מיון מיזוג, חיפוש בינארי, אסימפטוטיקה ועוד...

אסימפטוטיקה

$$1 < log(n) < nlogn < n^2 < n^3 < ... < 2^n < 3^n < n^n$$

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Merge Sort

סיבוכיות $O(n \cdot log(n))$ $\Theta(n \cdot log(n))$ $\Omega(n \cdot log(n))$

מיון זה הוא רקורסיבי. הוא מחלק את המערך לשתי קבוצות, כל קבוצה הוא שוב מחלק לשניים וכן האלה (באופן רקורסיבי) עד תנאי העצירה – קבוצות בנות איבר אחד.

בשלב השני הפונקציה ממזגת כל תת-קבוצה עם תת-קבוצה אחרת, באמצעות מיזוג של שתי קבוצות ממוינות, וכן הלאה עד להיווצרות מערך ממוין.

הסבר

בכל קריאה לפונקציה **Merge-Sort** מחלקים את המערך לשניים, לכן עלינו לחשב כמה פעמים יש לחלק מספר בשניים עד שנגיע ל-0. במילים אחרות, כמה פעמים נצטרך להכפיל את 2 בעצמו (חזקה) על מנת להגיע למספר המבוקש:

$$2^x = N \Rightarrow x = log_2 N \Rightarrow O(log_2 N)$$

לאחר כל חלוקה קוראים לפונקציה ***Merge***. הסיבוכיות שלה היא $O(n)$ משום שהיא עוברת באופן סדרתי על שני חלקי של המערך ומשוואה איבר לאיבר. בנוסף לזה הפונקציה עוברת עוד 3 פעמים על המערך:

$$3 \cdot O(n) + O(n) = O(n)$$

הסיבוכיות של הפונקציה **Merge-Sort** היא:

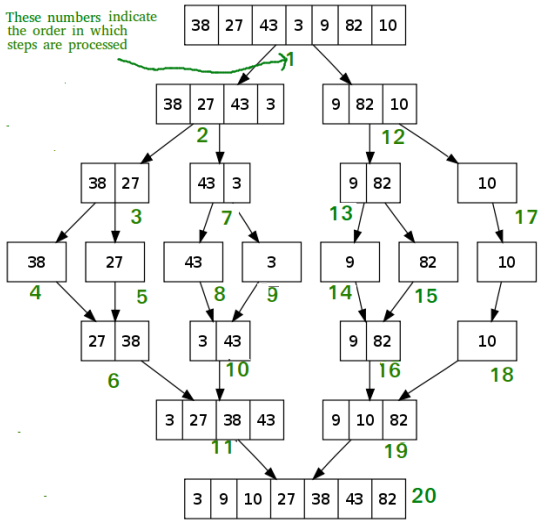
$$O(log_2 n) \cdot O(n) = O(n \cdot log_2 n)$$

סרטון אלגוריתם

מימוש

```
private static void mergeSort(int[] arr) {
   mergeSort(arr,0,arr.length-1);
}

private static void mergeSort(int[] arr, int left, int right) {
   if(left < right) {
       int middle = (left+right)/2;

       mergeSort(arr,left,middle); // left
       mergeSort(arr,middle+1,right); // right
       Merge(arr,left,middle,right);
```

```
    }
}

private static void Merge(int[] arr, int left, int middle, int right) {

    int[] temp = new int[right - left + 1];

    int i = left; // left half
    int j = middle + 1; // right half
    int k = 0; // The Running Pointer

    while( i <= middle && j <= right) {

        if(arr[i] < arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }

    while(i <= middle)
        temp[k++] = arr[i++];

    while(j <= right)
        temp[k++] = arr[j++];


    for(i = left, k = 0 ; k < temp.length && i <= right; k++, i++)
        arr[i] = temp[k];
}

public static void main(String[] args) {
    int[] arr = {48,3,7,9,43,1,2,4,6,8};
    mergeSort(arr);
    System.out.println(Arrays.toString(arr));
}
```



Binary Search

סיבוכיות $O(log(n))$ $\Theta(log(n))$ $\Omega(log(n))$

הסבר

בכל קריאה לפונקציה ***binarySearch*** אנו מחלקים את המערך לשניים, על כן עלינו לחשב את מספר הפעמים שבהן נצטרך לחלק מספר בשניים עד שנגיע ל-0. במילים אחרות, כמה פעמים נצטרך להכפיל את 2 בעצמו (חזקה) על מנת להגיע למספר המבוקש:

$$2^x = N \Rightarrow x = log_2 N \Rightarrow O(log_2 N)$$

סרטון אלגוריתם

מימוש

```
public static int binarySearch(int arr[], int left, int right, int x) {

   if(right >= 1) {

       int middle = left + (right - left)/2;

       if(arr[middle] == x)
           return middle;

       if(arr[middle] > x)
           return binarySearch(arr, left, middle - 1 , x);

       else
           return binarySearch(arr, middle + 1 , right, x);
   }
   return -1;
}

public static void main(String[] args) {
   int[] arr = {48,3,7,9,43,1,2,4,6,8};
   System.out.println(binarySearch(arr,0,arr.length-1,43)); // 4
}
```