

פרק 5 יעילות

יש יותר מדרך אחת לפצח אגוז. אפשר להניחו על הרצפה ולרקוע עליו, אפשר לפצחו בעזרת השיניים או להיעזר באגוז נוסף, ואפשר כמובן להשתמש במפצח אגוזים. בכל הדרכים נשיג את מטרתנו – אגוז מפוצח. מבחינת התוצאה, כל השיטות נכונות. השיטות השונות נבדלות ביניהן במאמץ הנדרש מאתנו לביצוע המשימה, במשך הזמן שהיא דורשת, באמצעים העומדים לרשותנו (מפצח אגוזים) ובעלות הביצוע (טיפול שיניים או נעל חדשה במקרים קיצוניים). אנו נעדיף את השיטה שתהיה היעילה ביותר עבורנו.

גם במדעי המחשב אנו רוצים שהאלגוריתמים שאנו מתכננים ושהתוכניות שאנו כותבים לפי אלגוריתמים אלה יהיו יעילים. כאשר מודדים יעילות של אלגוריתמים ותוכניות, מתחשבים בשני קריטריונים: זמן ומקום. הזמן הוא זמן חישוב, כלומר משך הזמן שבו מתבצעת הפעולה, הקריטריון השני – המקום – מתייחס לגודל זיכרון המחשב שיש להקצות לצורך ביצוע הפעולה.

הבעיות המתעוררות בהערכת אלגוריתמים לפי שני הקריטריונים דומות במידה רבה. חקר היעילות של קריטריון המקום עוסק בנושאים דומים לאלה של קריטריון הזמן. אנו נתרכז בחישובי היעילות של מדד הזמן.

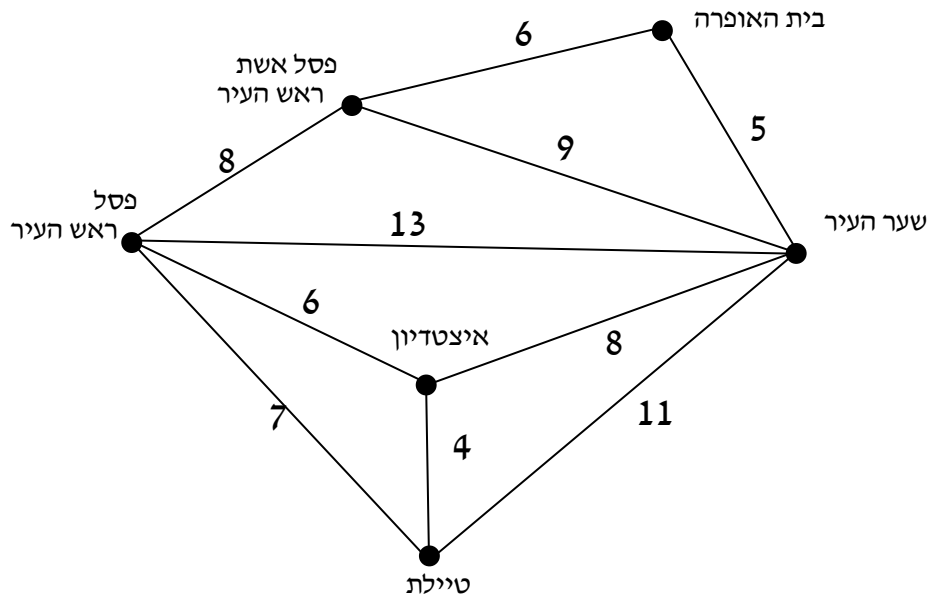
כל דור חדש של מחשבים מהיר בהרבה מהדור הקודם: המחשבים שאנו משתמשים בהם כעת מהירים מאוד יחסית לאלה שהיו קיימים לפני שנים ומבצעים מיליוני הוראות בשנייה. צפוי שכוח החישוב של מחשבים ימשיך לעלות במהירות בעתיד הנראה לעין. לפיכך, נשאלת השאלה מדוע יש להתחשב בזמן הריצה של אלגוריתמים? מדוע כדאי לדון בשאלה אם תוכנית תבצע שני מיליון הוראות או רק מיליון אחד? הרי בכל מקרה משך הריצה של תוכנית כזו יהיה חלק של שנייה.

גישה זו מוטעית מיסודה. ראשית, המחשבים אמנם מהירים יותר, אך המשימות שעליהם למלא וכמויות הנתונים לעיבוד ולחישוב אף הם גדלים במהירות. כשאנו רוצים לחפש ספר בקטלוג הממוחשב של הספרייה, לקבל מספר טלפון של מנוי מסוים במודיעין 144 או לבצע חיפוש באינטרנט בעזרת מנוע חיפוש, איננו רוצים לחכות יותר מכמה שניות עד שתינתן תשובה. מדובר בחיפוש באוספי נתונים גדולים מאוד. לכן יש חשיבות מרובה לשימוש באלגוריתם יעיל לחיפוש. במערכות לחיזוי מזג אוויר, כמויות הנתונים אף הן גדולות מאוד, וכדי להגיע לתחזיות מדויקות יותר מגדילים אותן בהתמדה, ובנוסף לכך החישובים מסובכים ביותר. המהירות במערכות אלה חשובה, שהרי מה הערך של התחזית למחר אם נקבלה רק מחרתיים? במערכת כזאת דרושים אלגוריתמים יעילים לעיבוד הכמויות הגדולות של הנתונים ולביצוע החישובים הרבים והמסובכים. גם במערכות לבקרת טיסה, הנחיית טילים, בקרת מפעלי תעשייה וכן במערכות המשתמשות בגרפיקה (הדורשת משאבי חישוב רבים) כגון משחקי מחשב, הזמן הוא גורם מכריע. מערכות אלה חייבות להגיב לגירויים קיצוניים, כגון לחיצה על כפתורים, בזמן אמת, כלומר כמעט מיד, אחרת הדבר עלול להוביל לתוצאות שליליות ואף קטלניות.

א. בעיות, פתרונות ויעילות

כדי להבין טוב יותר את חשיבות הדיון ביעילות, הבה נתבונן בבעיה זו:

עקב שפל חסר תקדים במספר התיירים הפוקדים עיר מסוימת, החליט ראש העירייה לקדם את עסקי התיירות בעירו. לשם כך הגה תוכנית לקו אוטובוס שיעבור באתרי התיירות המרכזיים בעיר. הוא זימן למשרדו את מנהל חברת האוטובוסים, הסביר לו את מבוקשו וצייד אותו במפת העיר. במפה צוינו האתרים החשובים, הכבישים המחברים ביניהם ואורכו של כל כביש.



מפת העיר (המספרים מציינים מרחקים בק"מ. אורכי הכבישים אינם לפי קנה מידה אחיד)

מנהל חברת האוטובוסים כינס את צוות המתכנתים של החברה והטיל עליו את המשימה הזו: "עליך למצוא את המסלול הקצר ביותר היוצא משער העיר ומסתיים בו. המסלול צריך לעבור בכל אתר תיירות פעם אחת בלבד".

? נסו למצוא שיטה אלגוריתמית לפתרון הבעיה עבור כל רשימה נתונה של אתרים ומרחקים.

הערה: בעיה זו נקראת בספרות מדעי המחשב: "בעיית הסוכן הנוסע", מהסיבה הבאה: בארצות הברית הגדולה היה מקובל שסוכני מכירה או שירות של חברות נוסעים מעיר לעיר כדי לבקר את לקוחות החברה. סוכן היה יוצא מעיר מגוריו, עובר דרך כל עיר שבה הוא צריך לבקר, וחוזר אחרי שבוע או יותר לביתו, למנוחה ואחריה יוצא לסיבוב נוסף. טבעי שסוכן נוסע כזה יהיה מעוניין במסלול הקצר ביותר העובר בין כל לקוחותיו.

ובכן, פתרון אפשרי לבעיה זו הוא מציאת כל המסלולים האפשריים, שמתחילים ומסתיימים בשער העיר ועוברים בכל אתר תיירות פעם אחת בלבד, חישוב אורכיהם ומציאת הקצר שבהם. פתרון זה נראה פשוט ויעיל, שהרי מספר המסלולים האפשריים במפת העיר לכאורה אינו גדול.

אם נכתוב תוכנית מחשב שתממש פתרון זה, נראה שאמנם זמן ריצתה של התוכנית יהיה קצר למדי.

כמה מסלולים כאלה קיימים? הנקודה הראשונה במסלול היא קבועה: שער העיר. לבחירת הנקודה השנייה יש לנו חמש אפשרויות שונות, מכיוון ששער העיר מחובר לכל האתרים שבעיר. לאחר שבחרנו את הנקודה השנייה, יש לנו לכל היותר ארבע אפשרויות שונות לבחירת הנקודה השלישית, וכן הלאה. אנו מציינים שיש לנו לכל היותר x אפשרויות, כיוון שלמעשה יכול להיות שיהיו לנו פחות אפשרויות, שהרי יש מקומות שאינם מקושרים לאחרים. למשל, אם הנקודה השנייה שנבחרה היא בית האופרה, לא נוכל לבחור את פסל ראש העיר כנקודה השלישית. אם כן, מספר המסלולים האפשריים הוא לכל היותר: $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

מה יקרה אם נכליל את האלגוריתם לעיר שבה n אתרים? מספר המסלולים האפשריים הוא לכל היותר $(n-1)!$. נניח שיש לנו מחשב היכול לחשב מיליון מסלולים בשנייה. לעיר שבה 11 אתרים יידרשו למחשב כ-4 שניות לעבור על כל המסלולים האפשריים ולמצוא את הפתרון. זהו משך זמן סביר. אך אם נעבור למפה גדולה רק במעט, שבה 15 אתרים, יזדקק אותו מחשב ליום שלם כדי למצוא את הפתרון. בעיר תיירותית מצויה שבה 20 אתרים, יידרשו למחשב כ-3,857 שנים למציאת הפתרון!

הטבלה הבאה מציגה את מספר המסלולים האפשריים עבור מספרים שונים של אתרים ואת הזמנים הדרושים לחישובם:

מספר האתרים	מספר המסלולים	זמן החישוב
6	120	8 מילישניות
11	3,628,800	כ-3.5 שניות
13	479,001,600	כ-8 דקות
16	1,307,674,368,000	כ-15 ימים
18	~355,000,000,000,000	כ-11 שנים
21	~2,430,000,000,000,000,000	כ-77,000 אלף שנים

מצב זה הוא בלתי סביר בעליל. הבעיה אמנם מנוסחת באופן ברור וקצר ובידינו אלגוריתם קצר וברור למציאת הפתרון, אך משך החיפוש עבור עיר שמספר אתריה גדול מ-14, ואפשר לבקור במסלולים רבים ושונים, לחלוטין אינו קביל.

האם יש אלגוריתם אחר, יעיל יותר, לפתרון בעיית הסוכן הנוסע? להלן תמונת מצב עדכנית:

1. כיום לא ידוע אלגוריתם יעיל לפתרון הבעיה.
2. בעיית הסוכן הנוסע אינה יחידה. יש אוסף גדול של בעיות, שלרבות מהן שימושים בכלכלה ובעשייה, שהפתרונות היחידים הידועים עבורן אינם יעילים עד מאוד. אוסף זה כולל בעיות כגון תכנון לוחות זמנים (להקצאת שעות הוראה למורים, להקצאת מכונות לסוגים שונים של עבודה וכיו"ב), ניהול מלאי והעברת משלוחים. כדי לפתור כל אחת מבעיות אלה, עבור

קלטים לא גדולים במיוחד, באמצעות האלגוריתם הטוב ביותר הידוע לנו, ואפילו על המחשבים המהירים ביותר הקיימים כיום (וגם באלה שייבנו בעתיד הנראה לעין), ניאלץ להמתין לתשובות שנים רבות.

3. ידוע, כי אם יימצא פתרון יעיל לאחת מהבעיות אלה, ניתן יהיה לכתוב לפיו פתרונות יעילים לכל הבעיות האחרות באוסף.

אוסף בעיות זה הוא מוקד מחקר במדעי המחשב. כיום מקובלת הסברה שאין לבעיות אלה פתרונות יעילים, וזאת בהתחשב במאמצים הרבים שהושקעו למציאת פתרונות יעילים. ואולם, לאמונה זו אין עדיין הוכחה מדעית.

לסיכום: ראינו שיש בעיות שאפשר לפתור במחשב בעזרת אלגוריתמים פשוטים למדי שאינם יעילים. זמן הרצת התוכנית במקרים אלה יהיה בלתי סביר, אפילו לקלטים קצרים יחסית. לקח ברור הוא שכאשר אנו מתכננים אלגוריתם לפתרון בעיה כלשהי, עלינו לבדוק את יעילותו ולוודא כי אפשר לפתור בעזרתו את הבעיה בזמן סביר. אם לא כך הדבר, רצוי למצוא אלגוריתם יעיל יותר. לפעמים לא קל למצוא אלגוריתם שכזה (כמו בבעיית הסוכן הנוסע), אך במקרים רבים הדבר אפשרי. לבעיות רבות יש פתרונות שונים, הנבדלים זה מזה ביעילותם. בהמשך הפרק נחיד את הבנתנו לגבי מדידת יעילות של אלגוריתמים, וכן נדגים שיטות לשיפור יעילות זו.

