Algoritmi e Strutture Dati – 17/01/23

Esercizio A1 – Complessità – Punti ≥ 8

L'equazione di ricorrenza associata al codice è la seguente:

$$T(n) = \begin{cases} 1 & n < 4 \\ 2T(3/4n) + 1 & n \geq 4 \end{cases}$$

Sebbene nel caso base si utilizzi la funzione sum(), viene applicata su un numero costante di elementi, quindi ha costo costante. Il codice divide il vettore in quattro quarti e applica la funzione ricorsiva sui primi tre quarti e sui secondi tre quarti, da cui il fattore moltiplicativo 2.
Applicando il Master Theorem, si nota che $a = 2$, $b = 4/3$, $\alpha = \log_{4/3} 2$, $\beta = 0$, quindi la complessità è pari a $\Theta(n^{\log_{4/3} 2})$. L'esponente può essere approssimato dal valore 2.4.

Esercizio A2 – Alberto binario massimale – Punti ≥ 10

Il problema può essere risolto con un approccio divide-et-impera nel modo seguente: si considerano i sottovettori con indici compresi fra i e j. Sono dati tre casi:

- se il sottovettore è vuoto, si restituisce un albero **nil**;
- altrimenti,
  - si cerca l'indice m del massimo tramite la funzione argmax(A, i, j), con costo computazionale $O(j - i + 1)$;
  - si crea un nuovo nodo che funge da radice;
  - si richiama ricorsivamente l'algoritmo sui valori a sinistra del massimo (compresi fra i e $m - 1$), inserendo il sottoalbero così ottenuto nel figlio sinistro;
  - si richiama ricorsivamente l'algoritmo sui valori a destra del massimo (compresi fra $m + 1$ e j), inserendo il sottoalbero così ottenuto nel figlio destro.

  Poichè uno o entrambi gli intervalli $[i, m - 1]$ e $[m + 1, j]$ possono essere vuoti, ci affidiamo al caso base per ottenere un nodo **nil** in queste situazioni.

```
TREE maxTree(int[] A, int n)
    return mtRec(A, 1, n)
```

```
TREE mtRec(int[] A, int i, int j)
    if j < i then
        return nil
    else
        int m = argmax(A, i, j)
        TREE t = Tree(A[m])
        t.insertLeft(mtRec(A, i, m − 1))
        t.insertRight(mtRec(A, m + 1, i))
        return t
```

Nel caso ottimo, il vettore viene costantamente diviso in due parti uguali. Questo corrisponde, per esempio, al caso in cui l'albero binario risultante corrisponda ad un albero max-heap (ma non solo). In questo caso, la complessità dell'algoritmo può essere espressa dalla seguente relazione di ricorrenza:

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

che corrisponde a $\Theta(n \log n)$, per il Teorema delle Ricorrenze lineari con partizione bilanciata.
Nel caso pessimo, il vettore è ordinato in senso crescente o decrescente, e quindi il valore massimo si trova sempre in ultima o prima posizione. In questo caso, la complessità è data dalla seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n - 1) + n & n > 1 \end{cases}$$

che corrisponde a $\Theta(n^2)$, per il Teorema delle ricorrenze lineari di ordine costante.
Risolvere l'algoritmo in questo modo ed effettuare correttamente l'analisi da origine ad un voto pari a 100%.

Soluzione con Segment Tree Per completezza, illustriamo una soluzione basata su Segment Tree, una struttura dati che non fa parte del programma del corso. Un Segment Tree è una struttura dati dinamica che permette, fra le altre cose, di memorizzare un vettore di valori e di rispondere rapidamente a domande del tipo "qual è il valore massimo compreso fra gli indici i e j?". Le funzionalità dinamiche non ci interessano qui; assumiamo quindi che il Segment tree disponga di due metodi:

- SegmentTree(**int**[] A, **int** n), un costruttore che prende in input un vettore A di dimensione n e costruisce un Segment Tree contenente tali valori;
- **int** query(**int** i, **int** j), un metodo che restituisce l'indice del valore massimo contenuto nel sottovettore $A[i \dots j]$;

Possiamo quindi modificare il codice sopra in questo modo:

```
TREE maxTree(int[] A, int n)
    SEGMENTTREE ST = SegmentTree(A, n)
    return mtRec(A, ST, 1, n)
```

```
TREE mtRec(int[] A, SEGMENTTREE ST, int i, int j)
    if j < i then
        return nil
    else
        int m = ST.query(i, j)
        TREE t = Tree(A[m])
        t.insertLeft(mtRec(A, ST, i, m − 1))
        t.insertRight(mtRec(A, ST, m + 1, i))
        return t
```

