Bachelor of Computer Science and Information Technology

DATA STRUCTURE AND ALGORITHM

Chapter(6)

Sorting

Lecturer: Er. Saroj Ghimire
Qualification: Msc.CSIT, BE(COMPUTER)

CONTENT

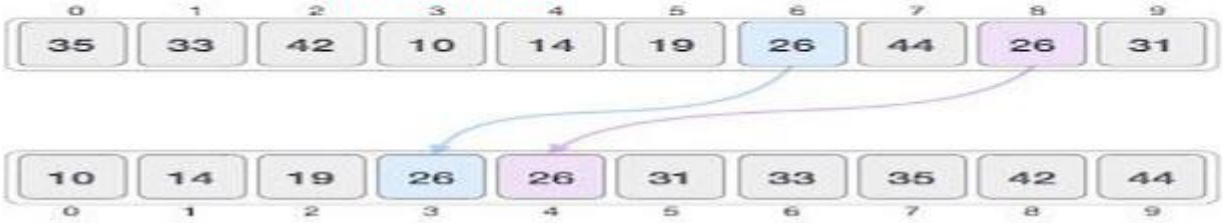
- ✓ Introduction, Internal & External Sort
- ✓ Comparison Sorting Algorithms: Bubble, Selection and Insertion Sort, Shell Sort
- ✓ Divide and Conquer Sorting: Merge, Quick and Heap Sort
- ✓ Efficiency of Sorting Algorithms

SORTING

- Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order.
- Sorting is a process through which the data is arranged in ascending or descending order
- The complexity of a sorting algorithm can be measured in terms of
  - number of algorithm steps to sort n records
  - number of comparisons between keys (appropriate when the keys are long character strings)
  - number of times records must be moved (appropriate when record size is large)
- Example:
  - 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.
  - 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.
  - For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.
  - For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.

SORTING

- Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called **in-place sorting.** Bubble sort is an example of in-place sorting.
- Sorting which uses equal or more space is called not-in-place sorting. Merge-sort is an example of **not-in-place sorting**.
- If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting.**



- If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear.

It is called **unstable sorting**.

INTERNAL & EXTERNAL SORTING

Internal sorting

- An internal sort is any data sorting process that takes place entirely within the main memory of a computer.
- This is possible whenever the data to be sorted is small enough to all be held in the main memory.There are 3 types of internal sorts.
  - (i) SELECTION SORT :- Ex:- Selection sort algorithm, Heap Sort algorithm
  - (ii) INSERTION SORT :- Ex:- Insertion sort algorithm, Shell Sort algorithm
  - (iii) EXCHANGE SORT :- Ex:- Bubble Sort Algorithm, Quick sort algorithm

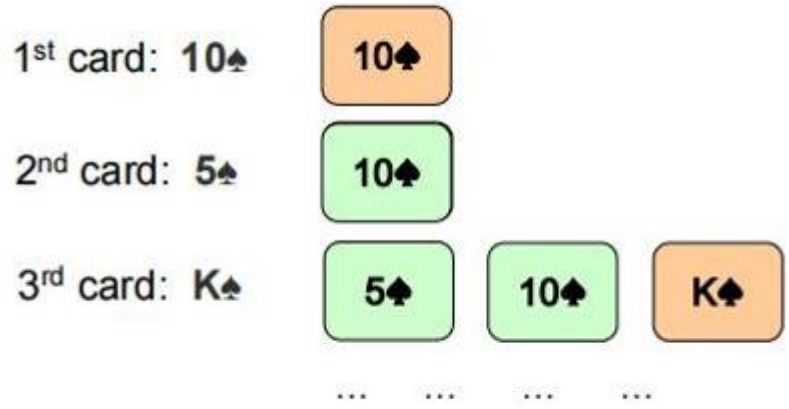
External sorting

- Sorting large amount of data requires external or secondary memory. This process uses external memory such as HDD, to store the data which is not fit into the main memory. So, primary memory holds the currently being sorted data only. All external sorts are based on process of merging. Different parts of data are sorted separately and merged together.
- Ex:- Merge Sort

NOTE: We will only consider internal sorting

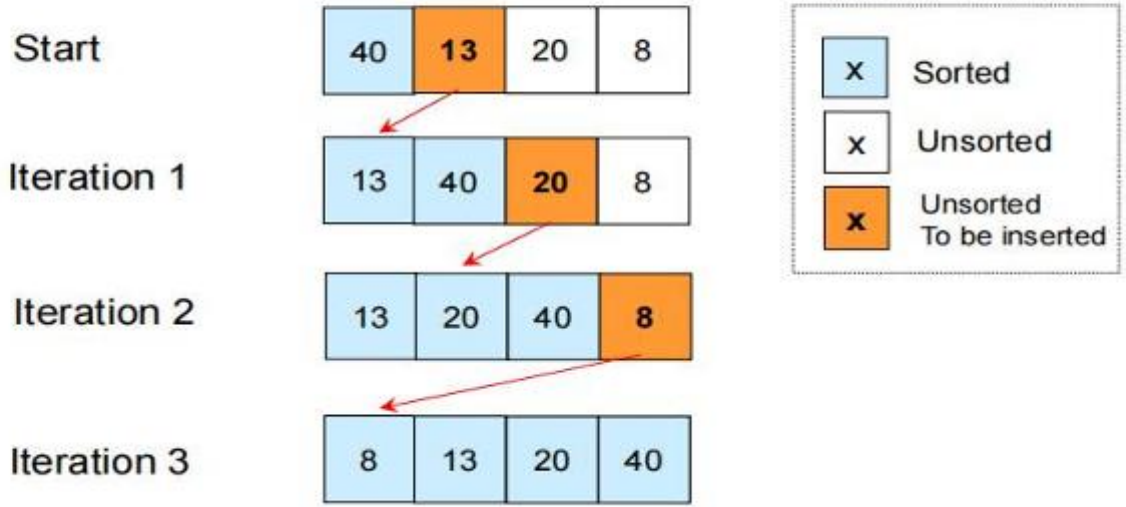
INSERTION SORT

- **Insertion sort** is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
- The array is virtually split into a sorted and an unsorted part.
- Values from the unsorted part are picked and placed at the correct position in the sorted part.



INSERTION SORT: ALGORITHM

1. We start with an empty left hand [sorted array] and the cards face down on the table [unsorted array].
2. Then remove one card [key] at a time from the table [unsorted array], and insert it into the correct position in the left hand [sorted array].
3. To find the correct position for the card, we compare it with each of the cards already in the hand, from right to left.



INSERTION SORT : IMPLEMENTATION

```
void insertionSort(int a[], int n) {
  for (int i = 1; i < n; i++) {
    int next = a[i];
    int j;
    for (j = i-1; j >= 0 && a[j] > next; j--)
      a[j+1] = a[j];

    a[j+1] = next;
  }
}
```

next is the item to be inserted

Shift sorted items to make place for next

Insert next to the correct location

INSERTION SORT: ANALYSIS

- Outer-loop executes (n−1) times
- Number of times inner-loop is executed depends on the input
  - Best-case: the array is already sorted and (a[j] > next) is always false
  - No shifting of data is necessary
  - Worst-case: the array is reversely sorted and (a[j] > next) is always true
  - Insertion always occur at the front
  - Average case: we have to look at all possible intial data organization
- Therefore,
  - best-case time is O(n)
  - worst-case time is $O(n^2)$
  - average- case time is $O(n^2)$

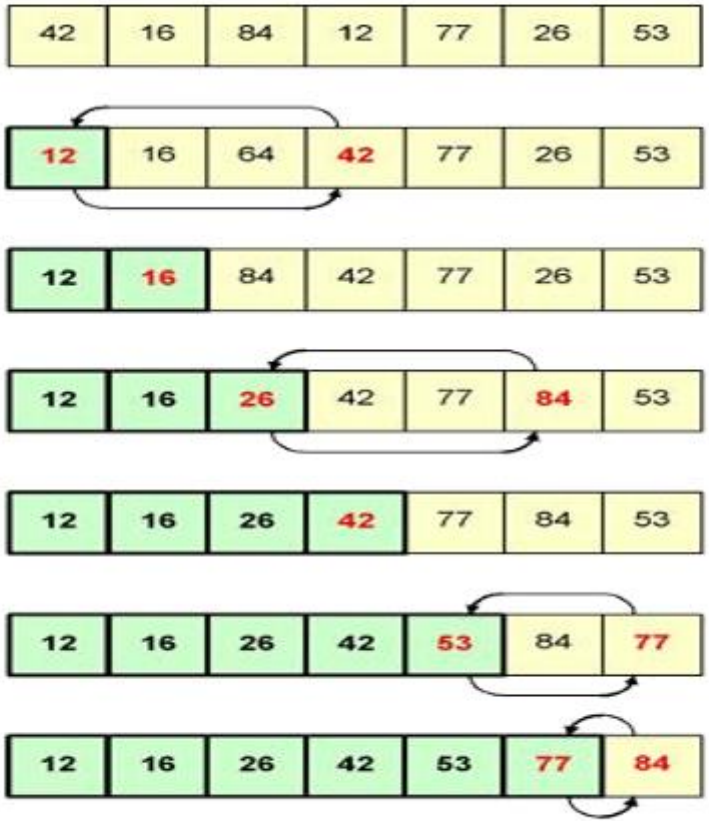
SELECTION SORT

- ✓ 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.

