

# Applied Programming

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

 armahmood786@yahoo.com


 alphapeeler.sf.net/pubkeys/pkey.htm

 pk.linkedin.com/in/armahmood

 www.twitter.com/alphapeeler

 www.facebook.com/alphapeeler

 abdulmahmood-sss  alphasecure

 armahmood786@hotmail.com

 <http://alphapeeler.sf.net/me>

 alphasecure@gmail.com

 <http://alphapeeler.sourceforge.net>

 <http://alphapeeler.tumblr.com>

 armahmood786@jabber.org

 alphapeeler@aim.com

 mahmood\_cubix  48660186

 alphapeeler@icloud.com

 <http://alphapeeler.sf.net/acms/>

# Data structures

Sorting (bubble, selection, insertion, shell, quick)

# Bubble sort

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

- **Example:**

**First Pass:**

( 5 1 4 2 8 )  $\rightarrow$  ( 1 5 4 2 8 ), Here, algorithm compares the first two elements, and swaps since  $5 > 1$ .

( 1 5 4 2 8 )  $\rightarrow$  ( 1 4 5 2 8 ), Swap since  $5 > 4$

( 1 4 5 2 8 )  $\rightarrow$  ( 1 4 2 5 8 ), Swap since  $5 > 2$

( 1 4 2 5 8 )  $\rightarrow$  ( 1 4 2 5 8 ), Now, since these elements are already in order ( $8 > 5$ ), algorithm does not swap them.

# Bubble sort

- **Second Pass:**

( 1 4 2 5 8 )  $\rightarrow$  ( 1 4 2 5 8 )

( 1 4 2 5 8 )  $\rightarrow$  ( 1 2 4 5 8 ), Swap since  $4 > 2$

( 1 2 4 5 8 )  $\rightarrow$  ( 1 2 4 5 8 )

( 1 2 4 5 8 )  $\rightarrow$  ( 1 2 4 5 8 )

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

# Bubble sort

- **Third Pass:**

( 1 2 4 5 8 )  $\rightarrow$  ( 1 2 4 5 8 )

( 1 2 4 5 8 )  $\rightarrow$  ( 1 2 4 5 8 )

( 1 2 4 5 8 )  $\rightarrow$  ( 1 2 4 5 8 )

( 1 2 4 5 8 )  $\rightarrow$  ( 1 2 4 5 8 )

# Ex01

```
public class BubbleSort01 {  
    void bubbleSort(int arr[]) {  
        int n = arr.length;  
        for (int i = 0; i < n-1; i++)  
            for (int j = 0; j < n-i-1; j++)  
                if (arr[j] > arr[j+1])  
                {  
                    // swap arr[j+1] and arr[i]  
                    int temp = arr[j];  
                    arr[j] = arr[j+1];  
                    arr[j+1] = temp;  
                }  
    }  
    /* Prints the array */  
    void printArray(int arr[]) {  
        int n = arr.length;  
        for (int i=0; i<n; ++i)  
            System.out.print(arr[i] + " ");  
        System.out.println();  
    }  
    public static void main(String[] args) {  
        BubbleSort01 ob = new BubbleSort01();  
        int arr[] = {5, 3, 1, 9, 8, 2, 4,7};  
        ob.bubbleSort(arr);  
        System.out.println("Sorted array");  
        ob.printArray(arr);  
    }  
}
```

**Output:**  
Sorted array  
1 2 3 4 5 7 8 9

# Ex 01 - Illustration

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
		1	2						

# Optimized Implementation

- The above function always runs  $O(n^2)$  time even if the array is sorted. It can be optimized by stopping the algorithm if inner loop didn't cause any swap.



