חישוב חזקה

תיאור הבעיה

נרצה לחשב חזקה בצורה היעילה ביותר.

⇐ נרצה לחשב את a^n.

בראייה ראשונית, אפשר לראות כי סיבוכיות זמן הריצה עבור $a^n = a \cdot a \cdot .... \cdot a$ הוא $O(n)$.
בדומה לנושא של אינדוקציה מול רקורסיה נראה את 2 הפתרונות עבורם ונחליט איזה שיטה יעילה יותר.

חישוב חזקה בשיטה **רקורסיבית**

```
public static int powerRecursion(int a, int n) {
   return n == 0 ? 1 : a * powerRecursion(a,n-1);
}
public static void main(String[] args) {
   System.out.println(powerRecursion(2,5)); // 32
   System.out.println(powerRecursion(3,3)); // 27
}
```

בנושא אינדוקציה מול רקורסיה הראינו כי רקורסיה מקצה זיכרון נוסף עבור כל קריאה לפונקציה כולל עבור כל הפרמטרים הנמצאים בתוך הפונקציה כולל המשתנים בארגומנט השליחה.
סה"כ סיבוכיות זמן הריצה היא $O(2n) \ = \ O(n)$.

חישוב חזקה בשיטה **אינדוקטיבית**

```
public static int powerInduction(int a, int n) {
   int solution = 1;
   while (n > 0) {
       solution *= a;
       n--;
   }
   return solution;
}
public static void main(String[] args) {
   System.out.println(powerInduction(2,5)); // 32
   System.out.println(powerInduction(3,3)); // 27
}
```

גם כאן קל לראות כי סיבוכיות זמן הריצה היא $O(n)$ אך בגלל שרקורסיה עושה פעולות נוספות של הקצאות זיכרון עבור כל קריאה, השיטה האינדוקטיבית יותר יעילה במקרה זה.

עד עכשיו למדנו חישוב חזקה בסיבוכיות זמן ריצה של $O(n)$, לדוגמה, עבור x^8 נחשב ידנית באופן הבא:

$$x^8 = x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x$$

האם אפשר לספק פתרון נוסף ביעילות טובה יותר מ- $O(n)$?
אפשר לראות כי עבור x^8 ביצענו 7 פעולות כפל עד שהגענו לפתרון, אנו יכולים לייעל את השיטה כך שנבצע פחות פעולות הכפלה ואז סיבוכיות זמן הריצה שלנו תרד משמעותית.
אם

$$x^4 = x^2 \cdot x^2$$

אז באופן דומה, אפשר לחשב את x^8 בצורה הבאה:

$$x^8 = x^2 \cdot x^2 \cdot x^2 \cdot x^2$$

ככה בעצם קיבלנו 3 פעולות הכפלה במקום 7 פעולות, שזה בעצם $\log_2 8 = 3 \;\Rightarrow\; \log n$.
לכן הסיבוכיות עשויה להיות $O(log\ n)$.

ידוע שכל מספר עשרוני ניתן לכתוב כסכום של מספרים בחזקות של 2.
נסתכל על x^{11} , ננסה לפתור זאת באמצעות הנוסחה הבאה: $x^{11} = x^8 \cdot x^2 \cdot x^1$.
אנו רואים שכאן השקענו סוג של $O(log\ n)$ פעולות, יש באפשרותינו כעת להעזר בחזקות הבאות:

$$x^1 ,\ x^2 ,\ x^4,\ x^8$$

מאחר ו- $11 = 8 + 2 + 1$ נרשום את הייצוג הבינארי של המספר הזה:

$$11 = 8 + 2 + 1 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

כלומר $11_{10} = 1011_2$.

דרך נוספת וקלה יותר לקראת הפתרון, הוא לחלק את מספר החזקה בכל צעד בחצי, אם השארית היא 1 אז נחבר את החישוב לתוצאה הסופית, אחרת לא נחשב כלום ונמשיך לצעד הבא, כלומר:

מספר	שלם	שארית
11/2	5	1
5/2	2	1
2/2	1	0
1/2	0	1

נקרא את המספר שקיבלנו בשארית מלמטה למעלה וזה באמת 1011.

חישוב חזקה בעזרת מספרים בינאריים בשיטה **רקורסיבית**

```
public static int recursion(int a, int n) {
   return rec(a, n , 1);
}

public static int rec(int a, int n, int answer) {

   if(n == 0)
       return answer;

   if(n % 2 == 1)
       answer = answer * a;

   return rec(a * a,n/2,answer);
}

public static void main(String[] args) {
   System.out.println(recursion(2,5)); // 32
   System.out.println(recursion(3,3)); // 27
}
```

אמנם אין לנו 2 קריאות לפונקציה בכל שלב בפונקציה, אך זה לא אומר שלא מדובר בסיבוכיות $O(log\ n)$ כי מאחר ותנאי העצירה שלנו הוא באחריות פרמטר n וכי בכל צעד אנו קוראים לצעד הבאה כאשר $n/2$ אז באמת מתקיים כאן סיבוכיות זמן ריצה של $O(log\ n)$.

חישוב חזקה בעזרת מספרים בינאריים בשיטה **אינדוקטיבית**

```
public static int induction(int a, int n) {
   int answer = 1;

   while (n > 0) {

       if(n % 2 == 1)
           answer = answer * a;

       n = n / 2;
       a = a * a;
   }

   return answer;
}
```

גם כאן סיבוכיות זמן הריצה היא $O(log\ n)$.