

# ALGORITHMS AND DATA STRUCTURES

## LECTURE 6 - SORTING

---

Askar Khaimuldin  
[askar.khaimuldin@astanait.edu.kz](mailto:askar.khaimuldin@astanait.edu.kz)



# CONTENT

1. Bubble sort
2. Heap Sort
3. Divide and Conquer
4. Merge Sort
5. Quick Sort



# Sorting (Bubble sort) – $O(n^2)$

- Bubble Sort is the simplest sorting algorithm
- Several passes through the array
- Successive pairs of elements are compared
- Repeatedly swaps the adjacent elements if they are in wrong order
- At each  $i$ 'th iteration of the outer loop the maximum (can be minimum) element is moved to the position of  $n-i-1$

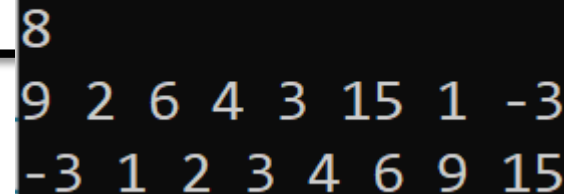
```
int a[10], n;
cin >> n; // actual number of needed elements
for (int i = 0; i < n; i++) {
    cin >> a[i]; // input each element
}

for (int i = 0; i < n - 1; i++) { // sorting
    for (int j = 0; j < n - i - 1; j++) {
        /* if (a[j] < a[j+1]) - descending order condition */
        if (a[j] > a[j + 1]) { // ascending order condition
            int temp = a[j]; // swapping
            a[j] = a[j + 1];
            a[j + 1] = temp;
        }
    }
}

for (int i = 0; i < n; i++) {
    cout << a[i] << " ";
}
```

Input:

Output:

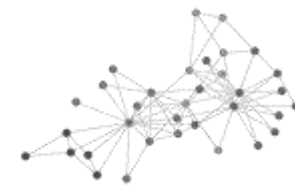
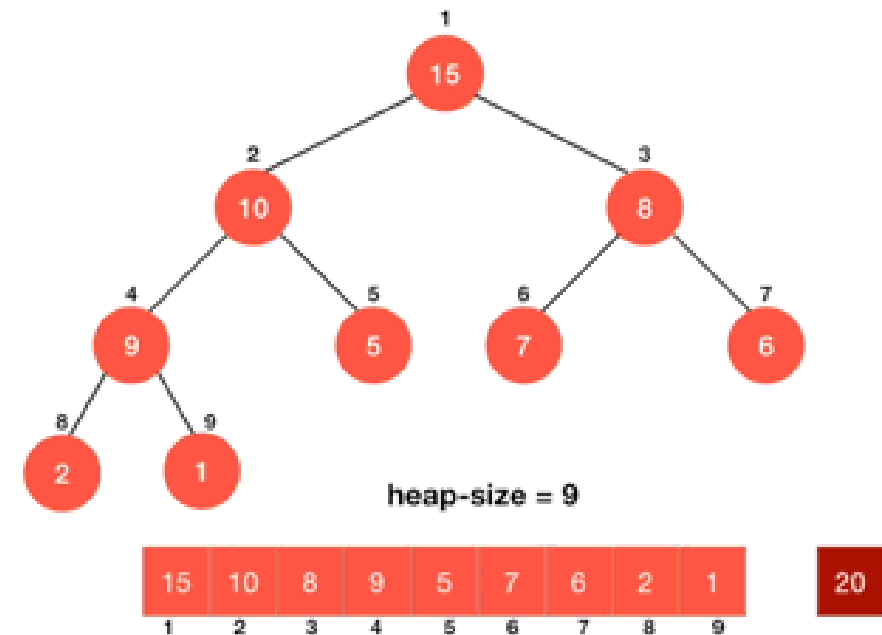


```
8
9 2 6 4 3 15 1 -3
-3 1 2 3 4 6 9 15
```



