מטלה 4 מבני נתונים

איתמר קרייטמן ת.ז 208925578

שאלה 1:

<u>סעיף א:</u>

האיבר המקסימלי בערימה יהיה באחד העלים.
מכיוון שמדובר בערימת מינימום, כל צומת קטן מבניו, ולכן ברמה האחרונה (הרמה של העלים) יהיו הערכים הגדולים ביותר בכל תת ערימה, מכיוון שאנו לא יודעים כלום על סדר האיברים בכל רמה אלא רק ביחס לרמה הקודמת נוכל לומר כי האיבר המקסימלי יהיה באחד מהעלים אך לא במדויק באיזה עלה.

<u>סעיף ד:</u>

(mergeTwoHeaps- complexity O(n *

```java
//**Q1b**
//O(n+k) + O(n) + O(k), O(n)
public static MaxHeap mergeTwoHeaps(MaxHeap h1, MaxHeap h2) {
    MaxHeap ansHeap = new MaxHeap( size: h1.size + h2.size);//array represent the new heap
/     pointers to run threw h1 and h2, starting from 0
    int h1Iterator = 0;
    int h2Iterator = h1.size;
/     adding elements from both max heaps arrays to one array
    while (h1Iterator < h1.size) {//O(n)
        ansHeap.arr[h1Iterator] = h1.arr[h1Iterator];
        h1Iterator++;
        ansHeap.last++;
    }
    h1Iterator = 0;
    while (h2Iterator < ansHeap.size) {//O(k)
        ansHeap.arr[h2Iterator] = h2.arr[h1Iterator];
        h2Iterator++;
        h1Iterator++;
        ansHeap.last++;
    }
/     building max heap from the array, we proved in the lecture it takes O(arr.length)
    for (int i = ansHeap.arr.length / 2 - 1 ; i >= 0; i--) {//O(n+k)
        ansHeap.Heapify_down(i);
    }
    return ansHeap;
```

הפונקציה מקבל כקלט שני ערימות מקסימום, נגדיר את $h1.size = n,\ h2.size = m$.
בשלב ראשון הפונקציה מכניסה את איברי $h1$ למערך חדש בגודל $n+m$, מעבר על $h1$ עולה $O(n)$ ולאחר מכן את איברי $h2$, מעבר על $h2$ עולה $O(m)$. בסך הכל $O(n)+O(m)$.
בשלב שני בונים maxHeap מאותו מערך, בנייה זו נעשית ב$O(n+m)$.
הוכחה:
פעולת heapify_down לוקחת $O(k)$ לכן במבט ראשון נראה כי הסיבוכיות היא בסך הכל $O(k \log k)$. אך

למעשה היא $O(k)$ כאשר k הוא גודל המערך החדש.
הוכחה:

$$T(k) \leqslant \frac{k}{2}\cdot 0+\frac{k}{4}\cdot 1+\frac{k}{8}\cdot 2+\dots +1\cdot log\ k \leqslant k\cdot \left[\frac{1}{4}+\frac{2}{8}+\frac{3}{16}+\dots\right]$$
$$\leqslant k\cdot \left[\left(\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\dots\right)+\left(\frac{1}{8}+\frac{1}{16}+\dots\right)+\left(\frac{1}{16}+\dots\right)\right]$$
$$\leqslant k\cdot \left[\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\dots\right] = k$$

לכן בסך הכל יש לנו סיבוכיות של $O(n)+O(m)+O(n+m)+O(k)=O(n)$.

* getMinHeap- complexity O(n logn)

```java
//**Q1c**
//O(nlogn)
public static int[] getMinHeap(MaxHeap h) {
    Arrays.sort(h.arr);//O(nlogn)
    int[] minHeap = Arrays.copyOf(h.arr, h.arr.length);//O(n)
    return minHeap;
}
```

הפונקציה מקבלת כקלט ערימת מקסימום h, ממיינת את המערך שלה בסיבוכיות של $O(n\ logn)$ בסדר עולה וממנו בונה ערימת מקסימום על ידי יצירת עותק של המערך הממויין $O(n)$ המערך הממויין החדש מייצג ערימת מינימום.
בסך הכל $O(n\ logn)+O(n) = O(n\ logn)$.