Nel caso ottimo, il vettore viene costantemente diviso in due parti uguali. Questo corrisponde, per esempio, al caso in cui l'albero binario risultante corrisponda ad un albero max-heap (ma non solo). In questo caso, la complessità dell'algoritmo può essere espressa dalla seguente relazione di ricorrenza:

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

che corrisponde a $\Theta(n)$ (per la versione estesa del Teorema delle Ricorrenze lineari con partizione bilanciata).
Nel caso pessimo, il vettore è ordinato in senso crescente o decrescente, e quindi il valore massimo si trova sempre in ultima o prima posizione. In questo caso, la complessità è data dalla seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n-1) + \log n & n > 1 \end{cases}$$

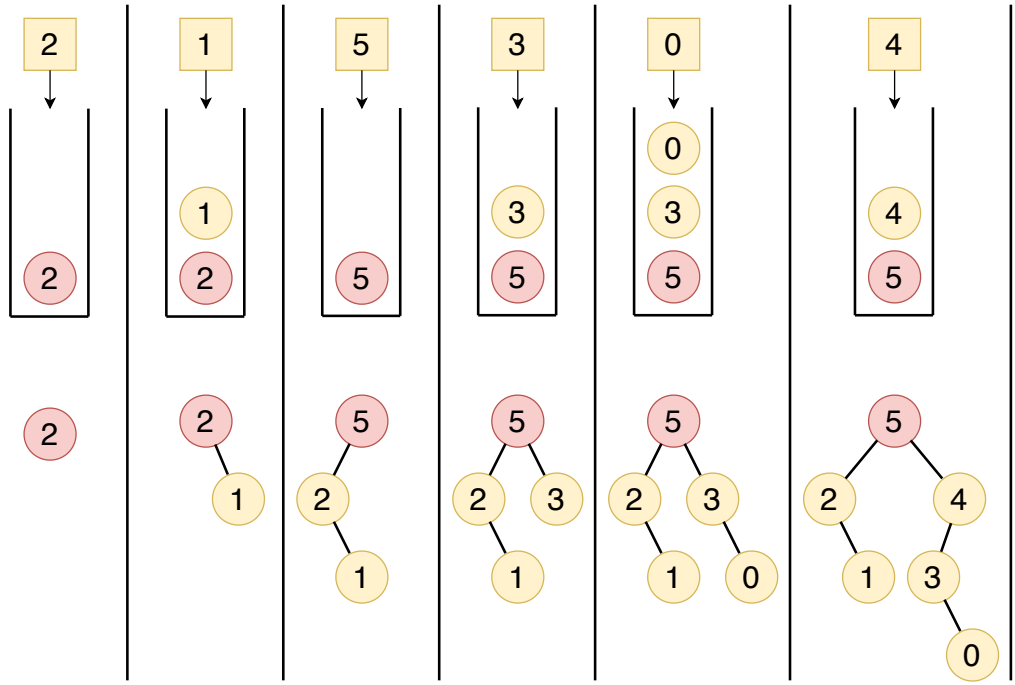
che corrisponde a $O(n \log n)$, estendendo l'interpretazione del Teorema delle ricorrenze lineari di ordine costante e successivamnente provandolo per sostituzione.
Gli studenti che conoscevano questa struttura dati e l'hanno usata correttamente nella loro soluzione, hanno preso 110%. Per chi è interessato, possibili riferimenti per comprendere il funzionamento dei SegmentTree sono i seguenti:

- https://cp-algorithms.com/data_structures/segment_tree.html#counting-zero-search-kth
- https://www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/tutorial/

Costruzione lineare dell'albero Come a volte capita, studiando meglio quello che era un esercizio scritto "al volo" per testare la vostra conoscenza della tecnica divide-et-impera, ho scoperto poi che questi alberi sono detti alberi cartesiani e hanno numerose applicazioni: https://en.wikipedia.org/wiki/Cartesian_tree.
Esiste una soluzione "ad hoc" per costruire tali alberi che opera in tempo lineare in tutti i casi. Al momento della stesura del compito, non ne ero al corrente. È stata proposta da alcuni studenti; propongo qui una versione ispirata a questa pagina web: https://leetcode.com/problems/maximum-binary-tree/solutions/106146/C++-O(N)-solution/.

```
TREE maxTree(int[] A, int n)
    STACK S = Stack()
    for i = 1 to n do
        TREE curr = Tree(A[i])
        TREE last = nil
        while S.size() > 0 and S.top() < A[i] do
            last = S.pop()
        if last ≠ nil then
            last.left = curr
            curr.parent = last
        if S.size() > 0 then
            TREE t = S.top()
            t.right = curr
            curr.parent = t
        S.push(curr)
    while S.size() > 1 do
        S.pop()
    return S.top()
```

Nella figura si può vedere un'esecuzione per il vettore $[2, 1, 5, 3, 0, 4]$.



