פרק 4
רקורסיה

כאשר אנו ניגשים לפתור בעיה, עומדים לרשותנו כמה כלים בסיסיים המקלים עלינו להגיע לפתרון. אחד הכלים לפתרון בעיה הוא שיטת "הַפְרֵד וּמְשׁוֹל", כלומר חלוקת הבעיה לתת-בעיות. כך פתרון כל אחת מהבעיות בנפרד מביא אותנו בסופו של דבר לפתרון הבעיה המקורית, על ידי הרכבת כל הפתרונות של התת-בעיות. גישת "הפרד ומשול" משמשת אותנו גם בחיי היומיום. לדוגמה, כאשר אנו מתכננים לצאת לטיול, אנו מפרקים את המשימה הגדולה "יציאה לטיול", לתת-משימות: קביעת התאריך והיעד, סוג הבגדים שיש לקחת, כמויות האוכל והשתייה הנחוצים וכדומה. עם פתרון כל המשימות הללו יש לקוות שנמצא את עצמנו בפתח הדלת, תרמיל על הגב ומקל ביד...

לפעמים אנו נתקלים בבעיות שניתן לפרקן לתת-בעיות שהן למעשה זהות לבעיה המקורית, אלא שהן "קטנות יותר". פתרון תת-בעיות אלה יאפשר לנו לפתור את הבעיה המקורית. תת-בעיה ניתן לפתור באופן ישיר, אם היא מספיק פשוטה. אם לא, ניתן לפתור אותה באותה שיטה, של פירוק לבעיות פשוטות יותר מאותו סוג. להלן דוגמה מחיי היומיום: כשעלינו לאסוף את ארבעת האחים הקטנים מהגן ומבית הספר בצהריים, ניתן לחשוב על הפתרון של 'איסוף האח הקטן' מהגן ו'איסוף שלושת האחרים' מבית הספר היסודי. איסוף האח הקטן הוא פעולה אחת. איסוף השלישייה מתפרט לכמה פעולות: 'איסוף האח הקטן בשלישייה', 'איסוף שני האחרים הגדולים' וכך הלאה. פתרון הבעיה באמצעות פירוקה לכמה בעיות זהות קטנות יותר נקרא **פתרון רקורסיבי**. בפרק זה נלמד כיצד לפתור בעיות בשיטה רקורסיבית.

לפני שנתעמק ברקורסיות מעולם התכנות, נבחן בעיה נוספת (מתוקה) מהחיים...

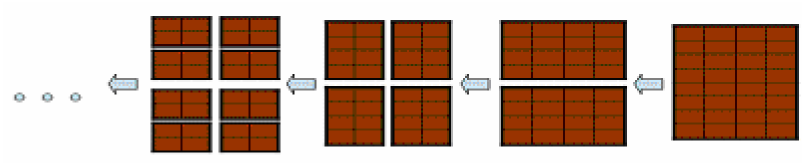
חפיסת השוקולד

מחנכת הכיתה החליטה להפתיע את 32 תלמידיה ולכבדם בטבלת שוקולד שבה 32 קוביות. מכיוון שיצא לך שם של תלמיד אחראי ביותר, היא מורה לך לקחת את הטבלה, ולחלקה כך שכל אחד מהתלמידים ייהנה מקוביית שוקולד אחת. זהו יום קיץ חם ויש סיבה לדאגה: אם תחלק את הקוביות אחת אחת, תגלה עד מהרה שהחפיסה נמסה ורק בני מזל מעטים ייהנו מהשוקולד, שכן רובו יישאר על אצבעותיך. כיצד תפתור את הבעיה?

לאחר כמה שניות של מחשבה מאומצת, אתה מציע את הרעיון הבא:

אחלק את טבלת השוקולד לשני חלקים שווים, ואתן כל חלק לתלמיד אחר. כל תלמיד שבידו מחצית הטבלה, ימשיך באותה השיטה, כלומר יחלק את החפיסה שקיבל לשתיים, וייתן כל מחצית לתלמיד אחר. כך יימשך התהליך עד שבידי כל תלמיד תהיה קובייה אחת בלבד. במקרה זה – יאכל התלמיד את הקובייה הבודדה שבידו.

קל לראות כי בדרך זו כל תלמיד מקבל קובייה אחת, ומה שיותר חשוב, החבילה מתחלקת במהירות. כל ילד בכיתה מקבל קוביית שוקולד מוצקה בתוך 5 שלבים בלבד, במקום שוקולד נמס בתוך 31 שלבים.



א. אז מהי (בדיוק) רקורסיה?

באופן כללי, רקורסיה היא הגדרה של מושג או של אלגוריתם (כלומר של פתרון לבעיה), המשתמשת במושג או באלגוריתם המוגדר, עצמו. בבואנו לתכנת אנו מעוניינים בעיקר בהגדרה רקורסיבית של אלגוריתמים. אלגוריתם רקורסיבי הוא אלגוריתם שמפעיל את עצמו פעם אחת או יותר על מופעים אחרים של בעיה שברצוננו לפתור, אם הבעיה אינה פשוטה דיה לפתרון ישיר. כדי שהגדרה רקורסיבית של אלגוריתם (פתרון) תוביל לפתרון, חשוב והכרחי שמופעי הבעיה הנפתרים בהפעלות הרקורסיביות יהיו "קטנים יותר" או פשוטים יותר מהמופע המקורי. תנאי זה מבטיח שתהליכי הפירוק מגיעים בסופו של דבר לבעיות פשוטות שניתן לפתור באופן ישיר.
לבעיה של חפיסת השוקולד הצענו פתרון רקורסיבי: פתרנו את הבעיה באמצעות חלוקתה לשתי בעיות דומות, שגודל כל אחת מהן הוא מחצית הבעיה המקורית. גם בעיות אלה נפתרו על ידי חלוקה לבעיות דומות שגודלן קטן בחצי (דהיינו שגודלן הוא רבע הבעיה המקורית), וכך הלאה, עד שהגענו למקרה "בסיסי", שאין עוד סיבה להמשיך ולפרק לבעיות קטנות יותר. בבעיית השוקולד המקרה הבסיסי ברור: כאשר נשארת ביד קובייה אחת בלבד, אין צורך להמשיך ולחלק. בשלב זה אוכלים את השוקולד.

בהמשך נראה שלמרבית הפתרונות הרקורסיביים שניתקל בהם יש מאפיינים דומים: ניתן להגדירם באמצעות **מקרה בסיסי** ו**מקרה מורכב**. **המקרה הבסיסי** "קל" להבנה, והוא ניתן לפתרון ישיר. את ה**מקרה המורכב** ניתן להגדיר במונחים פשוטים יותר, בעלי אפיון זהה לאפיון של הבעיה המקורית. כלומר, הבעיה המקורית ניתנת לחלוקה למקרים פשוטים יותר ממנה, שהם עדיין אינם בסיסיים אלא מורכבים. את המקרים המורכבים האלה אפשר לחלק למקרים פשוטים עוד יותר עד שמגיעים להגדרה של המקרה הבסיסי שפתרונו פשוט וידוע. חשיבות הפירוק למקרים פשוטים יותר של הבעיה ברורה: דבר זה מבטיח ששרשרת ההפעלות הרקורסיביות תסתיים במקרים בסיסיים הנפתרים ישירות, ולכן האלגוריתם הרקורסיבי יוצר תהליך רקורסיבי **סופי**. שרשרת ההפעלות נפסקת כאשר הרקורסיה מגיעה למקרה הבסיסי, ולכן המקרה הבסיסי נקרא גם **תנאי עצירה**.

ב. פתרון רקורסיבי של בעיה פשוטה

ב.1. הדפסת מספר במהופך

ברצוננו להדפיס את ספרותיו של מספר שלם וחיובי, במהופך (מהסוף להתחלה), מבלי להשתמש במערך או בצורת אחסון אחרת. לשם פתרון הבעיה נגדיר אלגוריתם רקורסיבי.
תחילה נזהה את המקרה הבסיסי: אם המספר הוא בין 1 ל-9, אזי הוא מיוצג על ידי ספרה אחת. במקרה זה נפתור את הבעיה על ידי הדפסת המספר.

אם המספר גדול מ-9, אזי הוא מיוצג על ידי שתי ספרות לפחות, ולפנינו מקרה מורכב. תהליך הפתרון ייראה כך: תחילה נדפיס את ספרת האחדות שלו (זו הספרה הימנית ביותר בייצוגו), אחריה נדפיס בסדר הפוך את המספר המיוצג על ידי הספרות של המספר הנתון בהשמטת ספרה זו. למשל, להדפסת הספרות של 1849 בסדר הפוך, נדפיס את 9 ואז נדפיס את הספרות של 184 בסדר הפוך. דבר זה נעשה כמובן על ידי הפעלה רקורסיבית של אותו תהליך. שימו לב שהפרמטר הנשלח לפעולה בהפעלה הרקורסיבית קטן מהפרמטר המקורי, כיון שהוא התקבל ממנו על ידי השמטת ספרת האחדות.

כיצד נבצע את הפירוק שתיארנו? נניח שהמספר הנתון הוא n, והוא גדול מ-9. את ספרת האחדות שלו ניתן לקבל על ידי ביצוע הפעולה n%10, המחזירה את השארית של חלוקת n ב-10. שארית זו היא כמובן מספר בן ספרה אחת בלבד. אל המספר המתקבל מ-n לאחר שהושמטה ספרת האחדות נוכל להגיע על ידי ביצוע הפעולה n/10 והכנסת התוצאה למשתנה מטיפוס שלם.