ב. איך מודדים יעילות?

אילו הציגו בפניכם שני אלגוריתמים שונים לפתרון בעיה מסוימת, והיה עליכם להכריע איזה מבין השניים יעיל יותר, האם הייתם מסתפקים בתשובה שלפניכם?

האלגוריתמים מומשו במחשב וזמני הריצה שלהם עברו אותו קלט נמדדו.

להלן התוצאות:

אלגוריתם א' – פתרון הבעיה נמשך 1.25 שניות.

אלגוריתם ב' – פתרון הבעיה נמשך 0.34 שניות.

מסקנה: אלגוריתם ב' יעיל יותר!

לא ניתן להכריע איזה אלגוריתם טוב יותר, כיוון שחסרים לנו פרטים רבים. למשל: האם מדידת זמן הריצה של שני האלגוריתמים בוצעה על אותו מחשב או על שני מחשבים שונים? הרי במחשבים שונים מותקנים מעבדים בעלי מהירויות שונות. ובכלל, האם משך הזמן הוא הממד המעניין אותנו? נראה שאנו צריכים להעריך את האלגוריתם עצמו ללא תלות במחשב שעליו הוא רץ.

ב.1. אורך קלט

משך הריצה של אלגוריתם תלוי ב"כמות העבודה" שעליו לבצע, וזו תלויה באורך הקלט שעליו פועל האלגוריתם. אלגוריתם שמסכם איברי מערך ירוץ זמן קצר יותר על מערך שבו 10 תאים, בהשוואה לאותו האלגוריתם שירוצץ על מערך שבו 100 תאים.

לכל בעיה יש להגדיר את אורך הקלט המתאים לה. באלגוריתם שמסכם איברי מערך, אורך הקלט הוא גודל המערך. אם אנו רוצים לחפש מילה ב־טקסט, אורך הקלט יכול להיות מספר האותיות או מספר המילים בטקסט. באלגוריתם המכפיל מספרים, אורך הקלט הוא מספר הספרות במספרים שעלינו להכפיל. שימו לב כי הנטייה הטבעית היא לחשוב על גודל המספרים כמדד לקלט. אך אלגוריתם לחישוב כפל פועל על ספרות, ולכן טבעי לקבוע שזמן ריצתו תלוי במספר הספרות.

לאחר שהגדרנו את אורך הקלט המתאים לבעיה, נוכל לעסוק ביעילותו של הפתרון האלגוריתמי המוצע, כתלוי באורך הקלט. בהמשך הפרק נעסוק גם בבעיות שבהן אין משמעות לאורך הקלט, והפתרונות האלגוריתמיים מתבצעים בזמן קבוע ללא תלות בגורם נוסף.

ב.2. פעולת יסוד

כדי לחשב את יעילות האלגוריתם, ללא קשר למחשב שעליו הוא ירוץ, עלינו לנתח את כמות העבודה שמבצע האלגוריתם על אורך קלט המתאים לבעיה. נגדיר כל אחת מהפעולות הפשוטות שמבצע אלגוריתם (שהן: קלט או פלט של נתון יחיד, השמה, פעולת חישוב, בדיקת תנאי בוליאני), כ-**פעולת יסוד**, בלי להתחשב בהבדלי הזמן הקלים הקיימים בין זמני הריצה של הפעולות השונות. למעשה, באופן טכני לחלוטין ניתן לומר שכל פעולה שאינה תלויה באורך הקלט אלא אורכת זמן קבוע, תוגדר לצורך החישוב כפעולת יסוד. ברוב המקרים, בעוד כל פעולת יסוד נפרדת אינה תלויה באורך הקלט, הרי כלל פעולות היסוד שמבצע האלגוריתם תלוי ממש באורך הקלט. אם אין תלות בין מספר הפעולות לבין אורך הקלט, אנו אומרים שזמן הריצה של האלגוריתם קבוע לכל אורך קלט שהוא.

לאחר שהגדרנו מהן פעולות היסוד, נספור בפועל כמה פעולות כאלה מבצע האלגוריתם על אורך קלט מסוים. המספר שיתקבל יהיה מדד טוב להגדרת היעילות של האלגוריתם. מדד זה הוא הערכה של משך זמן הריצה של האלגוריתם עבור קלט באורך נתון.

נדגים את אופן חישוב מדד היעילות בעזרת כמה דוגמאות. כמובן שאין הבדל אם אנו מבצעים את החישוב על אלגוריתם מילולי או על קוד הפעולה המיישמת את האלגוריתם.

בדוגמאות הבאות נבצע את חישוב היעילות על כמה פעולות פשוטות. נחשב באופן מדויק את מספר פעולות היסוד שאלגוריתמים שונים מבצעים. תחילה, נראה אלגוריתם שאינו תלוי באורך הקלט שלו, ולאחר מכן נחשב את מספר פעולות היסוד שמבצעים אלגוריתמים כפונקציה של אורך הקלט שלהם.

דוגמה 1: מינימום במערך ממוין

נתון מערך ממוין בסדר עולה. ברצוננו לקבל את הערך הקטן ביותר השמור במערך. לצורך כך עלינו לפנות לאיבר הראשון השמור במערך. נתבונן בפעולה הבאה:

```
public static int getMinInArray(int[] arr)
{
    return arr[0];
}
```

כפי שניתן לראות הפעולה מבצעת פעולת יסוד אחת בלבד, מכיוון שהיא אינה תלויה בגודלו של המערך. הפעולה פונה ישירות לתא הראשון במערך, ללא צורך במעבר על כל איברי המערך ובלי קשר לגודלו.

דוגמה 2: סכום איברים במערך

```
public static int sumArray(int[] arr)
{
    int sum = 0;
    int n = arr.length;    // אורך הקלט

    for (int i = 0; i < n; i++)
        sum = sum + arr[i];

    return sum;
}
```

אורך הקלט בבעיה שלפנינו הוא מספר האיברים במערך. נסמנו ב- n . הפעולה מבצעת שתי פעולות יסוד, שזמן ביצוען קבוע, לפני לולאת ה-`for`, ופעולת יסוד אחת אחריה. הלולאה עצמה עוברת על כל איברי המערך. לפיכך הלולאה מתבצעת n פעמים עבור מערך בגודל n . בכל שלב של ביצוע הלולאה, מתבצעת פעולת יסוד אחת (גוף הלולאה). כלומר, סך כל הפעולות המתבצעות הוא: $3+1 \cdot n$.

אפשר לדייק ולומר שבכל שלב ביצוע של הלולאה מתבצעות עוד שתי פעולות: הבדיקה אם הגענו לסוף הלולאה וקידומו של i השולט על הלולאה, ואם כך מתקיימת גם פעולה אחת נוספת בעת אתחולו של i . לשיטת הדייקנים, יהיה מספר הפעולות בסיכומו של דבר: $3+1+3n$ כלומר $4+3n$.

קיבלנו אם כן שני ביטויים התלויים באורך הקלט n , ביטויים לינאריים המתארים את מספר פעולות היסוד שמתבצע בתהליך, בשתי רמות דיוק: $3+n$, או, $4+3n$. בהמשך נראה שההבדל בין שתי תוצאות אלה אינו משמעותי, כל אחד מהחישובים מדויק במידה מספקת ומשמעותם זהה.

דוגמה 3: האם המספר k מופיע במערך

בדוגמה זו נתון מערך ממין בסדר עולה של מספרים, ואנו רוצים לבדוק אם מספר נתון מופיע בו. אם האיבר נמצא במערך, הפעולה מחזירה את מיקומו, אחרת הפעולה מחזירה את הערך -1 . לשם פשטות נניח כי המספרים במערך שונים זה מזה. אפשר לפתור את הבעיה בעזרת פעולה זו:

```
public static int search1(int[] arr, int k)
{
    int n = arr.length;    // אורך הקלט

    for (int i = 0; i < n; i++)
        if (arr[i] == k)
            return i;    // הביצוע הופסק עם החזרת מיקום המספר במערך

    return -1;
}
```

אורך הקלט בבעיה שלפנינו הוא מספר האיברים במערך. נסמנו ב- n . הפעולה עוברת על המערך, תא אחרי תא, באמצעות לולאה. אם הערך בתא שווה ל- k , הלולאה מופסקת והפעולה מחזירה את האינדקס של התא. אם הלולאה בוצעה בשלמותה, k אינו קיים במערך ולכן מוחזר הערך -1 .

נחשב את מספר פעולות היסוד המדויק שהפעולה מבצעת כאשר k אינו במערך, או נמצא בתא האחרון בו. אנו מתמקדים בשני מקרים אלה מכיוון שבהם הפעולה תבצע את מספר פעולות היסוד הגדול ביותר.

לפני הלולאה מתבצעות 2 פעולות אתחול, של n ושל i .

בכל ביצוע של גוף הלולאה מתבצעות 3 פעולות יסוד (על פי הגישה הדייקנית): 2 הפעולות בכותרת הלולאה (מתבצעת הבדיקה האם ערך i עדיין קטן מאורך המערך, ומתבצעת הגדלת ערכו ב-1), ופעולה נוספת בגוף הלולאה (קריאת ערכו של תא במערך והשוואת הערך הנקרא לערך של k).

מכיוון שהלולאה תלויה באורך הקלט היא תבצע בסך הכול $3n$ פעולות.

בסוף `search1(...)` תתבצע פעולת `return`. ה-`return` יתבצע מתוך הלולאה או מחוצה לה אך רק פעם אחת בלבד במהלך כל החיפוש.

אם כן, סך כל פעולות היסוד בזמן ביצוע הפעולה `search1(...)` הוא $3n+3$. גם כאן קיבלנו ביטוי לינארי התלוי באורך הקלט n .

דוגמה 4: האם מספר k מופיע בשני מערכים

הפעולה `existK(...)` מקבלת שני מערכים של שלמים ומספר שלם k , ומחזירה "אמת" אם k מופיע בשני המערכים, ו"שקר" אחרת. במימוש פעולה זו ניעזר ב-`search1(...)` שהגדרנו בסעיף הקודם:

```
public static boolean existK(int[] arr1, int[] arr2, int k)
{
    int res1 = search1 (arr1, k);
    int res2 = search1 (arr2, k);

    return (res1 != -1 && res2 != -1);
}
```

בשלב הראשון נניח ששני המערכים שווים בגודלם. נקבע גודל זה כאורך הקלט ונסמנו ב- n . נחשב כמה פעולות יסוד מתבצעות בפעולה. לפני התנאי מתבצע פעמיים אותו מהלך: מוגדר משתנה שלם שבו מוצבת תוצאת הזימון של `search1(...)` על אחד משני המערכים הנתונים.

זהו חידוש. הפעולה אינה "עצמאית", היא תלויה בפעולה אחרת שהיא מזמנת. השורות שבהן מופיע הזימון של `search1(...)` אינן יכולות להיחשב כפעולות יסוד. כאשר מזמנים את `search1(...)` יש להתחשב בכך שמספר פעולות היסוד שמבצעת פעולה זו בעצמה, הוא $3n+3$.

בכל זימון של `search1(...)` מתבצעות למעשה $3n+3$ הפעולות של `search1(...)` ועוד פעולת הצבה אחת. סך הכול $6n+8$ פעולות. פעולת ההחזרה, `return`, היא פעולה אחת נוספת. מספר פעולות היסוד של `existK(...)` הוא אם כן: $6n+9$.

נניח ששני המערכים אינם שווים בגודלם, כלומר הפעולה מתבצעת על שני אורכי קלט שונים, נסמנם ב- n_1 וב- n_2 בהתאמה. מספר פעולות היסוד של הפעולה $\text{existK}(\dots)$ יהיה תלוי אם כך באורך הקלט שהוא הסכום n_1+n_2 , ויראה כך: $(3n_1+3) + (3n_2+3) + 2 + 1 = 3(n_1+n_2) + 9$.

בהמשך נראה שאין הבדל משמעותי במספר פעולות היסוד בשני המצבים, ואין זה משמעותי אם מדובר בשני מערכים שווים בגודלם או בשניים שונים.

הערה: כאשר מחשבים את מספר פעולות היסוד של אלגוריתם המשתמש בפעולות עזר אחרות עלינו לדעת גם מהו מספר פעולות היסוד שמבצעות פעולות העזר. לעתים לא ניתן לחשב במדויק את מספר פעולות היסוד המתבצעות מכיוון שהאלגוריתם מזמן פעולת ממשק שלא אנו כתבנו, ואיננו יודעים מה קורה בתוכה. חשבו על מצב שבו אתם משווים מחרוזות ומשתמשים בפעולת הממשק: $\text{compareTo}(\dots)$ שמימושה אינו ידוע.