שאלה 2:

```java
public static List<int[]> ThreeSum(int[] arr, int target) {
    HashSet<HashSet<Integer>> triples = new HashSet<>();
    LinkedList<int[]> answer = new LinkedList<>();
    HashSet<Integer> arrAsHash = new HashSet<>();
    Arrays.sort(arr);//O(nlogn) average
    for (int p = 0; p < arr.length; p++) {//O(n)
        arrAsHash.add(arr[p]);
    }
    for (int i = 0; i <= arr.length - 3; i++) {//O(n)
        for (int j = i + 1; j < arr.length - 1; j++) {//O(n)
            if (arrAsHash.contains(target - arr[i] - arr[j])) {//O(1)
                int temp = target - arr[i] - arr[j];
                if (temp != arr[i] && temp != arr[j] && arr[i] != arr[j]) {
                    HashSet<Integer> matchTriple = new HashSet<>();
                    matchTriple.add(arr[i]);//O(1)
                    matchTriple.add(arr[j]);//O(1)
                    matchTriple.add(temp);//O(1)
                    if (!triples.contains(matchTriple)) {//O(1)
                        triples.add(matchTriple);//O(1)
                        answer.addLast(new int[]{arr[i], arr[j], temp});//O(1)
                    }
                }
            }
        }
    }
    return answer;
}
```

סיבוכיות זמן הריצה- $O(n^2)$.

הכנסת כל איברי המערך ל$HaseSet$ עולה $O(n)$.

נשים לב שבתוך הלולאה הפנימית כל הפעולות הן פעולות פשוטות שעולות $O(1)$.

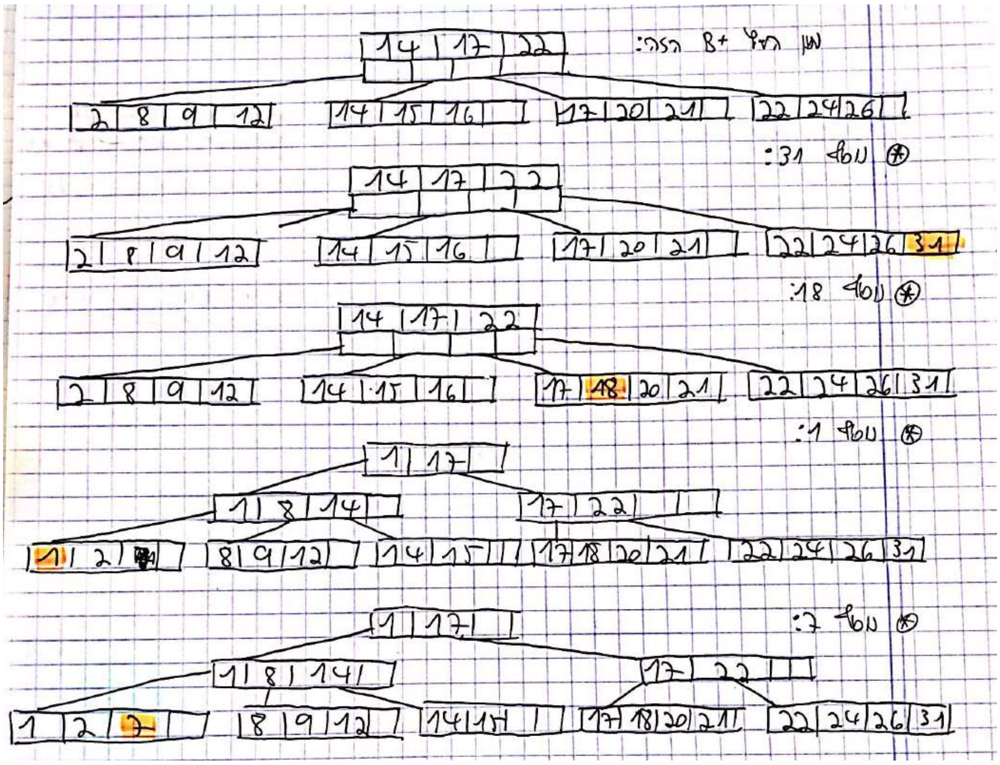
נחשב את זמן הריצה כל הלולאות: נסמן $n = arr.length$.