נסכם את מאפייני הרקורסיה בבעיה הנתונה (אנו מסמנים את המספר הטבעי הנתון ב-n, n>0):

מקרה בסיסי (=תנאי עצירה): n<10. הפתרון הוא הדפסת המספר.

מקרה מורכב: אם n>9, הייצוג של n הוא בן יותר מספרה אחת. הפתרון יבודד את ספרת האחדות וידפיס אותה ישירות, ואחר כך יזמן את הפעולה באופן רקורסיבי על המספר הנותר המיוצג על ידי יתר הספרות, לאחר שקופדה ספרת האחדות שלו.

ננסח את האלגוריתם:

הדפס-במהופך *(n)*

אם n<10, הדפס את n

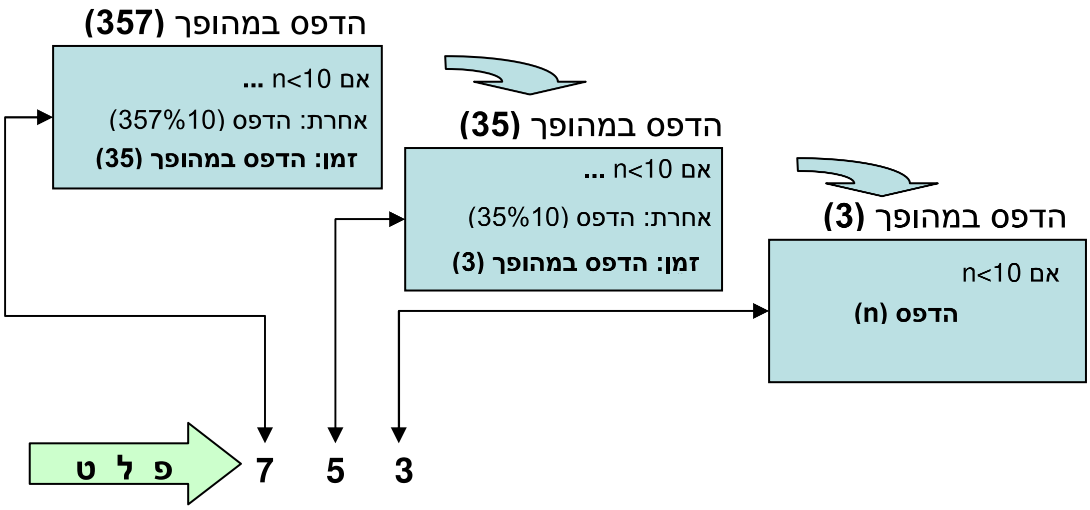
אחרת: הדפס את n%10 (השארית של חלוקת n ב-10)

זמֵן את ***הדפס-במהופך*** *(n/10)*

להלן פעולה המממשת את האלגוריתם:

```
public static void reverse(int n)
{
      if (n<10)
            System.out.println(n);
      else
      {
            System.out.print(n%10);
            reverse (n/10);
      }
}
```

נעקוב אחרי ביצוע האלגוריתם בעזרת האיור והטבלה. הפרמטר הוא המספר 357 :



מספר ההפעלה	הפרמטר	בדיקת תנאי העצירה	הפעולה המתבצעת	הערך לזימון הרקורסיבי
הפעלה 1	n = 357	! (n < 10)	הדפס (357%10) ← 7	(357/10) ← 35
הפעלה 2	n = 35	! (n < 10)	הדפס (35%10) ← 5	(35/10) ← 3
הפעלה 3	n = 3	n < 10	הדפס 3	סוף הרקורסיה

שימו לב כי כאשר זימנו את הפעולה הרקורסיבית בפעם הראשונה, הפרמטר לא התאים למקרה הבסיסי (תנאי העצירה), אולם, בסופו של דבר תזומן הפעולה הרקורסיבית עבור הפרמטר שמתאים למקרה זה והתהליך ייעצר. אם לא נגדיר מקרה בסיסי, או אם ההתכנסות לקראת תנאי העצירה לא תיכתב כראוי, ייווצרו זימונים חדשים בזה אחר זה. בגלל ריבוי הקריאות בריצת הפעולה, תזהה מערכת ג'אווה את הבעיה אחרי פרק זמן מסוים, תשלח הודעת שגיאה ותפסיק את ביצוע הפעולה.

? נסו לכתוב את הפעולה reverse(…) ללא תנאי העצירה וראו מה יקרה.

המשימה של הדפסת המספר במהופך הסתיימה כאשר זימנו את הפעולה הרקורסיבית עם מספר חד-ספרתי (המקרה הבסיסי).

סביר לממש את הפעולה reverse כפעולה סטטית הפועלת על מספר שלם. ניתן לכתוב את הפעולה הרקורסיבית הסטטית במחלקת שירות או בקובץ שבו מופיעה הפעולה הראשית ולזמן אותה מתוך הפעולה הראשית (או מכל פעולה אחרת), כמו בדוגמה:

```
public class TestReverse
{
      public static void main(String[] args)
      {
            reverse(4521);
      }
      public static void reverse(int n)
      {
            …
      }
}
```

אולם, באופן כללי, פעולה רקורסיבית אינה חייבת להיות סטטית. סיווג פעולה כרקורסיבית מציין את האופן שבו אנו משתמשים בפעולה. בג׳אווה אין הבדל בין פעולה רקורסיבית לפעולה לא רקורסיבית. לכל הפעולות, רקורסיביות או לא, אותן צורות זימון. כל מה שכבר למדנו ביחידה על פעולות, או שנלמד בהמשך, תקף בפרט גם לפעולות רקורסיביות.

ישנם פתרונות רקורסיביים מורכבים יותר, ובהם גם לאחר זימון הפעולה הרקורסיבית האחרונה יש המשך של תהליך הפתרון. בדוגמה הבאה נראה בעיה שכזו.

ב.2. בעיית העצרת

העצרת (factorial) של מספר טבעי כלשהו (0<n), מסומנת כך: n! והיא מוגדרת כמכפלת כל המספרים בין 1 ל-n.

לדוגמה:

$1! = 1$

$2! = 1 * 2 = 2$

$3! = 1 * 2 * 3 = 6$

$4! = 1 * 2 * 3 * 4 = 24$

$5! = 1 * 2 * 3 * 4 * 5 = 120$

...

$n! = 1 * 2 * 3 * \ldots * (n-1) * n$ (עבור n כלשהו)

כמו כן, מקובל להגדיר את 0! כ-1.

ניתן לחשב את העצרת בחישוב איטרטיבי, אך בפרק זה אנו מעוניינים להציג חישוב רקורסיבי. כדי לכתוב אלגוריתם רקורסיבי לפתרון בעיית חישוב העצרת של מספר נתון, נציג הגדרה נוספת של המושג עצרת, והפעם הגדרה רקורסיבית. מהגדרה זו נקבל בקלות חישוב רקורסיבי.

נשווה את העצרת של שני מספרים עוקבים, למשל 4 ו-5:

על פי ההגדרה: $1 * 2 * 3 * 4 * 5 = 5!$; $1 * 2 * 3 * 4 = 4!$

מכאן ברור ש: $4! * 5 = 5!$. שימו לב, שהמסקנה נכונה באופן כללי, שהרי גם $5! * 6 = 6!$ וכן הלאה, ובמקרה המורכב: $(n-1)! * n = n!$.

אם נוסיף להבנה זו את העובדה ש: $0! = 1$ (המקרה הבסיסי), נקבל **הגדרה רקורסיבית** של העצרת של מספר טבעי:

> העצרת (factorial) של מספר טבעי n מסומנת כ- $n!$
>
> ומוגדרת כך:
>
> - אם $n = 0$ (המקרה הבסיסי), ערך העצרת הוא 1
> - אם $n > 0$ (המקרה המורכב), ערך העצרת הוא המכפלה: $n * (n-1)!$

אלגוריתם רקורסיבי לבעיית העצרת

מצוידים בהבנת המבנה הרקורסיבי של העצרת, נוכל לכתוב בקלות אלגוריתם לחישובה:

חשב-עצרת (n)

אם $n = 0$, החזר 1

אחרת, החזר את המכפלה של n בתוצאה המוחזרת מזימון ***חשב-עצרת (n-1)***

ובקוד, בהנחה ש: $n \geq 0$:

```
public static int factorial(int n)
{
      if (n == 0)
            return 1;
      else
            return (n * factorial(n - 1));
}
```

שימו לב שהקריאה הרקורסיבית נעשית על ידי זימון עצמי של הפעולה, כלומר הפעולה מזמנת את עצמה, אך עם פרמטר שונה. על מנת להבין כיצד פועלת הרקורסיה, נעקוב אחרי מהלך ביצוע הפעולה בעזרת האיור הבא, עבור ערך הפרמטר 4. החִצים באיור מסמנים את שני שלבי הביצוע של אלגוריתם זה. הראשון הוא שלב "הלוך", שבו מתבצעים זימונים רקורסיביים בזה אחר זה, והוא מסומן בחץ . השלב השני הוא שלב "חזור", שבו מוחזרים ערכים לחישובים שממתינים והוא מסומן בחץ הפוך .

factorial (4): 4 * factorial (3)
factorial (3): 3 * factorial (2)
factorial (2): 2 * factorial (1)
factorial (1): 1 * factorial (0)
factorial (0): **return** 1
factorial (1): 1 * 1, **return** 1
factorial (2): 2 * 1, **return** 2
factorial (3): 3 * 2, **return** 6
factorial (4): 4 * 6, **return** 24

סוף הרקורסיה: הערך המוחזר הוא 24

בשורה הראשונה, הערך של n הוא 4. כיוון שערך זה אינו 0, יש לבצע כפל של n ב- factorial (n-1). כדי לקבל את הערך של factorial (n-1) הפעולה מחשבת את ערך n-1, מקבלת 3, ומבצעת זימון רקורסיבי של הפעולה factorial עם ערך הפרמטר 3. החישוב מופסק והפעולה ממתינה לתוצאות זימון זה.

