Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.  
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]

// and place it at beginning

**11** 25 12 22 64

// Find the minimum element in arr[1...4]

// and place it at beginning of arr[1...4]

11 **12** 25 22 64

// Find the minimum element in arr[2...4]

// and place it at beginning of arr[2...4]

11 12 **22** 25 64

// Find the minimum element in arr[3...4]

// and place it at beginning of arr[3...4]

11 12 22 **25** 64

Program:

|  |
| --- |
| // Java program for implementation of Selection Sort  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;          }      }        // 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 code to test above      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);      }  }  /\* This code is contributed by Rajat Mishra\*/ |

Run on IDE

Output:

Sorted array:

11 12 22 25 64

**Time Complexity:** O(n2) as there are two nested loops.

**Auxiliary Space:** O(1)  
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

# 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** )

## [Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/sort-the-array/0)

Following are C/C++, Python and Java implementations of Bubble Sort.

* C/C++
* Java
* Python

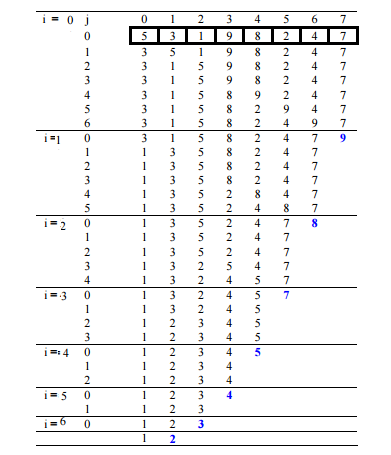
|  |
| --- |
| // C program for implementation of Bubble sort  #include <stdio.h>    void swap(int \*xp, int \*yp)  {      int temp = \*xp;      \*xp = \*yp;      \*yp = temp;  }    // A function to implement bubble sort  void bubbleSort(int arr[], int n)  {     int i, j;     for (i = 0; i < n-1; i++)           // Last i elements are already in place         for (j = 0; j < n-i-1; j++)             if (arr[j] > arr[j+1])                swap(&arr[j], &arr[j+1]);  }    /\* Function to print an array \*/  void printArray(int arr[], int size)  {      int i;      for (i=0; i < size; i++)          printf("%d ", arr[i]);      printf("n");  }    // Driver program to test above functions  int main()  {      int arr[] = {64, 34, 25, 12, 22, 11, 90};      int n = sizeof(arr)/sizeof(arr[0]);      bubbleSort(arr, n);      printf("Sorted array: \n");      printArray(arr, n);      return 0;  } |

Run on IDE

Output:

Sorted array:

11 12 22 25 34 64 90

**Illustration :**  
[](http://www.geeksforgeeks.org/wp-content/uploads/gq/2014/02/bubble-sort1.png)

**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.

|  |
| --- |
| // Optimized implementation of Bubble sort  #include <stdio.h>    void swap(int \*xp, int \*yp)  {      int temp = \*xp;      \*xp = \*yp;      \*yp = temp;  }    // An optimized version of Bubble Sort  void bubbleSort(int arr[], int n)  {     int i, j;     bool 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], &arr[j+1]);             swapped = true;          }       }         // IF no two elements were swapped by inner loop, then break       if (swapped == false)          break;     }  }    /\* Function to print an array \*/  void printArray(int arr[], int size)  {      int i;      for (i=0; i < size; i++)          printf("%d ", arr[i]);      printf("n");  }    // Driver program to test above functions  int main()  {      int arr[] = {64, 34, 25, 12, 22, 11, 90};      int n = sizeof(arr)/sizeof(arr[0]);      bubbleSort(arr, n);      printf("Sorted array: \n");      printArray(arr, n);      return 0;  } |

Run on IDE

Output:

Sorted array:

11 12 22 25 34 64 90

**Worst and Average Case Time Complexity:**O(n\*n). Worst case occurs when array is reverse sorted.

**Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.

**Auxiliary Space:** O(1)

**Boundary Cases:** Bubble sort takes minimum time (Order of n) when elements are already sorted.

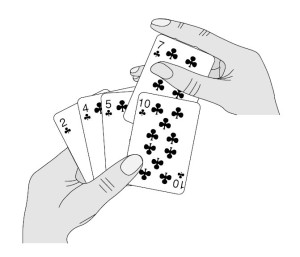
**Sorting In Place:**Yes

**Stable:** Yes

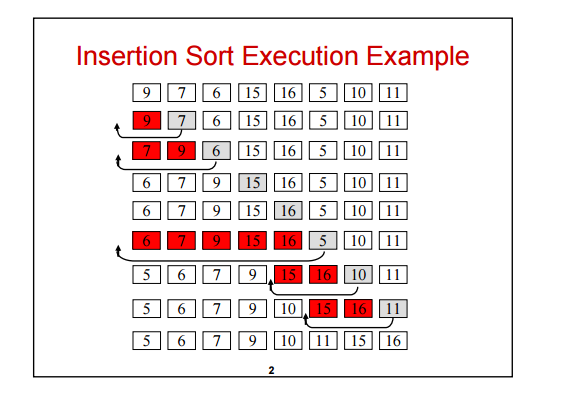
Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm.  
In computer graphics it is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n). For example, it is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to x axis) and with incrementing y their order changes (two elements are swapped) only at intersections of two lines (Source: [Wikipedia](http://en.wikipedia.org/wiki/Bubble_sort#In_practice))

# Insertion Sort

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

[](http://www.geeksforgeeks.org/wp-content/uploads/gq/2013/03/Insertion-Sort.jpg)

**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:**  
[](http://www.geeksforgeeks.org/wp-content/uploads/gq/2013/03/insertion-sort.png)  
**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**

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/probfunc-page.php?pid=700148)

* C/C++
* Python
* Java

|  |
| --- |
| // 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);      }  } /\* This code is contributed by Rajat Mishra. \*/ |

Run on IDE

Output:

5 6 11 12 13

**Time Complexity:** O(n\*n)

**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.

**What is Binary Insertion Sort?**  
We can use binary search to reduce the number of comparisons in normal insertion sort. Binary Insertion Sort find use binary search to find the proper location to insert the selected item at each iteration. In normal insertion, sort it takes O(i) (at ith iteration) in worst case. we can reduce it to O(logi) by using binary search. The algorithm as a whole still has a running worst case running time of O(n2) because of the series of swaps required for each insertion. Refer [this](http://quiz.geeksforgeeks.org/binary-insertion-sort/) for implementation.

**How to implement Insertion Sort for Linked List?**  
Below is simple insertion sort algorithm for linked list.

1) Create an empty sorted (or result) list

2) Traverse the given list, do following for every node.

......a) Insert current node in sorted way in sorted or result list.

3) Change head of given linked list to head of sorted (or result) list.

# Merge Sort

Like [QuickSort](http://quiz.geeksforgeeks.org/quick-sort/), Merge Sort is a [Divide and Conquer](http://www.geeksforgeeks.org/divide-and-conquer-set-1-find-closest-pair-of-points/) 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)

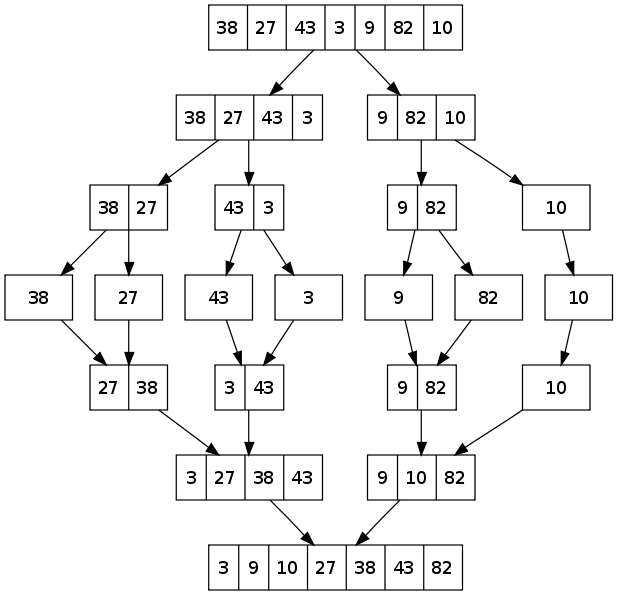
**3.** Call mergeSort for second half:

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](http://en.wikipedia.org/wiki/File:Merge_sort_algorithm_diagram.svg) 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.