Algorithm:

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat this process for all the elements until the entire array is sorted

SELECTION SORT: EXAMPLE



SELECTION SORT: ANALYSIS

- Selecting the lowest element requires scanning all n elements (this takes n-1 comparisons) and then swapping it into the first position.
- Finding the next lowest element requires scanning all n-1 elements and so on, for (n-1) + (n-2) + ... + 2 + 1 (O(n2)) comparisons. Each of these scans requires one swap for n-1 elements.
- Therefore,
  - Worst case performance: O(n2)
  - Best case performance: O(n2)
  - Average case performance: O(n2)

BUBBLE SORT

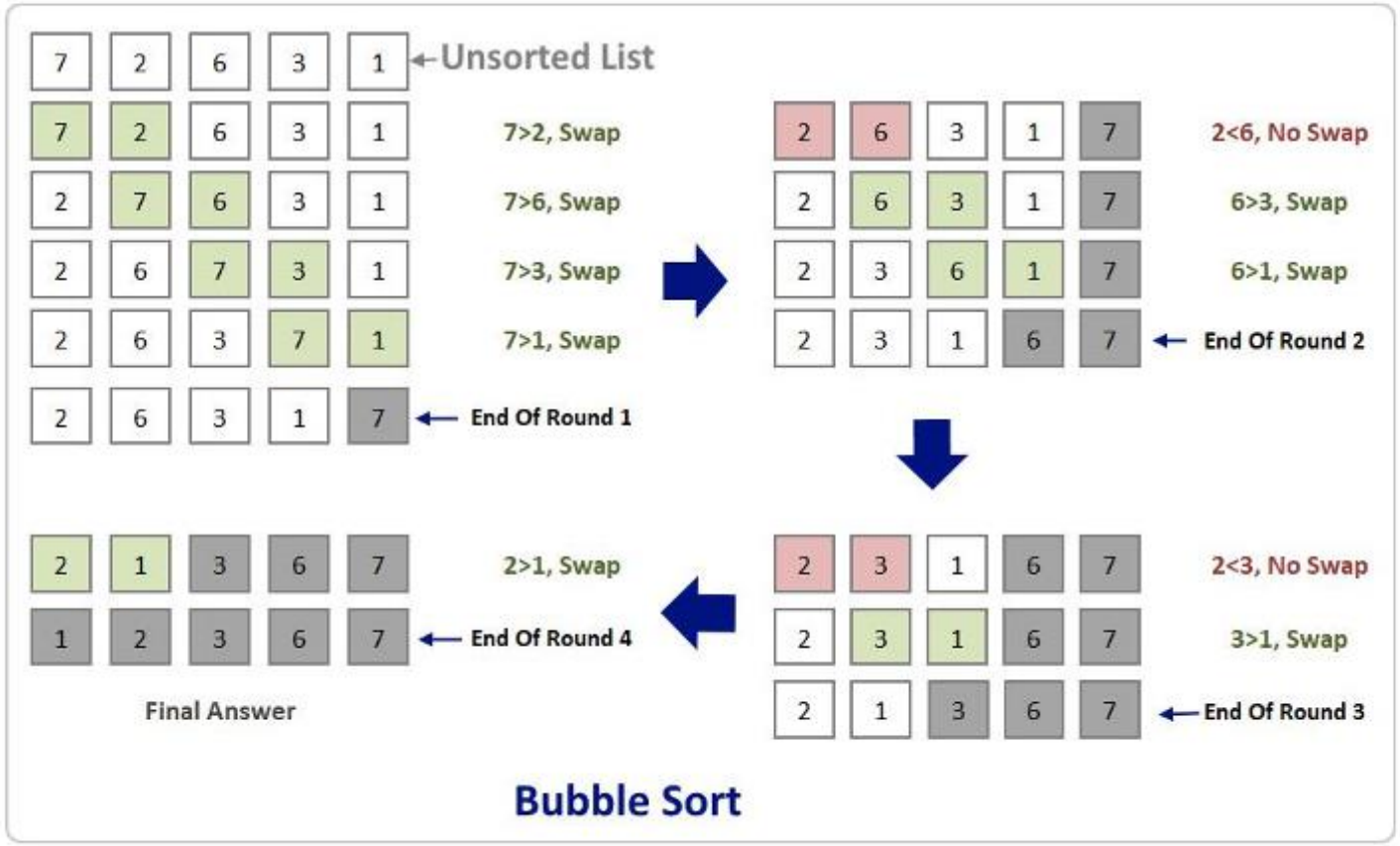
- ✓ Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order

Algorithm:

Given an array of n items

1. Compare each pair of adjacent elements in the list
2. Swap two element if necessary
3. Repeat this process for all the elements until the entire array is sorted
4. Reduce n by 1 and go to Step 1

BUBBLE SORT: EXAMPLE



BUBBLE SORT: ANALYSIS

- 1 iteration of the inner loop (test and swap) requires time bounded by a constant c
- Two nested loops Two nested loops
  - Outer loop: exactly n iterations
  - Inner loop:
    - when i=0, (n−1) iterations
    - when i=1, (n−2) iterations
    - … …
    - when i=(n−1), 0 iterations
- Total number of iterations = 0+1+…+(n−1) = n(n−1)/2
- Total time = c n(n−1)/2 = $O(n^2)$
- Therefore,
  - worst case:$O(n^2)$and best case:O(n)

SHELL SORT

- Shell sort: orders a list of values by comparing elements that are separated by a gap of >1 indexes
  - a generalization of insertion sort
  - invented by computer scientist Donald Shell in 1959
- Shell sort works by comparing elements that are distant rather than adjacent elements in an array or list where adjacent elements are compared.
- Shell sort makes multiple passes through a list and sorts a number of equally sized sets using the insertion sort.

SHELL SORT: ALGORITHM

1. Initialize the gap size.
2. Divide the array into subarrays of equal gap size.
3. Apply insertion sort on the subarrays.
4. Repeat the above steps until the gap size becomes 0 resulting into a sorted array.

SHELL SORT:EXAMPLE

Consider an unsorted array as follows.

81 94 11 96 12 35 17 95 28 58

1. Here N = 10, the first pass as K = 5 (10/2)

81 94 11 96 12 35 17 95 28 58

   - After first pass 35 17 11 28 12 81 94 95 96 58

2. In second Pass, K is reduced to 3

35 17 11 28 12 81 94 95 96 58

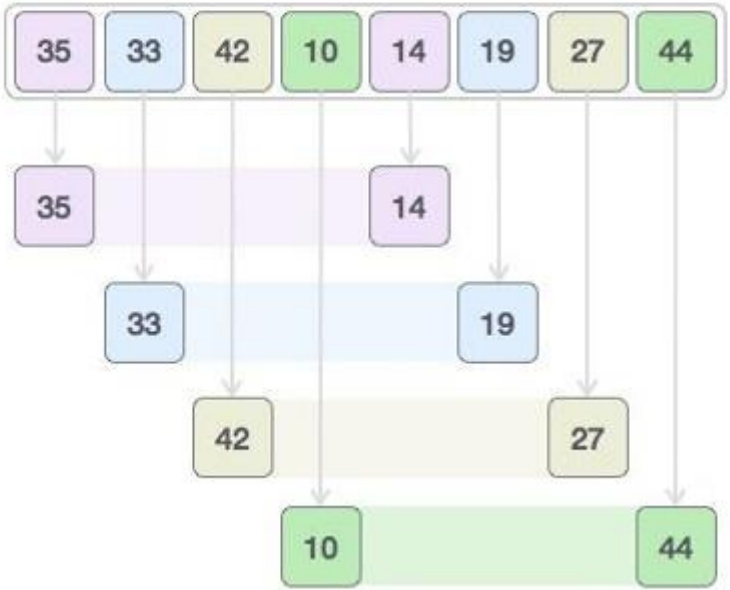
   - After second pass, 28 12 11 35 17 81 58 95 96 94