# Ex02

```
public class BubbleSort02 {
    static void bubbleSort(int arr[], int n) {
        int i, j, temp;
        boolean swapped;
        for (i = 0; i < n - 1; i++) {
            swapped = false;
            for (j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // swap arr[j] and arr[j+1]
                    temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }
            // IF no two elements were swapped by inner loop, then break
            if (swapped == false)
                break;
        }
    }

    static void printArray(int arr[], int size) {
        int i;
        for (i = 0; i < size; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        int arr[] = { 5, 3, 1, 9, 8, 2, 4, 7 };
        int n = arr.length;
        bubbleSort(arr, n);
        System.out.println("Sorted array: ");
        printArray(arr, n);
    }
}
```

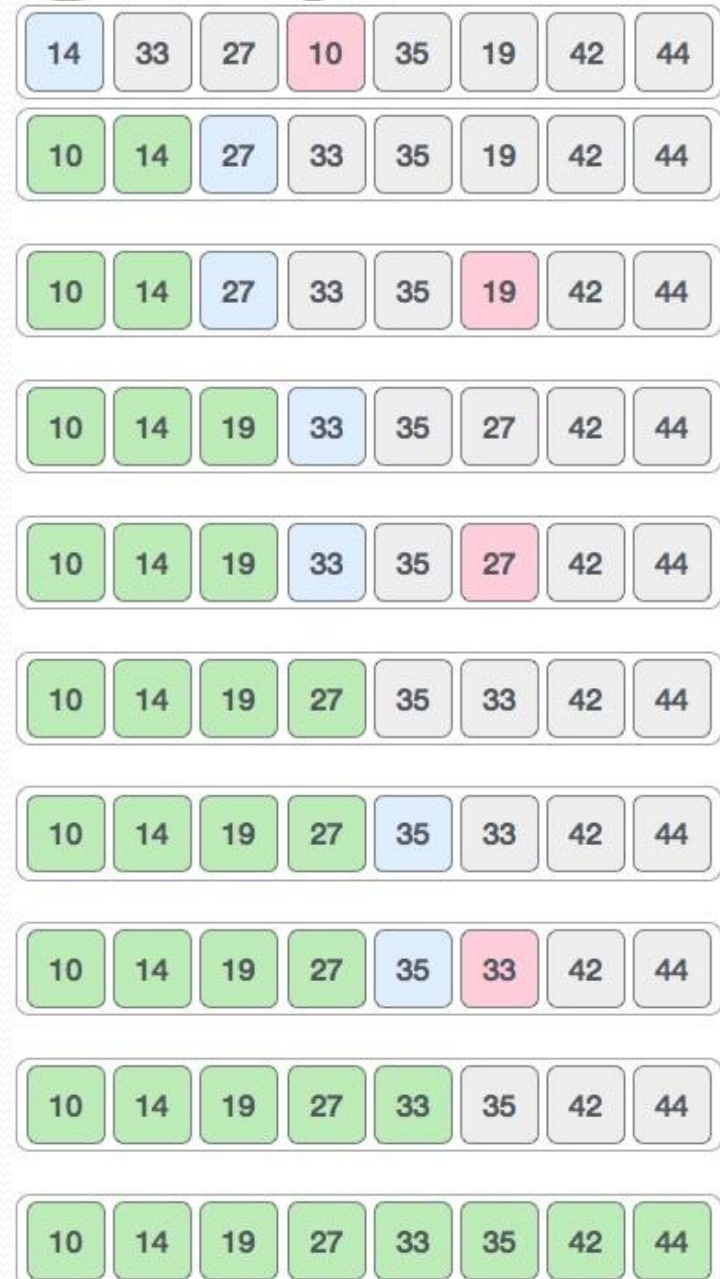
**Output:**  
Sorted array  
1 2 3 4 5 7 8 9

# Selection Sort

- Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
- The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

# How Selection Sort Works?

- The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value. So we replace 14 with 10.
- Next, We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.
- And so on.



