אינדוקציה-רקורסיה

מתי עדיף להשתמש ברקורסיה ומתי עדיף בלולאה?

פתרון:

מבחינת ניהול זיכרון של רקורסיה, בכל קריאה לפונקציה נוצר עותק חדש שלה, בעותק זה יש:

1. הקצאה חדשה של כל המשתנים הפנימיים, כולל הפרמטרים המופיעים בחתימה שלה.
2. השמה התחלתית של ערכים לפרמטרים לפי מה שהועבר בקריאה אליה.
3. אתחול של הפקודה הבאה לבצע בקוד- בעותק החדש זו הפקודה הראשונה בפונקציה.

נשתמש ברקורסיה כאשר האלגוריתם נותן עץ בגובה $\log_2 n$ או קטן יותר.

נשתמש בלולאה כאשר רקורסיה מביאה לעץ בגובה n עם מס' צמתים קרוב ל2^n וכאשר אין פיצולים והפונקציה קוראת לעצמה פעם אחת.

<u>דוגמאות למימושים :</u>

1. חישוב עצרת :
קלט – מספר אי שלילי שלם, פלט- $n!$.

מימוש רקורסיבי	מימוש אינדוקטיבי (לולאה)

`public static int factorial(int n) {`
`  if( n == 0 )`
`  return 1;`
 	
`  return n * factorial(n-1);`
`}`
	`public static int factorial(int n) {`
	
	`  int ans = 1;`
	`  for(int i = 1 ; i <= n ; i++) {`
	`  ans *= i;`
	`  }`
	
	`  return ans;`
	`}`

סיבוכיות נעבור $2n$ פעמים (נכנסים n פעמים ויוצאים n פעמים) סה"כ: $O(n)+O(n)=O(2n)=O(n)$	**סיבוכיות** נעבור פעם אחת מ1 עד n , סה"כ $O(n)$.

אמנם שני המימושים הם *O(n)*, אך עדיף לממש לולאה במקום רקורסיה כי כך אין שימוש נוסף בזיכרון.

2. חישוב פיבונאצ'י:
קלט- מספר אי שלילי, פלט- ערך באינדקס לפי סדר הסדרה.

מימוש רקורסיבי	מימוש אינדוקטיבי (לולאה)

`public static int fibo(int n) {`
`  if( n == 0 || n == 1) {`
`  return n;`
`  }`
`  return fibo(n-1) + fibo(n-2);`
`}`
	`public static int fibo(int n) {`
	`  int[] arr = new int[n+1];`
	`  arr[0] = 0;`
	`  arr[1] = 1;`
	
	`  for(int i = 2 ; i <= n ; i++)`
	`  arr[i] = arr[i-1] + arr[i-2];`
	
	`  return arr[n];`
	`}`

סיבוכיות כל קריאה לפונקציה גוררת שתי קריאות לפונקציה- לכן, הסיבוכיות היא $O(2^n)$.	**סיבוכיות** עבור ההשמה של איבר 0 ו-1 זה *O(2)*. עבור הלולאה אנו רצים *n-1* פעמים פחות 2 ההשמות שבוצעו כבר, ולכן התחלנו לרות החל מאינדקס 2, כלומר הלולאה מבצעת *O(n-3)=O(n)*. סה"כ *O(n)+O(2)=O(n)*

3. חישוב *merge sort*:
עדיף לחשב ברקורסיה ולא בלולאה כי שימוש ברקורסיה הופך את הקוד לפשוט וקריא.
באלגוריתם גובה העץ הוא $\log_2 n$ (בגלל החלוקה ב-2), ומספר הקריאות לפונקציה הוא 32, ומספר החזרות מהפונקציה הוא 32.
סה"כ 64 שלא יגרמו ל*stack overflow*.