דוגמה 5: איברים משותפים לשני מערכים

הפעולה שלפניכם מקבלת שני מערכים שגודלם שווה. כל האיברים במערך הראשון שונים זה מזה. הפעולה תדפיס את כל המספרים מהמערך הראשון המופיעים במערך השני, וכמה פעמים הם מופיעים בו. לשם כך, נעבור על כל איבר במערך הראשון ונבדוק האם הוא מופיע במערך השני ונמנה את מספר מופעיו. נעיין בפעולה המבצעת את המשימה:

```
public static void sharedNums(int[] arr1, int[] arr2)
{
    int counter;

    for (int i = 0; i < arr1.length; i++)
    {
        counter = 0;
        for (int j = 0; j < arr2.length; j++)
            if (arr1[i] == arr2[j])
                counter ++;
        if (counter > 0)
            System.out.println(arr1[i] + " appears " +
                               counter + " times");
    }
}
```

נחשב את מספר פעולות היסוד שמבצעת הפעולה. אורך הקלט הוא גודל אחד מהמערכים. נסמנו ב- n . זכרו ששני המערכים שווים בגודלם.

בתחילת הפעולה מתבצעת פעולת יסוד אחת – הגדרת המשתנה `counter`.

החידוש בדוגמה זו הוא קיומן של שתי לולאות מקוננות. נבחן ראשית את הלולאה הפנימית שתפקידה למנות כמה פעמים מספר מהמערך הראשון מופיע במערך השני. הלולאה מתבצעת n פעמים, בכותרתה מתבצעות שתי פעולות יסוד. בכל מעבר של הלולאה מתבצעת בדיקת תנאי שהיא פעולת יסוד אחת. אם תנאי זה מתקיים, יקודם גם המונה.

כלומר הלולאה הפנימית תבצע $3n$ פעולות יסוד לכל היותר.

נבדוק את הלולאה החיצונית. גם היא מתבצעת n פעמים. בכותרתה מתבצעות 2 פעולות יסוד, וכן אתחולו של j (לפני הכניסה ללולאה הפנימית). בגוף הלולאה יש: אתחול שהוא פעולה אחת, לולאה פנימית המבצעת $3n$ פעולות ועוד פעולה בסיסית אחת שהיא בדיקת תנאי וביצוע הדפסה. סך כל פעולות היסוד שמבצעת הלולאה החיצונית הוא: $n(2+2+3n+1) = 3n^2+5n$.

לסיכום, הפעולה `sharedNums(...)` מבצעת $2+3n^2+5n$ פעולות יסוד. הפעם מופיע אורך הקלט בחישוב, בחזקה הגבוהה מ-1. זהו ביטוי התלוי באורך הקלט בריבוע. בהמשך הפרק נלמד מה משמעות הדבר.

3.3. מקרים טובים, ממוצעים וגרועים

עד כה התעלמנו מעובדה חשובה: עבור אורך קלט נתון n , יש בדרך כלל הרבה (לעתים קרובות אין-סוף) קלטים אפשריים, ומספר פעולות היסוד שתבצע פעולה על קלטים אלה אינו שווה בהכרח. בדרך כלל קיימים קלטים שונים, באותו האורך, שמספר פעולות היסוד המתבצע עליהם שונה מאוד. נחזור לפעולה `search1(...)` שהצגנו קודם, ונבחן את פעילותה עבור מערכים שונים בגודל n . אנו החלטנו לחשב את מספר פעולות היסוד המתבצעות במקרה שהמספר מופיע אחרון או אינו מופיע כלל במערך. במקרים אלה יתבצעו $3n+3$ פעולות. אבל, מה יקרה אם המספר שאנו מחפשים מופיע במערך, במקום הראשון? אז נבצע רק צעד חיפוש יחיד, במחיר של 4 פעולות יסוד בלבד. במילים אחרות, הביטוי $3n+3$ מתאר את מספר פעולות היסוד המתבצעות במקרה הגרוע ביותר, בכל מקרה אחר נבצע פחות פעולות יסוד.

יש שתי גישות אפשריות לחישוב מספר פעולות היסוד שמבצע אלגוריתם מסוים, כתלות באורך הקלט: חישוב מספר הפעולות לפי המקרה הגרוע ביותר (worst case), או חישוב מספרן לפי המקרה הממוצע (average case).

נמנה את עיקרי היתרונות והחסרונות של שתי הגישות, מתוך הנחה שזמן ריצה של אלגוריתם עומד ביחס ישר למספר פעולות היסוד שהוא מבצע:

(1) כאשר יש לפתור בעיה מסוימת במערכת פעמים רבות, מאות ואלפי פעמים (ואף יותר) ביום, המחיר הממוצע של פעולה הפותרת את הבעיה נותן לנו מושג טוב על המחיר הכולל (בזמן ריצה) שאנו משלמים בכל יום עבור פתרון הבעיה. במקרה כזה, סביר שנעדיף פעולה שזמן ריצתה הממוצע נמוך, על פני פעולה אחרת שזמן ריצתה הממוצע גבוה, אפילו אם היחס בין זמני הריצה במקרה הגרוע שלהן הפוך.

(2) כדי לחשב את זמן הריצה של האלגוריתם במקרה הממוצע, יש למצוא מהו הממוצע של זמני הריצה על פני כל הקלטים האפשריים באורך n . לעתים קרובות חישוב זה הוא קשה הרבה יותר לביצוע מחישוב זמן הריצה במקרה הגרוע ביותר.

(3) גם אם נדע מהו זמן הריצה במקרה הממוצע, לא נוכל להסיק דבר באשר להרצה אחת של הפעולה. ייתכן שזמן זה יהיה נמוך מהזמן הממוצע, ייתכן שהוא יהיה גבוה יותר במידה משמעותית מהזמן הממוצע. במערכות שבהן תגובה תוך פרק זמן קצר היא חשובה, התנהגות לא צפויה מעין זו היא בעייתית. הדבר דומה למדידת הגובה של תלמידים בכיתה מסוימת.

אם נדע שהגובה הממוצע הוא 1.65 מ', האם נוכל להסיק מכך משהו על גובהו של דני הלומד בכיתה זו? לעומת זאת, אם נדע שהתלמיד הגבוה ביותר בכיתה מתנשא לגובה של 1.80 מ', נוכל להגיד בוודאות כי גובהו של דני אינו עולה על 1.80 מ'. מדידת זמן הריצה במקרה הגרוע מספקת הערכה שניתן לסמוך עליה בכל ריצה, על כל קלט.

בפרק זה נתייחס רק לחישוב פעולות היסוד המתבצעות במקרה הגרוע ביותר, דהיינו לזמן הריצה הארוך ביותר.

? אפשר להגדיר גם את זמן הריצה במקרה הטוב ביותר. האם זהו מדד טוב ליעילות?

סיכום: בסיכום סעיף זה אנו יכולים לענות על השאלה שהצגנו בפתחו: איך מודדים יעילות? מספר פעולות היסוד שמבצע אלגוריתם במהלך ריצתו, במקרה הגרוע ביותר, הוא פונקציה של אורך הקלט. פונקציה זו, הנקראת פונקציית זמן הריצה של האלגוריתם, משמשת כמדד ליעילותו של האלגוריתם. נסמן את הפונקציה ב- $f(n)$.

מדד זה אינו תלוי במחשב מסוים, אך אפשר בקלות להעריך כמה זמן תיארך ריצתו של אלגוריתם על מחשב נתון. לצורך זה, נעריך מהו משך הביצוע של פעולת יסוד על מחשב זה, ונכפיל משך זמן זה בפונקציית זמן הריצה (שהיא כאמור מספר פעולות היסוד שהפעולה מבצעת).

בהמשך הפרק נשתמש במדד של פונקציית זמן הריצה כדי להעריך את יעילותם של אלגוריתמים.

ג. שיפור יעילות של אלגוריתם

פונקציית זמן ריצה של אלגוריתם מתארת את יעילותו. האם אפשר לשפר את יעילותו של אלגוריתם כלומר להקטין את מספר פעולות היסוד שהוא מבצע? נשתמש בבעיית החיפוש שהצגנו לעיל כדי להציג שתי דרכים לשיפור יעילות של אלגוריתם. הראשונה תציג שיפור קל ולא משמעותי, והאחרת תציג שיפור משמעותי יותר.

ג.1. שיפור יעילות בקבוע

נחזור לפעולה `search1(...)` הבודקת האם מספר נתון מופיע במערך ממוין בסדר עולה. אם האיבר נמצא במערך, הפעולה מחזירה את מיקומו, אחרת הפעולה מחזירה את הערך -1. לשם פשטות נניח כי המספרים במערך שונים זה מזה:

```
public static int search1(int[] arr, int k)
{
    int n = arr.length;
    for (int i = 0; i < n; i++)
    {
        if (arr[i] == k)
            return i;
    }
    return -1;
}
```

הראינו כי במקרה הגרוע פונקציית זמן הריצה של הפעולה היא : $3n+3$. n מסמן את אורך הקלט, שהוא למעשה גודל המערך.

ניתן לוותר על האתחול של n ובכותרת הלולאה להשתמש ב-`arr.length`. כך פונקציית זמן הריצה תרד בפעולה אחת, ותהיה : $3n+2$.

האם ניתן לחסוך יותר?

בתוך הלולאה מתבצעות שלוש פעולות : האחת בודקת האם מצאנו את האיבר, מכיוון שהיא עיקר העניין לא נוכל לוותר עליה. פעולה נוספת היא קידומו של i לאורך המערך, וגם עליה לא נוכל לוותר, שכן עלינו להתקדם על פני כל המערך. נתמקד בפעולה הבודקת האם חרגנו מגבולות המערך ($i < n$) וננסה לחסוך אותה.

לשם כך נתחיל בבדיקה האם הערך שאנו מחפשים נמצא בתא האחרון. אם כן, נחזיר את האינדקס של תא זה, ונסיים מיד את הבדיקה. אם לא, נכניס את המספר שאותו אנו מחפשים, לתא האחרון במערך ונשמור את הערך שהיה שם במשתנה `temp` (בסוף הפעולה נבצע שוב את ההחלפה, כך שהערכים במערך יישארו כשהיו).

לאחר הכנסה "מלאכותית" זו אנו יכולים לוותר על הבדיקה האם עומדים לחרוג מגבולות המערך, ולהסתפק בבדיקה "האם הערך בתא הנוכחי שווה ל- k ". אם האינדקס שיוחזר כשימצא הערך במערך – יהיה האינדקס של התא האחרון, הרי משמעות הדבר היא שהערך לא היה קיים במערך. אם הערך קיים במערך, הרי הפעולה תמצא אותו לפני שתגיע לתא האחרון והיא תחזיר את האינדקס שלו.

הורדת הפעולה $i < n$, משמעה חסכון ב- n פעולות. חיסכון זה נראה משמעותי.

נעקוב אחר הפעולה הבאה המציגה פתרון זה :

```
public static int search2(int[] arr, int k)
{
    int temp = arr[arr.length-1];

    if (temp == k)
        return arr.length-1;
    arr[arr.length-1] = k; // המספר שמחפשים יוכנס לתא האחרון
    int i=0;

    while(arr[i] != k)
        i++;

    arr[arr.length-1] = temp;

    if(i == arr.length-1)
        return -1;
    else
        return i;
}
```

נבדוק את מספר פעולות היסוד שמבצעת פעולה זו במקרה הגרוע ביותר (במקרה ש- k אינו במערך). מספר פעולות היסוד בלולאה הוא 2, ואם כך במהלך ביצוע הלולאה יתבצעו $2n$ פעולות. מספר הפעולות לפני ואחרי הלולאה הוא כ-7 (לא נעסוק בפרטי הספירה, כיוון שגם אם המספר הוא 8 או 10, זה לא ישפיע על הטיעון שאנו עומדים להציג, וכפי שהבטחנו, נסביר זאת עוד מעט...). אם כן, בסך הכול מספר פעולות היסוד הוא כ: $2n+7$. הצלחנו להקטין את מספר הפעולות המתבצעות בגוף הלולאה מ-3 ל-2, אך המחיר של האלגוריתם המתוחכם יותר הוא הגדלת מספר הפעולות המתבצעות מחוץ ללולאה מ-3 ל-7. שתי פונקציות זמן הריצה הן ביטויים לינאריים התלויים באורך הקלט n .