La correttezza si prova tramite tre invarianti:

1. l'albero costruito al termine del passo i-esimo è un albero massimale definito sui primi i valori;
2. la sua radice si trova in fondo allo stack;
3. estraendo i nodi dallo stack ad esclusione della radice, si ottiene una sequenza di valori crescenti, contenuti in nodi che sono figli destri dei loro genitori.

Dimostrazione:

- **Inizializzazione**: Al termine del ciclo con $i = 1$, abbiamo un elemento nell'albero, ovvero la sua radice, che si trova in fondo allo stack. La terza proprietà è banalmente vera, perché non ci sono altri nodi oltre la radice.
- **Conservazione**: Supponiamo che le invarianti siano vere per $i - 1$, vogliamo dimostrare che sono vere anche per i.
  - Se il valore $A[i]$ è maggiore di tutti i valori precedenti, tutti i valori presenti nello stack vengono estratti dal ciclo **while** all'interno del ciclo **for**; il nuovo nodo viene inserito nello stack e la radice dell'albero precedente viene collegata come figlio sinistro della nuova radice.

– Altrimenti, sia $A[k]$ il primo valore più grande di $A[i]$; il ciclo **while** termina e il nuovo nodo diventa figlio destro del nodo contenente $A[k]$. Il nodo che era precedentemente figlio destro del nodo contenente $A[k]$ diventa figlio sinistro del nodo contenente $A[i]$. Questo perchè tutti i valori compresi nell'intervallo $A[k+1 \dots i-1]$ sono minori di $A[i]$ e si trovano prima di $A[i]$, quindi vanno inseriti a sinistra.

- **Conclusione**: al termine del passo n-esimo, l'albero così definito è un albero massimale sui tutti gli n valori, la cui radice si trova nello stack e può essere così restituita.

L'algoritmo è lineare perchè ogni nodo viene inserito ed estratto nello stack esattamente una volta. Quindi domina la complessità del ciclo for, che è lineare.

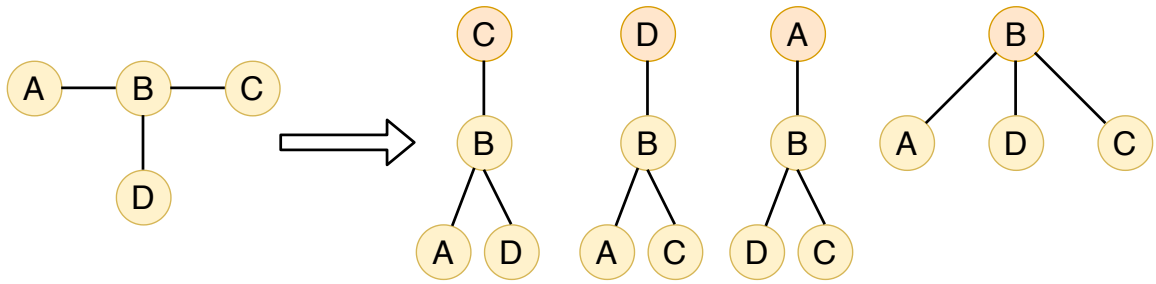
Ovviamente, una soluzione del genere non è assolutamente richiesta per il compito; gli studenti che l'hanno proposta hanno preso 110%.

Esercizio A3 – Albero di altezza minima – Punti ≥ 12

La prima soluzione che può venire in mente è quella di considerare ogni nodo come possibile radice ed eseguire una BFS a partire da esso, misurando le distanze. Si prende poi la distanza minima fra le distanze massime.

```
int minHeigth(Graph G)
    int[] distance = new int[1...n]
    int minSoFar = +∞
    foreach u ∈ G.V() do
        distance(G, u, distance)
        int height = max(distance)
        minSoFar = min(minSoFar, height)
    return minSoFar
```

Il problema di questa soluzione è che considera anche nodi che non possono essere radici di alberi binari, ovvero nodi che hanno tre vicini. Si consideri l'esempio seguente:



L'algoritmo visto sopra restituisce 1 come altezza minima, prendendo come radice B, ma questa è l'altezza minima di una qualunque alberizzazione, mentre tutti gli alberi binari hanno altezza 2 e quindi l'algoritmo deve restituire tale valore. Quindi l'algoritmo corretto è:

```
int minHeigth(Graph G)
    int[] distance = new int[1...n]
    int minSoFar = +∞
    foreach u ∈ G.V() do
        if G.adj(u).size() < 3 then
            distance(G, u, distance)
            int height = max(distance)
            minSoFar = min(minSoFar, height)
    return minSoFar
```

Il costo computazionale è quello di n visite in ampiezza di costo $\Theta(m+n)$. Poiché in un albero abbiamo che $m = n-1$, la complessità effettiva è $O(n^2)$. Una soluzione di questo tipo con la relativa analisi di complessità, evitando quindi di considerare nodi con tre figli, prende 100%.