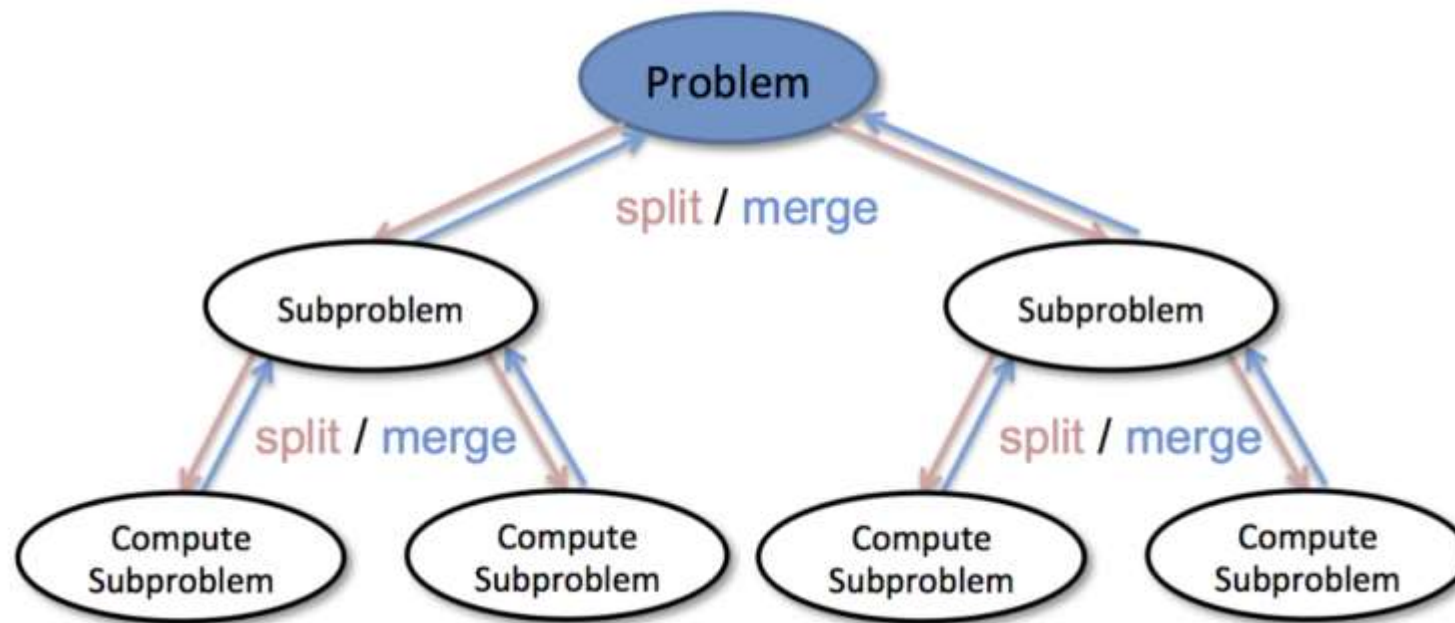
# HEAP SORT — $O(N \log N)$

1. Heap tree must be built from the input data to implement heap sort
2. Swap the root with the last item of the heap followed by reducing the size of heap by 1 (**removing** the root element)
3. **Heapify** the root of the tree
4. Repeat step 2 and 3 while size of heap is greater than 1



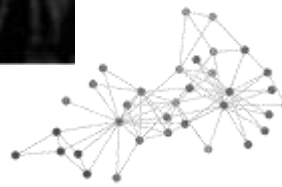
# DIVIDE-AND-CONQUER

Divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple enough to be solved directly



# JOHN VON NEUMANN

- Invented a Merge Sort algorithm in 1945, in which the first and second halves of an array are each sorted recursively and then merged



# MERGE SORT

Divide data sequence recursively till each data sub-sequence obtains its "**atomic**" representation

The sequence is said to be sorted if data is arranged in an ordered way OR **the length of sequence is exactly 1**

Merge Sort uses  $\leq N \lg N$  compares to sort an array of length  $N$

5	3	7	1	0	8	5
---	---	---	---	---	---	---

```
mergesort([5, 3, 7, 1, 0, 8, 5])
```