האם שיפרנו את האלגוריתם המקורי? כדי לענות על שאלה זו נבחן את פונקציות זמן הריצה של שתי פעולות החיפוש על אורכי קלט שונים:

אורך הקלט	פונקציית זמן ריצה של search1 : $3n+3$	פונקציית זמן ריצה של search2 : $2n+7$	היחס המקורב בין פונקציות זמני הריצה
1	6	9	1.5
3	12	13	1.08
5	18	17	0.94
10	33	27	0.81
100	303	207	0.68
1,000	3,003	2,007	2/3
30,000	90,003	60,007	2/3
3,000,000	9,000,003	6,000,007	2/3

מהטבלה נראה כי התחכום של הפעולה השנייה השתלם. לכל המערכים שאורכם 5 או יותר, הפעולה השנייה יעילה יותר מן הראשונה, ועבור n הגדול מ-100, הפעולה החדשה מורידה כ-33% ממספר פעולות היסוד שנעשו ב- $search1(\dots)$. היחס בין מספרי הפעולות הנעשות על ידי שתי הפעולות שווה (בקירוב גבוה מאוד) למנה $2/3$. מנה זו שווה ליחס בין המקדמים של n בשתי פונקציות זמן הריצה שאנו משווים. יתרונה של הפעולה $search2(\dots)$ נובע מכך שהיא מבצעת פחות פעולות יסוד בגוף הלולאה, שכן הפעולות שבגוף הלולאה מבוצעות מספר פעמים כאורך המערך, ואילו אלה שמחוץ ללולאה מבוצעות פעם אחת.

בנוסף, מהטבלה עולה מסקנה חשובה: הגורם המשמעותי לקביעת יעילות של אלגוריתם הוא אורך הקלט בלבד. הקבועים השונים הם חסרי משמעות כאשר עוסקים באורכי קלט גדולים, ולכן אין צורך להתחשב בהם בעת הדיון בהשוואת יעילותם של אלגוריתמים, אלא רק במקדמים של n .

הצלחנו לשפר את יעילות האלגוריתם בעזרת שינוי קל באלגוריתם. שיפור כזה ביעילות נקרא **שיפור בקבוע (improvement by factor)**, שכן היחס בין זמני הריצה של שתי הפעולות, כאשר

אורך הקלט גדול, הוא בקירוב קבוע. הטבלה שלעיל מדגימה כיצד ככל שגדל אורך הקלט, הולך היחס בין זמני הריצה ומתקרב לקבוע $2/3$.

הערה: אם פונקציית זמן הריצה של פעולה א' היא $5n+6$ וזו של פעולה ב' היא $4n+100,000$ אזי נכון כי עבור n מספיק גדול פעולה ב' עדיפה על פעולה א', אולם יש תחום לא זניח של ערכי n (לכל n הקטן מ-99,994) שבו פעולה א' עדיפה. במקרה כזה יתכן שנרצה להשתמש בכל אחת מהפעולות בתחום בו היא עדיפה. מקרים כאלה, שבהם הקבועים כה גדולים עד שלא ניתן להתעלם מהם, נדירים, וביחידת לימוד זו לא נעסוק במקרים כאלה כלל.

שיפור בקבוע הוא אמנם חשוב, אבל כפי שנראה מיד, לפעמים גישה אחרת לפתרון הבעיה – רעיון אלגוריתמי שונה – יכולה להוביל לשיפור ניכר יותר.

2.2. שיפור יעילות משמעותי יותר

נחזור לדוגמה של חיפוש מספר במערך. שתי הפעולות שהצגנו, $search1(...)$ ו- $search2(...)$, מממשות למעשה אותו רעיון אלגוריתמי, שלפיו עוברים על כל תאי המערך, לפי סדר, ומשווים את הערך שבתא לערך המבוקש. רעיון זה נקרא **חיפוש סדרתי (sequential search)**. העובדה ששתי הפעולות מבצעות חיפוש סדרתי מתבטאת בכך שלשתיהן אותה יעילות לינארית. בשני הפתרונות התעלמנו מהיות המערך ממוין בסדר עולה. האם העובדה שהמערך ממוין יכולה לסייע במציאת פתרון יעיל יותר מחיפוש סדרתי?

ההתייעלות היחידה שתושג מכך שהמערך ממוין היא שלעתים קרובות ניתן יהיה להפסיק את החיפוש לפני שמגיעים לסוף המערך. מכיוון שנתון שהמערך ממוין בסדר עולה, ניתן להפסיק את החיפוש כאשר נתקלים במערך במספר גדול מהמספר שאנו מחפשים. למרות שיפור זה היעילות תישאר לינארית.

? האם יש שינוי בחישוב היעילות עבור המקרה הגרוע ביותר, בשל העובדה שהמערך ממוין?

אנו מעוניינים לשנות את הגישה האלגוריתמית כך שתנצל את העובדה שהאיברים במערך ממוינים, ותציע שיפור משמעותי לבעיית החיפוש. ניזכר כיצד אנו מחפשים בספר טלפונים. אין אנו מבצעים חיפוש סדרתי בספר הטלפון – האם תוכלו לדמיין כמה זמן ייקח חיפוש כזה, וכמה משעמם יהיה? במקום זאת, אנו משתמשים בשיטה המנצלת את העובדה שספר הטלפונים **ממוין** – השמות בו מופיעים בסדר אלפביתי.

1.2.2. חיפוש בינרי

החיפוש בספר טלפונים מתבצע על פי שיטת החצייה: בשלב הראשון, חוצים את הספר לשניים, מגיעים לערך שבאמצע הספר ובודקים אם מצאנו את מבוקשנו. אם לא, מחליטים באיזה צד להמשיך בחיפוש, חוזרים וחוצים את מחצית הספר הנותרת וכך הלאה. בכל שלב ממשיכים למעשה את תהליך החיפוש רק על מחצית מהערכים שנותרו.

חיפוש בשיטת החצייה, המנצל את העובדה שספר הטלפונים ממוין, נקרא "חיפוש בינרי". ואותו הכרתם בלימודיכם הקודמים. נצל רעיון אלגוריתמי זה כדי לממש פעולת חיפוש יעילה יותר:

```
public static int binSearch(int[] arr, int k)
{
    int left = 0;
    int right = arr.length-1;

    while (left <= right)
    {
        int mid = (left + right)/2;
        if (arr[mid] == k)
            return mid;
        if (k < arr[mid])
            right = mid -1;
        else
            left = mid + 1;
    }
    return -1;
}
```

ג.2.2. חישוב זמן הריצה של חיפוש בינרי

אורך הקלט הוא גודל המערך המתקבל. נסמנו ב-n. פרט ללולאה הכלולה בקוד מתבצעות שלוש פעולות יסוד (שתיים לפני הלולאה ואחת אחריה).

נתמקד בלולאת ה-while ונבדוק כמה פעמים לולאה זו מתבצעת. הלולאה ממשיכה כל עוד left קטן מ-right או שווה לו, כלומר כל עוד יש במערך המצטמצם תאים לחיפוש. תנאי הסיום של הלולאה הוא צמצום גודלו של המערך העומד לחיפוש כך שלא יכלול עוד תאים (למעשה, כש-left יהיה גדול מ-right). טווח החיפוש מצטמצם כל פעם בחצי. נחשב כמה פעמים ניתן לחלק את n המקורי לשניים, עד לקיום תנאי העצירה של הלולאה. לשם פשטות נניח שגודל המערך הוא בדיוק חזקה של 2:

$$n / \underbrace{2 / 2 / 2 / 2 / \dots / 2}_k \text{ פעמים} = 1 \quad \text{אנו מחפשים } k \text{ אשר יקיים:}$$

$$n / \underbrace{(2 \cdot 2 \cdot 2 \cdot 2 \cdot \dots \cdot 2)}_k \text{ פעמים} = 1 \quad \text{או:}$$

$$1 \cdot \underbrace{2 \cdot 2 \cdot 2 \cdot 2 \cdot \dots \cdot 2}_k \text{ פעמים} = n \quad \text{נעביר אגפים ונקבל:}$$

$$2^k = n \quad \text{כלומר:}$$

$$k = \log_2 n \quad \text{ומכאן:}$$

k הוא מספר החלוקות המקסימלי שיידרש כדי להגיע בוודאות לתשובה מוחלטת. למעשה, זהו מספר האיטרציות שלולאת ה-while תבצע במקרה הגרוע ביותר.

לדוגמה, במערך שגודלו $n=1,000,000$, לולאת ה-**while** תתבצע כ-20 פעם. עבור מערך בגודל 1,000,000,000 (מיליארד) לולאת ה-**while** תתבצע כ-30 פעם. למספר 30 הגענו לפי הנוסחה שקיבלנו לעיל:

$$k = \log_2 n:$$

$$k = \log_2 1,000,000,000 \cong 30$$

אם כן, לולאת ה-**while** מבצעת $\log_2 n$ איטרציות ובתוכה מתבצעות כ-5 פעולות יסוד, אזי בסך הכול נאמר שהלולאה כולה מבצעת $5\log_2 n$ פעולות יסוד. בתוספת הפעולות שמחוץ ללולאה נקבל שפונקציית זמן הריצה של הפעולה $\text{binSearch}(\dots)$ היא $5\log_2 n + 3$, כלומר היא פונקציה לוגריתמית.

הטבלה הבאה משווה את פונקציות זמני הריצה של חיפוש סדרתי וחיפוש בינרי:

היחס המקורב בין פונקציות זמני הריצה	פונקציית זמן ריצה של binSearch : $5\log_2 n + 3$	פונקציית זמן ריצה של search2 : $2n + 7$	אורך הקלט
0.741	20	27	10
0.174	36	207	100
0.026	53	2,007	1,000
0.0034	69	20,007	10,000
0.00043	86	200,007	100,000
0.000051	103	2,000,007	מיליון
$7.6 * 10^{-8}$	152	2,000,000,007	מיליארד

מהטבלה ניתן לאשר באופן פורמלי את מה שאתם יודעים מזמן: החיפוש הבינרי יעיל בצורה משמעותית מהחיפוש הסדרתי, עבור מערך בגודל כלשהו. כלומר החיפוש הבינרי מציע שיפור גדול לבעיית החיפוש. ככל שהמערכים יגדלו השיפור יהיה משמעותי יותר. היחס בין זמני הריצה של האלגוריתמים הולך ומשתפר (לטובת החיפוש הבינרי) ככל שאורך הקלט גדל (השוו יחס זה לשיפור בקבוע, שם היחס בין זמני הריצה קבוע).

ד. הערכה כללית של יעילות – סדר גודל

האלגוריתם הפשוט והאלגוריתם המשופר לחיפוש מספר במערך ממוין דורשים במקרה הגרוע ביצוע של n צעדי חיפוש, כאשר n הוא אורך הקלט. ככל שננסה לדקדק בחישוב זמן הריצה, נקבל בסופו של דבר פונקציה מהצורה $f(n) = an + b$, הידועה בשם פונקציה לינארית, אשר בה a ו- b קבועים כלשהם. היחס בין כל שתי פונקציות מסוג זה יהיה קבוע עבור n גדול מספיק. לכן נאמר כי פונקציות זמני הריצה של אלגוריתמים אלה הן **מסדר גודל לינארי**. כיוון שהפונקציה הלינארית הפשוטה ביותר היא $f(n) = n$, נהוג לסמן יעילות מסדר גודל לינארי בסימון $O(n)$, קרי:

"או-גדול של n". באומרנו שסדר הגודל של יעילות של פעולה הוא $O(n)$ אנו מתכוונים לכך שיש תלות ישירה בין אורך הקלט ליעילות הפעולה, ואין שום גורם משמעותי אחר התורם למדד זה. כאן אנו ממלאים את הבטחתנו מראשית הפרק, להראות שכאשר פונקציות זמן הריצה של פעולות דומות במבנה שלהן אין הבדל משמעותי בין יעילותן. במילים אחרות, אין הבדל ביעילותן של פעולות הדומות בתלות שלהן ב-n ושונויות רק בגודל הקבועים או בגורמים הכופלים את n, שהוא אורך הקלט. בדוגמה 2 – מציאת סכום איברים במערך – טענו שאין הבדל בין הפונקציות $3n+3$, ו- $(4+3n)$, וששתיהן מציגות אותה היעילות. בדוגמה הרביעית – כאשר בדקנו אם k מופיע בשני מערכים נתונים – טענו שאין הבדל משמעותי בין הפונקציה המדויקת $3(n_1+n_2) + 12$ המתייחסת במדויק לשני מערכים באורכים שונים לבין הפונקציה הכללית יותר $6n+12$. בטענה זו כבר רמזנו למושג "סדר גודל", הנותן משמעות רק לתלות באורך הקלט ומצמצם את משמעות הקבועים הנלווים ל-n.

נאמר באופן כללי שבסיווג ראשוני של זמני ריצה של אלגוריתמים "נתעלם" מהבדלים דקים של קבועים בין פונקציות של זמן ריצה ונקבצן יחד תחת קורת גג אחת, שלה נקרא "סדר גודל" ושתשקף את התלות באורך הקלט בלבד. אנו אומרים כי שתי פונקציות זמן ריצה הן מאותו סדר גודל, אם היחס בין ערכיהן מתקרב ליחס קבוע עבור n מספיק גדול. קיימות כמה משפחות שימושיות של סדרי גודל, ונעמוד עליהן בהמשך.

א. האם הפונקציה $f(n) = n^2$ היא מסדר גודל לינארי?