בשורה השנייה מתוארת ההפעלה עבור ערך הפרמטר 3, בביצוע דומה: כדי לחשב את factorial (3), יש לזמן באופן רקורסיבי את factorial (2), וכך הלאה עד לשורה החמישית, שבה ערך הפרמטר בזימון המתואר בשורה הוא 0.

החישוב של factorial (0) הוא חישוב של המקרה הבסיסי (שכן הפרמטר הוא 0), ועל כן התוצאה שלו היא 1. תוצאה זו מוחזרת לזימון הקודם, של factorial (1), שהיה בהמתנה. משלב זה זימון זה ממשיך, מכפיל 1 ב-1, ומחזיר 1 לזימון factorial (2). זה מחדש את פעולתו, מכפיל 1 ב-2, ומחזיר 2. כך מוחזרים ערכים לזימונים הממתינים, בזה אחרי זה. כל זימון מכפיל את ערך הפרמטר שלו בתוצאה שהוחזרה אליו, ומחזיר את התוצאה לזימון שהפעיל אותו. כאשר הזימון הראשון מחזיר את התוצאה שלו, שהיא 24, החישוב מסתיים.

שימו לב להבדל בין התהליכים הרקורסיביים בשתי הבעיות שהצגנו: הפיכת ספרות של מספר ובעיית העצרת. בהפיכת ספרות של מספר ביצענו רק את תהליך ה-"הלוך" – הדפסנו את ספרת האחדות ואז זימנו את הפעולה הרקורסיבית. כאשר הסתיימה הקריאה האחרונה, סיימנו את התהליך הרקורסיבי. לעומת זאת, בחישוב העצרת היינו צריכים להחזיר את הערך של זימון הפעולה ולבצע הכפלה נוספת ב-n. לכן, כאשר פעולה מזמנת את עצמה עם ערך פרמטר קטן יותר, היא נשארת בהמתנה לערך שיוחזר מהזימון הרקורסיבי. כאשר מסתיים הזימון, הוא מחזיר ערך, והפעולה הממתינה "מתעוררת לחיים" וממשיכה בחישוב, עד שהיא מסיימת ומחזירה ערך. בדוגמה שראינו כל הזימונים התבצעו בזה אחר זה, ואחריהם התבצעו החזרות הערכים. בתהליך רקורסיבי זה החלוקה לשני שלבים ברורה, "הלוך" ואחריו "חזור".

ג. אלגוריתם רקורסיבי לחישוב מספרי פִיבּוֹנָצִ'י

סדרת פיבונצ'י היא סדרת מספרים הקרויה על שמו של לאונרדו פִיבּוֹנָצִ'י, המתמטיקאי המערבי הראשון שגילה אותה לפני כ-800 שנה. הסדרה מוגדרת בדרך הבאה: שני האיברים הראשונים בסדרה הם 0 ו-1, כל איבר נוסף בסדרה הוא סכומם של שני האיברים הקודמים לו בסדרה.

12 האיברים הראשונים של סדרת פיבונצ'י הם:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 . . .

אם נסמן את הערך של המספר במקום ה-k בסדרת פיבונצ'י כ: **מספר פיבונצ'י (k)**, נקבל הגדרה רקורסיבית של ערך כל מספר בסדרה:

מספר פיבונצ'י (k):

אם $k = 1$, הערך הוא 0

אם $k = 2$, הערך הוא 1

אחרת, הערך הוא מספר פיבונצ'י (k - 1) + מספר פיבונצ'י (k - 2)

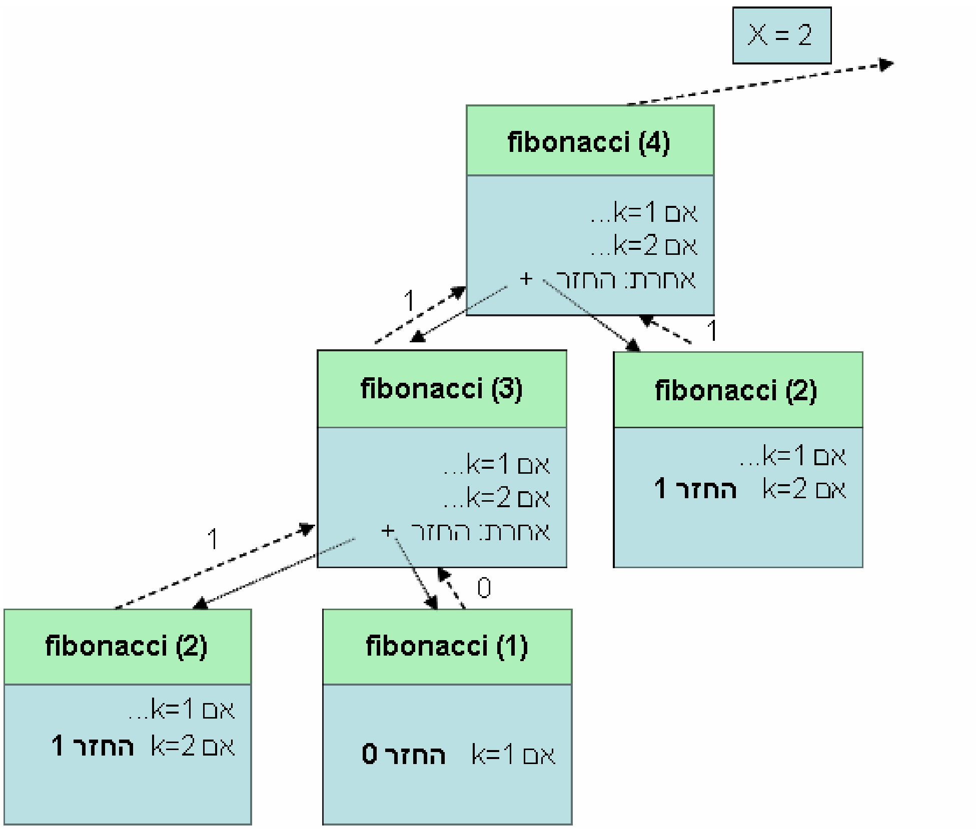
זוהי הגדרה רקורסיבית כיוון שהיא מגדירה כל איבר, פרט לשני הראשונים, באמצעות שני האיברים הקודמים. מהגדרה זו קל לקבל אלגוריתם רקורסיבי לחישוב מספרי פיבונצ'י. כדי לעשות זאת, המושג 'הערך הוא' יתורגם באלגוריתם להנחיה 'החזר'.

ובהנחה ש: $k \geq 1$, הקוד ייראה כך:

```java
public static int fibonacci(int k)
{
      if (k == 1)
            return 0;
      if (k == 2)
            return 1;
      return (fibonacci(k-1) + fibonacci(k-2));
}
```

נעקוב אחרי הביצוע של **int** x = fibonacci (4), בעזרת האיור הבא:

כל זימון של הרקורסיה מצוין בחץ שחור מלא . פעולת החיבור, הפעולה החשבונית שיש צורך לבצע בין ערכי הזימונים השונים (כשיחושבו ויוחזרו), מסומנת בשורה המתחילה ב"אחרת:". ערכי החזרה מסומנים על גבי החִצים המקווקווים הפונים חזרה אל מי שזימן את הרקורסיה. עם החזרת הערכים מתבצעת פעולת החיבור, והערך המתקבל מוחזר שוב "כלפי מעלה", למי שזימן את החישוב. כיוון שהקריאה הרקורסיבית מתבצעת פעמיים, נבצע את המעקב בעזרת דגם "עץ" שיאפשר לנו להסתעף לכיוונים שונים על פי התפתחות הרקורסיה:



כפי שנראה מהאיור, חלק מהחישובים הנעשים הם מיותרים. שוב ושוב נדרשת הרקורסיה לתת מענה על ערכם של fibonacci (1) ושל fibonacci (2), אף על פי שאלו חושבו כבר. ככל שהמספר הראשון שישלח לפעולה יהיה גדול יותר, כך יגדל מספרם של החישובים המיותרים החוזרים על עצמם. נראה שפתרון רקורסיבי לסדרת פיבונצ׳י אינו פתרון אידיאלי, ואינו חוסך במשאבים השונים.

? כאשר מזמנים את fibonacci עבור הערך 5, כמה פעמים מחושב fibonacci (1) וכמה פעמים מחושב fibonacci (2)?

ד. סוגי רקורסיה

ד.1. רקורסיית ״זנב״ מול רקורסיה ״הלוך-חזור״

ראינו שלוש דוגמאות של רקורסיה, כל אחת שונה במבנה או בהתנהגות מהאחרות. בדוגמה הראשונה (היפוך הספרות של המספר), יש באלגוריתם קריאה רקורסיבית יחידה. הרקורסיה מבצעת תהליך ״**הלוך**״, כלומר סדרה אחת של קריאות רקורסיביות. התהליך מסתיים אחרי תום ביצוע הקריאה הרקורסיבית האחרונה. רקורסיה אשר מבצעת רק תהליך של ״הלוך״ נקראת ״**רקורסיית זנב**״.

גם חישוב העצרת מתבצע על ידי אלגוריתם שבו קריאה רקורסיבית יחידה, אולם התהליך הרקורסיבי הזה הוא תהליך ״**הלוך-חזור**״. אחרי שמתבצעת הקריאה הרקורסיבית האחרונה,

הערך שחושב מוחזר לשלב הקודם של הרקורסיה, והוא בתורו מוחזר להמשך החישוב, וכך הלאה.