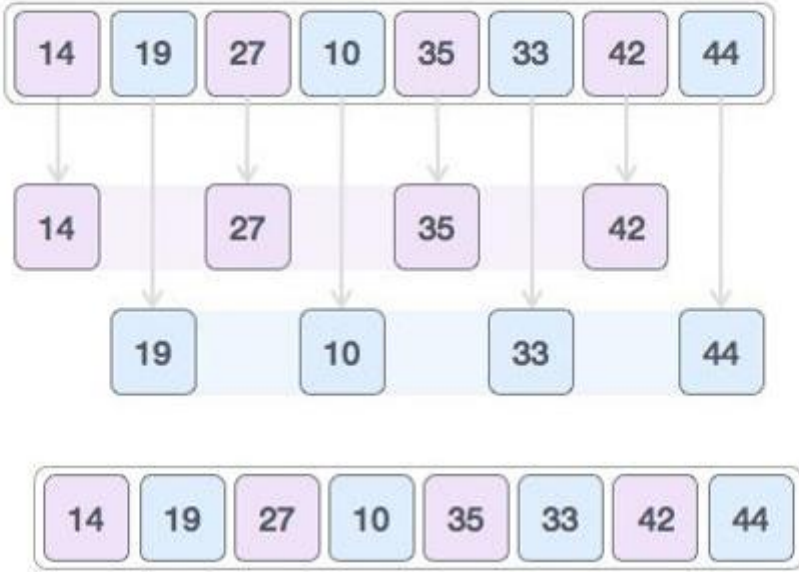
3. In third pass, K is reduced to 1

28 12 11 35 17 81 58 95 96 94

   - The final sorted array is 11 12 17 28 35 58 81 94 95 96

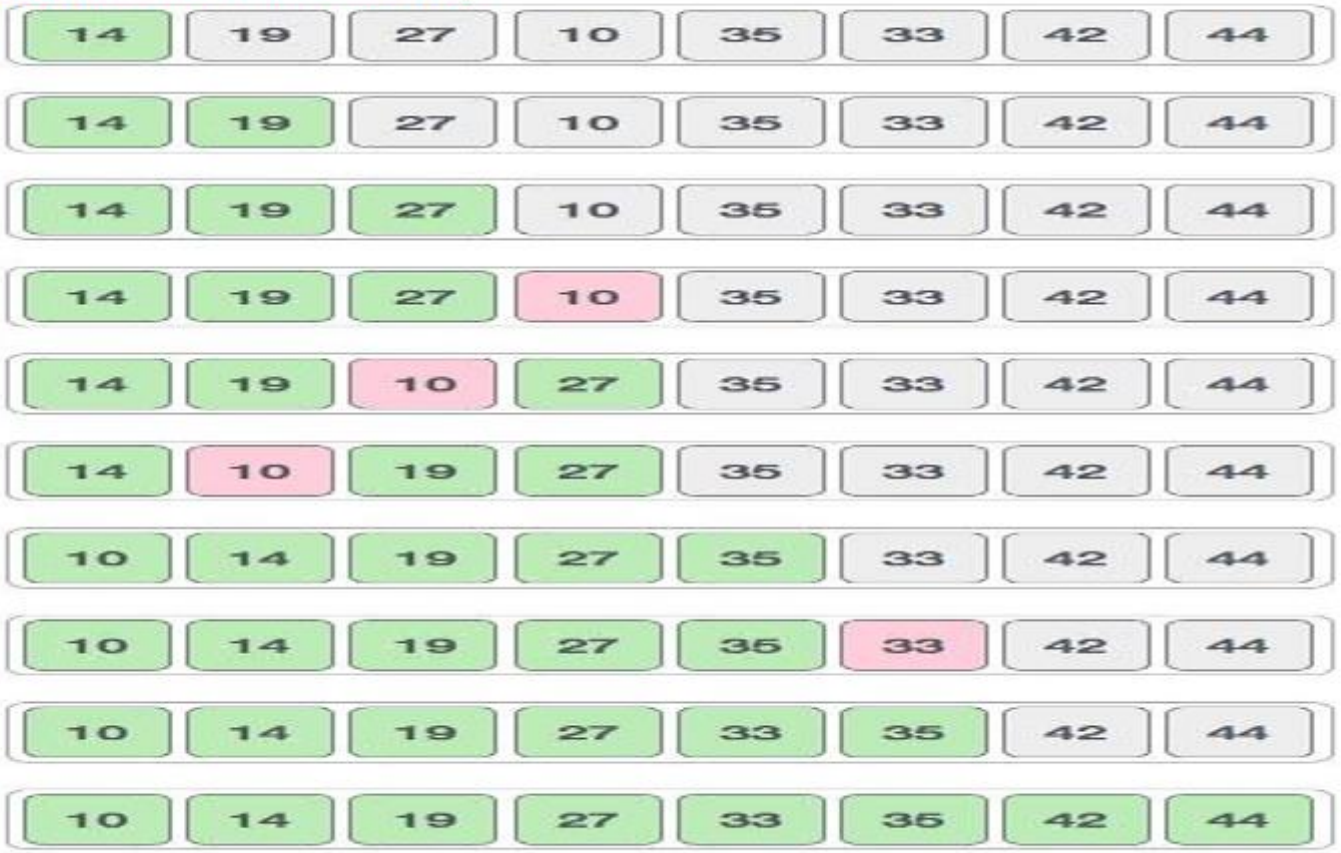
SHELL SORT: NEXT EXAMPLE





SHELL SORT: NEXT EXAMPLE

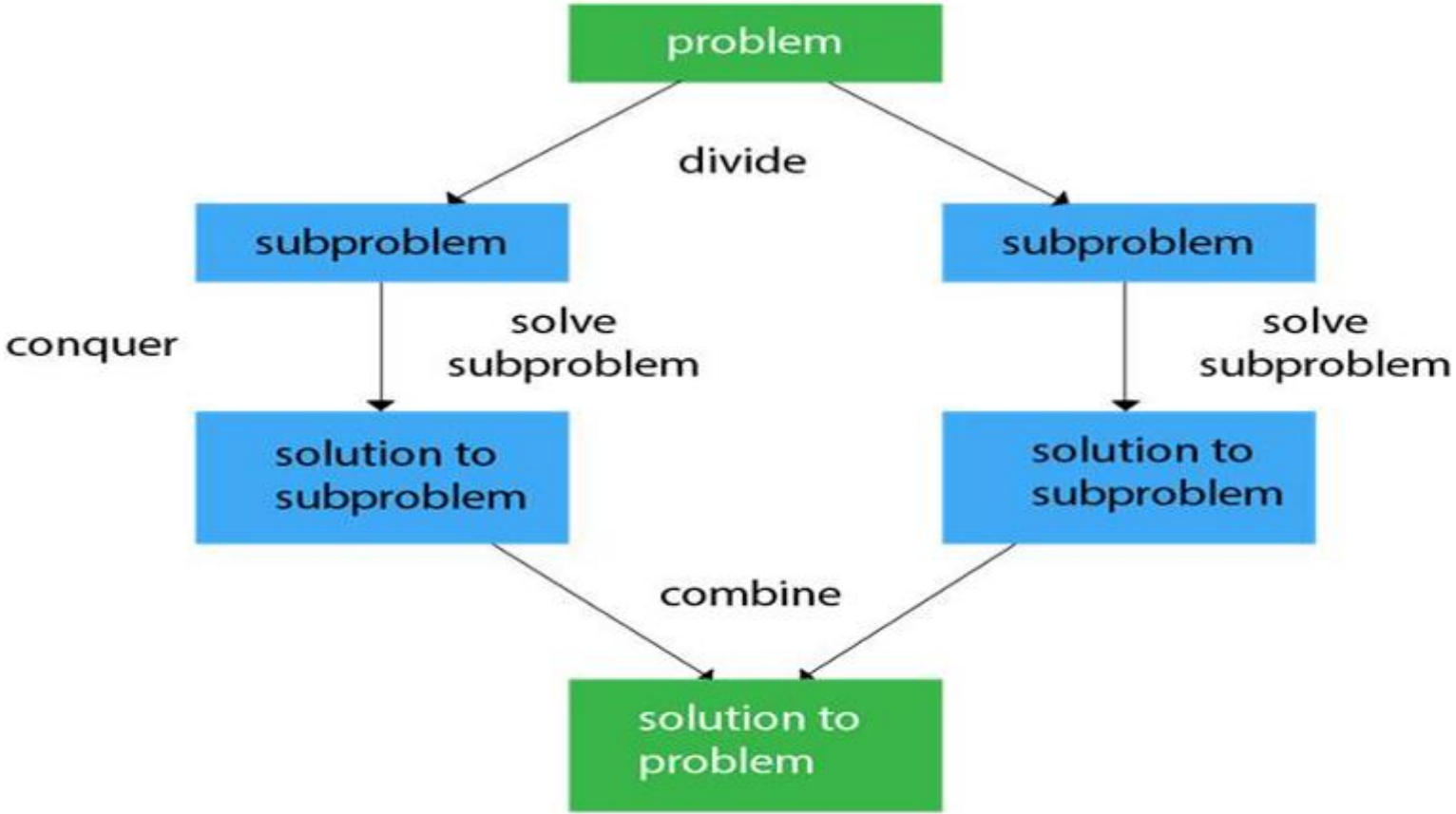
Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.