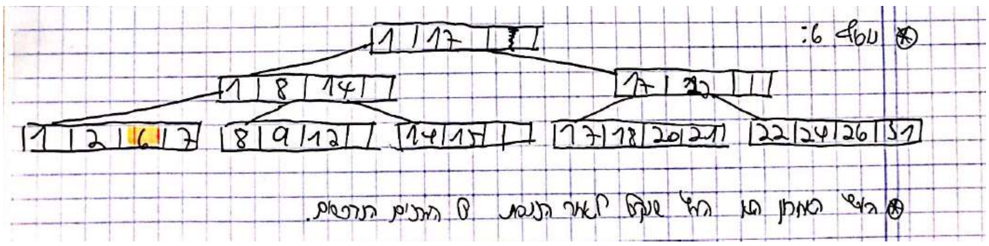
$$T(n) = \sum_{i=0}^{n-3} \sum_{i+1}^{n-1} 9 = 9 \cdot (n-2+n-3+n-4+ \ldots +n-n-1)$$

$$= 9 \cdot (n+n+n \ldots -n) = 9 \cdot (n \cdot (n-1)) = 9n^2 - n = O(n^2)$$

בסך הכל: $O(n)+O(n^2) = O(n^2)$

שאלה 3:





שאלה 4:

```java
public UnionFind(int size, double angle) {
    elements = Ex4Utils.generateRandomArray(size);//O(size)
    this.size = new int[size];
    id = new int[size];
    for (int i = 0; i < size; i++) {//O(size)
        id[i] = i;
        this.size[i]++;
    }
    this.angle = angle;
    UnionByAngularDist(new Point( x: 50, y: 50));
}
```

ננתח את זמן הריצה של הבנאי:
genreateRandomArray מאתחל מערך חדש של נקודות בגודל size, בריצה על i מ0 ועד size ובסך הכל זמן הריצה שלו הוא $O(size)$.
לאחר מכן מערך האבות id מאותחל כך שכל נקודה היא האב של עצמה, בריצה על i מ0 ועד size ובסך הכל $O(size)$.
UnionByAngularDist רץ ב$O(size\ \cdot log\ szie)$ (יוכח בהמשך) ולכן בסך הכל:

$$O(size) + O(size) + O(size \cdot log\ size) = 2 \cdot O(size) + O(size \cdot log\ size)$$
$$= O(size \cdot log\ size) = O(n\ logn)$$

```java
public void Union(int ind1, int ind2) {
    int parentNumInd1 = Find(ind1);//O(log n)
    int parentNumInd2 = Find(ind2);//O(log n)
    if (parentNumInd1 != parentNumInd2) {
        if (this.size[parentNumInd1] < this.size[parentNumInd2]) {
            id[parentNumInd1] = parentNumInd2;
            this.size[parentNumInd2] = this.size[parentNumInd2] + this.size[parentNumInd1];
        }
        else {
            id[parentNumInd2] = parentNumInd1;
            this.size[parentNumInd1] = this.size[parentNumInd2] + this.size[parentNumInd1];
        }
    }
}
```

```java
public int Find(int p) {//O(log n)
    if (p != id[p])
        id[p] = Find(id[p]);
    return id[p];
}
```