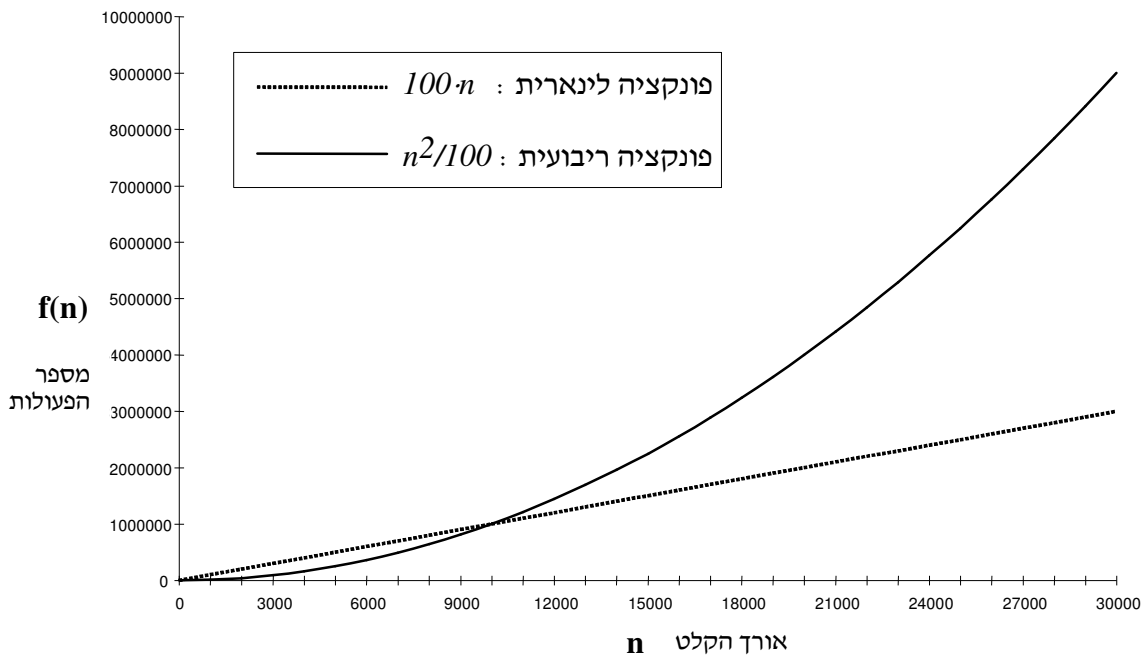
ב. האם הפונקציות $f(n) = n^2/100$ ו- $g(n) = n^2/10$ הן מאותו סדר גודל?

בדומה לסדר גודל לינארי, ניתן להגדיר גם **סדר גודל ריבועי**, המסומן ב- $O(n^2)$. זוהי משפחת כל הפונקציות שהיחס ביניהן לבין הפונקציה הריבועית הפשוטה ביותר, $f(n)=n^2$, הוא קבוע עבור n גדול מספיק. כשאנו אומרים שפונקציית זמן הריצה של אלגוריתם היא מסדר גודל ריבועי, פונקציה זו יכולה להיות n^2 , $5n^2+6$, $5n^2+100n+6$, או $5,000n^2$. גם אם זמן ריצתו של האלגוריתם הוא רק חלק מ- n^2 , למשל $n^2/6$, עדיין נאמר שזמן הריצה שלו הוא $O(n^2)$. אנו כוללים את כל הפונקציות האלה במשפחה אחת, תוך התעלמות מקבועים. אנו מגדירים את המשפחה על פי התלות המשמעותית ביותר באורך הקלט וזו כמובן תהיה התלות ב- n^2 .

רעיון זה נראה אבסורדי במקצת: האם ייתכן שנשכן אלגוריתם שזמן ריצתו n^2 ואלגוריתם שזמן ריצתו $n^2/100$ תחת קורת גג אחת?

האיור הבא מבהיר כי כאשר אנו משווים זמני ריצה אלו לזמני ריצה לינאריים, סדר הגודל הוא המכריע, וכי עבור n גדול מספיק הקבועים הופכים לחסרי משמעות. כך, אף על פי שהמקדם של הפונקציה הלינארית באיור הוא 100, ואילו המקדם של הפונקציה הריבועית הוא 1/100 (קטן פי 10,000), הרי שהחל ב-n מסוים ערך הפונקציה הריבועית גדול מערך הפונקציה הלינארית. היחס בין שתי הפונקציות הולך וגדל גם הוא, ככל שגדל ערכו של n.

אם לפתרון בעיה נתונים לנו שני אלגוריתמים ממשפחות שונות של סדרי גודל, האם נעדיף תמיד את האלגוריתם בעל סדר גודל היעילות הנמוך? התבוננו באיור הבא המציג פונקציות זמני ריצה מסדר גודל לינארי וריבועי:



עבור $n > 10,000$ אכן מתקיים: $n^2/100 > 100n$, אבל עבור $n < 10,000$ מתקיים ההפך: $n^2/100 < 100n$. בשל כך אפשר לומר, שאם נדע שאורכי הקלטים הצפויים הם לכל היותר 10,000, נעדיף אלגוריתם שזמן ריצתו $n^2/100$, אף על פי שפונקציית זמן הריצה שלו גדולה יותר מ- $100n$, כלומר שיעילותו באופן כללי, פחות טובה. אם ידוע לנו כי חלק ניכר מהקלטים הצפויים יהיו ארוכים מ-10,000, נעדיף את האלגוריתם הלינארי.

בנוסף לפונקציות זמן הריצה הקבועה, הלינארית והריבועית, הכרנו גם את פונקציית זמן הריצה הלוגריתמית המאפיינת את החיפוש הבינרי. פונקציה זו היא **מסדר גודל לוגריתמי** ומסומנת ב- $O(\log n)$.

לסיום הדיון במושג סדר גודל נציין כי נהוג לסמן את סדר הגודל של פונקציה קבועה, שאינה תלויה באורך הקלט (כגון משך ביצועה של פעולה בסיסית באלגוריתם), ב- $O(1)$.

הערה: בדיונים ביעילות של אלגוריתמים משתמשים (כמעט) רק בבסיס 2 עבור לוגריתמים, ולכן מקובל לכתוב $\log n$, כלומר להשמיט את בסיס הלוגריתם.

בהמשך הפרק נשתמש בסדרי גודל כדי להעריך את יעילותם של אלגוריתמים, ולרוב לא נזדקק לחישוב פונקציית זמן ריצה מדויקת.

1.4. צעד בסיסי

לאחר שהסכמנו שאין צורך לספור במדויק כל פעולה בסיסית ודי להתרכז בסדר הגודל של האלגוריתם, נוכל להגדיר מושג נוסף שלעתיים יקל עלינו בניתוח אלגוריתמים. בכל אלגוריתם

אפשר למצוא פעולה או כמה פעולות בסיסיות המשמשות יחד **צעד בסיסי** שעליו חוזר האלגוריתם במהלך ריצתו. משך ביצועו של צעד בסיסי זה, כמו משך ביצועה של פעולה בסיסית, אינו תלוי באורך הקלט, כלומר הוא נמשך זמן קבוע. כך למשל, אלגוריתם המסכם איברי מערך חוזר כמה פעמים על הצעד: "הוסף את האיבר הבא במערך לסכום המצטבר". אלגוריתם שבודק האם מספר x הוא ראשוני, יחזור על הצעד: "בדוק האם המספר y מחלק את x ". אלגוריתם המכפיל שני מספרים מבצע שני סוגים של צעדים בסיסיים: הכפלת שתי ספרות וחיבור שתי ספרות. לעומת זאת הפעולה "מצא האם איבר x נמצא במערך A " אינה צעד בסיסי, שכן משך ביצועה תלוי באורך הקלט, כלומר בגודל המערך.

לעתים קרובות יהיה נוח יותר להצביע על צעד בסיסי ולחשב כמה פעמים האלגוריתם מבצע את הצעד כולו, מאשר לחשב במדויק את מספר הפעולות הבסיסיות המתבצעות באלגוריתם. מספר הפעמים שהאלגוריתם מבצע את הצעד הבסיסי במהלך ריצתו תלוי באורך הקלט ויכול לשמש מדד טוב ליעילותו.

2.4. שיפור יעילות בסדר גודל – דוגמה נוספת

כדוגמה נוספת לשיפור יעילות בסדר גודל נבחן שני אלגוריתמים למיון מערך. הצורך במיון יעיל עולה במגוון רחב ביותר של שימושים במחשבים. כך, למשל, באחד הסעיפים הקודמים הוצג אלגוריתם לחיפוש בינרי של מספר במערך ממוין. אם המערך אינו ממוין, לא ניתן להפעיל את האלגוריתם ולשפר את זמן הריצה.

כפי שראינו, האיברים המאוחסנים במערך אינם בהכרח מספרים. כאשר ברצוננו למיין מערך שבו שמור מידע מורכב, עלינו להחליט תחילה לפי איזה **מפתח** ימוין המערך. המפתח צריך להיות ערך המאפיין את המידע השמור במערך ושערכיו ניתנים לסידור. בספר טלפונים המפתח הוא שם המשתמש. במאגר משרד הפנים, המפתח הוא מספר הזהות. לשם הפשטות, אנו נניח כאן, כפי שכבר הנחנו קודם, שנתון לנו מערך באורך n של מספרים, שונים זה מזה שאותו אנו מעוניינים למיין.

1.2.4. מיון בועות

האלגוריתם הראשון שנבחן הוא האלגוריתם **מיון בועות** (bubble sort), המוכר לכם מלימודיכם הקודמים. המיון מתבסס על סדרת מעברים על המערך, ובכל מעבר האיבר הגדול ביותר "מבעבע" לסוף המערך. שטח המיון לכל שלב הולך ומצטמצם.