# Ex 03

```
public class SelectionSort {
    void sort(int arr[]) {
        int n = arr.length;
        // One by one move boundary of unsorted subarray
        for (int i = 0; i < n-1; i++) {
            // Find the minimum element in unsorted array
            int min_idx = i;
            for (int j = i+1; j < n; j++)
                if (arr[j] < arr[min_idx])
                    min_idx = j;
            // Swap the found minimum element with the first element
            int temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
    }
    void printArray(int arr[]) {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i]+" ");
        System.out.println();
    }
    public static void main(String args[]) {
        SelectionSort ob = new SelectionSort();
        int arr[] = {64,25,12,22,11};
        ob.sort(arr);
        System.out.println("Sorted array");
        ob.printArray(arr);
    }
}
```

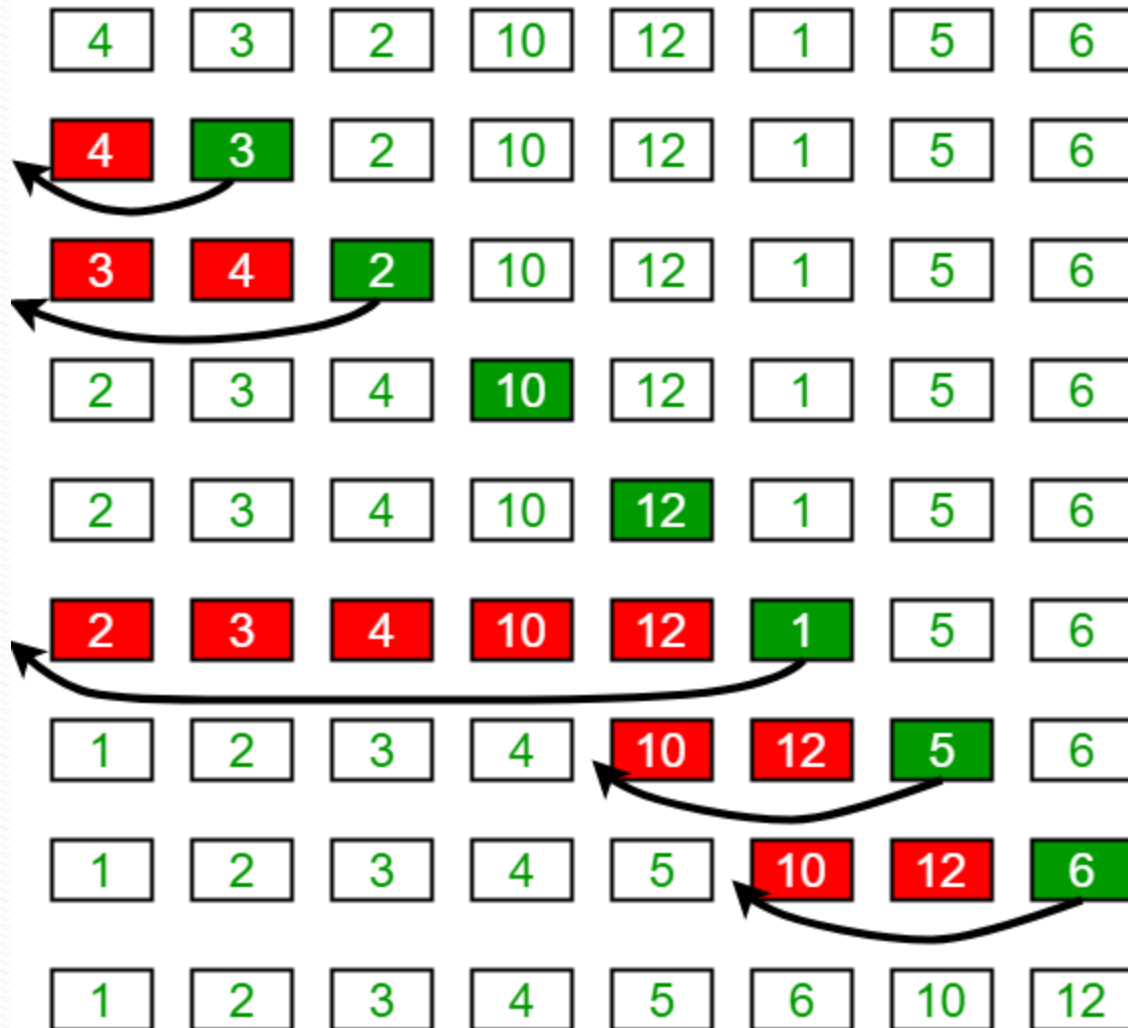
**Output:**  
Sorted array  
11 12 22 25 64