ניתן לזהות רקורסיית "הלוך-חזור" אם נענה בחיוב על השאלה: "האם אחרי הקריאה הרקורסיבית יש לבצע משהו נוסף?". בדוגמה שראינו, בחישוב העצרת, יש להכפיל את התוצאה ב-n, זוהי רקורסיית "הלוך-חזור", לעומת זאת בהיפוך הספרות של המספר אין צורך לבצע משהו נוסף ולכן זו אינה רקורסיית "הלוך-חזור".

ד.2. רקורסיה כפולה

בניגוד לשתי הדוגמאות הקודמות, בדוגמה של מספרי פיבונצ'י יש בגוף האלגוריתם שתי קריאות רקורסיביות: לחישוב מספר פיבונצ'י במקום k-1 ולחישוב מספר פיבונצ'י במקום k-2. רקורסיה כזו נקראת "**רקורסיה כפולה**". גם בדוגמה זו, אחרי שמוחזרים הערכים של הקריאות הרקורסיביות, קיימת פעולה נוספת שיש לבצע: חיבור הערכים. אפשר לומר שזו היא רקורסיית "הלוך-חזור", אך פריסתה דומה יותר לפריסה של עץ מאשר לפריסה של תהליך "הלוך" רציף אחד, שבסופו נפרס תהליך "חזור" רציף. במקרה זה זימונים והחזרות ערכים שלובים אלה באלה בתהליך אחד.

כמובן שאלה אינם כל הסוגים האפשריים של רקורסיה, אך הם מדגימים שני קריטריונים חשובים בניתוח אלגוריתם רקורסיבי: א) מספר הקריאות הרקורסיביות בגוף האלגוריתם; ב) האם אחרי קריאה רקורסיבית צריך לבצע עוד חישוב בפעולה המזמנת. חשוב להדגיש שלא בכל מקרה שקיימת רק קריאה רקורסיבית אחת באלגוריתם, מדובר בהכרח ברקורסיית זנב או רקורסיית הלוך-חזור. אם הקריאה היחידה מופיעה בתוך לולאה, ייתכן שתתבצע פעמים רבות.

להלן דוגמה:

```
public static int mysterySum(int n)
{
      if (n == 1)
            return 1;
      int sum = n;
      for (int i = 1; i < n; i++)
            sum = sum + mysterySum(i);
      return sum;
}
```

? מה מחזירה הפעולה mysterySum (…) עבור n = 3?

ד.3. רקורסיה הדדית

במצבים שונים משתתפות ברקורסיה שתי פעולות שונות. אם הפעולה f() מזמנת את פעולה g(), וזו מזמנת את f(), אזי הפעולות f() ו-g() הן פעולות רקורסיביות, גם אם אף פעולה לא מזמנת את עצמה ישירות. רקורסיה שכזו נקראת "**רקורסיה הדדית**". אם נסתכל על שתי הפעולות נראה שיחד הן מבצעות תהליך רקורסיבי, המפוצל לשתי פעולות.

לדוגמה, נראה את בעיית העצרת, אלא שהפעם היא נכתבת כרקורסיה הדדית של שתי הפעולות: factorial (…) ו-(…) g:

```
public static int factorial(int n)
{
      if (n == 0)
            return 1;
      else
            return g(n);
}

public static int g(int n)
{
      return n * factorial(n-1);
}
```

ה. רקורסיה על טיפוסי נתונים אחרים

כל הדוגמאות הקודמות הציגו פתרונות רקורסיביים לבעיות המטפלות במספרים. נבחן עתה פתרונות רקורסיביים לבעיות שהפרמטר שלהן הוא מסוג מחרוזת או מערך. לא נציג כאן סוגים חדשים של רקורסיה. העניין בדוגמאות אלה הוא בהגדרת המקרים הפשוטים והמורכבים, והמעבר ממקרה מורכב לפשוט יותר, עבור טיפוסי נתונים השונים מ-**int**.

ה.1. רקורסיה על מחרוזות

כדי לעבור ממחרוזת למחרוזת קטנה ממנה, ניתן להוריד מהמחרוזת ההתחלתית תו אחד או יותר, בעזרת הפעולות הקיימות במחלקה String (ראו ב-API).

דוגמה: האם מחרוזת היא פלינדרום

פלינדרום הוא מחרוזת שקריאתה משמאל לימין ומימין לשמאל מחזירה מחרוזת זהה. למשל, המחרוזות "אבא" ו-"כצפרשתשרפצכ" הן פלינדרומים.

בעיה: כתבו פעולה המקבלת מחרוזת ומחזירה 'אמת' אם המחרוזת הנתונה היא פלינדרום, ו'שקר' אחרת.

פתרון: ניתן לפתור בעיה זו גם על ידי פעולה שאינה רקורסיבית, למשל על ידי לולאה שתשווה כל זוג תווים הנמצאים באותו מרחק מתחילת המילה ומסופה, אך לצורך התרגול של פרק זה נעצב פתרון רקורסיבי.

המקרה הבסיסי: אם המחרוזת מכילה תו אחד או אינה מכילה תווים כלל, אזי היא פלינדרום.

המקרה המורכב: אם המחרוזת מכילה יותר מתו אחד, אזי: אם התו האחרון זהה לתו הראשון ואם המחרוזת ללא תווים אלה (הראשון והאחרון), מהווה פלינדרום, אזי המחרוזת כולה היא פלינדרום.

קל לראות כי בקריאה הרקורסיבית הפרמטר קטן יותר מזה שבקריאה המקורית, שכן שני תווים הורדו משני צדי המחרוזת.

תזכורת: הפעולות הבאות לעבודה עם מחרוזות לקוחות מתוך ממשק המחלקה String:

String substring (**int** beginIndex, **int** endIndex)	הפעולה מחזירה תת-מחרוזת המתחילה ב-beginIndex ומסתיימת ב-endIndex, אך אינה כוללת את endIndex עצמו
int length()	הפעולה מחזירה את אורך המחרוזת
char charAt (**int** index)	הפעולה מחזירה את התו שבמקום index

```
public static boolean palindrome(String str)
{
      if (str.length() <= 1)
            return true;
      if (str.charAt (0) != str.charAt(str.length()-1))
            return false;
      return palindrome(str.substring(1, str.length()-1));
}
```

שימו לב, תנאי העצירה בדוגמה זו כולל גם את המקרה של מחרוזת ריקה (0 תווים) וגם את המקרה של מחרוזת בת תו אחד.

לגבי המקרה המורכב, אם במחרוזת מספר אי-זוגי של תווים, ואנו מקטינים את הבעיה על ידי הורדת התו האחרון והתו הראשון, תישאר בסוף התהליך מחרוזת בת תו אחד. אם במחרוזת מספר זוגי של תווים, אזי בסוף התהליך תישאר מחרוזת ריקה שבה 0 תווים. בשני המקרים, אם הגענו לשלב הזה, סימן שהמחרוזת היא פלינדרום.

מעקב: נעקוב אחר התוכנית, עבור המחרוזת "ABCDA", בעזרת החִצים המתארים את תהליך הכניסה והיציאה מהרקורסיה ("הלוך-חזור"). שימו לב שהגדרת הרקורסיה כרקורסיית "הלוך-חזור", נובעת הפעם מכך שעם סיום הזימון האחרון יש עדיין ערכים להחזיר, עד לקבלת התשובה הסופית. אמנם זהו הדבר הנוסף היחיד שיתבצע, אך די בכך כדי להגדיר שיש כאן תהליך של "חזור".

palindrome ("ABCDA") => palindrome ("BCD")

palindrome ("BCD") => **return** false

palindrome ("ABCDA") => **return** false

הרקורסיה נגמרה ומחזירה ערך 'שקר'

ה.2. רקורסיה על מערכים

פעולה רקורסיבית על מערך פירושה בדרך כלל שבזימון הרקורסיבי, כדי להקטין את הפרמטר, מצמצמים את אורך המערך שפועלים עליו. לצורך כך, בדרך כלל, איננו מייצרים מערך חדש קטן יותר (חיסכון במקום), אלא מצמצמים את טווח הפעילות על המערך באמצעות שינוי ערכי האינדקסים. כך מצמצמים את השטח שעובדים עליו ומתכנסים לקראת המקרה הבסיסי (תנאי העצירה).

דוגמה: מציאת ערך מקסימלי במערך

בעיה זו ניתנת לפתרון ללא רקורסיה, ויש לשער שכבר פתרתם בעיות שכאלה בעזרת סריקה של מערך. בפרק זה ברצוננו להציג שני פתרונות רקורסיביים לבעיה, מתוך מספר פתרונות רקורסיביים אפשריים.

פתרון 1: נמצא את הערך המקסימלי בין הערכים במערך, פרט לערך הראשון בו.
נמצא את המקסימום שבין הערך הראשון במערך, לערך המקסימלי שמצאנו.

דוגמה: נבחן את המערך הבא שבו מאוחסנים הערכים: 4, 6, 9, 7, 1, 5, 8
הערך הראשון הוא 4 והוא מאוחסן בתא שהאינדקס שלו הוא 0. המקסימום בין יתר הערכים (החל מהמקום השני) הוא 9. המקסימום בין 4 ל-9 הוא 9, לכן זהו המקסימום בכל המערך.

כיצד נמצא את המקסימום בקרב יתר הערכים?

ניתן למצוא את המקסימום בעזרת לולאה, אך כדי להציג פתרון רקורסיבי לבעיה נזמן באופן רקורסיבי את הפעולה כך שתמצא את המקסימום במערך, החל מהמקום השני בו, וכך הלאה.