לפניכם הפעולה המבצעת מיון בועות:

```
public static void bubbleSort(int[] arr)
{
    for (int i = 1; i <= arr.length-1; i++)
    {
        for (int j = 0; j < arr.length-i; j++)
        {
            if (arr[j] > arr[j+1])
            {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

נחשב את סדר הגודל של הפעולה `bubbleSort(...)`. כפי שניתן לראות הפעולה מורכבת משתי לולאות מקוננות התלויות באורך הקלט, ולכן סדר הגודל של הפעולה יהיה $O(n^2)$. ישנם אלגוריתמים נוספים למיון מערך שהם מסדר גודל ריבועי. נשאלת השאלה, האם יש אלגוריתמים יעילים יותר?

2.2.4. מיון מיזוג

נחשוב על גישה שונה לפתרון הבעיה של מיון מערך, כך שנמצא אלגוריתם יעיל יותר, וניזכר בתהליך המיזוג של מערכים שהכרתם ביחידת הלימוד "יסודות". כדי למיין מערך ניתן לחלקו לשני תת-מערכים שווים פחות או יותר בגודלם, למיין כל אחד מהם, ולאחר מכן למזג בין המערכים הממוינים.

מהי דרך המיון שנשתמש בה כדי למיין את שני חלקי המערך? אפשר להשתמש, למשל, באלגוריתם למיון בועות שהכרנו בסעיף הקודם, אולם ניתוח פשוט יראה כי לא נשיג שיפור ביעילות האלגוריתם למיון; היעילות תישאר ריבועית.

המפתח לשיפור במקרה זה הוא שימוש ברקורסיה. את שני חלקי המערך נוכל למיין באותה גישה: נחלק כל אחד מהחלקים לשני חלקים נוספים, נמיין שוב כל חלק (שוב באותה שיטה), ונחזור ונמזגם לשני תת-מערכים ממוינים. בחלוקות החוזרות אנו יוצרים קטעים קטנים יותר ויותר של המערך. כאשר מגיעים לקטע באורך 1, הוא כמובן ממוין, ולכן אין צורך בזימון רקורסיבי למיונו. קטע באורך 1 הוא אם כן בסיס הרקורסיה.

האלגוריתם הבא מציג את שיטת המיון **מיון מיזוג (merge sort)**, תוך שימוש במערכי עזר של מספרים שלמים. כמו כן, שימו לב, האלגוריתם משתמש בפעולת עזר למיזוג מערכים:

מיון-מיזוג (arr)

אם גודל המערך גדול מ-1, בצע:

(1) חלק את המערך לשני תת-מערכים $arr1$ ו- $arr2$,

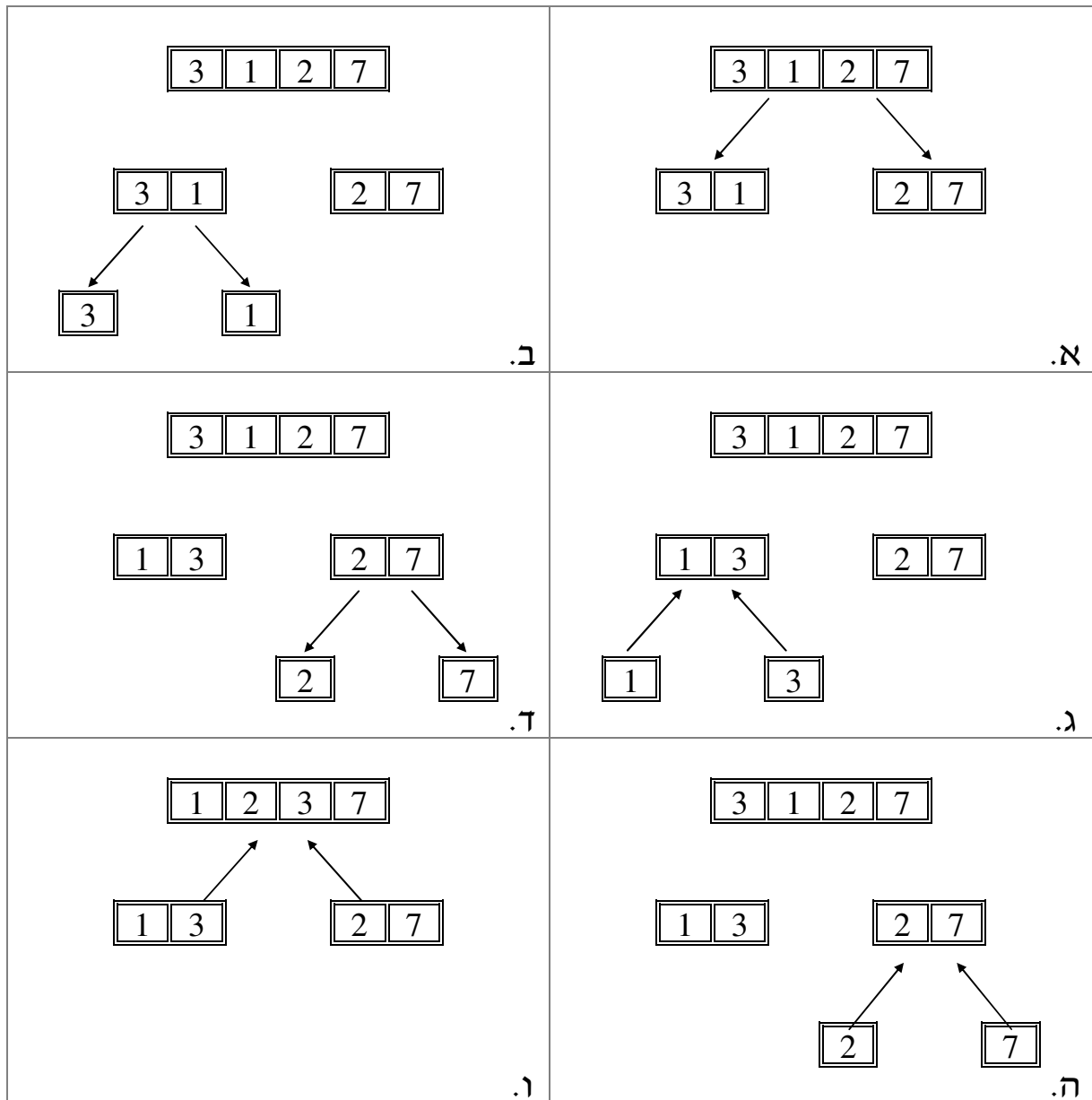
המכילים כל אחד כמחצית מאיברי המערך

(2) **מיון-מיזוג ($arr1$)**

(3) **מיון-מיזוג ($arr2$)**

(4) מזג את $arr1$ ואת $arr2$ לתוך arr

האיור הבא מתאר את ריצת האלגוריתם על מערך שבו ארבעה איברים. בשלב א' המערך הראשוני, שבו ארבעה איברים, מחולק לשני תת-מערכים בגודל 2 (שורה 1). בשלב ב' קוראים ברקורסיה לאלגוריתם המיון על התת-מערך הראשון (שורה 2). קריאה רקורסיבית זו יוצרת שני תת-מערכים בגודל 1, ובשלב זה נעצרת הרקורסיה. בשלב ג' ממוזגים שני תת-מערכים אלה לתוך המערך המקורי בגודל 2, וכעת הוא ממוין. בשלבים ד' ו-ה' מבוצעת אותה הסדרה של פעולות על התת-מערך השני. סדרת פעולות זו מופעלת בשל הקריאה הרקורסיבית השנייה (שורה 3). לבסוף, בשלב ו', ממוזגים שני מערכים אלה למערך ממוין בגודל 4 (שורה 4).



נחשב את סדר הגודל של האלגוריתם למיון-מיזוג, הפועל על מערך שבו n איברים. נניח לשם פשטות ש- n הוא חזקה של 2. ברמת הרקורסיה הראשונה מחולק המערך לשני מערכים, שבכל אחד מהם $n/2$ איברים. ברמת הרקורסיה השנייה מחולקים מערכים אלה לארבעה מערכים, שבכל אחד מהם $n/4$ איברים, וכך הלאה. הרקורסיה נעצרת כאשר המערך חולק ל- n מערכים, שבכל אחד מהם איבר בודד.

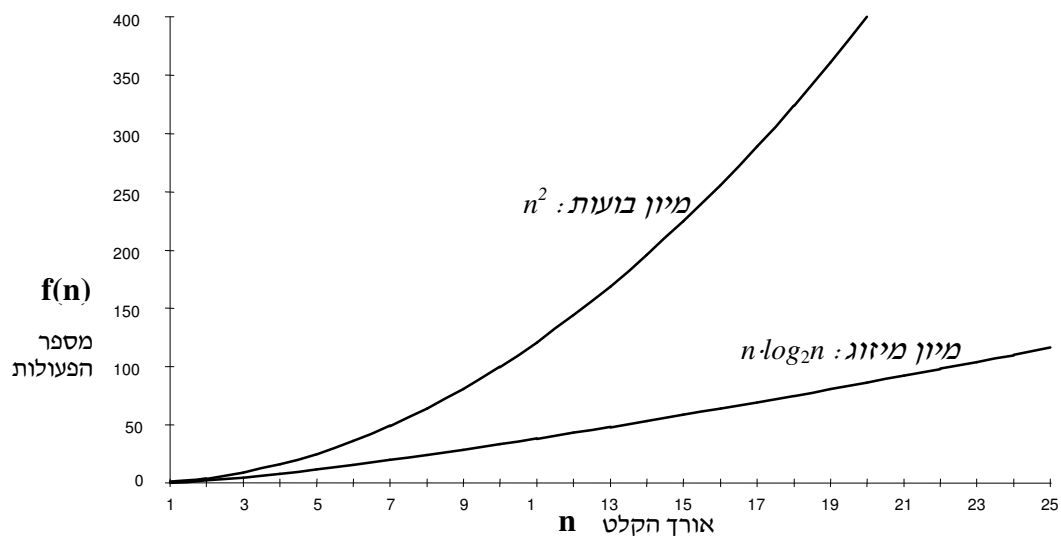
נבחן את מהלך הרקורסיה בכל אחת מרמותיה. ברמה הראשונה אורך המערך הבודד הוא n . שורה (4) ממוזגת שני תת-מערכים למערך אחד באורך n . מיזוג של שני מערכים מחייב סריקה ומעבר על כל איבריהם פעם אחת, כלומר הוא מסדר גודל לינארי, של סכום אורכי המערכים, $O(n)$. יידרש עוד מספר קבוע של פעולות לצורך חלוקת המערך לשניים ומיזוגו מחדש, בהתאם למימוש שנבחר.

יש קושי בחישוב מספר הפעולות המדויק שמבצע האלגוריתם, אך כבר הראינו שדי להבין באופן כללי את התלות שלו באורך הקלט כדי להגדיר את סדר הגודל של יעילות הפעולה.

רמת הרקורסיה השנייה מבוצעת על שני מערכים באורך $n/2$. על כל אחד ממערכים אלה מבוצעים $c \cdot n/2$ צעדים. בסך הכול מבוצעים ברמת הרקורסיה השנייה $2(c \cdot n/2) = c \cdot n$ צעדים. באופן דומה ניתן לוודא כי בכל אחת מרמות הרקורסיה מבוצעים $c \cdot n$ צעדים.

האלגוריתם מורכב מזימונים רקורסיביים, שלכל אחד מהם נשלחת מחצית המערך שהיה בשלב הקודם, לצורך מיון מיזוג. כמה זימונים שכאלה מתבצעים? הדיון על מספר הזימונים כאן דומה מאוד לזה שביצענו באלגוריתם לחיפוש בינרי. במקרה הכללי, כאשר המערך מכיל n תאים, מספר החציות שלו הוא $\log n$, בכל חצייה מתבצעים n מיזוגים, לכן יעילות האלגוריתם הכוללת היא $O(n \cdot \log n)$.

סדר גודל $n \cdot \log n$ מהווה שיפור של ממש בהשוואה לאלגוריתם מיון בועות, שהוא בעל יעילות ריבועית, $O(n^2)$. על מערכת הצירים הבאה נראית ההשוואה בין שני סדרי הגודל. השוואה זו מראה כי קיים הבדל ניכר ביעילות האלגוריתמים כבר במערכים קצרים, הבדל שהולך וגדל ככל שגדל אורך המערך.



בתרגילים שבסוף הפרק תתבקשו לממש את האלגוריתם "מיון מיזוג" כפעולה חיצונית על מערכים.

ה. פונקציית זמן ריצה מעריכית

נחזור לבעיית "הסוכן הנוסע" ולבעיות דומות לה, הנחשבות בעיות שאין להן פתרונות יעילים. באלגוריתמים אלה פונקציות זמן הריצה גדלות בקצב מהיר כל כך, שאפילו בקלטים קטנים אי-אפשר להשתמש בהם לפתרון הבעיות.

אילו פונקציות נחשבות כמתארות זמן ריצה בלתי סביר? משפחת הפונקציות החשובה ביותר המתארת זמני ריצה כאלה היא משפחת הפונקציות המעריכיות (exponential functions). בפונקציות אלה מופיע אורך הקלט כמעריך של חזקה.

הפונקציה 2^n היא פונקציה מעריכית. את האלגוריתם שזמן ריצתו $O(2^n)$ לא יהיה כדאי לממש, אלא במקרים שבהם ברור שהקלטים יהיו קצרים במיוחד. על ידי שימוש באלגוריתם כזה ניתן לפתור רק מופעים קטנים של הבעיה, כיוון שגידול קטן בגודל הבעיה מוביל לגידול משמעותי מאוד במספר הפעולות. כך, למשל, קלט שאורכו 10 ידרוש $2^{10} = 1,024$ פעולות, ולכן במקרה זה יהיה אפשר להיעזר במחשב. קלט שאורכו 20 ידרוש $2^{20} \sim 1,000,000$ פעולות – וגם זה עדיין מספר פעולות סביר לביצוע על ידי מחשב. אולם קלט שאורכו 100 ידרוש $2^{100} = 1.26 \cdot 10^{30}$ פעולות וביצוע האלגוריתם כבר לא יהיה אפשרי.

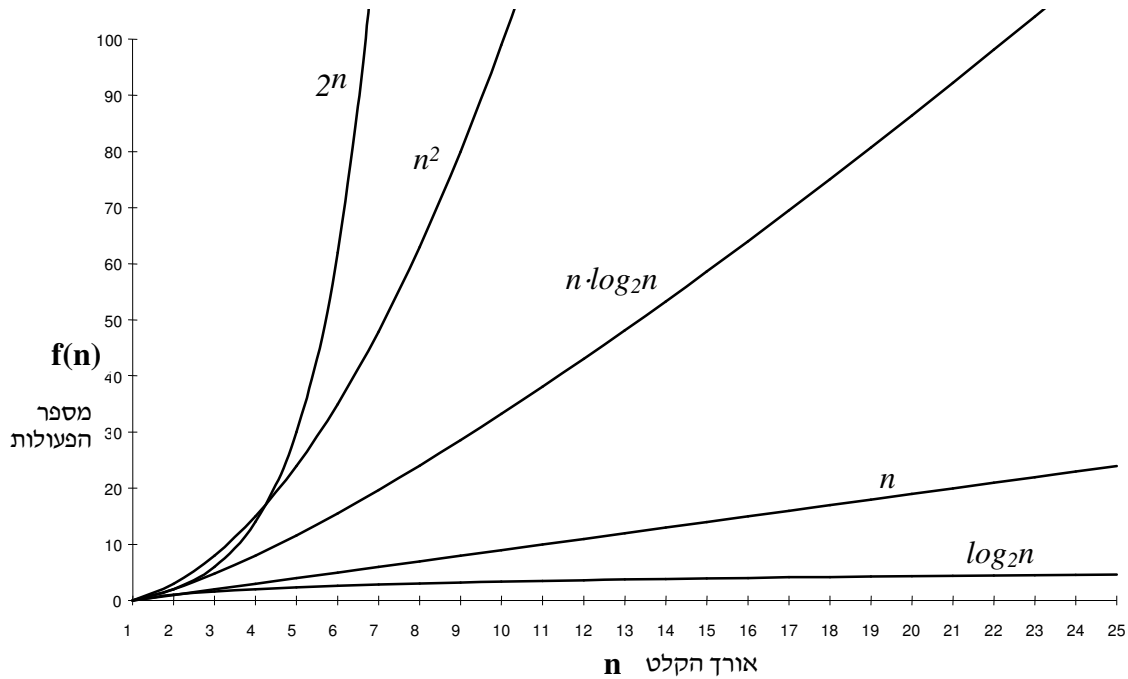
מובן שקיימות פונקציות זמן ריצה שגדלות בקצב מהיר אף יותר מפונקציות מעריכיות פשוטות. הפונקציות 2^n , $n \cdot 2^n$, n^n ו- $n!$ גדלות בקצב מהיר יותר מפונקציות מעריכיות, וכולן נחשבות בלתי סבירות מבחינת זמן הריצה שלהן.

ו. משפחות של סדרי גודל – סיכום

סדרי הגודל שהכרנו עד כה, הלינארי $O(n)$, הריבועי $O(n^2)$ והלוגריתמי $O(\log n)$, אינם היחידים. קיימים גם סדרי הגודל $O(n^k)$ לכל מספר חיובי k . כשם שהפונקציות במשפחה $O(n^2)$ גדלות מהר יותר מאלה שבמשפחה $O(n)$, כך אלה שבמשפחה $O(n^3)$ גדלות מהר יותר מאלה שבמשפחה $O(n^2)$. גם סדר הגודל $(n \log n)$ מאפיין אלגוריתמים רבים. פונקציה שזה סדר הגודל שלה, גדלה יותר מהר מפונקציה לינארית (אם כי המנה שלהן גדלה לאט, כיוון ש- $\log n$ גדלה לאט), וכמובן שהיא גדלה לאט יותר מפונקציות שסדר הגודל שלהן ריבועי.

1.1. גרפים של סדרי גודל

לסיכום נתבונן במערכת הצירים הבאה שעליה משורטטים גרפים שונים המציגים פונקציות מסדרי גודל שונים. הדוגמאות מדגישות את השיפור בסדר הגודל. שימו לב להבדלים הגדלים והולכים ככל ש-n גדל.



2.1. טבלה מסכמת

נסכם את הפעולות שניתחנו בפרק עד כה, ונציין לכל אחת את סדר הגודל של יעילותה. הטבלה מסודרת כך שסדר הגודל של היעילות הולך וגדל במורד הטבלה. סדר הגודל הקבוע הוא הקטן מכל סדרי הגודל הקיימים וממנו והלאה סדרי הגודל הולכים וגדלים:

סדר הגודל	הסימון	הסבר ודוגמאות מייצגות
קבוע	$O(1)$	זהו סדר הגודל של פעולות שאינן תלויות באורך הקלט (פונקציית זמן הריצה היא מספר קבוע)
לוגריתמי	$O(\log n)$	סדר גודל זה מתאים פעמים רבות לפעולות בהן אורך הקלט מצטמצם בכל שלב כדי מחציתו. דוגמה: חיפוש בינרי
לינארי	$O(n)$	פונקציות זמן ריצה מהצורה $f(n) = an+b$. דוגמה: חיפוש סדרתי
לינארי-לוגריתמי	$O(n \log n)$	דוגמה: אלגוריתם למיון מיזוג

פונקציות זמן ריצה שהן פולינום מדרגה $k > 1$, $f(n) = an^2 + bn + c$ דוגמה : מיון בועות הוא מסדר גודל $O(n^2)$ ריבועי, $k=2$	$O(n^k)$	פולינומיאלי
דוגמה : אלגוריתם לפתרון בעיית "מגדלי האנוי"	$O(2^n)$	מעריכי
דוגמה : אלגוריתם "הסוכן הנוסע"; אלגוריתם להדפסת קומבינציות	$O(n!)$	עצרת

בכל הבעיות שבהן נצטרך לבחור בין אלגוריתמים שונים לפתרון בעיות, נוכל להיעזר בטבלה כדי להשוות בין אלגוריתמים ממשפחות שונות. בדרך כלל נעדיף את האלגוריתם שסדר הגודל שלו נמוך יותר, אך כפי שראינו, בדיקה מדויקת של אורכי קלט שונים עלולה להצביע על תחומים שבהם אלגוריתם בעל סדר גודל גדול יותר – עדיף. במקרים שבהם נצטרך לבחור בין אלגוריתמים מאותה המשפחה, נצטרך לשוב אל פונקציית זמן הריצה ולחשבה באופן מדויק, כדי להכריע בין האפשרויות השונות.

ז. סיכום

- לבעיות נתונות קיימים פתרונות אלגוריתמיים שונים. מדדי זמן ומקום מאפשרים להשוות בין הפתרונות השונים. יעילותו של פתרון אלגוריתמי נקבעת לפי זמן ביצועו ולפי מרחב הזיכרון שהוא צורך.
- פעולת יסוד היא כל פעולה פשוטה שמבצע המחשב, והיא אינה תלויה באורך הקלט.
- מדד זמן מוגדר על פי התלות שבין מספר הפעמים שהאלגוריתם מבצע את פעולות היסוד לבין אורך הקלט. מדד זה נותן הערכה של משך זמן הריצה של האלגוריתם עבור קלט באורך נתון.
- פונקציית זמן הריצה של אלגוריתם מתארת את מספר פעולות היסוד שמבצע אלגוריתם על קלט באורך n , במקרה הגרוע ביותר.
- אפשר לשפר את יעילותו של אלגוריתם על ידי הקטנת מספר ההוראות הנלוות לפעולות היסוד. שיפור כזה מכונה **שיפור בקבוע**, והוא מאופיין ביחס קבוע בין זמני הריצה של האלגוריתם הראשוני והאלגוריתם המשופר, עבור אורכי קלט גדולים מספיק.
- רעיון אלגוריתמי שונה לפתרון בעיה נתונה יכול להוביל ל**שיפור בסדר גודל**, המאופיין ביחס הולך וגדל בין זמני הריצה של האלגוריתם המקורי ושל האלגוריתם המשופר, ככל שגדל אורך הקלט. שיפור בסדר גודל מועיל בדרך כלל יותר משיפור בקבוע.
- נהוג להגדיר משפחות של סדרי גודל, המאגדות פונקציות זמני ריצה שהיחס ביניהן מתקרב לקבוע, באורכי קלט גדולים מספיק. המשפחות שהכרנו בפרק הן: $O(\log n)$, $O(n)$, $O(n^k)$, $O(2^n)$.

- הכרנו שני אלגוריתמים לחיפוש במערך ממוין :
 - חיפוש סדרתי שפונקציית זמן הריצה שלו לינארית.
 - חיפוש בינרי שפונקציית זמן הריצה שלו לוגריתמית.
- הכרנו שני אלגוריתמים למיון מערך :
 - מיון בועות שפונקציית זמן הריצה שלו ריבועית.
 - מיון מיזוג שפונקציית זמן הריצה שלו $O(n \cdot \log n)$.
- קיימות בעיות שלפתרון ידועים רק אלגוריתמים הדורשים זמן ריצה בלתי סביר.
- אלגוריתמים שיעילותם מעריכית, או פחות טובה ממעריכית, אינם ישימים במחשב אלא לאורכי קלט קצרים במיוחד.

מושגים

input size	אורך קלט
run time	זמן ריצה
binary search	חיפוש בינרי
linear search	חיפוש סדרתי
efficiency	יעילות
merge sort	מיון מיזוג
worst case	מקרה גרוע ביותר
best case	מקרה טוב ביותר
average case	מקרה ממוצע
order of magnitude	סדר גודל
elementary step	פעולת יסוד
improvement by a factor	שיפור בקבוע

תרגילים

שאלה 1

מהו אורך הקלט בכל אחד מהאלגוריתמים שלפניכם, וכמה פעולות בסיסיות מתבצעות בכל אחד?

א.