# Insertion Sort

- This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.
- **Algorithm**  
insertionSort(arr, n)  
Loop from  $i = 1$  to  $n-1$ .  
.....a) Pick element  $arr[i]$  and insert it into sorted sequence  $arr[0..i-1]$

# How Insertion Sort Works?

## Insertion Sort Execution Example



# Another Example

- 12, 11, 13, 5, 6
- loop for  $i = 1$  (2<sup>nd</sup> element of array) to 4 (last element of array)
- $i = 1$ . Since 11 is smaller than 12, move 12 and insert 11 before 12  
11, 12, 13, 5, 6
- $i = 2$ . 13 will remain at its position as all elements in  $A[0..i-1] < 13$   
11, 12, 13, 5, 6
- $i = 3$ . 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.  
5, 11, 12, 13, 6
- $i = 4$ . 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.  
5, 6, 11, 12, 13

# Ex 04

```
public class InsertionSort {
void sort(int arr[]) {
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;
        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

static void printArray(int arr[]) {
    int n = arr.length;
    for (int i = 0; i < n; ++i)
        System.out.print(arr[i] + " ");
    System.out.println();
}

public static void main(String args[]) {
    int arr[] = { 12, 11, 13, 5, 6 };
    InsertionSort ob = new InsertionSort();
    ob.sort(arr);
    printArray(arr);
}
}
```

**Output:**  
5 6 11 12 13



# Shell Sort

- ShellSort is mainly a variation of Insertion Sort. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shellSort is to allow exchange of far items. In shellSort, we make the array  $h$ -sorted for a large value of  $h$ . We keep reducing the value of  $h$  until it becomes 1. An array is said to be  $h$ -sorted if all sublists of every  $h$ 'th element is sorted.

# How Shell Sort Works?



Temp

Start with gap =  $n/2$  (2 in this case)

One by one select elements to the right of gap and place them at their appropriate position.

# How Shell Sort Works?



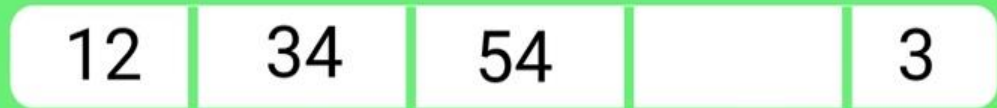
54

Temp

Elements left of 54 are already smaller, so no change.

One by one select elements to the right of gap and place them at their appropriate position.

# How Shell Sort Works?



Temp

Compare 2 with  $\text{arr}[3-2] = 34$  and shift it to  $\text{arr}[\text{gap}+1 = 3]$ .

# How Shell Sort Works?



Compare 2 with  $\text{arr}[3-2] = 34$  and shift it to  $\text{arr}[\text{gap}+1 = 3]$ .

# How Shell Sort Works?



2

Temp

Since  $3 > 2$

Now gap reduces to  $1(n/4)$ .

Select all elements starting from `arr[ 1 ]` and compare them with elements within the distance of gap.

