ממ"ן 14 – מבנה נתונים ומבוא לאלגוריתמים

מטלה תכנותית – מסמך נלווה

מגיש: עומרי ברכה [redacted]

רקע.

במטלה זו, נדרשתי בהתייחס לאלגוריתם שמקורו בממ"ן 11, לבחון מספר אסטרטגיות שונות לביצוע אותה משימה. כזכור, **מטרתו האלגוריתם הנ"ל היא למצוא את מספר האיברים השונים במערך**.

במטלה זו נבחן את הדרכים השונות והמגוונות לביצוע אותה משימה, תוך הסבר קצר עבור כל שיטה, ניתוח השוואות/השמות/העתקות המתבצעות בכל האלגוריתם (אם בכלל) וניתוח סיבוכיות זמן ריצה.

פסאודו-קוד עבור השגרה המדוברת.

```
D(A) // A is an array of numbers
U_Size = 1
For i=2 to length(A)
  U=TRUE
  For j=1 to U_Size
    If A[j]=A[i]
     Then U = FALSE
      j = U_Size
   if U = TRUE

     Then U_Size = U_Size +1
         A[U_Size] = A[i]
return U_Size
```

קוד המטלה.

- קוד המטלה נכתב בשפת JAVA בשיטת TOP-DOWN כך שהאלגוריתמים השונים הם אלו הראשונים שיופיעו ואילו פונקציות העזר יהיו בתחתית העמוד.
- מצורפים בקובץ המטלה המחלקות "RBT.java", "BST.java" אלו הם הקבצים עבור המחלקות של עצים
- כל אלגוריתם קיבל את השם הגנרי `methodi` כך ש-$1 \leq i \leq 7$ הוא מספר הסעיף בממ"ן (לדוגמא, האלגוריתם המקורי, אותו עלינו לממש בסעיף 1, קיבל את השם `method1`)
- הוגדר משתנה גלובאלי `private static final int MAX_NUM = 100;` בכדי לקבוע את מספר האיברים אותם נגריל באופן אקראי לבניית המערך
- עבור כל שיטה, הוגדרו גם כן המשתנים הגלובאליים אשר מטרתם לכמת את המדדים (השוואות / העתקות / השמות), לדוגמא עבור האלגוריתם המקורי (סעיף 1):

```
static int comp1 = 0;
static int ass1 = 0;
static int copy1 = 0;
```

האלגוריתמים השונים.

נבחן את השיטות השונות למציאת מספר האיברים השונים במערך באמצעות:

1. **האלגוריתם המקורי –**

רקע: בשיטה זו מגדירים "סופר", הלוא הוא ה`int U_size = 1` וכאן אנו יוצאים מנקודת הנחה כי האיבר הראשון במערך הוא שונה באופן ריק.

נרוץ על כל המערך בלולאה ובה נגדיר משתנה בוליאני כ"אמת". לאחר מכן, בתוך הלולאה הקיימת נרוץ בלולאה נוספת עד לגודל ה"סופר" שהגדרנו למעלה ונבדוק – אם איבר במקום ה-0 שווה לאיבר במקום ה-1 (הרצה ראשונה), נשנה את המשתנה הבוליאני לשקר ונעצור. במידה ולא כך הדבר, נצא מהלולאה הפנימית ובהכרח המשתה נשאר עדין "אמת". ובכן, מעולה, נעביר את האיברים השונים לתחילת המערך ונגדיר את ה"סופר" שלנו ב-1 וכן הלאה עד לסיום האיטרציה האחרונה.

הרעיון: הרעיון מאחורי השגרה הוא למעשה להשוות כל איבר עם קודמו לאורך כל גודל המערך ולבדוק בעזרת משתנה עזר בוליאני, האם מתקיים מצב של שוויון בין האיברים או שמא האיברים שונים. כך אינדיקציית ה"אמת" או ה"שקר" עוזרת לנו להבין האם מהו מספר האיברים השונים במערך.