DIVIDE AND CONQUER ALGORITHM

- Divide and Conquer is an **algorithmic pattern**.
- In **algorithmic methods**, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the **Divide & Conquer Strategy.**
- Divide and Conquer algorithm consists of a dispute using the following three steps.
  1. Divide the original problem into a set of subproblems.
  2. Conquer: Solve every subproblem individually, recursively.
  3. Combine: Put together the solutions of the subproblems to get the solution to the whole problem.
- Examples: The specific computer algorithms are based on the Divide & Conquer approach:
  - Maximum and Minimum Problem
  - Binary Search
  - Sorting (merge sort, quick sort)
  - Tower of Hanoi.

DIVIDE AND CONQUER ALGORITHM



DIVIDE AND CONQUER

Advantages of Divide and Conquer

1. Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
2. It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.
3. It is more proficient than that of its counterpart Brute Force technique.
4. Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating parallel processing.

Disadvantages of Divide and Conquer

1. Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
2. An explicit stack may overuse the space.
3. It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

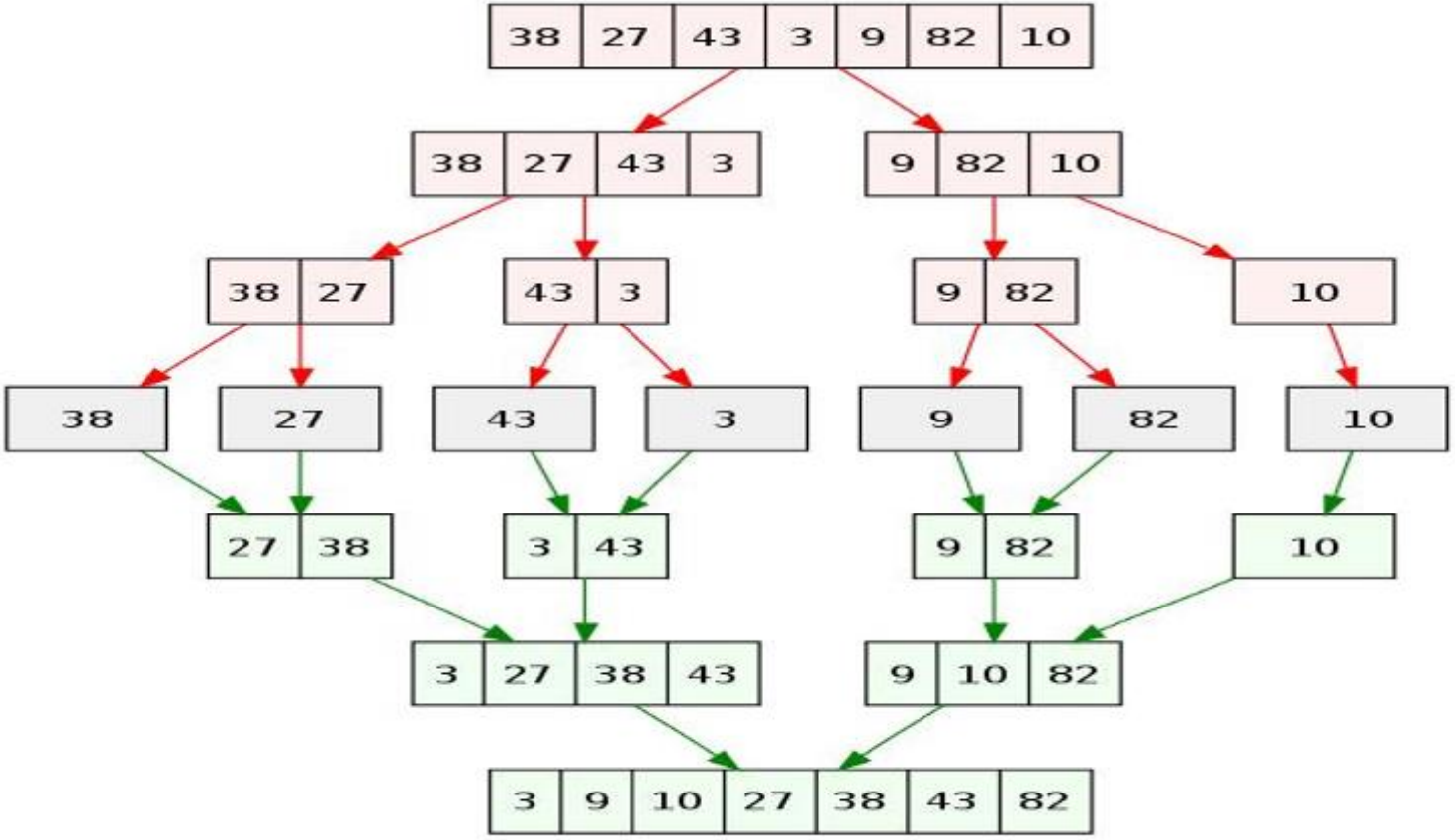
MERGE SORT

- Merge Sort is a divide-and-conquer sorting algorithm
- This is a simple and very efficient algorithm for sorting a list of numbers.
- We are given a sequence of n numberswhich we will assume is stored in an array A [1...n].
  - Divide step
    - Divide the array into two (equal) halves
    - Recursively sort the two halves
  - Conquer step
    - Merge the two halves to form a sorted array

MERGE SORT: ALGORITHM

- Conceptually, a merge sort works as follows:
  1. If the list is of length 0 or 1, then it is already sorted. Otherwise:
  2. Divide the unsorted list into two sublists of about half the size.
  3. Sort each sublist recursively by re-applying merge sort.
  4. Merge the two sublists back into one sorted list.

MERGE SORT: EXAMPLE



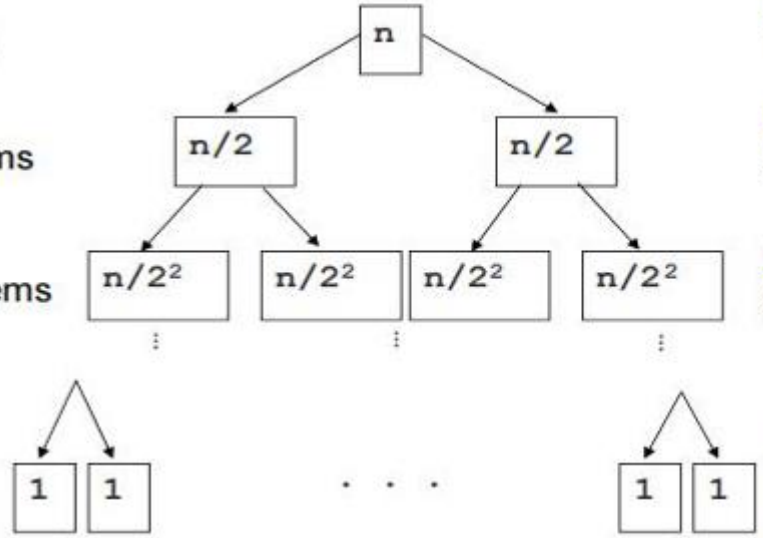
MERGE SORT: ANALYSIS

Level 0:
mergeSort n items

Level 1:
mergeSort n/2 items

Level 2:
mergeSort $n/2^2$ items

Level (lg *n*):
mergeSort 1 item



Level 0:
1 call to mergeSort

Level 1:
2 calls to mergeSort

Level 2:
2^2 calls to mergeSort

Level (lg *n*):
$2^{\lg n} (= n)$ calls to mergeSort

$n/(2^k) = 1 \rightarrow n = 2^k \rightarrow k = \lg n$

Total time complexity = O(n lg(n))

QUICK SORT

- Quick Sort is one of the different Sorting Technique which is based on the concept of Divide and Conquer
- It is also called partition-exchange sort. This algorithm divides the list into three main parts:
  - Elements less than the Pivot element
  - Pivot element(Central element)
  - Elements greater than the pivot element
- Pivot element can be any element from the array, it can be the first element, the last element or any random element.