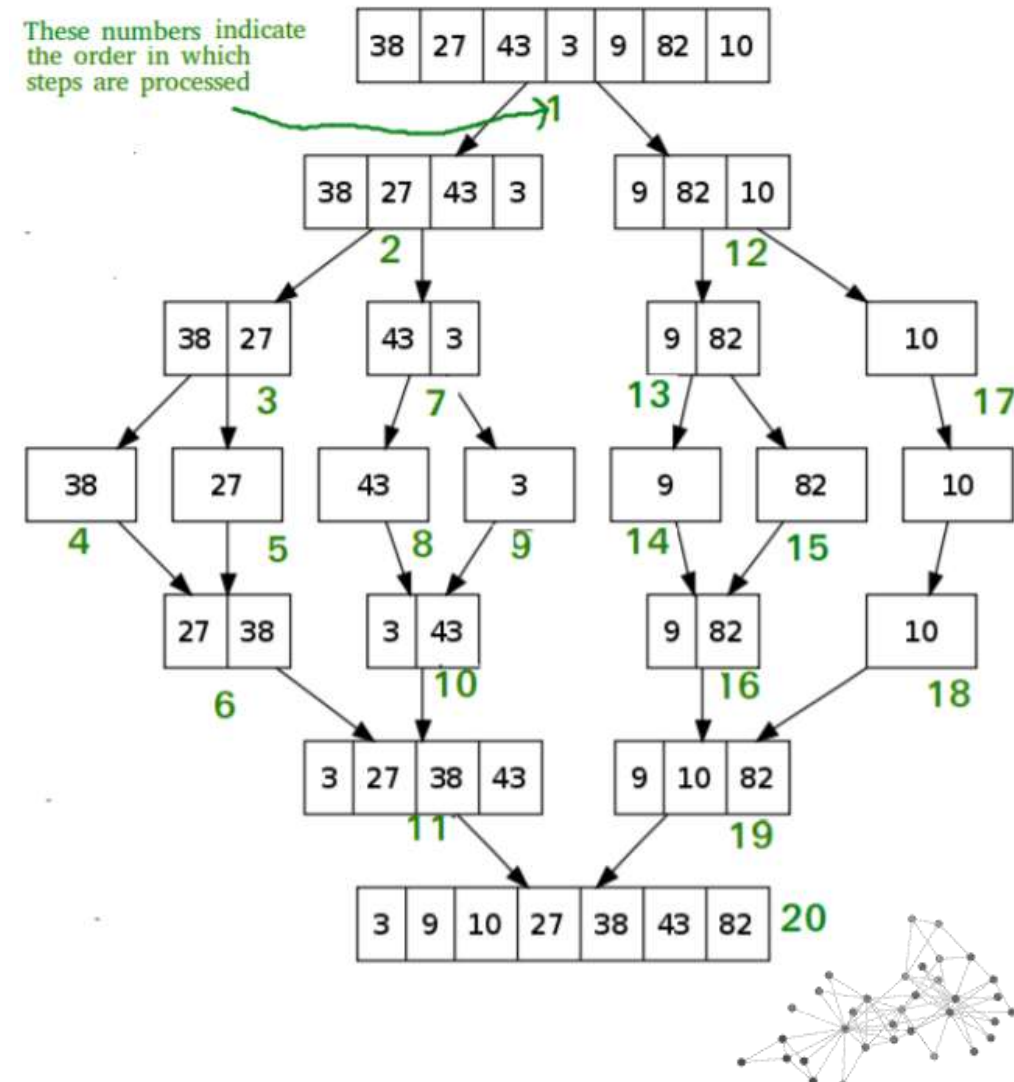
# MERGE SORT (CONTINUED)