ניתוח זמן ריצה: בהנחה כי $A \leftarrow array; |A| = n$

כידוע, הלולאה הראשונה רצה על כל איברי המערך (למעט הראשון) ובמקרה הגרוע, כאשר כל איברי המערך שונים, הלולאה הפנימית תרוץ על כל איברי המערך. כאמור, מהסיבה שמדובר בלולאה מקוננת קיבלנו, במקרה הגרוע, זמן ריצה ריבועי כך –

$$O\big(n \cdot (n-1)\big) = O(n^2 - n) \sim O(n^2)$$

2. **מיון הכנסה –**

רקע: בשיטה זו נתבסס על האלגוריתם המוכר מיון הכנסה כך שבאמצעותו נקבל מערך ממוין, נגדיר `int count = 1` בלולאה על כל איברי המערך כדי לבדוק האם איבר במערך שונה מקודמו.

הרעיון: כאשר אנו ממיינים מערך, בהכרח נוצר מצב כי כל האיברים מסודרים ב"גושים" מסוימים ועל כן מעבר על איברי המערך, אחד אחרי השני, מבטיח לנו כי לא פספסנו מקרה קצה.

לדוגמא: *בהינתן המערך* $A = \{1,99,6,1\}$*, ידוע כי הפלט אמור להיות 3. אם נעבור על המערך בכדי למצוא את מספר האיברים השונים מבלי למיין אותו, הפלט יהיה 4 (כי הרי שבאיטרציה הלפני אחרונה, נבדוק אם* $6 =? 1$ *והתנאי יחזיר כמובן "שקר" והשגרה תעלה את ה"סופר" ב-1, אבל למעשה 1 כבר הופיע במערך, אבל לא הושווה ישירות אל מופעו הקודם). לכן השימוש במיון הכנסה (או כל מיון אחר לצורך העניין) הכרחי להצלחת השגרה, במקרה זה.*

ניתוח זמן ריצה: בהנחה כי $A \leftarrow array; |A| = n$

כידוע, הלולאה בשיטה רצה על כל איברי המערך, אך אנו נעזרים בפונקציית עזר `private static void insertionSort(int arr[])` וידוע כי זמן הריצה של מיון הכנסה הוא ריבועי ולכן בסך הכול קיבלנו –

$$O(n^2)$$

3. **מיון מיזוג –**

רקע: בשיטה זו נתבסס על האלגוריתם המוכר **מיון מיזוג** כך שבאמצעותו נקבל מערך ממוין, נגדיר `int count = 1` בלולאה על כל איברי המערך כדי לבדוק האם איבר במערך שונה מקודמו.

הרעיון: באופן זהה לחלוטין לסעיף 2, גם כאן המיון חיוני להצלחת השגרה. השוני בסעיף זה הוא כי נדרשנו למצוא מיון מבוסס השוואות אופטימלי ועל כן בחרתי במיון שזמן הריצה שלו אינו ריבועי, אלא קטן מכך.

ניתוח זמן ריצה: בהנחה כי $A \leftarrow array; |A| = n$

כידוע, הלולאה בשיטה רצה על כל איברי המערך, אך אנו נעזרים בפונקציית עזר

```
private static void mergeSort(int[] arr, int low, int high)
```

וידוע כי זמן הריצה של מיון מיזוג הוא פולינומי-לוגריתמי ולכן בסך הכול קיבלנו –

$$O(n) + O(n \cdot \log n) = O(n \cdot \log n)$$

4. **באמצעות מערך C של מיון מניה –**

רקע: במיון מניה, אנו למעשה נעזרים במערך "מנייה", מערך עזר בו למעשה נספרים כמות המופעים של כל איבר המערך, שהרי אם אחד מאיברי אותו מערך עזר הוא 0, אזי האיבר אליו הוא משויך אינו מופיע כלל במערך. עבור כל איבר במערך המקורי, נציין כל הופעה שלו כמספר במערך C, נעבור על איברי מערך המנייה ונבדוק האם יש איבר שאינו אפס. במידה ואין, נגדיר את ה"סופר" ובמידה ויש, כנראה שאיבר מסוים אינו מופיע במערך כלל.