QUICK SORT: ALGORITHM

Following are the steps involved in quick sort algorithm:

1. After selecting an element as pivot, which is the last index of the array in our case, we divide the array for the first time.
2. Define two variables i and j. Set i and j to first and last elements of the list respectively.
3. Increment i until list[i] > pivot then stop.
4. Decrement j until list[j] < pivot then stop.
5. If i < j then exchange list[i] and list[j].
6. Repeat steps 3,4 & 5 until i > j.
7. Exchange the pivot element with list[j] element.

QUICK SORT: EXAMPLE

Consider the following unsorted list of elements...

Define pivot, left & right. Set pivot = 0, left = 1 & right = 7. Here '7' indicates 'size-1'.

Compare List[left] with List[pivot]. If **List[left]** is greater than **List[pivot]** then stop left otherwise move left to the next.

Compare List[right] with List[pivot]. If List[right] is smaller than List[pivot] then stop right otherwise move right to the previous.

Repeat the same until **left>=right**.

If both left & right are stoped but left<right then swap List[left] with List[right] and countinue the process.

If left>=right then swap List[pivot] with List[right].

QUICK SORT: EXAMPLE

Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 8.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 2.

Here left & right both are stoped and left is not greater than right so we need to swap List[left] and List[right]

Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 6.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 4.

Here left & right both are stoped and left is greater than right so we need to swap List[pivot] and List[right]

QUICK SORT: EXAMPLE

Here we can observe that all the numbers to the left side of 5 are smaller and right side are greater. That means 5 is placed in its correct position.

Repeat the same process on the left sublist and right sublist to the number 5.

In the left sublist as there are no smaller number than the pivot left will keep on moving to the next and stops at last number. As the List[right] is smaller, right stops at same position. Now left and right both are equal so we swap pivot with right.

QUICK SORT: EXAMPLE

In the right sublist left is grester than the pivot, left will stop at same position.

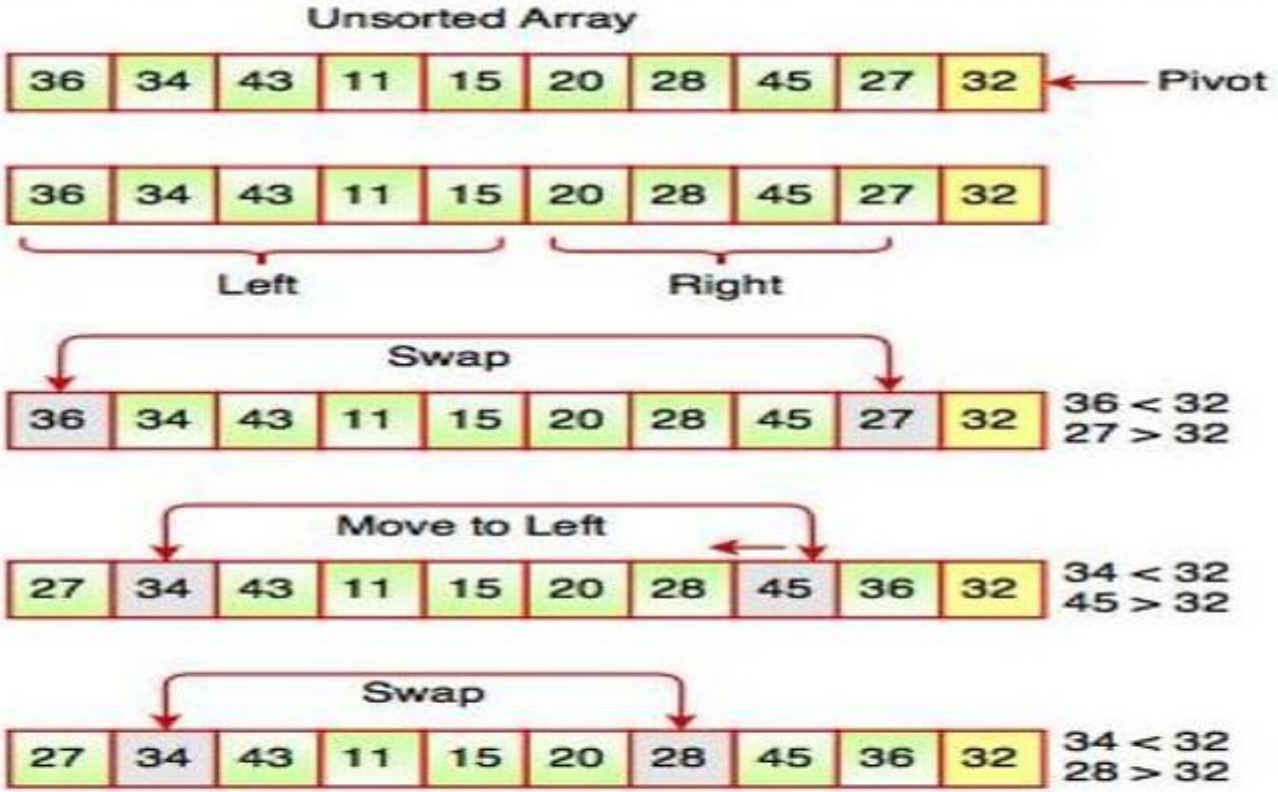
As the List[right] is greater than List[pivot], right moves towords left and stops at pivot number position.

Now left > right so we swap pivot with right. (6 is swap by itself).

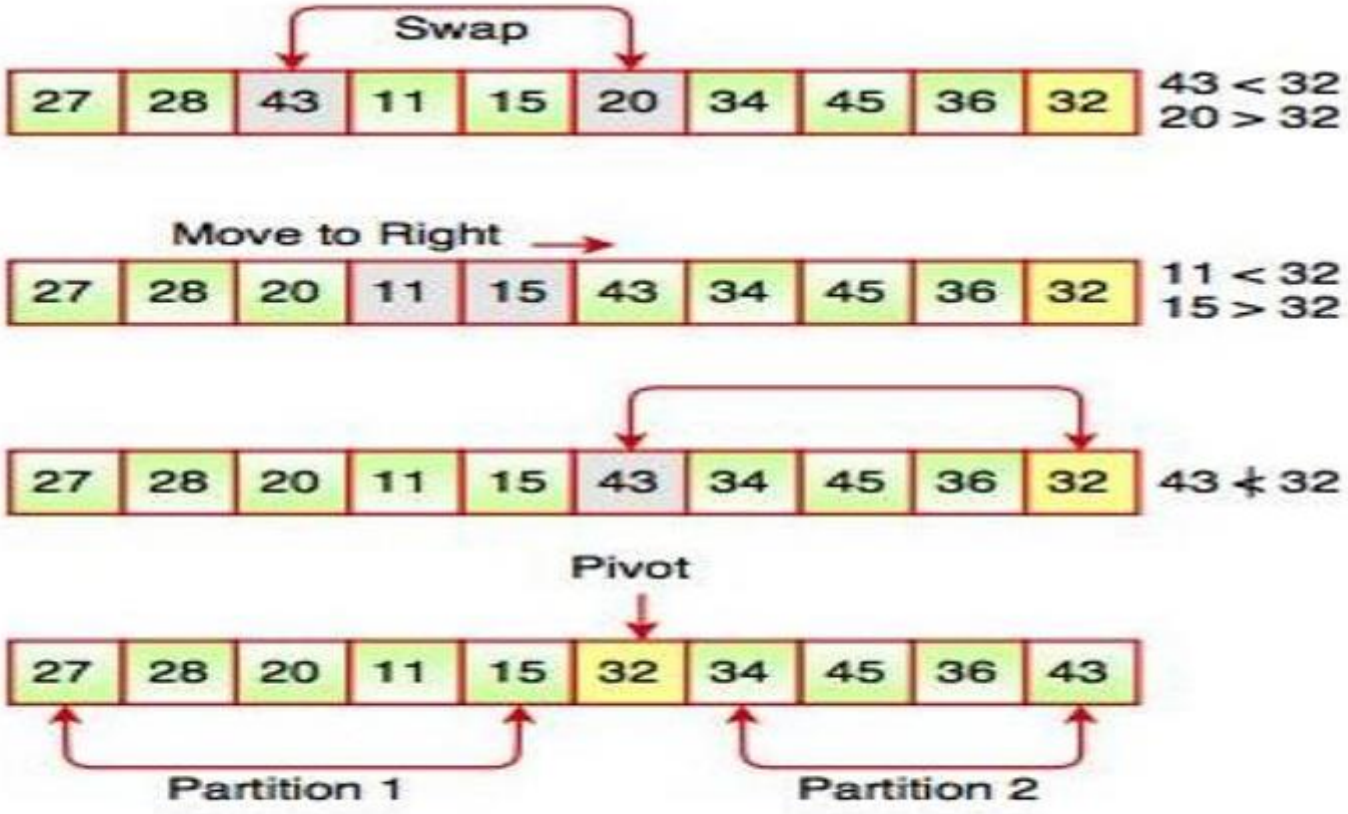
Repeat the same recursively on both left and right sublists until all the numbers are sorted.

