non prendere coppie di caratteri uguali, cercando il massimo fra tre casi. Tuttavia, la scelta non è in realtà necessaria: conviene sempre scegliere i caratteri uguali. Siano $T[n] = U[m]$. Sia t la lunghezza della LCNCS contenuta in $T(n-2)$ e $U(n-2)$; la lunghezza della LCNCS in T, U è quindi pari a $t+1$. Supponiamo che la lunghezza della LCNCS in $T(n-1)$ e U sia maggiore di $t+1$ (il caso in cui si considerano T e $U(n-1)$ è simmetrico). Siano i e j gli indici dell'ultimo carattere nella LCNCS tale per cui $T[i] = U[j]$. Sappiamo che $i \leq n-1$ e $j \leq m$. Quindi, esiste una LCNCS in $T(n-3)$ e $U(m-2)$ che ha lunghezza maggiore di t. Ma questo è assurdo, perché $T(n-3)$ e $U(m-2)$ sono prefissi di $T(n-2)$ e $U(m-2)$, la cui LCNCS ha lunghezza t.

La formula può quindi essere semplificata nel modo seguente:

$$DP[i][j] = \begin{cases} 0 & i \leq 0 \ \vee j \leq 0 \\ DP[i-2][j-2]+1 & i > 0 \wedge j > 0 \wedge T[i] = U[j] \\ max\{DP[i-1][j], DP[i][j-1]\} & i > 0 \wedge j > 0 \wedge T[i] \neq U[j] \end{cases}$$

Il codice per risolvere il problema è il seguente:

```
int lancs(Item[] T, Item[] U, int n, int m)
    int[][] DP = new int[0...n][0...m] = {0}
    for i = 1 to n do
        for j = 1 to m do
            if T[i] == U[j] then
                if i == 1 or j = 1 then
                    DP[i][j] = 1
                else
                    DP[i][j] = DP[i − 2][j − 2] + 1
            else
                DP[i][j] = max(DP[i − 1][j], DP[i][j − 1])
    return DP[n][m]
```

La complessità è $\Theta(nm)$, a causa dei due cicli **for**.