```
public static void stam1(int n, int big)
{
    int num;
    int count = 0;

    for (int i=1; i<=n; i++)
    {
        num = (int)(Math.random()*100);
        if (num>big)
        {
            System.out.println(num);
            count++;
        }
    }

    System.out.println(count);
}
```

ב.

```
public static void stam2(int[] vec)
{
    int sum = 0;

    for (int i=0; i<vec.length; i=i+2)
        sum = sum + vec[i];
    int avg = sum / (vec.length/2);

    System.out.println(avg);
}
```

ג.

```
// הנחה: המטריצה המתקבלת ריבועית
public static void stam3(int[][] mat)
{
    int sum = 0;

    for (int i=0; i<mat.length; i++)
        for (int j=0; j<mat.length; j++)
            sum = sum + mat[i][j];
    double avg = (double)sum/(mat.length*mat.length);

    System.out.println(avg);
}
```

ד.

```
// הנחה: המטריצה המתקבלת ריבועית
public static void stam4(int[][] mat)
{
    int sum = 0;

    for (int i=0; i<mat.length; i++)
        for (int j=0; j<=i; j++)
            sum = sum + mat[i][j];

    System.out.println(sum);
}
```

שאלה 2

א. לפניכם רשימה של זמני ריצה. מיינו את הזמנים בסדר יעילות יורד.

1. $10n$

2. n^2

3. 2^n

4. $n^2 + n$

5. n^3

6. $n \cdot \log n + n$

7. $2^n + n \cdot \log n$

8. $n \cdot \log n + 10 \cdot \log n$

ב. האם תשובתכם ל-א תשתנה אם נתון: $n = 10$, $n = 25$? נמקו.

ג. מהם סדרי הגודל של זמני הריצה שבסעיף א?

שאלה 3

האם זוג הפונקציות הרשומות בכל סעיף הוא מסדר גודל זהה?

1. $50 \cdot n + 100$ $2 \cdot n^2$

2. $300 \cdot \log n + 6$ $n / 100$

3. $6 \cdot n^2 + n$ n^2

4. $6 \cdot n^3 + n^2 - 100$ $0.4 \cdot n^3$

5. $6 \cdot n^3 + n^2 - 100$ $40 \cdot n^2 - 4$

6. $n + \log n$ $10 \cdot n$

7. $n \cdot \log n$ n

ב. מהו סדר הגודל של פולינום כלשהו, לדוגמה: $4 \cdot n^5 + 30 \cdot n^4 - n^3 + 9 \cdot n + 8$?

שאלה 4

מיינו את הפונקציות בסדר יעילות יורד:

- | | | |
|----------------------------------|--------------------------------|-----------------------|
| א. $n^2 + 3$ | ב. $50 \cdot \log n - 300$ | ג. $0.5 \cdot n - 4$ |
| ד. $n^3 + 100 \cdot n^2 - 10$ | ה. $\log n / 1,000$ | ו. $n^3 + n^2 / 100$ |
| ז. $n^2 / 1,000 + 5,000 \cdot n$ | ח. $6 \cdot n^3 - 500 \cdot n$ | ט. $50 \cdot n + n^3$ |
| י. $1,000 \cdot \log n + 7$ | | |

שאלה 5

לפניכם כמה קטעי תוכניות. בהנחה שאורך הקלט הוא n , כתבו עבור כל קטע:

א. מהי פונקציית זמן הריצה של האלגוריתם?

ב. מהו סדר הגודל של יעילות האלגוריתם?