[](http://www.geeksforgeeks.org/wp-content/uploads/gq/2013/03/Merge-Sort.png)

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/merge-sort/1)

* C/C++
* Java
* Python

|  |
| --- |
| /\* 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);      }  }  /\* This code is contributed by Rajat Mishra \*/ |

Run on IDE

Output:

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

**Time Complexity:** Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.  
T(n) = 2T(n/2) +   
The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is .  
Time complexity of Merge Sort is  in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

**Auxiliary Space:** O(n)

**Algorithmic Paradigm:**Divide and Conquer

**Sorting In Place:** No in a typical implementation

**Stable:** Yes

**Applications of Merge Sort**

1. [Merge Sort is useful for sorting linked lists in O(nLogn) time](http://www.geeksforgeeks.org/merge-sort-for-linked-list/).In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

1. [Inversion Count Problem](http://www.geeksforgeeks.org/counting-inversions/)
2. Used in [External Sorting](http://en.wikipedia.org/wiki/External_sorting)

# QuickSort

Like [Merge Sort](http://quiz.geeksforgeeks.org/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[p] is now

at right place \*/

pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); // Before pi

quickSort(arr, pi + 1, high); // After pi

}

}

[](http://www.geeksforgeeks.org/wp-content/uploads/gq/2014/01/QuickSort2.png)

**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[p] 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.

## [Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/sort-the-array/0)

**Implementation:**  
Following are C++, Java and Python implementations of QuickSort.

* C/C++
* Java
* Python

|  |
| --- |
| // 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);      }  }  /\*This code is contributed by Rajat Mishra \*/ |

Run on IDE

Output:

Sorted array:

1 5 7 8 9 10

**Analysis of QuickSort**  
Time taken by QuickSort in general can be written as following.

T(n) = T(k) + T(n-k-1) + (n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.  
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

***Worst Case:*** The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

T(n) = T(0) + T(n-1) + (n)

which is equivalent to

T(n) = T(n-1) + (n)

The solution of above recurrence is (n2).

***Best Case:*** The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

T(n) = 2T(n/2) + (n)

The solution of above recurrence is (nLogn). It can be solved using case 2 of [Master Theorem](http://en.wikipedia.org/wiki/Master_theorem).

***Average Case:***  
To do average case analysis, we need to [consider all possible permutation of array and calculate time taken by every permutation which doesn’t look easy](http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/).  
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

T(n) = T(n/9) + T(9n/10) + (n)

Solution of above recurrence is also O(nLogn)

Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms like [Merge Sort](http://quiz.geeksforgeeks.org/merge-sort/) and [Heap Sort](http://quiz.geeksforgeeks.org/heap-sort/), QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

**What is 3-Way QuickSort?**  
In simple QuickSort algorithm, we select an element as pivot, partition the array around pivot and recur for subarrays on left and right of pivot.  
Consider an array which has many redundant elements. For example, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}. If 4 is picked as pivot in Simple QuickSort, we fix only one 4 and recursively process remaining occurrences. In 3 Way QuickSort, an array arr[l..r] is divided in 3 parts:  
a) arr[l..i] elements less than pivot.  
b) arr[i+1..j-1] elements equal to pivot.  
c) arr[j..r] elements greater than pivot.  
See [this](http://www.geeksforgeeks.org/3-way-quicksort/) for implementation.

**How to implement QuickSort for Linked Lists?**  
[QuickSort on Singly Linked List](http://www.geeksforgeeks.org/quicksort-on-singly-linked-list/)  
[QuickSort on Doubly Linked List](http://www.geeksforgeeks.org/quicksort-for-linked-list/)

**Can we implement QuickSort Iteratively?**  
Yes, please refer [Iterative Quick Sort](http://www.geeksforgeeks.org/iterative-quick-sort/).

**Why Quick Sort is preferred over MergeSort for sorting Arrays**  
Quick Sort in its general form is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn’t occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

**Why MergeSort is preferred over QuickSort for Linked Lists?**  
In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

# Heap Sort

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**](http://geeksquiz.com/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](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees))

A [Binary Heap](http://geeksquiz.com/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.

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](http://practice.geeksforgeeks.org/problems/heap-sort/1)

* C++
* Java
* Python

|  |
| --- |
| // 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);      }  } |

Run on IDE

Output:

Sorted array is

5 6 7 11 12 13

[Here](http://code.geeksforgeeks.org/rFO7Lm) is previous C code for reference.

**Notes:**  
Heap sort is an in-place algorithm.  
Its typical implementation is not stable, but can be made stable (See [this](http://www.geeksforgeeks.org/stability-in-sorting-algorithms/))

**Time Complexity:**Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

**Applications of HeapSort**  
**1.** [Sort a nearly sorted (or K sorted) array](http://www.geeksforgeeks.org/nearly-sorted-algorithm/)  
**2.**[k largest(or smallest) elements in an array](http://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/)

Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used. See [Applications of Heap Data Structure](http://www.geeksforgeeks.org/applications-of-heap-data-structure/)

s