The final sorted list will be as follows...

QUICK SORT: NEXT EXAMPLE



QUICK SORT: NEXT EXAMPLE



QUICK SORT: ANALYSIS

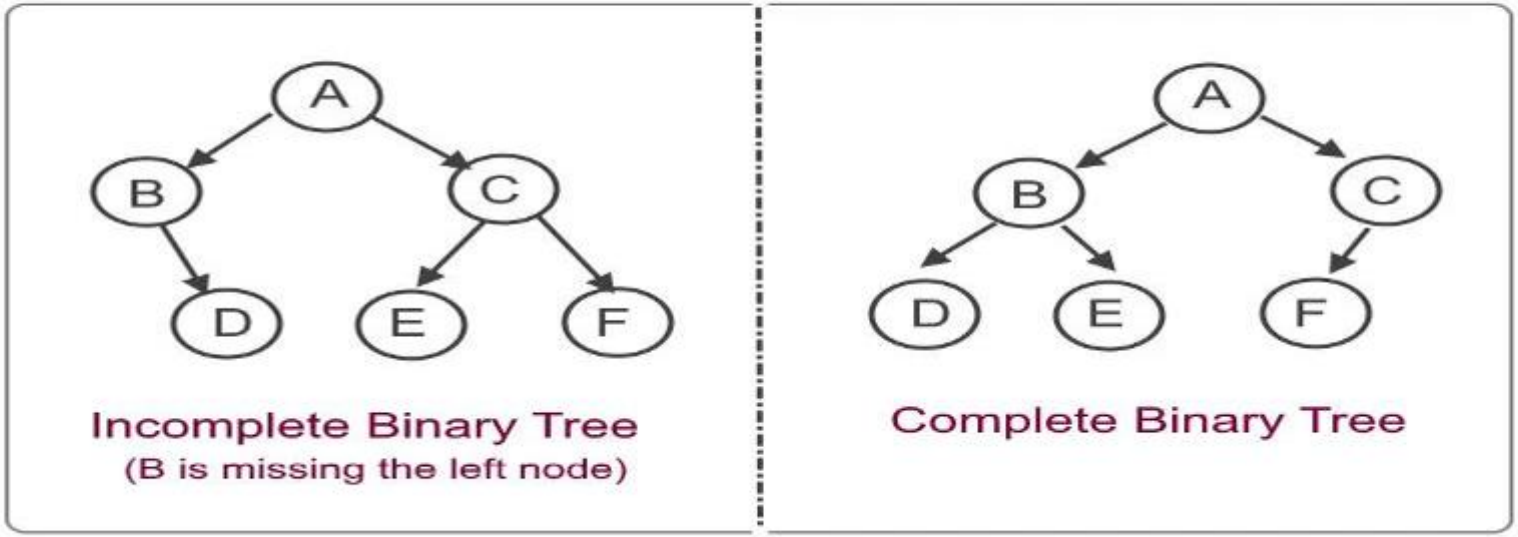
- Best-case: Pivot is always the median
  - T(0)=T(1)=1
  - T(n)=2T(n/2) + n -- linear-time partition
  - Same recurrence as mergesort: O(n log n)
- Worst-case: Pivot is always smallest or largest element
  - T(0)=T(1)=1
  - T(n) = 1T(n-1) + n
  - Basically same recurrence as selection sort: $O(n^2)$
- Average-case (e.g., with random pivot)
  - T(n) = ? + $\frac{(n-1)!}{n!}[\sum_{i=1}^{n} T(i-1) + T(n-i)]$
  - O(n log n),

HEAP

- Heap is a data structure that stores a collection of objects (with keys), and has the following properties:
  - Complete Binary tree
  - Heap Order
- The heap sort algorithm has two major steps :
  - The first major step involves transforming the complete tree into a heap.
  - The second major step is to perform the actual sort by extracting the largest or lowest element from the root and transforming the remaining tree into a heap.

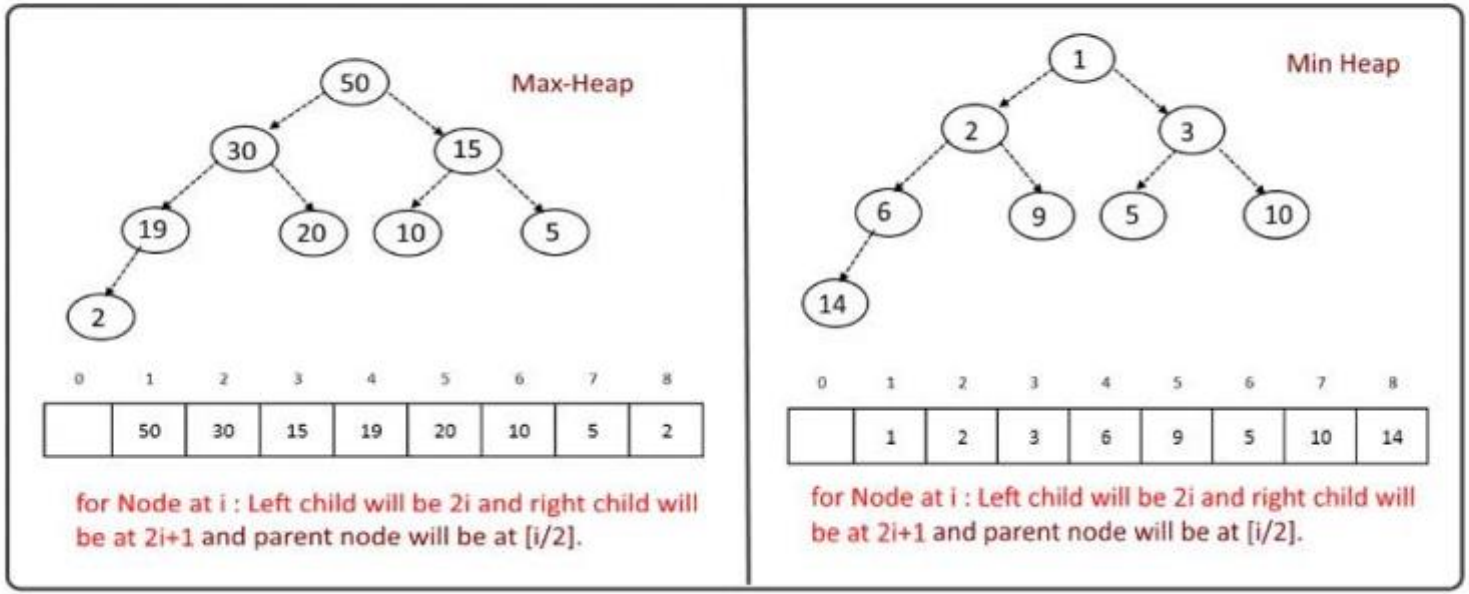
BINARY HEAP

- Shape Property:
  - Heap data structure is always a Complete Binary Tree. The Complete Binary tree is a binary tree which is completely filled (means all the nodes has 2 children) except the last level which might not be completely full.



BINARY HEAP

- Heap Property:
  - All nodes are either greater than equal to (Max-Heap) or less than equal to (Min-Heap) to each of its child nodes. This is called heap property.