הרעיון: כאן, בשונה מסעיפים 2-3, אין חובה למיין את המערך בכדי שהשגרה תצליח לבצע את המשימה, אפילו, אין חובה לבצע ולו השוואה אחת בין כל שני איברים. אך, כידוע אנו מוגבלים בטווח ובהיות המספרים שלמים כפי שלמדנו בספר ובמפגשים על מיון מנייה.

ניתוח זמן ריצה: בהנחה כי $A \leftarrow array; |A| = n$

כידוע, הלולאה בשיטה רצה על כל איברי המערך המקורי כדי לבצע העתקה לערך מסוים במערך העזר ואז אנו רוצים על מערך העזר בלולאה נוספת כדי לבדוק. כמובן כי מיון מניה הוא מיון לינארי ובהכרח קיבלנו זמן ריצה לינארי כך –

$$O(n) + O(n) = O(n)$$

5. **טבלת גיבוב –**

רקע: נעזר במבנה נתונים מוכר בשפת JAVA הנקרא `HashSet`

מבנה זה מעניק לנו פעולות מילון חיוניות למטרתנו בזמן ריצה מעולה, כגון:

עבור - `HashSet<Integer> h = new HashSet<Integer>();`

`h.contains`

`h.add`

נעבור על כל המערך בלולאה ונשתמש בפעולות המילון כדי לבדוק האם האיבר אינו נמצא בטבלת הגיבוב. במידה וכך הדבר, נגדיל את ה"סופר" ונוסיף את אותו איבר לטבלה כך שבאיטרציה הבאה,

אם איבר זהה ייבדק, האלגוריתם ידע כי האיבר כבר נמצא בטבלה ועל כן, לא נגדיל את "הסופר" ולא נכניס אותו לטבלה שוב.

הרעיון: בעזרת מבנה נתונים מובנה של טבלת גיבוב, אנו משתמשים בבדיקה פשוטה לכל אורך המערך של איברים, העתקתם לטבלת גיבוב ע"י פעולות מילון "זולות" מבחינת זמני ריצה ובכך האלגוריתם יבצע את המשימה.

ניתוח זמן ריצה: בהנחה כי $A \leftarrow array; |A| = n$

נעיר: טבלת גיבוב מעניקה לו זמני ריצה טובים במקרה הממוצע (תוחלת זמן)

כידוע, הלולאה בשיטה רצה על כל איברי המערך המקורי כדי ומשתמשת בפעולות מילון בעלות של $O(1)$ ולכן בסך הכול קיבלנו זמן ריצה לינארי המושפע בעיקר מגודל המערך –

$O(n)$

6. **עץ חיפוש בינארי –**

רקע: כידוע, עץ חיפוש בינארי הוא עץ בו לכל קודקוד יש שני בנים לכל היותר, ימני ושמאלי. הבן הימני יחד עם כל צאצאיו, נמצא ביחס הסדר אחרי הקודקוד שממנו הוא יוצא ואילו הבן השמאלי וכל צאצאיו קודמים לקודקוד האב ביחס הסדר. גם כאן, נתבסס על פעולות המילון המהירות של העץ.

