Animation is Here: http://www.cs.armstrong.edu/liang/animation/animation.html

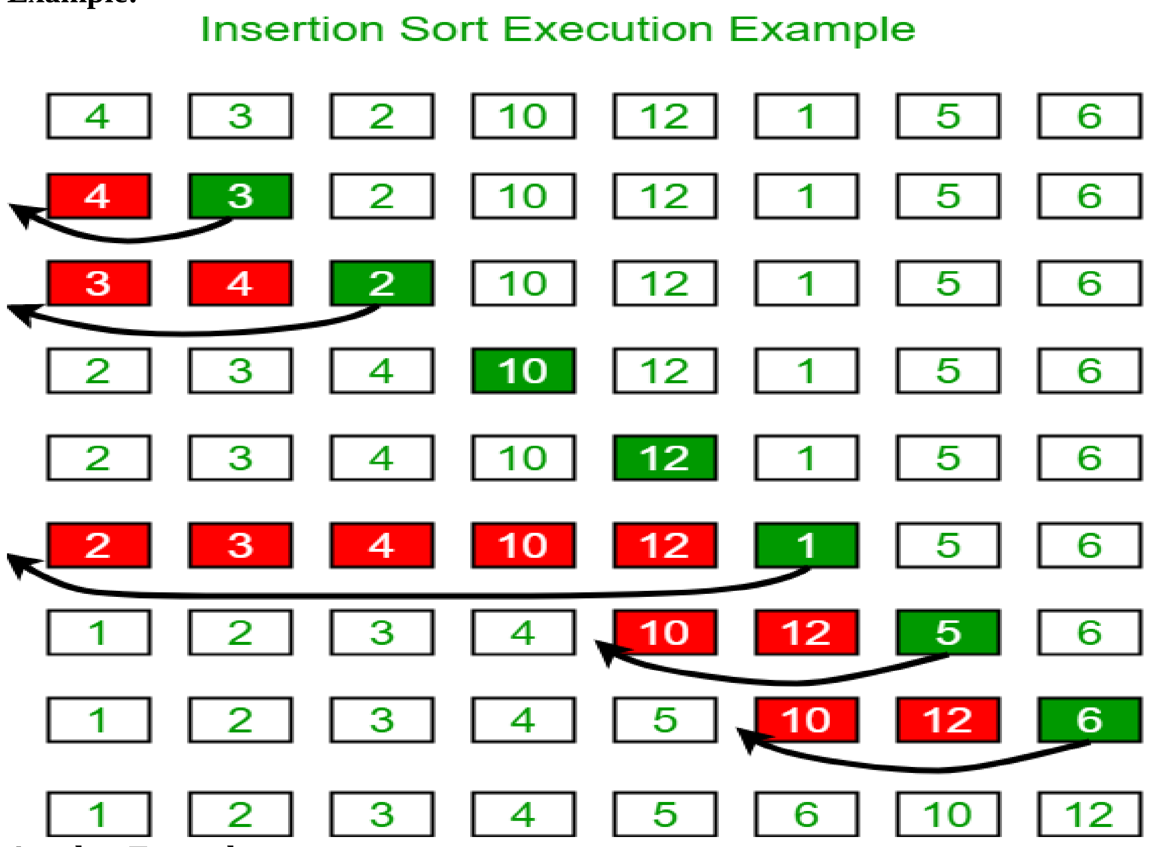
Insertion Sort:

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

Algorithm

```
// Sort an arr[] of size n
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]
```

Example:



Another Example:
12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 5 (Size of input 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] are smaller than 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

Example:

```
// Java program for implementation of Insertion Sort

class InsertionSort
{
    /*Function to sort array using insertion sort*/
    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;
        }
    }

    /* A utility function to print array of size n*/
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");

        System.out.println();
    }

    // Driver method
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6};

        InsertionSort ob = new InsertionSort();
        ob.sort(arr);
        printArray(arr);
    }
}
```

Time Complexity: O(n*2)

Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes

Stable: Yes

Online: Yes

Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

================================xxxxxxxxxxxxxxx================================

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) –> (**1** **5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5** **4** 2 8) –> (1 **4** **5** 2 8), Swap since 5 > 4
(1 4 **5** **2** 8) –> (1 4 **2** **5** 8), Swap since 5 > 2
(1 4 2 **5** **8**) –> (1 4 2 **5** **8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(**1** **4** 2 5 8) –> (**1** **4** 2 5 8)
(1 **4** **2** 5 8) –> (1 **2** **4** 5 8), Swap since 4 > 2
(1 2 **4** **5** 8) –> (1 2 **4** **5** 8)

(1 2 4 **5 8**) –> (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.

Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Example:

```
// Java program for implementation of Bubble Sort

class BubbleSort

{

        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();
        }

        // Driver method to test above
        public static void main(String args[])
        {
            BubbleSort ob = new BubbleSort();
            int arr[] = {64, 34, 25, 12, 22, 11, 90};
            ob.bubbleSort(arr);
            System.out.println("Sorted array");
            ob.printArray(arr);
        }
}
```

===================================xxxxxxxxxxxxxxxxxxxxxxxxxxx=======================

Merge Sort

Like

Merge Sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

```
MergeSort(arr[], l,  r)

If r > l
```

1. Find the middle point to divide the array into two halves:

    middle m = (l+r)/2

2. Call mergeSort for first half:

    Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

    Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

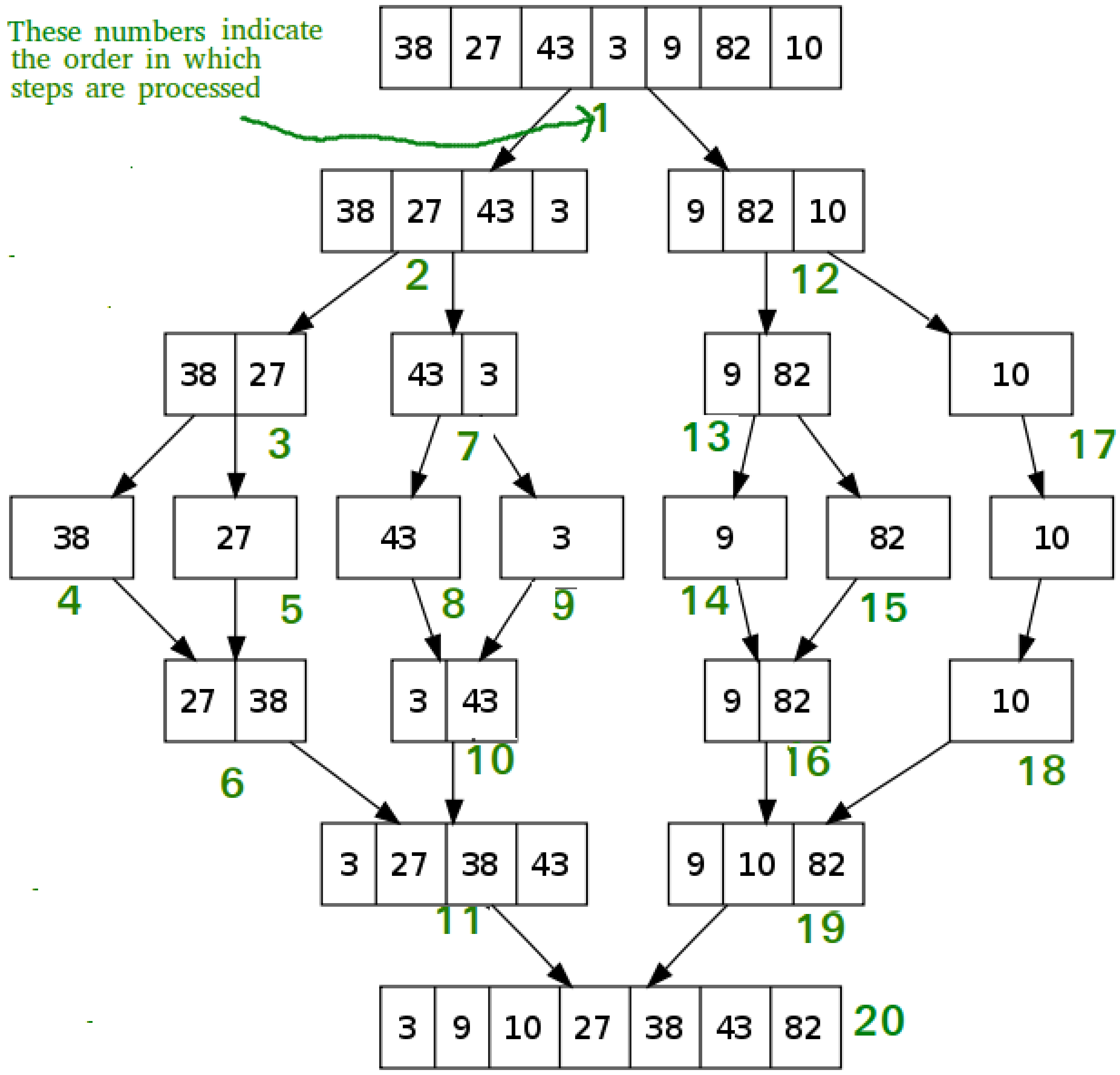
    Call merge(arr, l, m, r)

The following diagram from wikipedia shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.