The initial array length is 7

It is divided to two subarrays with lengths 4 and 3

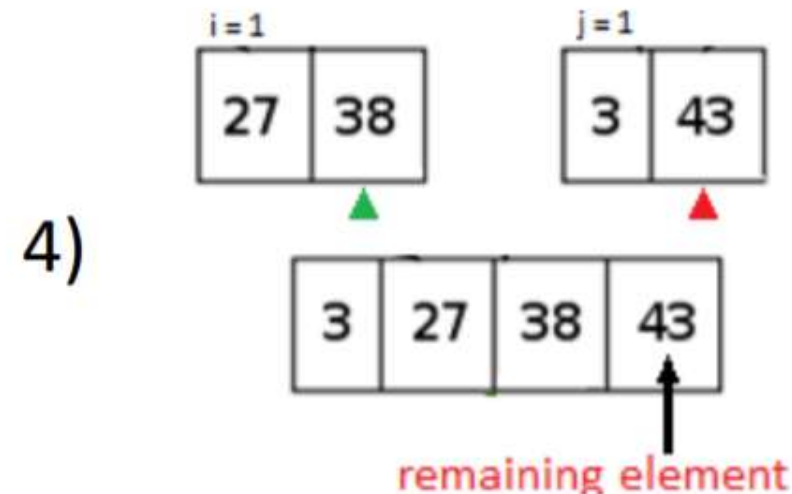
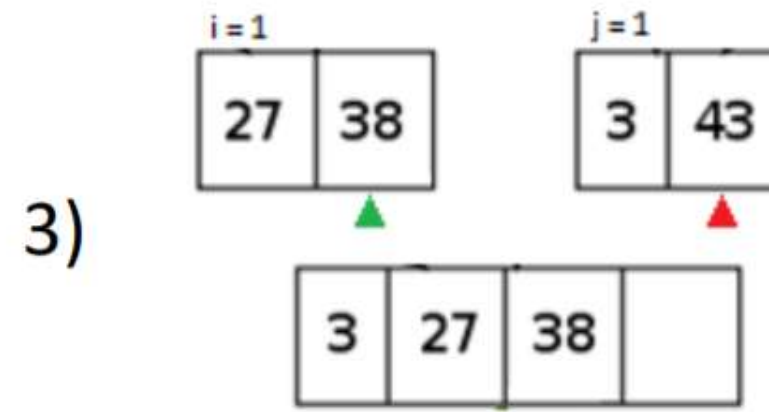
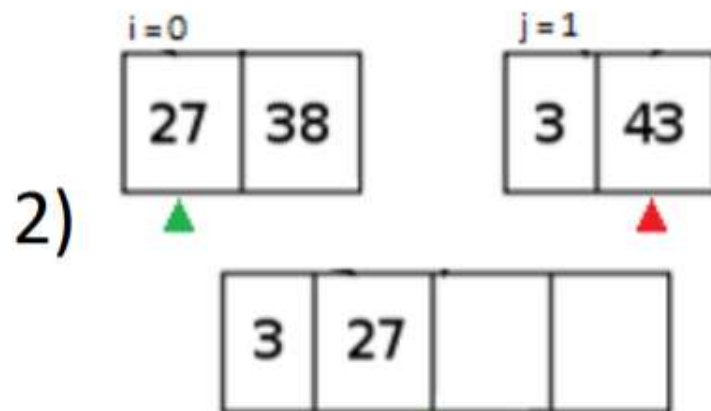
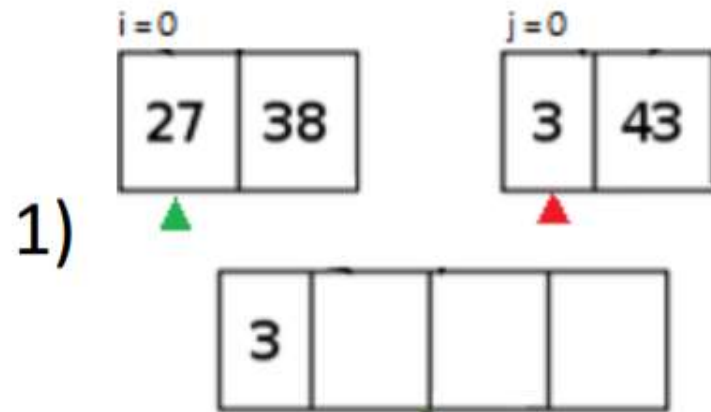
Division is repeated recursively till all subarrays reach an atomic view (one element in each array)

Since array that contains only one element is said to be sorted, merging process is started





# MERGING TWO SORTED SUBARRAYS INTO ONE



# SAMPLE CODE TO START

```
public static void sort(int[] x) {
    sort(x, 0, x.length-1);
}

private static void sort(int[] x, int start, int end) {
    if (start < end) {
        int middle = ???;
        sort(x, ???, ???);
        sort(x, ???, ???);

        merge(x, start, end, middle);
    }
}

public static void merge(int[] x, int start, int end, int middle) {

    int[] a = new int[???];
    int[] b = new int[???];

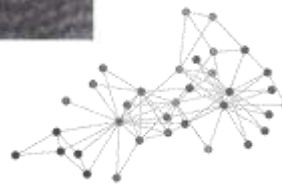
    // take a copy of a left half
    for (int j = 0; j <= ???; j++) {
        a[j] = x[???];
    }

    // take a copy of a right half
    for (int j = 0; j <= ???; j++) {
        b[j] = x[???];
    }
}
```