# How Shell Sort Works?

2

3

12

34

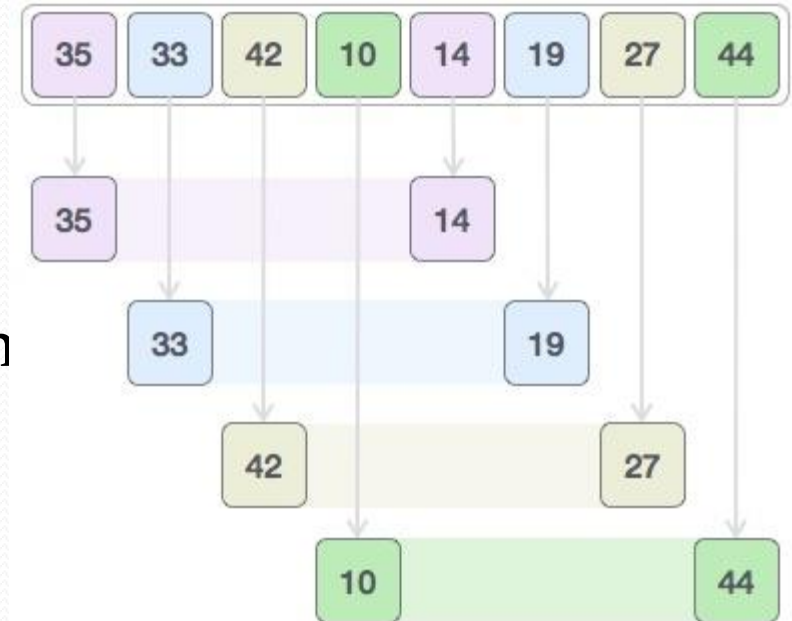
54

Now gap reduces to 0

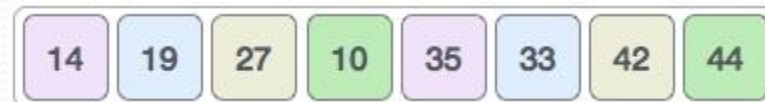
Sorting stops and array is sorted.

# How Shell Sort Works?

- 1. For ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



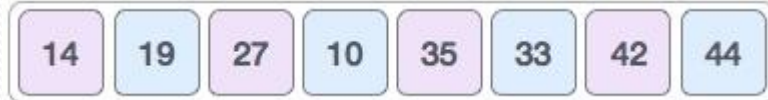
- 2. We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this :





# How Shell Sort Works?

- Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



- Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.
- Following is the step-by-step depiction –

# How Shell Sort Works?

14	19	27	10	35	33	42	44
----	----	----	----	----	----	----	----

14	19	27	10	35	33	42	44
----	----	----	----	----	----	----	----

14	19	27	10	35	33	42	44
----	----	----	----	----	----	----	----

14	19	27	10	35	33	42	44
----	----	----	----	----	----	----	----

14	19	10	27	35	33	42	44
----	----	----	----	----	----	----	----

14	10	19	27	35	33	42	44
----	----	----	----	----	----	----	----

10	14	19	27	35	33	42	44
----	----	----	----	----	----	----	----

10	14	19	27	35	33	42	44
----	----	----	----	----	----	----	----

10	14	19	27	33	35	42	44
----	----	----	----	----	----	----	----

10	14	19	27	33	35	42	44
----	----	----	----	----	----	----	----

# Ex 05

```
public class shellSort {
    int sort(int arr[]) {
        int n = arr.length;
        // Start with a big gap, then reduce the gap
        for (int gap = n/2; gap > 0; gap /= 2) {
            // Do a gapped insertion sort for this gap size.
            // The first gap elements a[0..gap-1] are already
            // in gapped order keep adding one more element
            // until the entire array is gap sorted
            for (int i = gap; i < n; i += 1) {
                // add a[i] to the elements that have been gap
                // sorted save a[i] in temp and make a hole at
                // position i
                int temp = arr[i];
                // shift earlier gap-sorted elements up until
                // the correct location for a[i] is found
                int j;
                for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                    arr[j] = arr[j - gap];
                // put temp (the original a[i]) in its correct
                // location
                arr[j] = temp;
            }
        }
        return 0;
    }
}
```

# Ex 05

```
static void printArray(int arr[]) {  
    int n = arr.length;  
    for (int i=0; i<n; ++i)  
        System.out.print(arr[i] + " ");  
    System.out.println();  
}  
public static void main(String args[]) {  
    int arr[] = {12, 34, 54, 2, 3};  
    System.out.println("Array before sorting");  
    printArray(arr);  
    shellSort ob = new shellSort();  
    ob.sort(arr);  
    System.out.println("Array after sorting");  
    printArray(arr);  
}  
}
```

**Output:**  
5 6 11 12 13

# Video tutorial – shell sort

- <https://www.youtube.com/watch?v=SHcPqUe2GZM>

# Quick Sort

- It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.
  - Always pick first element as pivot.
  - Always pick last element as pivot (implemented below)
  - Pick a random element as pivot.
  - Pick median as pivot.