Un'analisi più raffinata È possibile domandarsi quante visite in ampiezza sono necessarie, ovvero se il costo computazionale è $\Omega(n^2)$ o più basso, tenuto conto che la visita non viene effettuata quando il grado del nodo è pari a 3. Per saggiare il problema, consideriamo due casi particolari:

- nel caso degli alberi completi, in cui tutti i livelli sono riempiti tranne l'ultimo, il numero nodi con degree 1 è pari a $\lfloor n/2 \rfloor$, un nodo ha degree 2, quindi la maggioranza dei nodi ha grado < 3 e vengono effettuate un numero lineare di visite in ampiezza.

- nel caso dell'albero lineare, in cui tutti i nodi hanno grado 2 tranne due (che hanno grado 1), vengono effettuate n visite in ampiezza.

Ora, questi costituiscono gli estremi rispetto all'altezza - con altezza $\lceil \log n \rceil$, effettuiamo $\approx n/2$ visite; con altezza $n-1$, effettuiamo esattamente n visite. Verrebbe da dire che qualunque albero fra questi due estremi avrà un numero di nodi con grado < 3 comunque lineare, ma questa non è una dimostrazione.

Siano n_1, n_2, n_3 il numero di nodi con grado 1, 2, 3, rispettivamente. Sappiamo che:

- la somma di questi tre valori è pari a n:
$$n_1 + n_2 + n_3 = n$$
- la somma dei gradi di tutti i nodi è pari a $2n-2$ (il doppio degli archi); quindi
$$n_1 + 2n_2 + 3n_3 = 2n - 2$$
- sostituendo n_2 con $n - n_2 - n_3$ (ottenuto dalla prima equazione) nella seconda, otteniamo:
$$\begin{aligned} n_1 + 2n_2 + 3n_3 &= 2n - 2 \\ n_2 + 2n - 2n_1 - 2n_3 + 3n_3 &= n - 2 \\ n_3 &= n_1 - 2 \end{aligned}$$
- n_1 deve essere minore o uguale a $\lceil n/2 \rceil$, che corrisponde al numero di foglie in un albero completo con n nodi, che è quello con il maggior numero di foglie fra gli alberi con n nodi;
- da cui deduciamo che $n_3 \leq \lceil n/2 \rceil - 2$ e quindi il numero di nodi da cui far partire la visita è lineare nella dimensione n.

Una soluzione più efficiente, ma comunque sbagliata Sia k il diametro del grafo G, ovvero il più lungo cammino breve fra tutte le coppie di nodi (misurato in numero di archi). Essendo G un albero binario, l'albero binario di altezza minima avrà la radice lungo questo cammino, con metà degli archi di questo cammino in un sottoalbero e metà nell'altro (se gli archi sono dispari, $\lceil k/2 \rceil$ da una parte, $\lfloor k/2 \rfloor$ dall'altra). Quindi la risposta al nostro quesito è $\lceil k/2 \rceil$. Il problema può essere quindi trasformato nel trovare il diametro del grafo. Per grafi generali, il costo per trovare il diametro è $O(n(n+m))$ (vedi esercizi di laboratorio), nel nostro caso $O(n^2)$, quindi la complessità è simile a quella precedente.
Esiste un algoritmo per trovare il diametro di un albero che funziona in questo modo: si fa una visita in ampiezza di un nodo qualunque x. Si prende uno nodo y a distanza massima da x. Si effettua una nuova visita in ampiezza da y. La distanza massima che si ottiene da questa seconda visita è il diametro k dell'albero. L'algoritmo restituisce quindi altezza $\lceil k/2 \rceil$.
Il problema di questo approccio è che soffre dello stesso problema della visita a partire da tutti i nodi: può selezionare un diametro che corrisponde ad un albero generale, non a un albero binario. Si prenda il grafo mostrato sopra: partendo da un nodo a caso, si arriva ad uno dei nodi con grado 1 come nodi a distanza massima. Partendo da uno di questi nodi per la seconda visita, si ottiene un diametro pari a 2. Quindi l'algoritmo restituisce 1, che è sbagliato.
Nonostante l'algoritmo non sia corretto, ho deciso di dare 110% a chi ha proposto questa soluzione, perché come sopra, dimostra la conoscenza di concetti più avanzati; inoltre, la specifica "albero binario" era stata creata per complicare l'algoritmo $O(n^2)$, senza considerare che avrebbe complicato anche questo algoritmo.
Questo compito è "nascosto" perché verrà utilizzato come simulazione compito durante la primavera del 2024. So che avete altri modi per accedere ai vecchi compiti, ma vi consiglio di non farlo prima. L'idea è di assegnarvi il compito in aula, lo fate, mi mandate le soluzioni, io ve le correggo con calma prima del compito vero.