```
int x = 1;
for (int i=1; i<=10; i++)
    x = x * i;
System.out.print(x);
```

```
int x = 1;
for (int i=1; i<=n; i++)
    x = x * i;
System.out.print(x);
```

```
int x = 1;
for (int i=n; i>=1; i--)
{
    x = x + i;
    System.out.print(x);
}
System.out.print(x);
```

```
int x = 1;
for (int i=1; i<=n/2; i++)
    x = x * i;
System.out.print(x);
```

```
int x = 1;
for (int i=1; i<=n; i=i+2)
    x = x * i;
System.out.print(x);
```

```

int x = 1;
int m = n;
for (int i=1; i<=m; i++)
{
    x = x + i;
    m = m - 1;
}
System.out.print(x);
System.out.print(m);

```

```

int i = 1;
while (i <= n)
{
    System.out.print(i);
    i = i * 2;
}

```

```

int i = n;
while (i >= 1)
{
    if(i%2==0)
        System.out.print(i);
    else
    {
        System.out.print("***");
        System.out.print(i);
    }
    i = i / 2;
}

```

```

int i = 1;
int sum1;
int sum2 = 0;
while (i <= n)
{
    sum1 = 0;
    for (int k=1; k<=n; k++)
        sum1 = sum1 + k;
    System.out.println(sum1);
    sum2 = sum2 + sum1;
    i = i + 1;
}
System.out.println(sum2);

```

```

int x = 1;
for (int i=1; i<=n; i++)
{
    for (int j=1; j<=100; j++)
        x = x * j;
    System.out.print(x);
}

```

```

int x = 1;
for (int i=1; i<=n; i++)
{
    for (int j=1; j<=i; j++)
        x = x * j;
    System.out.print(x);
}

```

```

int x = 1;
for (int i=1; i<=n; i++)
{
    m = n;
    for (int j=1; j<=m; j++)
    {
        x = x * i;
        m = m / 2;
    }
    System.out.print(x);
}

```

שאלה 6

נניח שברשותנו שני אלגוריתמים המבצעים הגהות כתיב לטקסט בטורקית (טורקית נכתבת באותיות לטיניות). זמן ריצתו של האלגוריתם הראשון הוא $f_1(n) = 100,000n$, ואילו זמן ריצתו של השני הוא:

$$f_2(n) = n^2$$

- א. אם ברצוננו לשלב את האלגוריתם בתוכנית הקוראת מהמשתמש משפט ומודיעה האם יש בו שגיאות כתיב או לא, איזה אלגוריתם נעדיף?
- ב. אם נרצה לשלב את האלגוריתם במעבד תמלילים, האם תשתנה תשובתנו? (רמז: פרויקט הגמר של דני בחוג "טורקית למתקדמים" הוא תרגום ספרו הארוך של טולסטוי, "מלחמה ושלוש", מעברית לטורקית).
- ג. עבור איזה תחום של אורכי קלט נוכל לומר כי האלגוריתם השני יעיל פחות?

שאלה 7

המורה זרובבל אחסן את ציוני התלמידים בבחינה המסכמת בקצרנות במערך grades. למורת רוחו הוא גילה שהציונים נמוכים, ולכן החליט לנרמל את הציונים כך שהציון הגבוה בכיתה יהיה 100. כדי לבצע זאת הוא נעזר בפעולה הזו:

```

public static void normalizeGrades(int[] grades)
{
    int max = getMaxGrade(grades);

    for (int i=0; i<grades.length; i++)
        grades[i] = grades[i]*100/max;
}

```

א. מהו אורך הקלט של הפעולה?

ב. ממשו את הפעולה: `public static int getMaxGrade(int[] grades)`

- ג. מהם הצעדים הבסיסיים המתבצעים בפעולה `?normalizeGrades(...)` (רמז: הבחינו בין שני סוגים של צעדים).
- ד. חשבו את פונקציית זמן הריצה של הפעולה `.normalizeGrades(...)`.

שאלה 8

התבוננו בפעולה זו המקבלת מספרים חיוביים בלבד:

```
public static int mystery(int num)
{
    int x = 0;
    while (num > 0)
    {
        num = num / 10;
        x++;
    }
    return x;
}
```

- א. איזה ערך תחזיר הפעולה עבור המספרים: 1000, 1, 45, 527?
- ב. מהי טענת היציאה של הפעולה?
- ג. נגדיר את אורך הקלט להיות מספר הספרות במספר. מה יהיה סדר הגודל של יעילות הפעולה?
- ד. נגדיר את אורך הקלט להיות המספר עצמו (num). מה יהיה סדר הגודל של יעילות הפעולה?
- ה. האם יש סתירה בין סעיפים ג ל-ד? נמקו.

שאלה 9

- א. כתבו פעולה המקבלת מערך `arr` של מספרים ומספר `x`. הפעולה מחזירה כמה איברים במערך `arr` גדולים מ-`x` או שווים לו.
- ב. מהו סדר הגודל של יעילות הפעולה שכתבתם בסעיף א?
- ג. נוסף למשימה מסעיף א את ההנחות האלה:
- המערך `arr` ממוין בסדר עולה.
 - אין במערך מופעים כפולים של ערכים.
 - `x` מופיע במערך.
- השתמשו בפעולת עזר שהכרתם בפרק כדי לפתור את המשימה בצורה יעילה יותר. כתבו את הפתרון החדש ונתחו את יעילותו.

שאלה 10

התבוננו באלגוריתמים א ו-ב וענו על השאלות:

אלגוריתם א

(1) קרא תו *ch*.

(2) הדפס את *ch*.

אלגוריתם ב

(1) קרא את *n*.

(2) עבור *i* מ-1 ועד *n* בצע:

(2.1) עבור *j* מ-1 ועד *n* בצע:

(2.1.1) **אלגוריתם א**.

(2.1.2) הדפס ':

(3) הדפס " : סוף הקלט."

א. מהו הצעד הבסיסי באלגוריתם_ב? מה סדר הגודל של יעילות האלגוריתם?

ב. מה יהיה זמן הריצה של האלגוריתם אם שורה (2.1) תוחלף בשורה:

(2.1) עבור *j* מ-*i* ועד *n* בצע:

ג. התבוננו באלגוריתם_ג וענו על השאלות שאחריו:

אלגוריתם ג (n)

(1) קרא תו *ch*.

(2) בצע *n* פעמים:

(2.1) הדפס את *ch*.

(2.2) הדפס ' ,

כיצד תשתנה התשובה לסעיף א אם באלגוריתם_ב שורה (2.1.1) תוחלף בשורה:

(2.1.1) **אלגוריתם ג (n)**

ד. כיצד תשתנה התשובה לסעיף ב אם באלגוריתם_ב שורה (2.1.1) תוחלף בשורה:

(2.1.1) **אלגוריתם ג (j)**

שאלה 11

מיון-בחירה של מערך *arr* מתבצע כך: בשלב ה-*i* מוצאים את האיבר הקטן בקטע המערך *arr.length*, *i...arr*, ומחליפים אותו באיבר שבמקום ה-*i*.

א. כתבו פעולה למיון-בחירה.

ב. נתחו את יעילות הפעולה. מהי יעילותו במקרה הטוב ביותר?

שאלה 12

- התבוננו בפעולה bubbleSort(...) למיין בועות, שהוצגה בפרק.
- כיצד אפשר לדעת האם בתום אחד המעברים על המערך הוא כבר ממוין?
 - כיצד אפשר לשפר את הפעולה כך שלא תבצע מעברים מיותרים?
 - האם השיפור שהצעתם בסעיף הקודם יגרום לשיפור ביעילות הפעולה? אם כן, איזה סוג של שיפור? (חשבו מהו המקרה הגרוע ביותר).
 - האם אפשר לחשוב על שיפור דומה גם באלגוריתם מיון-מיזוג?

שאלה 13

מספר ראשוני הוא מספר המתחלק ללא שארית ב-1 ובעצמו בלבד. למשל, 2, 7, 13, 19, 23 הם מספרים ראשוניים. אם מספר כלשהו i מחלק מספר אחר n ללא שארית, נאמר כי n מתחלק ב- i , אחרת נאמר כי n אינו מתחלק ב- i .

הפעולה הבאה מקבלת מספר שלם ומחזירה 'אמת' אם המספר ראשוני, ו-'שקר' אחרת:

```
// הנחה: n>2
public static boolean isPrime(int n)
{
    for (int i = 2; i < n; i++)
        if (n%i==0)
            return (false);
    return (true);
}
```

- נתחו את יעילות הפעולה.
- אם n הוא זוגי ושונה מ-2, הרי אינו ראשוני. אם n אינו זוגי, אפשר לחסוך מחצית ממספר הפעמים שבהם מבוצעת הלולאה בשורה (1). כיצד? כתבו פעולה משופרת. איזה סוג של שיפור בזמן הריצה נשיג בצורה זו?
- אפשר לשפר את הפעולה גם בצורה הבאה: במקום לבדוק אם כל המספרים בין 2 ובין $n-1$ הם מחלקים של n , ניתן לבדוק את המספרים בין 2 ל- \sqrt{n} . מדוע מספיק לבדוק את המספרים עד \sqrt{n} ?
- רמז: אם קיים מספר i שגדול מ- \sqrt{n} המחלק את n , נקבל $n = k \cdot i$ עבור k שלם כלשהו. מה אפשר לומר על k ?
- האם הפונקציות n ו- \sqrt{n} הן מאותו סדר גודל? ציירו גרף של הפונקציות או בדקו באמצעות טבלה. מה תוכלו להסיק מכך על השיפור ביעילות הפעולה שהכנסנו בסעיף ג?

שאלה 14

כתבו פעולה המקבלת מערך של מספרים. הפעולה תדפיס את שני האיברים במערך שההפרש ביניהם הוא הגדול ביותר. הקפידו שסדר הגודל של הפעולה יהיה $O(n)$.

שאלה 15

א. כתבו פעולה המקבלת מערך שמאוחסנים בו ציונים (ערכים שלמים בין 0 ל-100). הפעולה תחזיר את הציון השכיח – הציון שהופיע הכי הרבה פעמים (הניחו שקיים אחד כזה). סדר הגודל של הפעולה יהיה $O(n)$.

ב. מה יקרה אם המספרים המאוחסנים במערך לא יהיו מוגבלים לטווח מסוים?

שאלה 16

הפעולה `mystery1(...)` מקבלת מערך שמאוחסנים בו ציונים (ערכים בין 0 ל-100). הפעולה נעזרת בפעולת-עזר בשם `mystery2(...)`. להלן הפעולות:

```
public static double mystery1(int[] arr)
{
    double res;
    mystery2(arr);

    if (arr.length%2 != 0)
        res = arr[arr.length/2];
    else
        res = (double) (arr[arr.length/2 - 1] +
                        arr[arr.length/2]) / 2;

    return res;
}

public static void mystery2(int[] arr)
{
    int temp;
    for (int i = 1; i <= arr.length-1; i++)
    {
        for (int j = 0; j < arr.length-i; j++)
        {
            if (arr[j]>arr[j+1])
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

- א. מה יעילות הפעולה `mystery2(...)`? נמקו. כתבו את טענת היציאה של הפעולה `mystery2(...)`.
- ב. מה יעילות הפעולה `mystery1(...)`? נמקו. כתבו את טענת היציאה של הפעולה `mystery1(...)`.
- ג. ממשו את הפעולה `mystery1(...)` מחדש כך שתשפר את יעילות הפעולה המקורית, בסדר גודל. אין חובה להיעזר בפעולה `mystery2(...)`.

שאלה 17

כתבו פעולה המקבלת שני מערכים של מספרים שלמים, a ו-b. הפעולה מחזירה מערך חדש, שהוא מכפלת שני המערכים a ו-b. הקפידו שסדר הגודל של הפעולה יהיה $O(n)$.
 הסבר: נקרא למערך החדש שיוחזר בשם c. המכפלה $c=a \cdot b$ תתבצע כך: האיבר הראשון במערך c שווה לסכום המכפלות של האיבר הראשון במערך a בכל אחד מאיברי המערך b, האיבר השני במערך c שווה לסכום המכפלות של האיבר השני במערך a בכל אחד מאיברי המערך b, וכך הלאה.

לדוגמה, עבור שני המערכים a ו-b האלה:

a	3	7	2
---	---	---	---

b	4	1	3	5
---	---	---	---	---

יוחזר המערך c הזה:

c	39	91	26
---	----	----	----

על פי החישוב:

$$c[0] = 3 \cdot 4 + 3 \cdot 1 + 3 \cdot 3 + 3 \cdot 5 = 39$$

$$c[1] = 7 \cdot 4 + 7 \cdot 1 + 7 \cdot 3 + 7 \cdot 5 = 91$$

$$c[2] = 2 \cdot 4 + 2 \cdot 1 + 2 \cdot 3 + 2 \cdot 5 = 26$$

שאלה 18

א. ממשו את האלגוריתם "מיון מיזוג", שנלמד בפרק, כפעולה חיזונית על מערכים.
 ב. הריצו את הפעולה שכתבתם על מערך בגודל 1,000 תאים, ועל מערך בגודל מיליון תאים, ומדדו את זמני הריצה בפועל. כדי למדוד את זמן ריצת הפעולה במחשב, במילי-שניות, עליכם להשתמש בפעולה `System.currentTimeMillis()` השייכת לשפת java. כדי למדוד את משך הריצה יש לבצע את הקוד הזה:

```
long startSortTime = System.currentTimeMillis();
mergeSort(arr); // מיון מיזוג
long finishSortTime = System.currentTimeMillis();
System.out.print(finishSortTime - startSortTime);
```

בסיום יודפס כמה מילי-שניות נמשכה הריצה של הפעולה למיון מיזוג.

ג. בצעו מדידת זמן ריצה דומה עבור הפעולה למיון מערך על פי אלגוריתם מיון בועות. השוו את הזמנים שקיבלתם לזמנים שנמשך אלגוריתם מיון מיזוג על מערכים בגדלים אלה. נסחו את מסקנותיכם.

שאלה 19

א. כתבו פעולה המקבלת שני מספרים שלמים n_1 ו- n_2 ומדפיסה את המחלק המשותף הגדול ביותר של שניהם.

ב. נתחו את יעילות הפעולה שכתבתם.

שאלה 20

שתי הפעולות שלפניכם מבצעות את המשימה: הדפסת כל מחלקיו של מספר (למעט 1 והמספר עצמו):

```
public static void printAllDividers1(int num)
{
    for (int i=2; i<num; i++)
        if (num % i == 0)
            System.out.println(i);
}
```

```
public static void printAllDividers2(int num)
{
    for (int i=2; i<=num/2; i++)
        if (num % i == 0)
            System.out.println(i);
}
```

א. נתחו את יעילות שתי הפעולות.

ב. איזו פעולה יעילה יותר? נמקו.