Merge-Sort-Tutorial, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

The following diagram from wikipedia shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



Example:

```
/* Java program for Merge Sort */

class MergeSort

{

        // Merges two subarrays of arr[].

        // First subarray is arr[l..m]

        // Second subarray is arr[m+1..r]

        void merge(int arr[], int l, int m, int r)
```

```
{
        // Find sizes of two subarrays to be merged
        int n1 = m - l + 1;
        int n2 = r - m;

        /* Create temp arrays */
        int L[] = new int [n1];
        int R[] = new int [n2];

        /*Copy data to temp arrays*/
        for (int i=0; i<n1; ++i)
                L[i] = arr[l + i];
        for (int j=0; j<n2; ++j)
                R[j] = arr[m + 1+ j];



        /* Merge the temp arrays */

        // Initial indexes of first and second subarrays
        int i = 0, j = 0;

        // Initial index of merged subarry array
        int k = l;
        while (i < n1 && j < n2)
        {
                if (L[i] <= R[j])
                {
                        arr[k] = L[i];
                        i++;
                }
                else
                {
                        arr[k] = R[j];
                        j++;
                }
                k++;
        }

        /* Copy remaining elements of L[] if any */
        while (i < n1)
        {
                arr[k] = L[i];
                i++;
                k++;
        }

        /* Copy remaining elements of R[] if any */
        while (j < n2)
        {
                arr[k] = R[j];
                j++;
                k++;
        }
}

// Main function that sorts arr[l..r] using
// merge()
void sort(int arr[], int l, int r)
```

```
{
        if (l < r)
        {
                // Find the middle point
                int m = (l+r)/2;

                // Sort first and second halves
                sort(arr, l, m);
                sort(arr , m+1, r);

                // Merge the sorted halves
                merge(arr, l, m, r);
        }
}

/* A utility function to print array of size n */
static void printArray(int arr[])
{
        int n = arr.length;
        for (int i=0; i<n; ++i)
                System.out.print(arr[i] + " ");
        System.out.println();
}

// Driver method
public static void main(String args[])
{
        int arr[] = {12, 11, 13, 5, 6, 7};
```

```
                System.out.println("Given Array");

                printArray(arr);


                MergeSort ob = new MergeSort();

                ob.sort(arr, 0, arr.length-1);


                System.out.println("\nSorted array");

                printArray(arr);

        }

}
```

QuickSort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. 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.

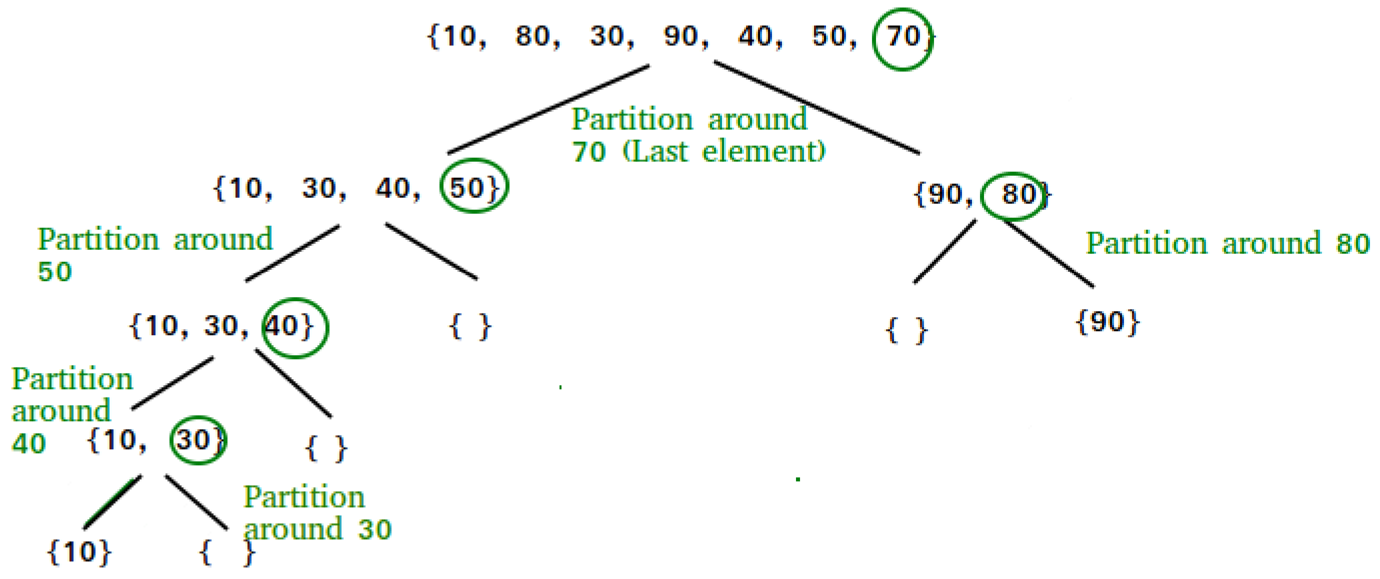
1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```