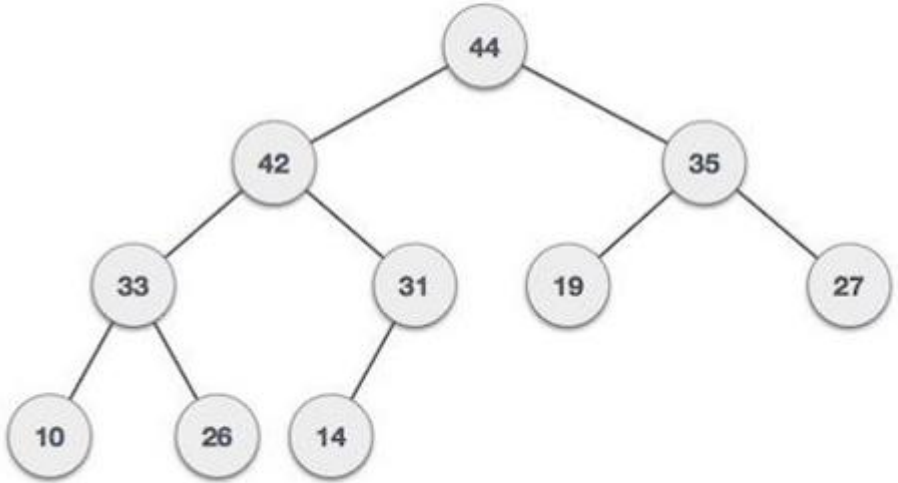


MAX-HEAP

1. In a **max-heap**, the **max-heap property** is that for every node i other than the root,

$$A[PARENT(i)] \geq A[i] .$$

- the largest element in a max- heap is stored at the root
- the subtree rooted at a node contains values no larger than that contained at the node itself

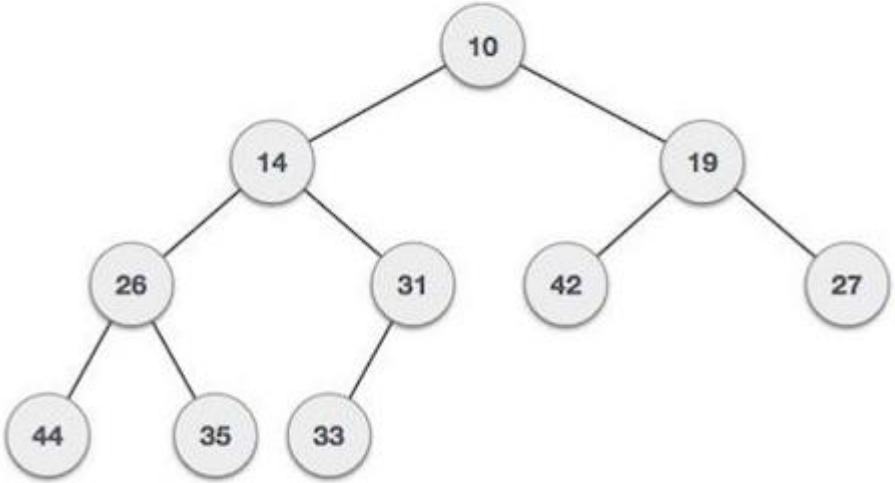


MIN-HEAP

2. In a **min-heap**, the **min-heap property** is that for every node i other than the root,

$$A[PARENT(i)] \leq A[i] .$$

- the smallest element in a min- heap is at the root
- the subtree rooted at a node contains values no smaller than that contained at the node itself



HEAP SORT

- A sorting algorithm that works by first organizing the data to be sorted into a special type of **binary tree called a heap.**
- Heap sort processes the elements by creating the min heap or max heap using the elements of the given array.
- **Min heap or max heap** represents the ordering of the array in which root element represents the minimum or maximum element of the array.
- At each step, the root element of the heap gets deleted and stored into the sorted array and the heap will again be heapified.
- The heap sort basically recursively performs two main operations.
  - Build a heap H, using the elements of array.
  - Repeatedly delete the root element of the heap formed in phase 1.

HEAP SORT ALGORITHM

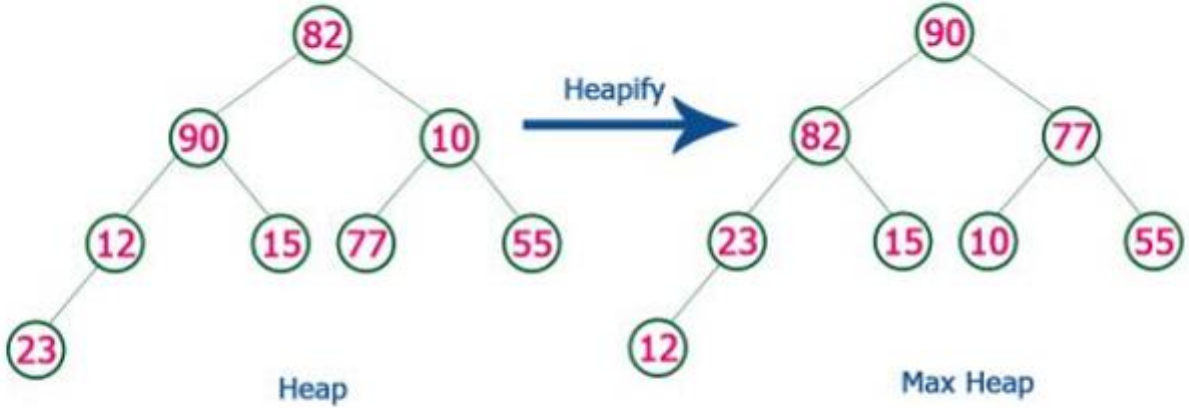
- Sorting can be in ascending or descending order.
- Either Max heap or min heap logic can be taken depending on the need.
  1. Build a max/min heap using Heapify() from the input data.
  2. At this point, the largest/smallest item is stored at the root of the heap. Replace it with the last item of the 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.
- To sort an unsorted list with 'n' number of elements, following are the complexities...
  - Worst Case : O(n log n)
  - Best Case : O(n log n)
  - Average Case : O(n log n)

HEAP SORT: EXAMPLE

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap

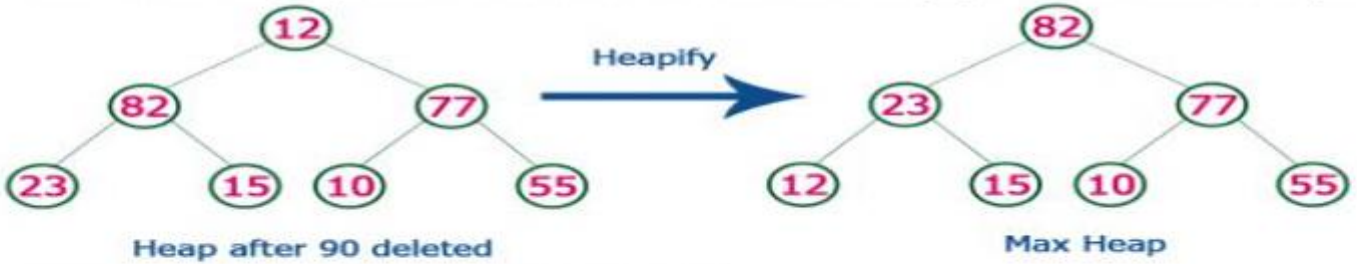


list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

HEAP SORT: EXAMPLE

Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, **90**

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, **82, 90**

HEAP SORT: EXAMPLE

Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, **77, 82, 90**

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 55 with 15.

12, 15, 10, 23, **55, 77, 82, 90**

HEAP SORT: EXAMPLE

Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 23 with 12.

12, 15, 10, 23, 55, 77, 82, 90

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after Deleting 15, 12 & 10 from the Max Heap.

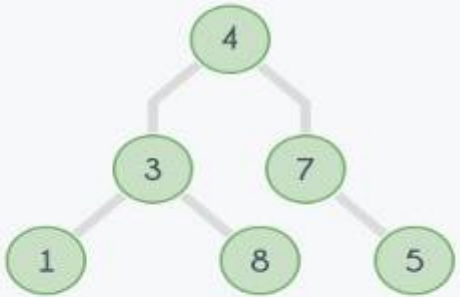
10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