מהו תנאי העצירה? כאשר קטע המערך שבידינו מכיל ערך אחד בלבד, ברור שערך זה הוא המקסימום בקטע. החזרת ערך זה תסיים את תהליך הזימון הרקורסיבי.

לפניכם קוד הפתרון. יש לזמן את הפעולה לראשונה עם הערך begin = 0, כדי להתחיל את חיפוש המקסימום במערך החל מהאיבר הראשון בו:

```
public static int findMax(int[] arr, int begin)
{
      if (begin == arr.length-1)  // אורך הקטע הנוכחי במערך הוא 1
            return arr[begin];
      else
      {
            int temp = findMax(arr, begin+1);
            if (arr[begin] > temp)
                  return arr[begin];
            return temp;
      }
}
```

זהו אלגוריתם עם זימון רקורסיבי יחיד. שימו לב כי אחרי החזרה מהזימון הרקורסיבי, עלינו לבצע פעולה נוספת: השוואת הערך המוחזר מהקריאה והשמור ב-temp, לערך arr[begin]. כיוון שכך, זוהי רקורסיית "הלוך-חזור".

מעקב: נעקוב אחר מציאת המקסימום במערך arr = {4 ,6 ,9 ,7 ,1 ,5, 8}

נחזור ונדגיש: בכל זימון רקורסיבי נשמרים גבולות המערך המקורי. מיקוד ההסתכלות על הערכים השמורים במערך בכל שלב משתנה, ובאיור הוא מודגש בקו תחתי. רעיון ההמתנה של התהליכים עד לשובם של הערכים המחושבים (הצורך בהמשך החישובים), מודגש באיור על ידי כתיבת ... לאחר הזימון עצמו:

findMax ({**4, 6, 9, 7, 1, 5, 8**}, 0): temp = findMax ({4, **6, 9, 7, 1, 5, 8**}, 1); …

findMax ({4, **6, 9, 7, 1, 5, 8**}, 1): temp = findMax ({4, 6, **9, 7, 1, 5, 8**}, 2); …

findMax ({4, 6, **9, 7, 1, 5, 8**}, 2): temp = findMax ({4, 6, 9, **7, 1, 5, 8**}, 3); …

findMax ({4, 6, 9, **7, 1, 5, 8**}, 3): temp = findMax ({4, 6, 9, 7, **1, 5, 8**}, 4); …

findMax ({4, 6, 9, 7, **1, 5, 8**}, 4): temp = findMax ({4, 6, 9, 7, 1, **5, 8**}, 5); …

findMax ({4, 6, 9, 7, 1, **5, 8**}, 5): temp = findMax ({4, 6, 9, 7, 1, 5, **8**}, 6); …

findMax ({4, 6, 9, 7, 1, 5, **8**}, 6) => **return** arr[6] = 8

findMax ({4, 6, 9, 7, 1, **5, 8**}, 5) => temp = 8; arr[5] = 5, 5!>8, **return** 8

findMax ({4, 6, 9, 7, **1, 5, 8**}, 4): temp = 8; arr[4] = 1, 1!>8, **return** 8

findMax ({4, 6, 9, **7, 1, 5, 8**}, 3): temp = 8; arr[3] = 7, 7!>8, **return** 8

findMax ({4, 6, **9, 7, 1, 5, 8**}, 2): temp = 8, arr[2] = 9, $\boxed{9>8}$, **return** 9

findMax ({4, **6, 9, 7, 1, 5, 8**}, 1): temp = 9, arr[1] = 6, 6!> 9, **return** 9

findMax ({**4, 6, 9, 7, 1, 5, 8**}, 0): temp = 9, arr[0] = 4, 4!>9**, return** 9

סוף הרקורסיה, הערך המוחזר הוא 9

פתרון 2: נחלק את המערך לשני חלקים שווים (אם מספר האיברים זוגי), או כמעט שווים (אם מספר האיברים אינו זוגי). נחפש באופן רקורסיבי את המקסימום בכל אחד מהחלקים. אחרי שנמצא, נשווה בין שני הערכים המקסימליים ונחזיר את המספר הגדול מביניהם.

כפי שאמרנו קודם, חלוקת המערך אינה נעשית על ידי יצירת מערכים קטנים יותר, אלא בעזרת מיקוד ההסתכלות על תת-מערכים מתוך המערך המקורי, בעזרת האינדקסים. לכן בנוסף למערך, הפעולה תקבל שני פרמטרים: אינדקס נקודת ההתחלה של תת-המערך הנוכחי, ואינדקס נקודת הסוף שלו.

מהו תנאי העצירה? כאשר הגענו לתת-מערך בגודל 1 (אינדקס ההתחלה שווה לאינדקס הסוף), נחזיר את הערך השמור במערך זה, כמקסימום.

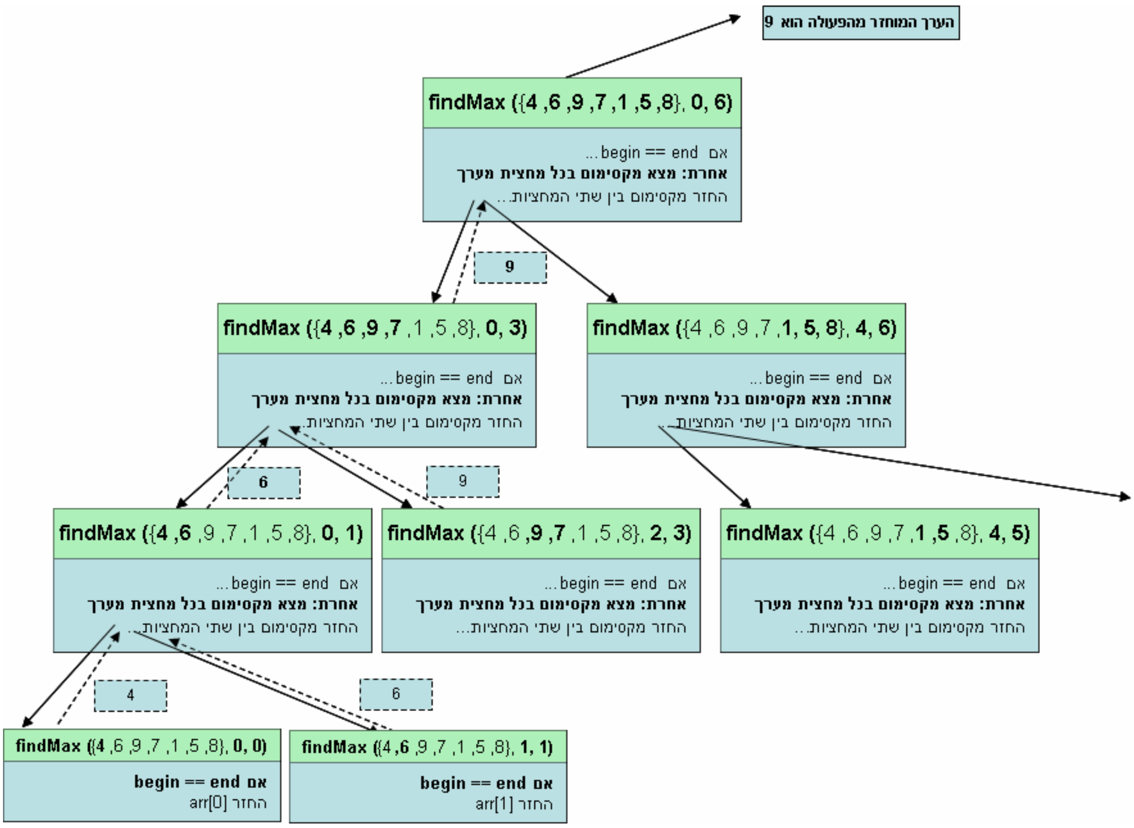
להלן קוד הפתרון. המשתמש צריך לזמן את הפעולה עם ערכים begin = 0, end = arr.length-1:

```
public static int findMax(int[] arr, int begin, int end)
{
      if (begin == end)
            return arr[begin];
      else
      {
            int middle = (begin + end) / 2;
            int max1 = findMax(arr, begin, middle);
            int max2 = findMax(arr, middle+1, end);
            if (max1 > max2)
                  return max1;
            return max2;
      }
}
```

כיוון שהמשתנה middle הוא מטיפוס **int**, החלוקה ב-2 תמיד תיתן לנו מספר שלם. עבור כל קטע המכיל לפחות שני איברים, תהליך החלוקה של הקטע ייצור שני קטעים קצרים ממנו, לכן אנו בטוחים שבסופו של דבר תמיד נגיע לתנאי העצירה (begin == end).

שימו לב שבאלגוריתם יש שני זימונים רקורסיביים. התהליך יהיה דומה לזה שראינו בחישוב מספרי פיבונצ׳י (תהליך ״הלוך-חזור״ לא רציף), ואנו נפרוס אותו במבנה של עץ מעקב.

דוגמה: נעקוב חלקית אחרי הפעולה findMax(...) לגבי אותו מערך: arr = {8, 5, 1, 7, 9, 6, 4}.
כיוון שאורך המערך הוא 7, והאינדקס של האיבר האחרון הוא 6, יהיה הזימון הראשון עם הערכים begin = 0, end = 6:



? השלימו את הציור ובדקו שאכן הערך המוחזר מהרקורסיה הוא הערך 9.

