4. i.1. המקרה הגרוע ביותר מתרחש כאשר X לא נמצא במערך ולכן יש לעבור על כל המערך בחיפוש אחריו. נחלק את חישוב החסם העליון לצעדים קדימה + צעדים אחורה כאשר n הוא גודל המערך:

מספר הצעדים קדימה הוא סכום של סדרה חשבונית שמתחילה מ-1 ועד לאמצע המערך+1 מכיוון שכשמגיעים לאמצע המערך האינדקס מתקדם $\left\lfloor\frac{n}{2}\right\rfloor$ צעדים קדימה אך עדיין לא יוצא מגבולות המערך ולכן נותרת איטרציה נוספת שתוביל לחריגה מגבולות המערך וכתוצאה מכך להפסקת הפונקציה. ההתקדמות אחורה מתבצעת $\left\lfloor\frac{n}{2}\right\rfloor$ פעמים כאשר בכל פעם מתבצע מספר גדול ב-1 של צעדים.

החישוב עבור צעדים קדימה הוא : $1+\left\lfloor\frac{n}{2}\right\rfloor + \dots + 3 + 2 + 1$ כלומר :

$$\sum_{i=1}^{\left\lfloor\frac{n}{2}\right\rfloor+1} i = \frac{(1+(\left\lfloor\frac{n}{2}\right\rfloor+1)(\left\lfloor\frac{n}{2}\right\rfloor+1))}{2}$$

והחישוב עבור צעדים אחורה הוא: $\left\lfloor\frac{n}{2}\right\rfloor + \dots + 3 + 2 + 1 + 0$ כלומר :

$$\sum_{i=0}^{\left\lfloor\frac{n}{2}\right\rfloor} i = \frac{(0+\left\lfloor\frac{n}{2}\right\rfloor)(\left\lfloor\frac{n}{2}\right\rfloor+1))}{2}$$

נחבר את מספר הפעולות:

$$\frac{(1+(\left\lfloor\frac{n}{2}\right\rfloor+1)(\left\lfloor\frac{n}{2}\right\rfloor+1))}{2} + \frac{(0+\left\lfloor\frac{n}{2}\right\rfloor)(\left\lfloor\frac{n}{2}\right\rfloor+1))}{2} = \left\lfloor\frac{n}{2}\right\rfloor^2 + 2\left\lfloor\frac{n}{2}\right\rfloor + 1$$

נמצא חסם עליון עבור הביטוי הנ"ל : $\left\lfloor\frac{n}{2}\right\rfloor^2 + 2\left\lfloor\frac{n}{2}\right\rfloor + 1 < 10 \cdot n^2, C = 10, n_o = 1$

נמצא חסם תחתון: $\frac{1}{10} \cdot n^2 < \left\lfloor\frac{n}{2}\right\rfloor^2 + 2\left\lfloor\frac{n}{2}\right\rfloor + 1, C = \frac{1}{10}, n_o = 1$

ולכן זמן הריצה של פונקציה זו עבור המקרה הגרוע ביותר מתאים ל $\Omega(n^2)$.

הערה: כל צעד, כלומר גישה למערך מתבצע בזמן ריצה של O(1) לכן כל הפעולות שנספרו היו שוות והיה ניתן לחברן.

ii. במקרה הגרוע ביותר נעשה מעבר על כל המערך בזמן החיפוש ו-X אינו נמצא במערך. בכל שלב באיטרציה מתבצע התקדמות בפועל של צעד אחד מכיוון שהפעולה מתחילה בהתקדמות של צעד אחד קדימה ולאחר מכן Y צעדים אחורה ומיד לאחר מכן Y צעדים קדימה כאשר Y תלוי ב-i וגם מספר הפעמים ש-Y מתבצע תלוי ב-i כאשר i מסמל את מספר האיטרציה. בסה"כ בכל איטרציה מתבצעים Y+Y+1 פעולות, כל פעולה היא בזמן ריצה של צעד במערך כלומר גישה למערך ומכיוון שגישה ישירה לתא במערך מתבצעת בזמן ריצה של O(1) כך גם כל אחת מ 2Y + 1 הפעולות שמתבצעות בכל איטרציה. נסמן ב-n את גודל המערך ונקבל ש-i רץ מ-1 עד n כאשר בכל i נסכמים הצעדים i-2 + ... 3 + 2 + 1 (מוכפל ב-2 עבור צעדים אחורה וקדימה) ובנוסף בכל איטרציה נסכם צעד בודד של התקדמות קדימה לכן הנוסחה המתקבלת:

$$Y = \sum_{j=1}^{i-2} j$$

$$T(n)=\sum_{i=1}^{n}\left(1 + \sum_{j=1}^{i-2} j + \sum_{j=1}^{i-2} j\right) = \sum_{i=1}^{n}\left(1 + 2\sum_{j=1}^{i-2} j\right) =$$

$$f(n)=\sum_{i=1}^{n} 1 + 2\sum_{i=1}^{n}\sum_{j=1}^{i-2} j = n + 2\sum_{i=1}^{n}\frac{(i-1)(i-2)}{2} = n + \sum_{i=1}^{n} i^2 - 3i + 2 = n + \frac{n^2+n}{2} + \frac{2n^3+3n^2+n}{6} - \frac{3(n+1)n}{2} + 2n = 3n + \frac{1}{2}\left(-2n^2 - 2n + \frac{2n^3+3n^2+n}{3}\right) = \boldsymbol{2n - n^2 + \frac{1}{6}(2n^3 + 3n^2 + n)}$$

נמצא חסמים עליון:

$$f(n) < 5n^3, c = 5, n_o = 1$$

נמצא חסם תחתון:

$$f(n) > \frac{1}{10}n^3, c = \frac{1}{10}, n_o = 1$$

לכן זמן הריצה של פונקציה זו עבור המקרה הגרוע ביותר מתאים ל $\Omega(n^3)$.

iii. חישוב זמן הריצה עבור המקרה הגרוע ביותר הינו כאשר X לא נמצא במערך ולכן יש לעבור על כל המערך בחיפוש אחריו ובנוסף כאשר חוזרים אחורה מספר מקסימלי של צעדים לאחר כל התקדמות קדימה, כלומר חוזרים n2 צעדים אחורה n2=n1-1 כאשר n1= מס' הצעדים קדימה ו- n= גודל המערך. הנוסחה לחישוב כמות הצעדים קדימה הינה:
(n+2-n1)n1 - 1 כאשר הביטוי שבתוך הסוגריים מייצג את כמות הפעמים שצריך לבצע התקדמות קדימה ומכיוון שבכל פעם שצריך להתקדם קדימה ההתקדמות נעשית n1 פעמים כלומר צעדים, מכפילים את מספר האיטרציות קדימה במספר הצעדים של כל איטרציה.
מוסיפים 2 עבור האיטרציה הראשונה שמתחילה מחוץ למערך ועבור האיטרציה האחרונה שעוקפת את המערך. ובנוסף מחסירים n1 מכיוון שלא מתבצעת התקדמות קדימה בצעדים האחרונים (כי עוברים את גבולות המערך והפיצוי על הצעד האחרון שכן מחושב מתבצע בעזרת ההוספה וההחסרה) מחסירים 1 מחוץ לסוגריים מכיוון שבאיטרציה האחרונה מתקדמים n1-1 צעדים קדימה ולא את כל n1 הצעדים.
ומספר הצעדים אחורה הוא: n2(n+1-n1) הביטוי שבתוך הסוגריים מייצג את כמות הפעמים שמתבצעת איטרציה אחורה ולכן מכפילים בכמות הצעדים אחורה שמתבצעת בכל איטרציה.
הביטוי שבתוך הסוגריים- מחסירים את מספר הצעדים האחרונים קדימה שלא מתבצע בהם צעדים אחורה ומוסיפים 1 עבור הפעם הראשונה שמתחילה מחוץ למערך שגם שם מתבצעים צעדים אחורה.
הערה: כל צעד, כלומר גישה למערך מתבצע בזמן ריצה של O(1) לכן כל הפעולות שנספרות שוות וניתן לחברן.
נסכום את הצעדים קדימה + הצעדים אחורה ונקבל:

(n+2-n1)n1 – 1 + n2(n+1-n1) ומכיוון ש n2=n1-1 נחליף את המשתנה:
(n+2-n1)n1 – 1 + (n1-1)(n+1-n1) = $-2n1^2 + 2n \cdot n1 - n + 4n1 - 2$
בכדי למצוא את מספר הצעדים קדימה שמביאים למספר המירבי של פעולות חיפוש נגדיר פונקציה f(n1)= $-2n1^2 + 2n \cdot n1 - n + 4n1 - 2$
נחפש נקודת מקסימום :
f'(n1)= -4n1+2n+4
נשווה ל-0 במטרה למצוא נקודת מקסימום: -4n1+2n+4 = 0
התוצאה שמתקבלת: n1=$\frac{n+2}{2}$ כלומר: $n1 = \frac{n}{2} + 1$
נראה שהיא אכן מקסימום באמצעות הנגזרת השנייה:
f''(n1)= -4
f''($\frac{n}{2} + 1$) < 0 ולכן זוהי נקודת מקסימום. כלומר קיבלנו פונקציה שמייצגת את המספר המקסימלי של צעדים :
f($\frac{n}{2} + 1$)= $\frac{n^2}{2}$ +n
נמצא חסם עליון: $\frac{n^2}{2} + n < 5n^2, c = 5, n_o = 1$
וחסם תחתון: $\frac{n^2}{5} < \frac{n^2}{2} + n, c = \frac{1}{5}, n_o = 1$
לכן זמן הריצה של פונקציה זו עבור המקרה הגרוע ביותר מתאים ל $\Omega(n^2)$.

.2i. Unsorted array:

```
public void backtrack() {
    if(!stack.isEmpty()) {                                  C1 = 1
        int myPop = (int)stack.pop();                       C2 = 1
        if(myPop==-1) {                                     C3 = 1
            int index = (int)stack.pop();                   C4 = 1
            arr[index]=arr[nextEmpty-1];                    C5 = 1
            nextEmpty--;                                    C6 = 1
        }else {                                             C7 = 1
            arr[nextEmpty]=arr[myPop];                      C8 = 1
            arr[myPop]=(int)stack.pop();                    C9 = 1
            nextEmpty++;                                    C10 = 1
        }
        System.out.println("backtracking performed");       C11 = 1
    }
```

backTrack delete:
T(n) = C1 + C2 + C3 + C4 + C5 + C6 + C11 = 7= constant = O(1)
backTrack insertion:
T(n) = C1 + C2 + C3 + C7 + C8 + C9 + C10 + C11 = 8 = constant = O(1)

ii. Sorted array:

```
public void backtrack() {
  if(!stack.isEmpty()){                                  C1 = 1
    String st = (String) stack.pop();                    C2 = 1
    if(st.equals("insert")){                             C3 = 1
      this.insert((int)stack.pop());                     C4 =(7+2n)*
      stack.pop();                                       C5 = 1
      stack.pop();                                       C6 = 1
    }     else {                                         C7 = 1
        int index = (int)stack.pop();                    C8 = 1
        this.delete(index);                              C9 =(6+2n)*
        stack.pop();                                     C10 = 1
        stack.pop();                                     C11 = 1
    }
    System.out.println("backtracking performed");        C13 = 1
  }

}
```