# SIR CHARLES ANTONY RICHARD HOARE

- Invented a Quick Sort algorithm in 1959/60. His sorting method is based on divide-and conquer algorithm



# QUICK SORT

Select a pivot point which could be:

- First element
- Last element
- Middle element in the sequence

5	4	2	1	3
---	---	---	---	---

Partition the other elements into two sub-arrays, according to whether they are less or greater than the pivot point

Generally Quick Sort uses  $\leq O(N \log N)$  compares to sort an array of length  $N$

Worst case:  $O(N^2)$  – when the selected pivot point is always the smallest or largest



# QUICK SORT (CONTINUED)

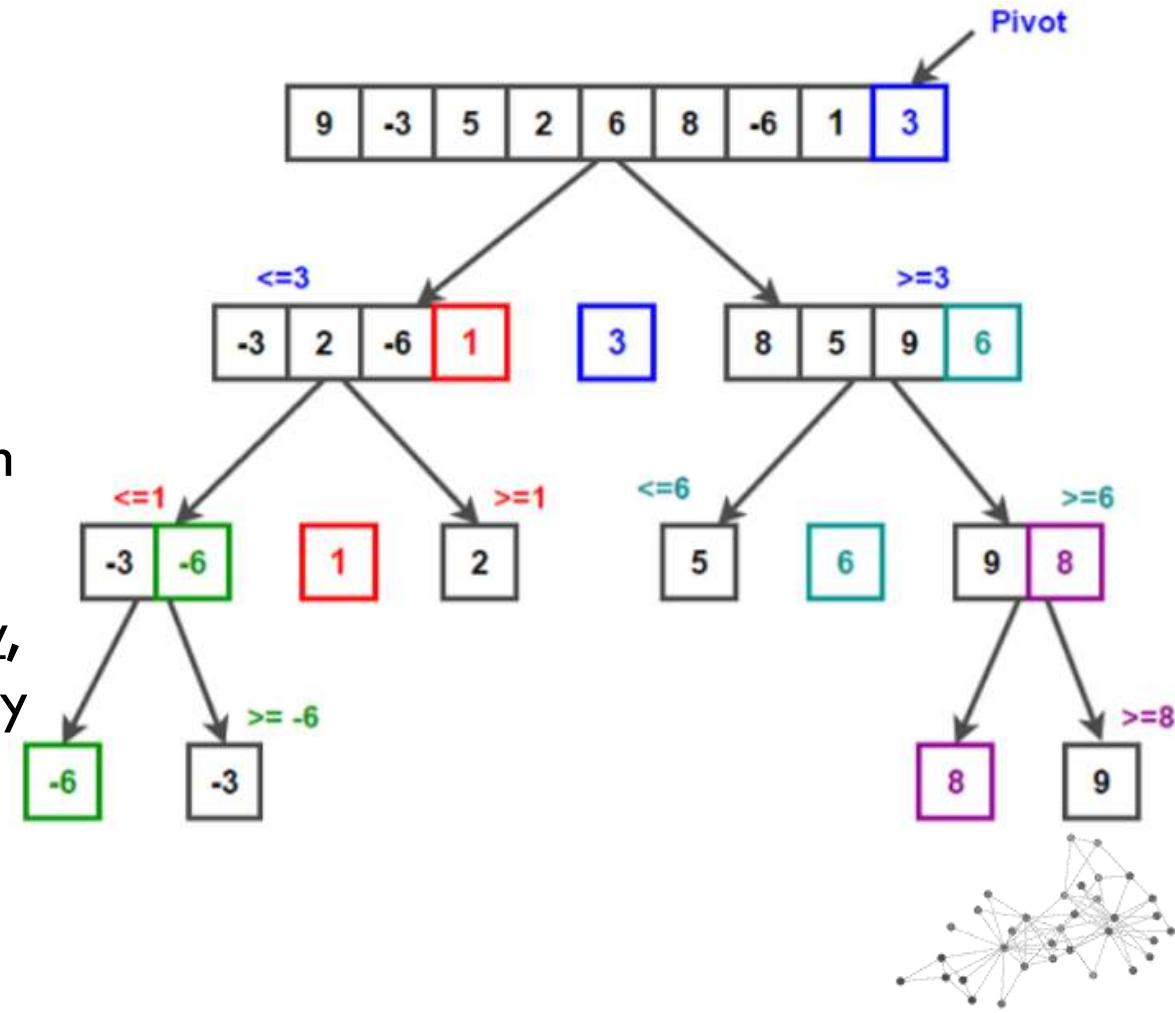
Select a pivot point (e.i. last element)