ו. דוגמה: האלגוריתם של אוֹקְלִידֵס

אחד האלגוריתמים המפורסמים במתמטיקה הומצא על ידי אוקלידס, מתמטיקאי יווני מפורסם שחי במאה השלישית לפני הספירה. האלגוריתם מוצא את **המחלק המשותף הגדול ביותר** של שני מספרים שלמים חיוביים (שימו לב שתמיד יש כזה, גם אם הוא רק המספר 1). בתיאור להלן אנו מניחים שהמספרים שבהם אנו עוסקים הם שלמים וחיוביים. המחלק המשותף המקסימלי של שני מספרים, הוא המספר הגדול ביותר המחלק את שניהם. לדוגמה: המחלק המשותף המקסימלי עבור שני המספרים 42 ו-28, הוא 14, כיוון שזהו המספר הגדול ביותר המחלק את שני המספרים.

האלגוריתם מתבסס על העובדה, כי אם מספר מחלק שני מספרים, אזי הוא מחלק גם את השארית של חלוקת הגדול שביניהם במשנהו. למשל, 3 מחלק את 24 וגם את 18, וכן את 6 שהוא השארית של חלוקת 24 ב-18. עובדה זו נכונה בפרט למחלק המשותף המקסימלי. מכאן שכדי למצוא מחלק משותף מקסימלי של 24 ו-18, די למצוא את המחלק המשותף המקסימלי של 18 ושל 6. כאשר נחלק את 18 ב-6, נקבל שארית 0, כלומר 6 מחלק את 18. מכאן ש-6 הוא המחלק המשותף המקסימלי של 18 ו-6, ולכן גם של 24 ו-18. באופן כללי נאמר שכדי למצוא מחלק משותף מקסימלי של שני מספרים, די למצוא את המחלק המשותף המקסימלי של הקטן מהם, ושל שארית חלוקת הגדול בקטן, אם שארית זו אינה 0. אם היא 0, אזי הקטן שבהם הוא המחלק

המשותף המקסימלי. החלפת זוג המספרים הנתון בזוג חדש, שהוא "פשוט יותר" (כיוון שהמספר הגדול הוחלף בשארית החלוקה שלו, שהוא מספר קטן יותר), מובילה לאלגוריתם רקורסיבי. האלגוריתם הרקורסיבי מקבל שני מספרים חיוביים j, k, ומחזיר את המחלק המשותף המקסימלי שלהם. האלגוריתם הרקורסיבי מגיע לתוצאת החישוב במהירות מפתיעה.

? מהו תנאי העצירה של האלגוריתם של אוקלידס?

ההנחה המקדימה של האלגוריתם של אוקלידס היא: $j > 0$, $k \geq 0$:

```
public static int euclidGcd(int j, int k)
{
      if (k == 0)
            return j;
      else
            return euclidGcd(k, j%k);
}
```

להלן פירוט שלבי הרקורסיה עבור קלט של שני מספרים גדולים למדי (j=1071, k=1029):

k	j	חלוקה	שארית החלוקה	מחליפים	ערך החזרה
1029	1071	1071/1029	42	j ← 1029 k ← 42	
42	1029	1029/42	21	j ← 42 k ← 21	
21	42	42/21	0	j ← 21 k ← 0	21

האלגוריתם של אוקלידס פשוט ואלגנטי, ומפתיע במהירות שבה הוא מגיע לתוצאה, גם עבור מספרים גדולים יחסית. נסו לדמיין כמה מסובך היה חישוב "רגיל" של המכנה המשותף הגדול ביותר, שהרי חישוב כזה מבצע לולאה בין המספרים 1 עד k (המספר הקטן מבין השניים) ובודק האם j ו- k מתחלקים בכל אחד מהמספרים האלה. במקרה שלנו הלולאה הייתה מתבצעת יותר מ-1000 פעמים!

זוהי דוגמה מובהקת לכוחה הרב של הרקורסיה, המאפשרת לעתים למצוא פתרונות אלגנטיים, פשוטים ויעילים לבעיות הנחשבות "קשות". לעתים קרובות, אחרי שנמצא פתרון כזה, אפשר למצוא גם פתרון איטרטיבי המבוסס על אותו רעיון. לפעמים, גם הפתרון האיטרטיבי יהיה אלגנטי ופשוט, אך בלא מעט מקרים הוא יהיה מסובך יותר מהפתרון הרקורסיבי (גם אם יעילותם זהה).

ז. פעולת עזר רקורסיבית

כדי להבין את הצורך בפעולות עזר רקורסיביות נתבונן בדוגמה שראינו בפרק על מציאת מקסימום במערך (ה.2). הכותרת של הפעולה היא:

```
public static int findMax(int[] arr, int begin, int end)
```

והקריאות הרקורסיביות מצמצמות את טווח החיפוש באמצעות האינדקסים begin ו-end.

אך ישנה אי נוחות מסוימת מבחינת המשתמש בקריאה מסוג זה. המשתמש רוצה לדעת מהו המקסימום במערך מסוים. ברור לו כי הוא צריך להעביר את המערך כפרמטר. אך הדרישה להעביר הן את תחילת המערך והן את סופו אינה מובנת (יתרה מזאת, הרי בקריאה שהמשתמש מבצע begin=0, ו- end=arr.length-1, לכן ההכנסה שלהם כפרמטרים נראית מיותרת לחלוטין).

לעתים קרובות נרצה לעטוף את הפעולה הרקורסיבית בפעולה נוספת אשר מסתירה פרטים טכניים מסוג זה.

נכתוב את פעולת ה"מעטפת":

```
public static int findMax(int[] arr)
{
      return findMax(arr, 0, arr.length-1);
}
```

פעולת ה"מעטפת" מבצעת את הקריאה לפעולה הרקורסיבית. נגדיר את הפעולה הרקורסיבית כפעולה פרטית, כיוון שהמשתמש אינו אמור להכיר אותה או להשתמש בה ישירות:

```
private static int findMaxHelp(int[] arr, int begin, int end)
{
      ...
}
```

בצורה זו המשתמש מזמן את הפעולה בצורה נוחה והגיונית:

```
findMax(arr);
```

ואינו שולח את האינדקסים כפרמטרים.

מובן שגם אם משתמשים בפתרון הרקורסיבי הראשון לבעיית החיפוש במערך נוצרת אותה הבעיה, משום שאנו נדרשים להעביר גם ערך של אינדקס אחד. גם כאן הפתרון המוצע הוא להשתמש בכותרת של פעולת חיפוש שאינה מציינת את האינדקס במערך שבו החיפוש מתחיל אלא מעבירה את המערך עצמו בלבד. בגוף הפעולה יהיה זימון לפעולת עזר פרטית ובין הפרמטרים שלה יהיה כלול גם האינדקס הנדרש.

ח. סיכום

אלגוריתם רקורסיבי הוא אלגוריתם שמשמש לפתרון בעיות שאינן פשוטות דיין לפתרון ישיר. האלגוריתם הרקורסיבי מפעיל את עצמו פעם אחת או יותר, על מופעים אחרים של אותה בעיה. כדי שהגדרה רקורסיבית של אלגוריתם (פתרון) תהיה יעילה, חשוב שמופעי הבעיה הנפתרים בהפעלות הרקורסיביות יהיו "קטנים יותר" או "פשוטים יותר" מהמופע המקורי. תנאי זה מבטיח שתהליכי הפירוק מגיעים בסופו של דבר לבעיות פשוטות שאותן ניתן לפתור ישירות.

- רקורסיה היא **הגדרה של מושג או אלגוריתם (כלומר של פתרון לבעיה), המשתמשת במושג או באלגוריתם המוגדר**. אלגוריתם רקורסיבי יזמן את עצמו שוב ושוב על ערכי פרמטרים הולכים וקטנים, עד שיגיע למקרה הבסיסי שהפתרון שלו מתקבל באופן ישיר.
- כשכותבים פתרון רקורסיבי לבעיה יש להגדיר שני מצבים: מקרה מורכב שאותו נפרק למקרים פשוטים יותר, ומקרה בסיסי, כלומר תנאי עצירה, המגדיר את הפתרון למקרה הפשוט בלי זימון נוסף של אלגוריתם הפתרון.
- קיימים סוגי רקורסיה רבים. בפרק זה התייחסנו במפורט לארבעה: "רקורסיית זנב" המסתיימת עם פתרון המקרה הבסיסי; "רקורסיית הלוך-חזור" שבה לאחר פתרון המקרה הבסיסי יש לחזור עם הערך המחושב ולבצע חישובים קודמים שממתינים לסיומם; "רקורסיה כפולה" המזמנת את ההליך הרקורסיבי פעמיים בכל שלב; "רקורסיה הדדית", המתבצעת באמצעות שתי פעולות שונות (לפחות).
- באלגוריתמים על מערכים, הדרך הטיפוסית לקבלת פתרון רקורסיבי היא על ידי הגדרת הבעיה מחדש, כך שתתייחס לקטעים של מערכים. כך מתקבלים מקרים פשוטים יותר על ידי צמצום קטעי המערך רק לחלקים שעליהם האלגוריתם פועל.

מושגים

הגדרה רקורסיבית	recursive definition
מקרה בסיסי או תנאי עצירה	base case
מקרה מורכב	
רקורסיה	recursion
רקורסיה הדדית	
רקורסיה כפולה	double recursion
רקורסיית הלוך-חזור	
רקורסיית זנב (או: רקורסיית הלוך)	tail recursion

תרגילים

א. מעקב אחר רקורסיות

לפניכם כמה פעולות רקורסיביות. עבור כל אחת מהן רשמו את טענת היציאה.