במימוש לפי דחיסת מסלול, שגורמת לכל צומת בחיפוש להיות הבן של שורש העץ, זמן הריצה ישאר $O(logn)$ אך נשים לב שרוב הפעולות יתבצעו במהירות של כמעט קבוע!
נוכיח תחילה כי Union רץ ב$O(log\ h)$ במימוש Union by Weight כאשר h הוא גובה העץ, סיבוכיות Union היא כגובה העץ.
נוכיח באינדוקציה:
בסיס: עבור עץ במשקל $s = 1$ נקבל $log\ s\ =\ 0$ ואכן הגובה של העץ הוא 0.
צעד: נניח כי הטענה נכונה עבור עץ עם משקל גדול מk, ונוכיח שעבור שני עצים הטענה מתקיימת גם עבור האיחוד שלהם.
נגדיר שני עצים, יהיו s_1, s_2 משקלי העצים וגובהם h_1, h_2. נסמן את משקל העץ המאוחד כ$s = s_1 + s_2$.
נניח בה"כ כי $s_1 > s_2$. לפי הנחת האינדוקציה $h_1 \leqslant log\ s_1$ ו$h_2 \leqslant log\ s_2$.
לאחר איחוד העצים גובה העץ החדש הוא המקסימלי מבין שני הגבהים.
נשים לב כי $s > s_1, s_2$ ולכן גובה העץ החדש שנסמן אותו בh יהיה-
$h = max(log\ s_1, log\ s_2) \leqslant max(logs, logs) = logs$.

כעת נוכיח כי Find רץ ב$O(log\ n)$ על ידי שימוש במימוש דחיסת מסלול. כאשר ממשים Union בUnion by Weight העץ תמיד נשאר מאוזן, ולכן כמו שלמדנו מציאת האב תיקח במקסימום כגובה העץ שהוא log n ולכן זמן הריצה של Find הוא $O(log\ n)$.

```java
public void doisJoin() {//O(h) when h is tree height
    for (int i = 0; i < this.size.length; i++) {
        this.id[i] = i;
        size[i] = i;
    }
}
```

תפקיד הפונקציה doIsJoin הוא להחזיר את כל הנקודות לקבוצות זרות (המקוריות שלהן) ובעצם לשייך כל נקודה i לקבוצה מספר i. לכן מה שנדרש הוא מעבר על מערך האבות ולהחזיר כל אינדקס להצביע על עצמו.
מעבר על מערך האבות (id) מתבצע ב$O(n)$ כאשר n הוא גודל המערך.

```java
public void increaseAngle(int d) {//O(size * log n)
    angle += d;
    UnionByAngularDist(new Point( x: 50, y: 50));
}
```

פונקצית increaseAngle רצה בזמן של $O(size\ \cdot log\ size)$. מכיוון שלאחר פעולת הוספת d לangle שהיא ב $O(1)$ היא מפעילה את UnionByAngularDist.

הוכחה תעשה להלן בהוכחת זמן הריצה של UnionByAngularDist:

```java
public void UnionByAngularDist(Point p) {//O(size * log size)
    Hashtable<Integer, Integer> check = new Hashtable<>();
    for (int i = 0; i < this.elements.length; i++) {//O(size)
        double newAngle = Ex4Utils.angleFrom(p,elements[i]);//O(1)
        if (!check.containsKey((int) (newAngle / angle))) {
            int newGroup = (int) (newAngle / angle);
            check.put(newGroup ,i);//O(1)
        }
        else {
            Union(check.get((int) (newAngle / angle)) ,i);//O(log size)
        }
    }
}
```

UnionByAngularDist רצה בזמן של $O(size \cdot log\ size)$.
לולאת for שרצה על מערך elements בגודל size תרוץ size פעמים.
פעולות שמבוצעות על check יתבצעו ב$O(1)$ לפי תכונות haseTable פעולת Union כמו שהוכחנו לעיל רצה בזמן של $O(log\ size)$ כאשר במקרה הגרוע היא תתבצע $O(size)$ ולכן בסך הכל:
$O(size) \cdot\ O(log\ size) = O(size \cdot log\ size) = O(n\ logn)$.