במימוש המבנה, כאשר מדברים על פעולת ההוספה, העץ בודק האם האיבר שהוכנס קטן מהשורש או לא (בין היתר) ובכך ממשיך כדי לדעת היכן למקמו. באופן דומה, העץ בודק איפה האיבר לחיפוש נמצא. נעבור על כל המערך בלולאה ונשתמש בפעולות המילון כדי לבדוק האם האיבר אינו נמצא בעץ. במידה וכך הדבר, נגדיל את ה"סופר" ונוסיף את אותו איבר לטבלה כך שבאיטרציה הבאה, אם איבר זהה ייבדק, האלגוריתם ידע כי האיבר כבר נמצא בעץ ועל כן, לא נגדיל את "הסופר" ולא נכניס אותו לטבלה שוב

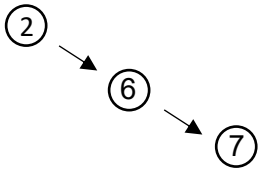
הרעיון: בעזרת עץ חיפוש בינארי, אנו משתמשים בבדיקה פשוטה לכל אורך המערך של איברים, העתקתם לעץ ע"י פעולות מילון "זולות" מבחינת זמני ריצה ובכך האלגוריתם יבצע את המשימה.

ניתוח זמן ריצה: בהנחה כי $A \leftarrow array; |A| = n$

כידוע, זמן הריצה של כל הפעולות בעץ חיפוש בינארי תלוי בגובה העץ. נקרא לו h.

כאשר במקרה הגרוע העץ איננו מאוזן,

לדוגמא:



ואז למעשה אין יתרון לעץ על-פני מערך לא ממוין, אך זמני הריצה עדין לינאריים. לעומת זאת, בעבור עץ בינארי מאוזן גובה העץ בהכרח $h = \log n$ ובכך פעולות המילון שלו "יעלו" לכל היותר $O(\log n)$. מכאן קיבלנו שבמקרה הגרוע, זמן הריצה של האלגוריתם הינו –

$O(n)$

7. **עץ אדום-שחור –**

רקע: כמה תכונות על עץ אדום-שחור

- הוא עץ חיפוש בינארי מאוזן, כאשר עבור כל צומת נשמור נתון נוסף – הצבע שלו שיכול להיות אדום או שחור.
- מוגדרת חוקיות לגבי אופציות הצביעה כך שנוכל להבטיח שכל מסלול מהשורש לעלה יהיה לכל היותר כפליים מכל מסלול אחר מהשורש לעלה, כך שהעץ יהיה מאוזן.
- כל צומת הוא או אדום או שחור והשורש הראשי של העץ תמיד שחור
- העלים הם צמתי ***NULL*** וצבעם תמיד שחור
- אם צומת הוא אדום אז שני בניו שחורים
- כל המסלולים צומת כלשהו לעלים מכילים אותו מספר של צמתים שחורים

באופן דומה לסעיף 6, נעזר בפעולות המילון אשר עץ אדום-שחור מספק לנו על מנת להוסיף איברים או לבדוק האם איברים נמצאים בעץ.

הרעיון: בעזרת עץ חיפוש בינארי, אנו משתמשים בבדיקה פשוטה לכל אורך המערך של איברים, העתקתם לעץ ע"י פעולות מילון "זולות" מבחינת זמני ריצה ובכך האלגוריתם יבצע את המשימה.

ניתוח זמן ריצה: בהנחה כי $A \leftarrow array; |A| = n$

ע"פ האמור, עץ אדום-שחור הינו עץ חיפוש בינארי מאוזן ולכן במקרה הגרוע ביותר, פעולות המילון נעשות בסיבוכיות זמן ריצה של $O(\log n)$ במקרה הגרוע ביותר. תיקון העץ, נעשה בזמן ריצה של $O(1)$. לכן בסך הכול, כאשר אנו משתמשים בעץ אדום-שחור נקבל סיבוכיות זמן ריצה לוגריתמית – $O(\log n)$

סיכום ביניים

זמני ריצה.

מציאת מספר איברים שונים במערך באמצעות	סיבוכיות זמן ריצה
האלגוריתם המקורי	$O(n^2)$
מיון הכנסה	$O(n^2)$
מיון מיזוג	$O(n \cdot \log n)$
מערך *C* של מיון מנייה	$O(n)$
טבלת גיבוב *HASHSET*	$O(n)$
עץ חיפוש בינארי	$O(n)$
עץ אדום-שחור	$O(\log n)$

נבדוק וננתח את המדדים עבור כל אלגוריתם.