1. *<u>טענת כניסה</u>: הפעולה מקבלת תו ומספר שלם אי-שלילי.*

```
public static void mystery(char ch, int n)
{
      if (n > 0)
      {
            System.out.print(ch);
            mystery(ch, n-1);
      }
}
```

2. *<u>טענת כניסה</u>: הפעולה מקבלת שני מספרים שלמים חיוביים.*

```
public static int mystery(int a, int b)
{
        if (a < b)
                return (0);
        else
                return (1 + mystery(a-b, b));
}
```

3. *<u>טענת כניסה</u>: הפעולה מקבלת מספר שלם חיובי.*

```
public static int mystery(int n)
{
      if (n < 10)
            return (n);
      else
      {
            int x = n % 10;
            int y = mystery(n / 10);
            if (x > y)
                  return x;
            else
                  return y;
      }
}
```

4. _טענת כניסה_: הפעולה מקבלת מספר שלם חיובי.

```
public static int mystery(int num)
{
      if (num == 1)

            return (1);

      else

            return (mystery(num – 1) + 2 * num – 1);
}
```

5. _טענת כניסה_: הפעולה מקבלת מערך של מספרים שלמים שאינו ריק, ומספר נוסף שלם וחיובי

הקטן או שווה לגודל המערך.

```
public static float mystery(int[] a, int k)
{
      float x;

      if (k == 1)
            return (a[0]);
      x = mystery(a, k–1) * (k–1);
      return ((a[k–1] + x) / k);
}
```

6. _טענת כניסה_: הפעולה מקבלת מספר שלם חיובי.

```
public static int mystery(int num)
{
      if (num < 10)
            return (num);
      else
      {
            int i = 10;
            while (num % i != num)
                  i *= 10;
            return ((num % 10) * i / 10) + mystery(num / 10);
      }
}
```

7. _טענת כניסה_: הפעולה מקבלת שני מספרים שלמים, המספר השני הוא גם אי-שלילי.

```
public static int mystery(int a, int b)
{
      if (b == 0)
            return 0;
      if (b % 2 == 0)
            return mystery(a + a, b / 2);
      return mystery(a + a, b / 2) + a;
}
```

לפניכם כמה פעולות רקורסיביות ובצמוד לכל אחת טענת הכניסה שלה וטענת היציאה שלה.
בכל פעולה חסרים ביטויים אחדים שעליכם להשלים כדי שהפעולה תבצע את הנדרש.

.8

<u>טענת כניסה</u> : הפעולה מקבלת מספר שלם וחיובי.

<u>טענת יציאה</u> : הפעולה מחזירה את סכום הספרות של המספר המתקבל.

```
public static int sumDigits(int num)
{
      if (____________)

            return (num);

      return (num % 10 + sumDigits(____________));
}
```

.9

<u>טענת כניסה</u> : הפעולה מקבלת מספר שלם וחיובי.

<u>טענת יציאה</u> : הפעולה מחזירה את מספר הספרות של המספר המתקבל.

```
public static int numDigits(int num)
{
      if (num < 10)

            return (____________);

      return (____________ + numDigits(____________));
}
```

.10

<u>טענת כניסה</u> : הפעולה מקבלת מחרוזת.

<u>טענת יציאה</u> : הפעולה מדפיסה את המחרוזת המתקבלת בסדר הפוך.

```
public static void reverseString(String str)
{
      if (str.length() > 0)
      {
            char ch = str.charAt(____________);
            reverseString(____________);
            System.out.print(ch);
      }
}
```

.11

<u>טענת כניסה</u>: הפעולה מקבלת מספר שלם חיובי וספרה.

*<u>טענת יציאה</u>: הפעולה מחזירה **true** אם הספרה נמצאת במספר, אחרת מחזירה **false**.*

```
public static boolean isDigitExist(int num, int digit)
{
      boolean ans = num%10 == digit;
      if (____________)
            return (ans);
      return (____________);
}
```

.12

<u>טענת כניסה</u>: הפעולה מקבלת מספר שלם חיובי.

*<u>טענת יציאה</u>: הפעולה מחזירה **true** אם כל הספרות במספר זוגיות, אחרת מחזירה **false**.*

```
public static boolean isAllDigitsEven(int num)
{
      boolean ans = ____________;
      if (num < 10)
            return (____________);
      return (ans && isAllDigitsEven(____________));
}
```

בכל אחת מהשאלות הבאות עליכם לבחור פעולה רקורסיבית אחת מהשלוש המוצעות, כך שתתאים לטענת הכניסה והיציאה המוצגות.

.13

<u>טענת כניסה</u>: הפעולה מקבלת שני מספרים שלמים: $m > 0$, $n >= 0$.

<u>טענת יציאה</u>: הפעולה מחזירה $n \% m$.

א.

```
public static int mod(int n, int m)
{
      if (n > m)
            return m;
      else
            return mod(m-n, m);
}
```

ב.

```
public static int mod(int n, int m)
{
      if (m < n)
            return 1;
      else
            return mod(n-m, m);
}
```

ג.

```
public static int mod(int n, int m)
{
      if (n < m)
            return n;
      else
            return mod(n-m, m);
}
```

.14

<u>טענת כניסה</u> : הפעולה מקבלת מחרוזת.

<u>טענת יציאה</u> : הפעולה מחזירה מחרוזת הפוכה לזו שהתקבלה.

א.

```
public static String rev(String str)
{
      if (str.length() == 0)
            return str;
      else
            return rev(str.substring(1)) + str.charAt(0);
}
```

ב.

```
public static String rev(String str)
{
      if (str.length() == 0)
            return str;
      else
            return rev(str.substring(0)) + str.charAt(0);
}
```

ג.

```
public static String rev(String str)
{
      if (str.length() == 0)
            return str;
      else
            return rev(str.substring(0)) + str.charAt(1);
}
```

הַצָּב הרקורסיבי – הריצו כל אחת מהתוכניות הבאות ובדקו מה היא מבצעת.

15. תוכנית ראשונה:

```
public class RecursiveTurtleDemo1
{
      public static void main(String[] args)
      {
            Turtle t = new Turtle();
            t.setDelay(50);
            t.turnRight(90);
            t.moveBackward(200);
            t.tailDown();
            t.setTailColor(Color.BLUE);
            draw(t, 15, 4);
            t.tailUp();
            t.moveForward(40);
      }
      public static void draw(Turtle t, int width, int step)
      {
             if (step <= 1)
                   t.moveForward(width);
             else
             {
                   draw(t, width, step-1);
                   t.turnLeft(60);
                   draw(t, width, step-1);
                   t.turnRight(120);
                   draw(t, width, step-1);
                   t.turnLeft(60);
                   draw(t, width, step-1);
             }
      }

}
```

16. תוכנית שנייה:

```
public class RecursiveTurtleDemo2
{
      public static void main(String[] args)
      {
            Turtle t = new Turtle();
            t.setDelay(50);
            t.turnRight(90);
            t.moveBackward(200);
            t.tailDown();
            t.setTailColor(Color.RED);
            hilbert0(t, 4, 10);
            t.tailUp();
            t.moveForward(40);
      }
```

```
    public static void hilbert0(Turtle t, int n, int step)
    {
        if (n > 0)
        {
            t.turnRight(90);
            hilbert1(t, n-1, step);
            t.moveForward(step);
            t.turnLeft(90);
            hilbert0(t, n-1, step);
            t.moveForward(step);
            hilbert0(t, n-1, step);
            t.turnLeft(90);
            t.moveForward(step);
            hilbert1(t, n-1, step);
            t.turnRight(90);
        }
    }

    public static void hilbert1(Turtle t, int n, int step)
    {
        if (n > 0)
        {
            t.turnLeft(90);
            hilbert0(t, n-1, step);
            t.moveForward(step);
            t.turnRight(90);
            hilbert1(t, n-1, step);
            t.moveForward(step);
            hilbert1(t, n-1, step);
            t.turnRight(90);
            t.moveForward(step);
            hilbert0(t, n-1, step);
            t.turnLeft(90);
        }
    }
}
```

ב. כתיבת רקורסיות

לפני שתתחילו את כתיבת כל אחד מהאלגוריתמים, ענו על השאלות האלה:

א) ציינו את קלט הכניסה לפעולה הרקורסיבית, ומהו מקרה הבסיס (תנאי העצירה).

ב) מה קורה במקרי קצה שונים? האם יש קלטים המחייבים התייחסות מיוחדת?

ג) הגדירו לעצמכם מהו סוג הרקורסיה שלפניכם (רקורסיית זנב, רקורסיית הלוך-חזור, רקורסיה הדדית).

ד) ציינו מה צריך להתבצע בגוף הרקורסיה ומה היא מחזירה.

תרגילים על ספרות ומספרים:

17. כתבו פעולה רקורסיבית המקבלת מספר מטיפוס **int**, ומחזירה את מספר ספרותיו.

18. כתבו פעולה רקורסיבית המקבלת שני מספרים חיוביים ומחזירה את מספר הספרות הזהות בערכן ובמיקומן בשני המספרים (כלומר עליכם לערוך השוואה בין ספרת האחדות, ספרת העשרות, ספרת המאות וכן הלאה).

19. כתבו פעולה רקורסיבית המקבלת מספר מטיפוס **int**, ומחזירה את ממוצע ספרותיו.

20. כתבו פעולה רקורסיבית המקבלת מספר חיובי מטיפוס **int** ומדפיסה אותו בבסיס בינרי.

21. הפעולה numPrefix מקבלת מספר שלם שונה מאפס ומדפיסה את השורות האלה:

    בשורה הראשונה יופיע המספר כולו.

    בשורה הבאה – המספר ללא ספרת האחדות שלו.

    בשורה שאחריה – המספר המקורי ללא ספרות העשרות והאחדות, וכך הלאה.

    עד שבשורה האחרונה תודפס הספרה המשמעותית ביותר של המספר המקורי.

    לדוגמה, עבור קלט: 29807 יודפסו השורות הבאות:

    ```
    29807
    2980
    298
    29
    2
    ```

    כתבו את הפעולה numPrefix.