Partition Algorithm

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Illustration of partition() :

```
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0   1   2   3   4   5   6

low = 0, high =  6, pivot = arr[h] = 70
Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping
arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than
```

70 are before it and all elements greater than 70 are after it.

Example:

```
// Java program for implementation of QuickSort
class QuickSort
{
        /* This function takes last element as pivot,
        places the pivot element at its correct
        position in sorted array, and places all
        smaller (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 or
                        // equal to 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;
        }


        /* 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);
```

```
            }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
            int n = arr.length;
            for (int i=0; i<n; ++i)
                    System.out.print(arr[i]+" ");
            System.out.println();
    }

    // Driver program
    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);
    }
}
```
======================================xxxxxxxxxxxxxxxxx=====================

HeapSort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

What is Binary Heap?
Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source Wikipedia)

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Why array based representation for Binary Heap?
Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting in increasing order:
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

How to build the heap?
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Lets understand with the help of an example:

```
Input data: 4, 10, 3, 5, 1
         4(0)
        /   \
     10(1)   3(2)
    /   \
 5(3)    1(4)

The numbers in bracket represent the indices in the array
representation of data.

Applying heapify procedure to index 1:
         4(0)
        /   \
    10(1)    3(2)
    /   \
5(3)    1(4)

Applying heapify procedure to index 0:
        10(0)
        /  \
     5(1)  3(2)
    /   \
 4(3)    1(4)
The heapify procedure calls itself recursively to build heap
 in top down manner.
```

Example:

```
// Java program for implementation of Heap Sort

public class HeapSort

{

      public void sort(int arr[])

      {

            int n = arr.length;
```

```
        // Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
                heapify(arr, n, i);

        // One by one extract an element from heap
        for (int i=n-1; i>=0; i--)
        {
                // Move current root to end
                int temp = arr[0];
                arr[0] = arr[i];
                arr[i] = temp;

                // call max heapify on the reduced heap
                heapify(arr, i, 0);
        }
}

// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
        int largest = i; // Initialize largest as root
        int l = 2*i + 1; // left = 2*i + 1
        int r = 2*i + 2; // right = 2*i + 2

        // If left child is larger than root
        if (l < n && arr[l] > arr[largest])
                largest = l;
        // If right child is larger than largest so far
        if (r < n && arr[r] > arr[largest])
            largest = r;

        // If largest is not root
        if (largest != i)
        {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

            // Recursively heapify the affected sub-tree
            heapify(arr, n, largest);
        }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i]+" ");
        System.out.println();
    }

    // Driver program
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6, 7};
        int n = arr.length;


        HeapSort ob = new HeapSort();

        ob.sort(arr);


        System.out.println("Sorted array is");

        printArray(arr);

    }

}
```