אלגוריתם	מדדים ליעילות	מדוע המדד/ים האלו מייצגים?
האלגוריתם המקורי	השוואות והשמות	בהתאם להסבר על האלגוריתם המקורי, כמות ההשמות של המשתנה `boolean U = true;` וההשוואות בין כל איבר במיקומים ה-i, j במערך אלו המדדים המייצגים מהסיבה שבמקרה הגרוע, כל אחד מהם ישפיע באופן ישיר על זמן הריצה של המערך
מיון הכנסה	השוואות והשמות	כידוע, מיו הכנסה הינו מיון מבוסס השוואות ובמיון זה מתבצעת השמה `arr [i+1] = arr [i];` ולכן יעילותו אלגוריתם `method2` המתבסס על מיון זה מושפעת באופן ישיר מהתהליכים המתרחשים במיון הכנסה
מיון מיזוג	השוואות, השמות והעתקות	שימוש במיון מיזוג, בו אנו נעזרים במערך עזר, משווים איברים, מבצעים השמות ומעתיקים למערך העזר ערכים, למעשה משפיע על יעילותו של האלגוריתם ה-`method3` באופן ישיר
מערך *C* של מיון מנייה	העתקות	כפי שלמדנו, מיון מניה הינו מיון בזמן לינארי שאינו מבוסס השוואות. אנו נעזרים במערך עזר, אשר הוא מייצג את כמות המופעים של כל איבר במערך המקורי. לכן, הפעולה היחידה שנעשית היא למעשה העתקה
טבלת גיבוב *HASHSET*	השוואות והעתקות	בשימוש בטבלת גיבוב, אנו למעשה מעתיקים את איברי המערך לטבלה כל עוד טרם הופיעו ובכך המדדים המייצגים בשיטה זו הם השוואות והעתקות
עץ חיפוש בינארי	השוואות והעתקות	באופן זהה לטבלת גיבוב, כך גם בעץ חיפוש בינארי ובעץ אדום-שחור. ההבדל היחיד הוא שעל עץ חיפוש בינארי ועץ אדום-שחור מתבצעות באופן ניכר יותר השוואות מהסיבה שהכנסת איבר לעץ, מלווה בלא מעט הגבלות (בשל המבנה הייחודי של כל עץ)
עץ אדום-שחור	השוואות והעתקות	

דוגמאות הרצה.