# How Quick Sort Works?

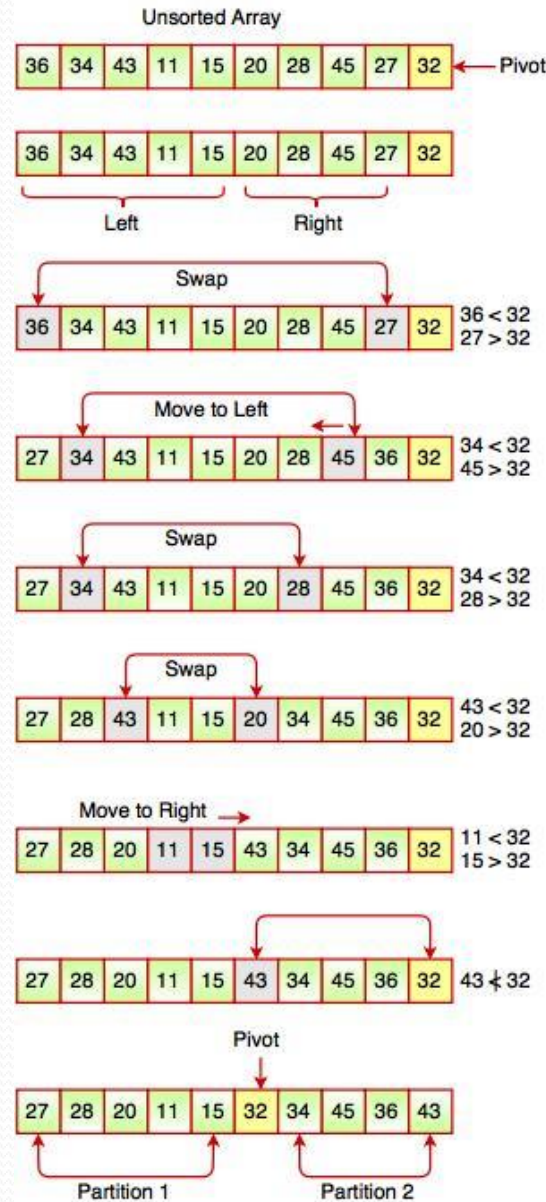


Fig. Finding Pivot Value in an Array

# Ex 06

```
public class QuickSort {  
    /* takes last element as pivot, places the pivot at its correct  
    position in sorted array, & places all smaller (than pivot) to left of  
    pivot and all greater elements to right of pivot */  
    int partition(int arr[], int low, int high) {  
        int pivot = arr[high];  
        int i = (low-1); // index of smaller element  
        for (int j=low; j<high; j++) {  
            // If current element is smaller than the pivot  
            if (arr[j] < pivot) {  
                i++;  
                // swap arr[i] and arr[j]  
                int temp = arr[i];  
                arr[i] = arr[j];  
                arr[j] = temp;  
            }  
        }  
        // swap arr[i+1] and arr[high] (or pivot)  
        int temp = arr[i+1];  
        arr[i+1] = arr[high];  
        arr[high] = temp;  
        return i+1;  
    }  
}
```



# Ex 06

```
/* The main function that implements QuickSort()
   arr[] --> Array to be sorted, low --> Starting index, high --> Ending index */
void sort(int arr[], int low, int high) {
    if (low < high) {
        /* pi is partitioning index, arr[pi] is
           now at right place */
        int pi = partition(arr, low, high);
        // Recursively sort elements before
        // partition and after partition
        sort(arr, low, pi-1);
        sort(arr, pi+1, high);
    }
}

static void printArray(int arr[]) {
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i]+" ");
    System.out.println();
}

public static void main(String args[]) {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = arr.length;
    QuickSort ob = new QuickSort();
    ob.sort(arr, 0, n-1);
    System.out.println("sorted array");
    printArray(arr);
}
}
```

**Output:**  
sorted array  
1 5 7 8 9 10

# Quick sort video tutorials

- Taking the last element as pivot:
- <https://www.youtube.com/watch?v=PgBzjlCcFvc>
- Taking the pivot randomly:
- <https://www.youtube.com/watch?v=SLauY6PpjW4>
- Taking the first element as pivot:
- <https://www.youtube.com/watch?v=3OLTJlwylqQ>