22. כתבו אלגוריתם **חשב-חזקה (x, y)**, המחשב את x^y – תוצאת העלאת x בחזקת y, כאשר שני המספרים שלמים, y לא שלילי, x חיובי ממש. ממשו את הפעולה.

תרגילים על מערך:

23. כתבו פעולה רקורסיבית המקבלת מערך של מספרים ומחזירה את סכום איבריו.

24. כתבו פעולה רקורסיבית הבודקת האם מערך נתון ממוין בסדר עולה.

25. כתבו פעולה רקורסיבית הבודקת האם ערכי המערך מהווים סדרה חשבונית (כלומר קיים הפרש קבוע בין איברי המערך).

תרגילי הדפסה:

26. כתבו פעולה רקורסיבית המקבלת כפרמטר מספר טבעי n ומדפיסה משולש של כוכביות, ובו n שורות. בשורה העליונה של המשולש יהיו n כוכביות צמודות, בשורה שמתחת יהיו n-1 כוכביות צמודות, וכך הלאה. בשורה התחתונה תודפס כוכבית אחת.

לדוגמה, עבור n = 4 יודפס:

```
****
***
**
*
```

הגדירו מהו סוג הרקורסיה שאתם מבצעים בפתרון שכתבתם? הסבירו.

27. כתבו פעולה רקורסיבית המקבלת כפרמטר מספר טבעי n ומדפיסה משולש של כוכביות, ובו n שורות. בשורה העליונה של המשולש תהיה כוכבית אחת, בשורה השנייה שתי כוכביות צמודות וכך הלאה. בשורה האחרונה יודפסו n כוכביות צמודות.

לדוגמה, עבור n = 4 יודפס:

```
*
**
***
****
```

28. כתבו פעולה רקורסיבית המקבלת כפרמטר מספר טבעי n, ומדפיסה צורה של שני משולשים המחוברים בקודקודם. בשורה הראשונה שלו n כוכביות ובשורה האחרונה שלו n כוכביות:

לדוגמה, עבור n = 3:

```
***
**
*
**
***
```

ג. תרגילי אתגר

29. פרמוטציות

תמורה (פרמוטציה, permutation) של מערך היא סידור כלשהו של איברי המערך.

כתבו פעולה רקורסיבית המדפיסה את כל התמורות של מערך נתון.

לדוגמה: עבור המערך שמכיל את 2, 4, 8

יודפסו התמורות הבאות:

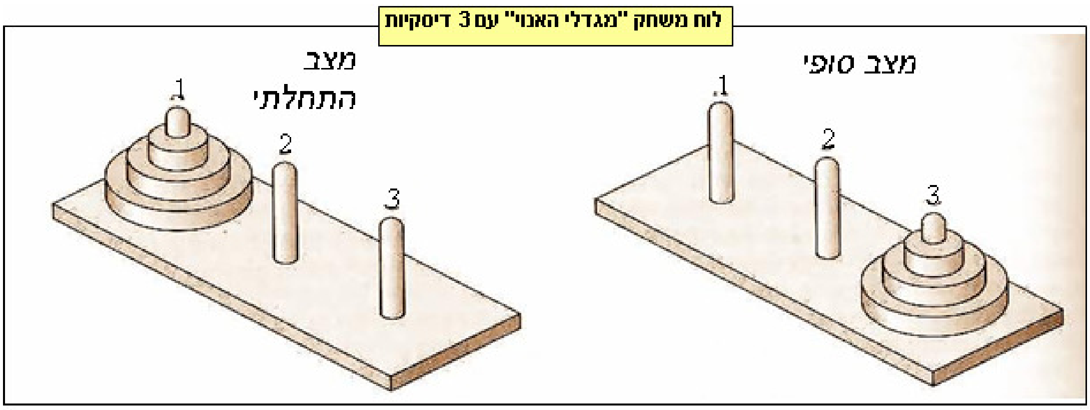
8, 4, 2
4, 8, 2
8, 2, 4
2, 8, 4
4, 2, 8
2, 4, 8

30. חידת מגדלי האנוי (Towers of Hanoi)

חידת מגדלי האנוי היא חידה מתמטית שהומצאה על ידי המתמטיקאי הצרפתי אדוארד לוקאס ב-1883, והפכה למשחק חביב. המשחק בנוי מלוח שבו נעוצים שלושה מוטות הממוספרים מ-1 עד 3 ומדיסקיות בהיקפים שונים. בתחילת המשחק, על מוט מספר 1 מושחלות n דיסקיות מהדיסקית שהיקפה גדול בסדר יורד (כמתואר באיור). הדיסקיות יוצרות צורה של מגדל ומכאן שמו של המשחק. מטרת המשחק היא להעביר את מגדל הדסקיות בשלמותו ממוט 1 למוט 3 על פי הכללים האלה:

א. ניתן להעביר דיסקית אחת בלבד בכל שלב.

ב. באף שלב אסור שדיסקית גדולה תהיה מונחת על דיסקית קטנה ממנה.



כתבו פעולה רקורסיבית:

```
public static void solveHanoi(int n)
```

המקבלת מספר טבעי n, ומדפיסה את סדרת ההזזות שיש לבצע על מנת להעביר את n הדיסקיות ממוט מספר 1 למוט מספר 3 על פי כללי המשחק.

למשל עבור הזימון solveHanoi (2), כלומר משחק ב-2 דיסקיות, יתקבל פלט זה:

הזז מ-1 ל-2

הזז מ-1 ל-3

הזז מ-2 ל-3

פירושה של הפקודה "הזז מ-a ל-b" היא הזזת הדיסקית העליונה ממוט מספר a למוט מספר b.

רמז: היעזרו בפעולת עזר רקורסיבית, אשר תקבל כפרמטר את מספרי המוטות ואת מספר הדיסקיות הכולל:

```
private static void solveHanoi(int a, int b, int c, int n)
```

פעולה זו תעביר את n הטבעות ממוט מספר a למוט מספר c בעזרת מוט מספר b.

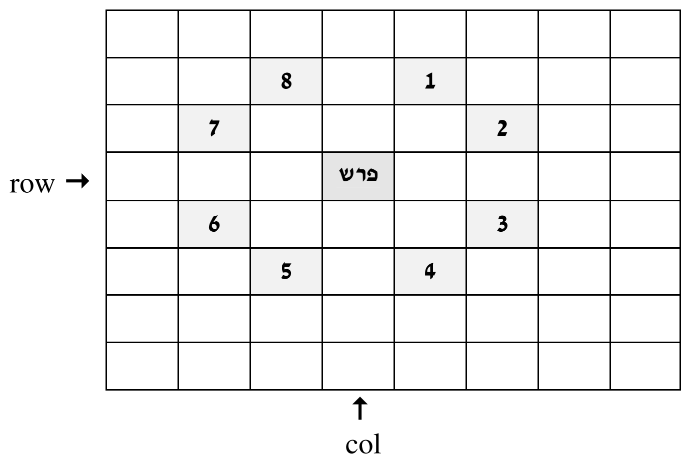
31. בעיית שמונה המלכות:

מלכה (Queen) במשחק שח-מט יכולה לנוע בשורה, בטור או באלכסון יחסית למקום שהיא עומדת בו. המלכה יכולה לנוע מספר בלתי מוגבל של משבצות בכיוון הנבחר, כל עוד כלי אחר אינו חוסם את דרכה, וכן כל עוד לא יצאה מהלוח. מלכה יכולה להכות כל כלי העומד בדרכה אך אחרי שהכתה כלי (כלומר הוציאה אותו מהמשחק ונעמדה במקומו), היא אינה יכולה לנוע יותר עד לתור הבא. מלכה מאיימת על כל הכלים הנמצאים בשורה, בעמודה או באלכסון של המשבצת בה היא ניצבת (כלומר יכולה להוציאם בתורה).

כתבו פעולה רקורסיבית שתציב שמונה מלכות על לוח שחמט ריק (מערך דו-ממדי בגודל 8x8), כך שאף אחת לא תאיים על האחרות.

32. בעיית מסעי פרש

פרש (knight) במשחק שח-מט (מערך דו-ממדי בגודל 8x8) העומד על משבצת שמיקומה (row, col) יכול להגיע לכל אחת מ-8 המשבצות (אם קיימות), כמתואר באיור זה:



מעבר הפרש למשבצת אפשרית נקרא **מסע פרש (knight move)** המורכב ממעבר על שתי משבצות רצופות בכיוון מאוזן ואחת בכיוון מאונך, או שתי משבצות רצופות בכיוון מאונך ואחת בכיוון מאוזן. שימו לב, כי אם הפרש נמצא על אחת המשבצות שבמסגרת הלוח או קרוב לה, הוא אינו יכול לבצע את כל 8 המסעות!

כתבו פעולה רקורסיבית שתאפשר ביקור של הפרש **בכל** משבצות לוח השח-מט. הביקור יתחיל במשבצת (0,0). בסוף התוכנית בכל משבצת בלוח יופיע מספר (מ-1 ועד 64) המתאר את מספר הביקור של הפרש במשבצת. לדוגמה התבוננו בלוח מסעי הפרש שהתקבל:

1	30	47	52	5	28	43	54
48	51	2	29	44	53	6	27
31	46	49	4	25	8	55	42
50	3	32	45	56	41	26	7
33	62	15	20	9	24	39	58
16	19	34	61	40	57	10	23
63	14	17	36	21	12	59	38
18	35	64	13	60	37	22	11

33. בעיית המבוך

נתון מערך דו-ממדי המדמה מבוך שלו נקודת כניסה אחת ונקודת יציאה אחת. יש לכתוב פעולה רקורסיבית המדפיסה את המסלול ליציאה מהמבוך.