ע"פ המטלה נבחן דוגמאות הרצה עבור $N = 20,100,1000,10000,100000$

```
For N = 100:
--------------------------------
The Original Algorithm performs:
Compares: 3003
Assignments: 162
Copies: 0
Total distinct numbers: 64
--------------------------------
The 2nd Algorithm performs:
Compares: 99
Assignments: 2715
Copies: 0
Total distinct numbers: 64
--------------------------------
The 3rd Algorithm performs:
Compares: 442
Assignments: 672
Copies: 672
Total distinct numbers: 64
--------------------------------
The 4th Algorithm performs:
Compares: 0
Assignments: 0
Copies: 100
Total distinct numbers: 64
--------------------------------
The 5th Algorithm performs:
Compares: 100
Assignments: 0
Copies: 64
Total distinct numbers: 64
--------------------------------
The 6th Algorithm performs:
Compares: 8220
Assignments: 0
Copies: 64
Total distinct numbers: 64
--------------------------------
The 7th Algorithm performs:
Compares: 8220
Assignments: 0
Copies: 64
Total distinct numbers: 64
--------------------------------
```

```
For N = 1000:
--------------------------------
The Original Algorithm performs:
Compares: 47611
Assignments: 1098
Copies: 0
Total distinct numbers: 100
--------------------------------
The 2nd Algorithm performs:
Compares: 999
Assignments: 249991
Copies: 0
Total distinct numbers: 100
--------------------------------
The 3rd Algorithm performs:
Compares: 6640
Assignments: 9976
Copies: 9976
Total distinct numbers: 100
--------------------------------
The 4th Algorithm performs:
Compares: 0
Assignments: 0
Copies: 1000
Total distinct numbers: 100
--------------------------------
The 5th Algorithm performs:
Compares: 1000
Assignments: 0
Copies: 100
Total distinct numbers: 100
--------------------------------
The 6th Algorithm performs:
Compares: 103320
Assignments: 0
Copies: 100
Total distinct numbers: 100
--------------------------------
The 7th Algorithm performs:
Compares: 103320
Assignments: 0
Copies: 100
Total distinct numbers: 100
--------------------------------
```

```
For N = 10000:
--------------------------------
The Original Algorithm performs:
Compares: 500717
Assignments: 10098
Copies: 0
Total distinct numbers: 100
--------------------------------
The 2nd Algorithm performs:
Compares: 9999
Assignments: 25023946
Copies: 0
Total distinct numbers: 100
--------------------------------
The 3rd Algorithm performs:
Compares: 88081
Assignments: 133616
Copies: 133616
Total distinct numbers: 100
--------------------------------
The 4th Algorithm performs:
Compares: 0
Assignments: 0
Copies: 10000
Total distinct numbers: 100
--------------------------------
The 5th Algorithm performs:
Compares: 10000
Assignments: 0
Copies: 100
Total distinct numbers: 100
--------------------------------
The 6th Algorithm performs:
Compares: 1006952
Assignments: 0
Copies: 100
Total distinct numbers: 100
--------------------------------
The 7th Algorithm performs:
Compares: 1006952
Assignments: 0
Copies: 100
Total distinct numbers: 100
--------------------------------
```

```
For N = 100000:
--------------------------------
The Original Algorithm performs:
Compares: 5046117
Assignments: 100098
Copies: 0
Total distinct numbers: 100
--------------------------------
The 2nd Algorithm performs:
Compares: 99999
Assignments: 2472607852
Copies: 0
Total distinct numbers: 100
--------------------------------
The 3rd Algorithm performs:
Compares: 1047339
Assignments: 1668928
Copies: 1668928
Total distinct numbers: 100
--------------------------------
The 4th Algorithm performs:
Compares: 0
Assignments: 0
Copies: 100000
Total distinct numbers: 100
--------------------------------
The 5th Algorithm performs:
Compares: 100000
Assignments: 0
Copies: 100
Total distinct numbers: 100
--------------------------------
The 6th Algorithm performs:
Compares: 9997764
Assignments: 0
Copies: 100
Total distinct numbers: 100
--------------------------------
The 7th Algorithm performs:
Compares: 9997764
Assignments: 0
Copies: 100
Total distinct numbers: 100
--------------------------------
```

```
For N = 1000000:
--------------------------------
The Original Algorithm performs:
Compares: 50471214
Assignments: 1000098
Copies: 0
Total distinct numbers: 100
--------------------------------
The 3rd Algorithm performs:
Compares: 19620184
Assignments: 19951424
Copies: 19951424
Total distinct numbers: 100
--------------------------------
The 4th Algorithm performs:
Compares: 0
Assignments: 0
Copies: 1000000
Total distinct numbers: 100
--------------------------------
The 5th Algorithm performs:
Compares: 1000000
Assignments: 0
Copies: 100
Total distinct numbers: 100

--------------------------------
The 6th Algorithm performs:
Compares: 13059381
Assignments: 0
Copies: 100
Total distinct numbers: 100
--------------------------------
```

אציין כי ניסיתי להריץ במחשב שלי עבור $N = 1,000,000$, אבל התכנית רצה לאורך יותר מידי זמן ולכן הראיתי בנפרד דוגמאות עבור חלק מן השיטות