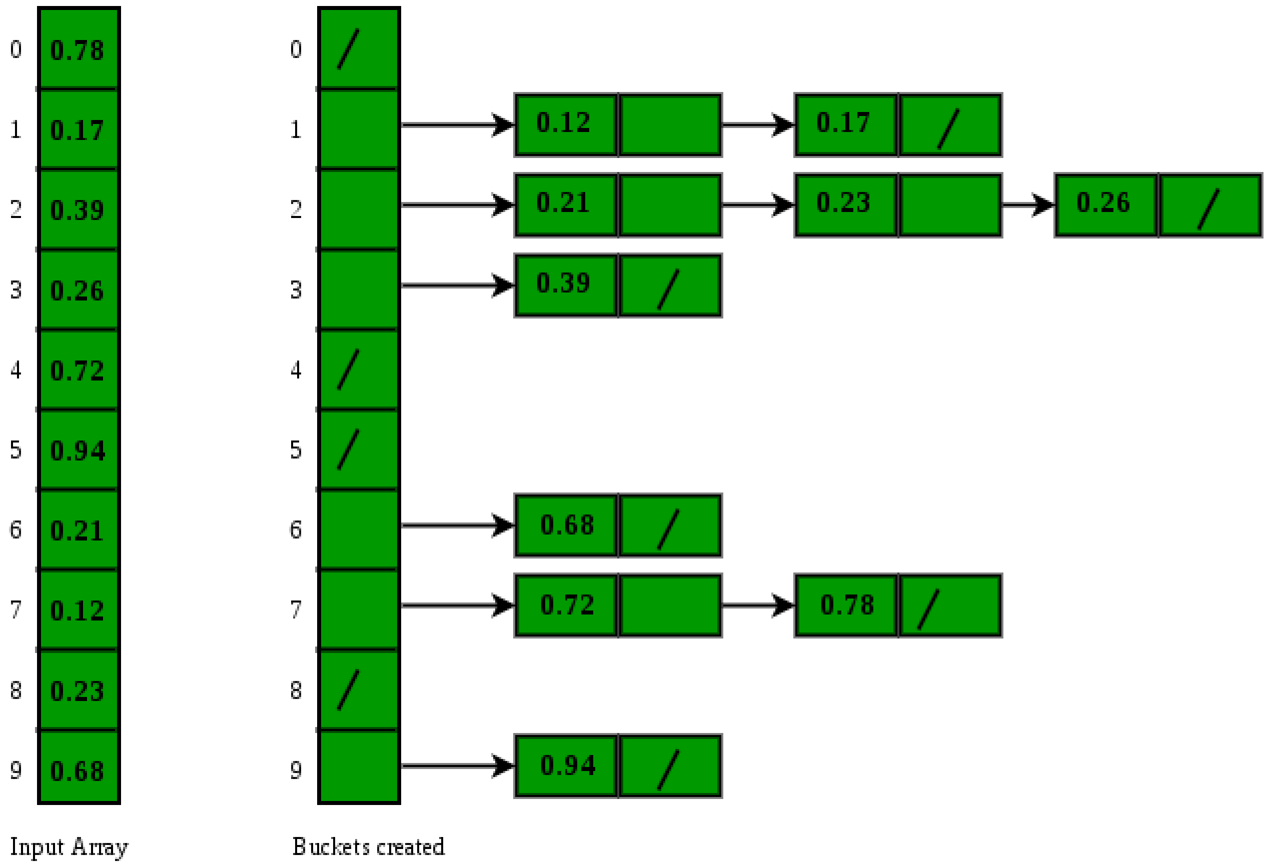
================================xxxxxxxxxxxxxxxxxxxxxxxx===============================

Bucket Sort

Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.
Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?

A simple way is to apply a comparison based sorting algorithm. The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(n Log n), i.e., they cannot do better than nLogn.
Can we sort the array in linear time? Counting sort can not be applied here as we use keys as index in counting sort. Here keys are floating point numbers.
The idea is to use bucket sort. Following is bucket algorithm.

```
bucketSort(arr[], n)
1) Create n empty buckets (Or lists).
2) Do following for every array element arr[i].
.......a) Insert arr[i] into bucket[n*array[i]]
3) Sort individual buckets using insertion sort.
4) Concatenate all sorted buckets.
```



Time Complexity: If we assume that insertion in a bucket takes O(1) time then steps 1 and 2 of the above algorithm clearly take O(n) time. The O(1) is easily possible if we use a linked list to represent a bucket (In the following code, C++ vector is used for simplicity). Step 4 also takes O(n) time as there will be n items in all buckets.
The main step to analyze is step 3. This step also takes O(n) time on average if all numbers are uniformly distributed (please refer CLRS book for more details)