*C4 and C9 is analyzed separately at the next page

```
(C9) public void delete(Integer index) {
    if (!(index < 0 | index >= nextEmpty)) {                    C14 = 2
        int insert =arr[index];                                 C15 = 1
        for (int i = index; i < nextEmpty; i++) {               C16 = n+1
            arr[i] = arr[i + 1];                                C17 = n-1
        }
        stack.push(insert);                                     C18 = 1
        stack.push("insert");                                   C19 = 1
        nextEmpty--;                                            C20 = 1
    }
```

delete:
T(n) = C14 + C15 + C16 + C17 + C18 + C19 + C20 = 2 + 1 + n + 1 + n – 1 + 1 + 1 + 1 = 6 + 2n = O(n)

```
(C4) public void insert(Integer x) {
    int i;                                                      C21 = 1
    if (nextEmpty != arr.length) {                              C22 = 1
        for (i = nextEmpty - 1; i >= 0 && arr[i] > x; i--) {    C23 = n+1
            arr[i + 1] = arr[i];                                C24= n
        }
        nextEmpty++;                                            C25 = 1
        arr[i + 1] = x;                                         C26 = 1
        stack.push(i+1);                                        C27 = 1

        stack.push("delete");                                   C28 = 1
    }
```

insert:
T(n) = C21 + C22 + C23 + C24 + C25 + C26 + C27 + C28 = 1 + 1 + n + 1 + n 1 + 1 + 1 + 1 = 7 + 2n = O(n)

backTrack delete:

$T(n) = C1 + C2 + C3 + C7 + C8 + C9 + C10 + C11 + C13 = 1 + 1 + 1 + 1 + 1 + 7 + 2n + 1 + 1 = 14 + 2n$

$\Omega(n) = n < 14 + 2n < 17n = O(n)$, $n_o=1$, $C_1=1$,$C_2=17$

backTrack insertion:

$T(n) = C1 + C2 + C3 + C4 + C5 + C6 + C13 = 1 + 1 + 1 +7 + 2n + 1 + 1 + 1= 13 + 2n$

$\Omega(n) = n < 13 + 2n < 16n = O(n)$, $n_o=1$, $C_1=1$,$C_2=16$

iii. Backtrack BST based

1. **Delete**

   The only case there is "delete" action in backtrack stack is after the "insert" function has been called. This function inserts a new node to its place at the "end" of the tree (the inserted node is always a leaf), therefor the delete function which called in the Backtrack function will run in O(1) runtime.

   Note that this assumption is correct only for backtrack (that committed after insertion exclusively) and not for retrack .

   Therefor the runtime is O(1).

2. **Insert**

   In insertion we take the node to insert and its replacement node (the node that took his place when he was deleted) from the stack witch both of the nodes include all of the relevant information we need in order to insert them back to there previous position in the tree. All of this actions run in O(1) runtime.

   Therefor the runtime is O(1).

iv. AVL-tree based

1. **Delete**
   When the action is "delete" the function acts like the backtrack of BST(in the case of deleting) witch is **O(1)** except of a few more actions that AVL tree does in addition to those that the BST does.
   The additional actions that AVL tree does:
   1) Search if there is an unbalanced node due to the deletion, if there is no such node and the node that was deleted was a leaf (the worst case) – the search goes all the way up until it reached the root = the high of the tree = **log(n)**
   2) During the searching if there found such a node which broke the balance it leads to several actions that responsible of changing the pointers of constant numbers of nodes = constant time = **O(1)**
   3) After the deletion all of the values of high that every node own should be changed in all of the nodes that are above the deleted node because the deletion reduces the high of the nodes by 1. This action depends on the high of the tree in the worst case = **log(n)**

2. **Insert – symmetry to delete**
   When the action is "insert" the function acts like the backtrack of BST(in the case of insertion) which is **O(1)** except of a few more actions that AVL tree does in addition to those that the BST does.
   The additional actions that AVL tree does:

1) Search if there is an unbalanced node due to the insertion, if there is no such node (the worst case) – the search goes all the way up until it reached the root = the high of the tree = **log(n)**
2) During the searching if there found such a node which broke the balance it leads to several actions that responsible of changing the pointers of constant numbers of nodes = constant time = **O(1)**
3) After the insertion all of the values of high that every node own should be changed in all of the nodes that are above the inserted node because the insertion increasing the high of the nodes by 1. This action depends on the high of the tree = **log(n)**