דוגמאות הרצה נוספות –

```
The array is:
26 93 81 26 13

For N = 5:
--------------------------------
The Original Algorithm performs:
Compares: 7
Assignments: 7
Copies: 0
Total distinct numbers: 4
--------------------------------
The 2nd Algorithm performs:
Compares: 4
Assignments: 15
Copies: 0
Total distinct numbers: 4
--------------------------------
The 3rd Algorithm performs:
Compares: 9
Assignments: 12
Copies: 12
Total distinct numbers: 4
--------------------------------
The 4th Algorithm performs:
Compares: 0
Assignments: 0
Copies: 5
Total distinct numbers: 4
--------------------------------
The 5th Algorithm performs:
Compares: 5
Assignments: 0
Copies: 4
Total distinct numbers: 4
--------------------------------
The 6th Algorithm performs:
Compares: 19
Assignments: 0
Copies: 4
Total distinct numbers: 4
--------------------------------
The 7th Algorithm performs:
Compares: 19
Assignments: 0
Copies: 4
Total distinct numbers: 4
--------------------------------
```

```
The array is:
50 27 49 55 91 24 33 58 84 33 52 23 67 52 58 28 24 92 58 50

For N = 20:
--------------------------------
The Original Algorithm performs:
Compares: 131
Assignments: 32
Copies: 0
Total distinct numbers: 14
--------------------------------
The 2nd Algorithm performs:
Compares: 19
Assignments: 120
Copies: 0
Total distinct numbers: 14
--------------------------------
The 3rd Algorithm performs:
Compares: 62
Assignments: 88
Copies: 88
Total distinct numbers: 14
--------------------------------
The 4th Algorithm performs:
Compares: 0
Assignments: 0
Copies: 20
Total distinct numbers: 14
--------------------------------
The 5th Algorithm performs:
Compares: 20
Assignments: 0
Copies: 14
Total distinct numbers: 14
--------------------------------
The 6th Algorithm performs:
Compares: 361
Assignments: 0
Copies: 14
Total distinct numbers: 14
--------------------------------
The 7th Algorithm performs:
Compares: 361
Assignments: 0
Copies: 14
Total distinct numbers: 14
--------------------------------
```

מסקנות.

כפי שניתן להסיק, ככל שגדל המערך, באופן די טריביאלי כל שיטה "תעלה" יותר זמן, אך באופן די מדהים, לאחר שבחנו 7 שיטות שונות, אנו רואים כי כל שיטה "תעלה" באופן שונה. מכאן אנו רואים כי למדדים המייצגים יש השפעה מכרעת בזמני הריצה של האלגוריתמים למציאת מספר המספרים השונים במערך. לדוגמא, כאשר נשתמש במיון לינארי (שאינו מבוסס השוואות), כאשר אנו מעתיקים בלבד את ערכי המערך למערך עזר ומגלים כך את מספר המופעים, קל לראות כי אכן מדובר בזמן ריצה לינארי למספר המופעים וכי מספר ההעתקות גדל בהתאם לגודל המערך (באופן ישיר).

סיכום.

לסיכום, בחנתי 7 שיטות שונות למציאת המספרים השונים במערך. חלקן מהירות יותר וחלקן מהירות פחות. תיארתי בקצרה את נכונות האלגוריתמים, את זמני הריצה ואת הרעיון מאחוריהם. הראיתי דוגמאות הרצה כנדרש ואף צירפתי סרטון המחשה שהכנתי על "מאחורי הקלעים" של האלגוריתם `method4` , אשר משתמש במערך ***C*** של מיון מניה.

מקרא קובץ.

1. קובץ המטלה "***Matal14***"
2. קבצי מחלקות העזר (עץ חיפוש בינארי ועץ אדום שחור)
3. סרטון הממחיש את פעולת האלגוריתם `method4`