Partition to two sub-arrays (using indices)

- Less elements to the left
- Greater elements to the right

Repeat recursively till all subarrays reach an atomic view (one element in each array)

No need to create two sub-arrays physically,  
since swaps are applied to the original array  
by **manipulating indices**



# SAMPLE CODE TO START

```
public void quickSort(int[] x) {
    quickSort(x, 0, x.length - 1);
}

private void quickSort(int[] x, int start, int end)
{
    if (start < end)
    {
        int pi = partition(x, start, end);
        quickSort(x, ???, ???); // sort left half
        quickSort(x, ???, ???); // sort right half
    }
}

private int partition(int[] x, int start, int end)
{
    int pivot = x[???]; // select pivot point

    // Your code goes here

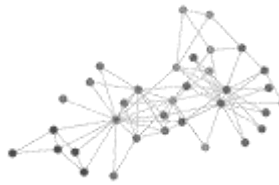
    return ???;
}
```



# USING COMPARABLE

Using array of comparable objects

```
public static void Sort(Comparable[] a) {  
    Comparable t;  
    for (int i = 0; i < a.length - 1; i++) {  
        for (int j = i+1; j < a.length; j++) {  
            if (a[i].compareTo(a[j]) > 0) {  
                t = a[i];  
                a[i] = a[j];  
                a[j] = t;  
            }  
        }  
    }  
}
```





# USING COMPARATOR

Instead of **Comparable**, one may use **Comparator** for convenience

Sort using an alternate order

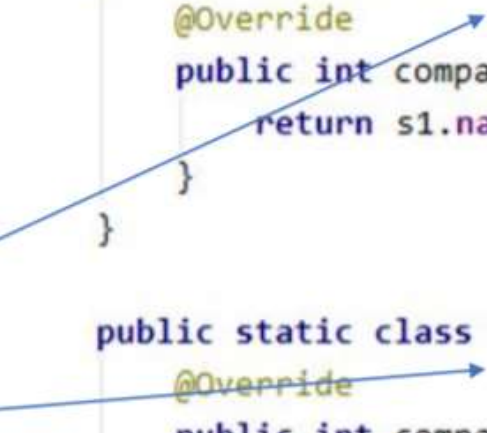
Firstly, define a (nested) class that implements the Comparator interface

```
import java.util.Comparator;

public class Student {
    public int ID;
    public String name;
    public Student(int ID, String name) {
        this.ID = ID;
        this.name = name;
    }
}

public static class ByName implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name);
    }
}

public static class ByID implements Comparator<Student>{
    @Override
    public int compare(Student s1, Student s2) {
        return s1.ID - s2.ID;
    }
}
}
```



# USING COMPARATOR

Use Comparator as second input parameter for sorting method

It affords to use a compare() method of Comparator object

Send a desired type of Comparator for Students as a second argument

```
public static void Sort(Object[] a, Comparator c) {  
    Object t;  
    for (int i = 0; i < a.length - 1; i++)  
        for (int j = i+1; j < a.length; j++)  
            if (c.compare(a[i], a[j]) > 0) {  
                t = a[i];  
                a[i] = a[j];  
                a[j] = t;  
            }  
}
```

```
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
    int n = sc.nextInt();  
  
    Student[] students = new Student[n];  
    for (int i = 0; i < students.length; i++) {  
        students[i] = new Student(sc.nextInt(), sc.next());  
    }  
    Sort(students, new Student.ByName());  
    for (int i = 0; i < students.length; i++) {  
        System.out.println(students[i]);  
    }  
    System.out.println();  
}
```



# LITERATURE

Algorithms, 4th Edition, by Robert Sedgewick and Kevin Wayne, Addison-Wesley

- Chapter 2.1-2.3

Grokking Algorithms, by Aditya Y. Bhargava, Manning

- Chapter 4



**GOOD LUCK!**