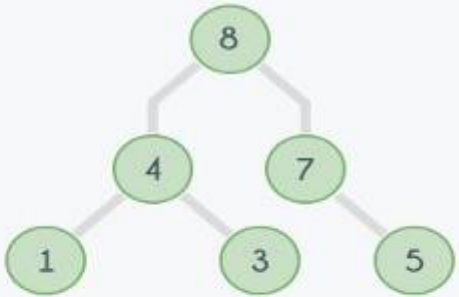
HEAP SORT: NEXT EXAMPLE

Arr		4	3	7	1	8	5
	0	1	2	3	4	5	6

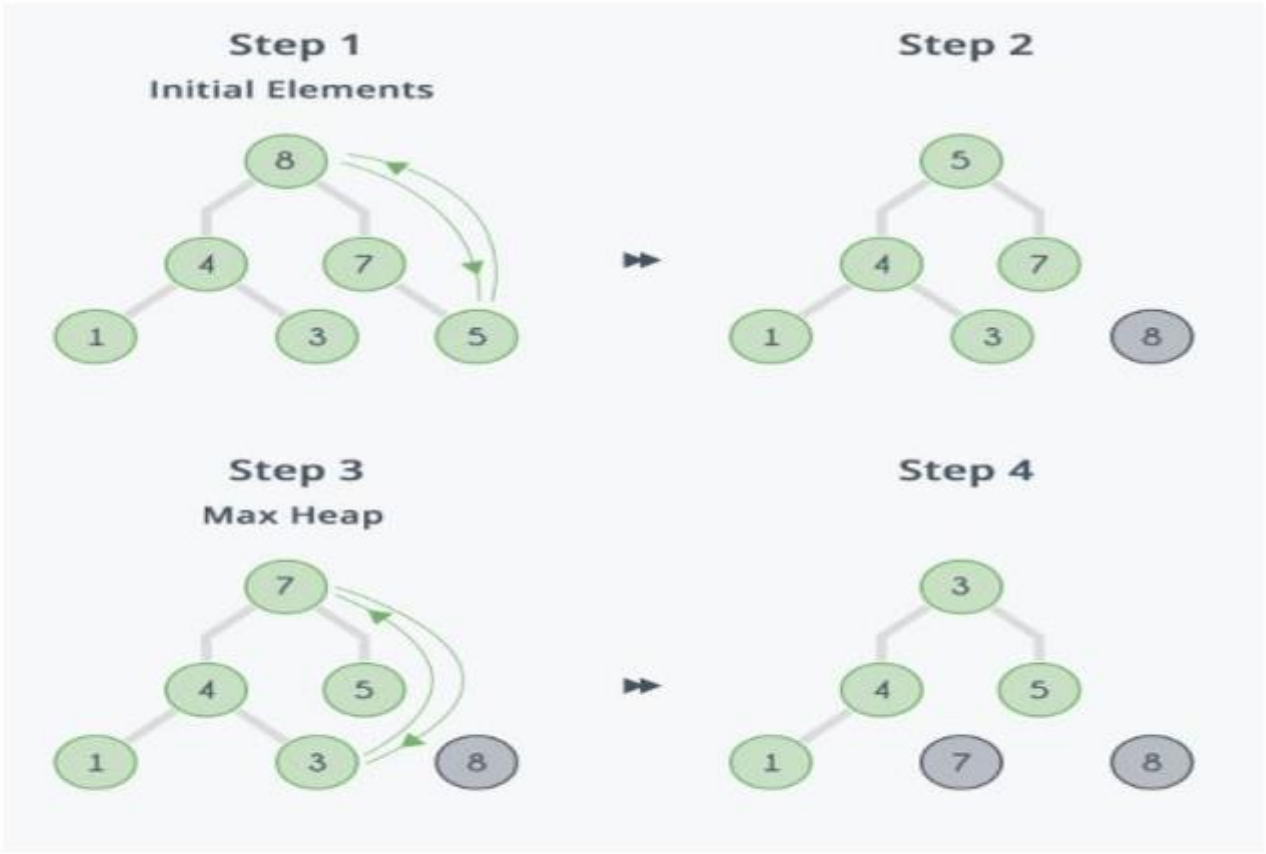
Initial Elements



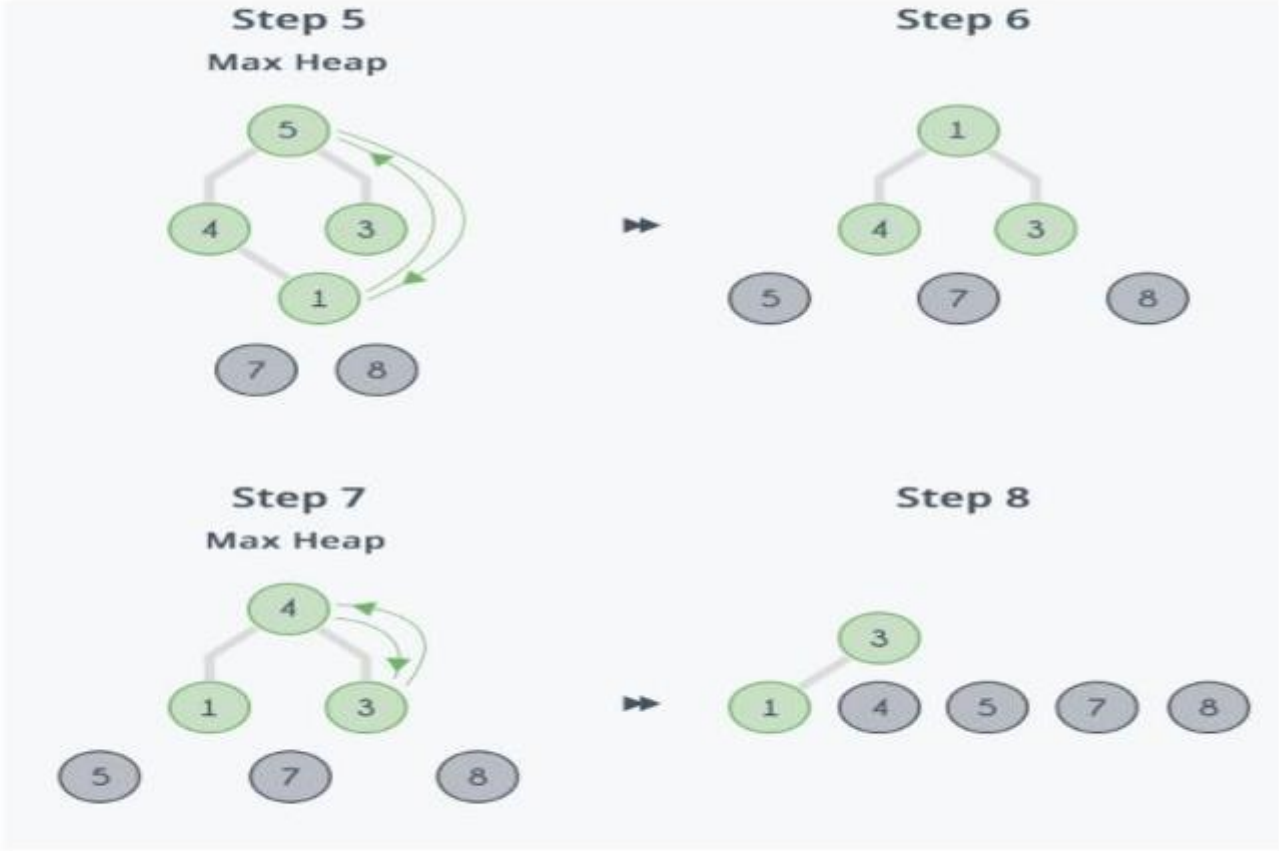
Max Heap



HEAP SORT: NEXT EXAMPLE



HEAP SORT: NEXT EXAMPLE



HEAP SORT: NEXT EXAMPLE

EFFICENCY OF SORTING ALGORITHM

- The complexity of a sorting algorithm measures the running time of a function in which n number of items are to be sorted.
- The choice of sorting method depends on efficiency considerations for different problems.
- Tree most important of these considerations are:
  - The length of time spent by programmer in coding a particular sorting program
  - Amount of machine time necessary for running the program
  - The amount of memory necessary for running the program

EFFICENCY OF SORTING ALGORITHM

- Various sorting methods are analyzed in the cases like – **best case, worst case or average case.**
- Most of the sort methods we consider have requirements that range from **0(nlogn) to 0(n^2).**
- A sort should not be selected only because its sorting time is **0(nlogn)**; the relation of the file size n and the other factors affecting the actual sorting time must be considered
- Determining the time requirement of sorting technique is to actually run the program and measure its efficiency.
- Once a particular sorting technique is selected the need is to make the program as efficient as possible.
- Any improvement in sorting time significantly affect the overall efficiency and saves a great deal of computer time

EFFICENCY OF SORTING ALGORITHM

Algorithm	Data Structure	Time Complexity		
		Best	Average	Worst
Quicksort	Array	O(n log(n))	O(n log(n))	O(n^2)
Mergesort	Array	O(n log(n))	O(n log(n))	O(n log(n))
Heapsort	Array	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	Array	O(n)	O(n^2)	O(n^2)
Insertion Sort	Array	O(n)	O(n^2)	O(n^2)
Select Sort	Array	O(n^2)	O(n^2)	O(n^2)