Chapter 7 Sorting

- Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending.
- Sorting can be done based on key references which can be numbers, alphabets etc.
- if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] < A[1] < A[2] < ..... < A[N-1].
- For example, if we have an array that is declared and initialized as
- A[] = {21, 34, 11, 9, 1, 0, 22};
- Then the sorted array (ascending order) can be given as: A[] = {0, 1, 9, 11, 21, 22, 34};
- Examples of sorting in real life scenarios :
    - Telephone directories in which names are sorted by location, category (business or residential), and then in an alphabetical order.
    - In a library, the information about books can be sorted alphabetically based on titles and then by authors' names.
    - Customers' addresses can be sorted based on the name of the city and then the street.
- Sorting can be categorized in two different categories:

Types of sorting:

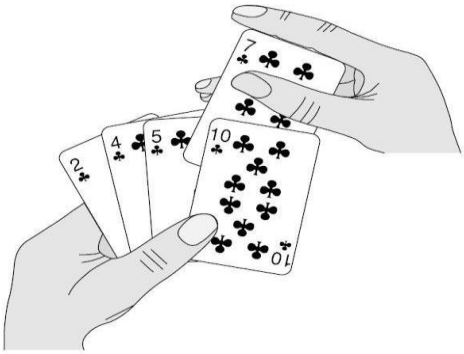
1. **Internal Sorting:**
    - The sorting is done within the computer main memory and all the data to be sorted is stored in main memory.
    - It is performed when the data to be sorted is small enough to fit in main memory. (It is used when the size of input is small.)
    - In it, the storage device used is only main memory (RAM).
    - Examples: Insertion sort, Quick Sort, Bubble Sort, etc.

2. **External Sorting:**
    - The sorting is done in external file disk and data is stored outside the main memory like on disk and only loaded into memory in small chunks.
    - It is usually applied when data can't fit in main memory entirely. (It is used when the size of input is large.)
    - In it, the storage device used are main memory (RAM) and secondary memory (Hard Disk).
    - Examples: External Merge Sort, External Radix Sort, Four Tape Sort.

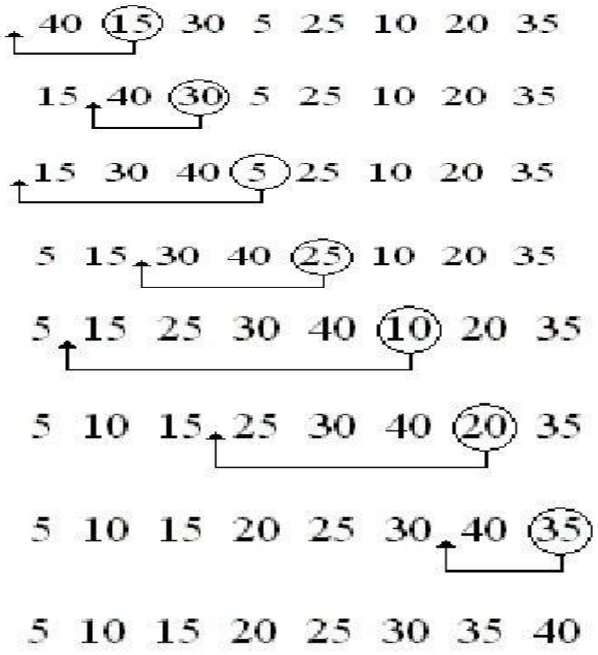
Chapter 7 Sorting

Insertion sort:

- Insertion sort is implemented by inserting a particular data item in its proper position.
- Any unsorted data item is kept on swapping with its previous data items until its proper position is not found.
- The number of swapping makes the previous data items to shift for the new data item to take its position in order.
- Once the new data item is inserted, the next data item after it is chosen for next insertion.
- The process continues until all data items are sorted.
- We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards.



- It is efficient for smaller data sets, but very inefficient for larger lists.
- Less efficient as compared to other more advanced algorithms such as quick Sort, heap sort, and merge sort.

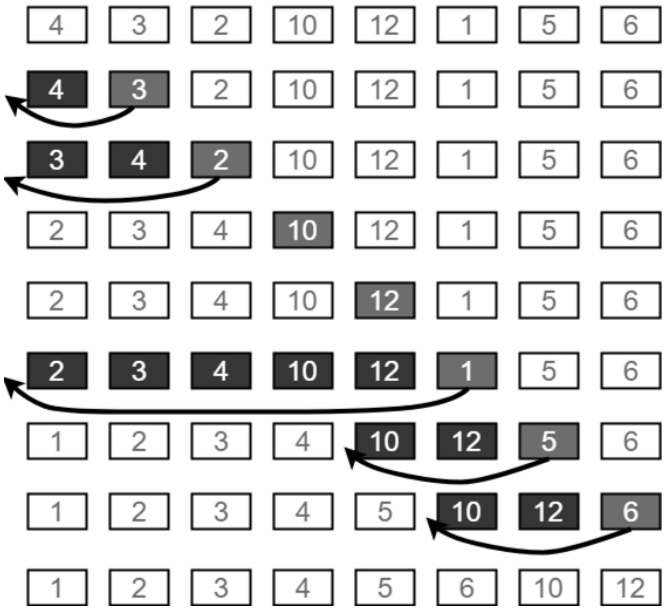


Algorithm:

Step 1 - If it is the first element, it is already sorted. Return 1;
Step 2 – Pick next element
Step 3 – Compare with all elements in the sorted sub-list
Step 4 – If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.
Step 5 – Insert the value
Step 6 – Repeat until list is sorted

```
//Insertion sort logic
 For( i = 1 ; i < size ; i++)
{
   temp = list[i];
   j = i – 1;
   while ((temp < list[j])&& (j ≥ 0))
{
    list[j + 1] = list[j];
    j = j – 1;
   }
   list[j + 1] = temp;
 }
```

Example 1:



Efficiency:
Nested loops

Worst Case : $O(n^2)$
Best Case : $\Omega(n)$
Average Case : $\Theta(n^2)$

Example 2:

Consider the following unsorted list of elements...

Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...

Move the first element 15 from unsorted portion to sorted portion of the list.

To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.

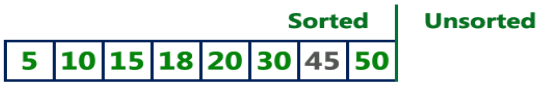
To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.

To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.

To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.



Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...

Chapter 7 Sorting

Selection Sort:

Selection Sort 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.

Algorithm:

Step 1 – Set MIN to location 0

Step 2 – Search the minimum element in the list

Step 3 – Swap with value at location MIN

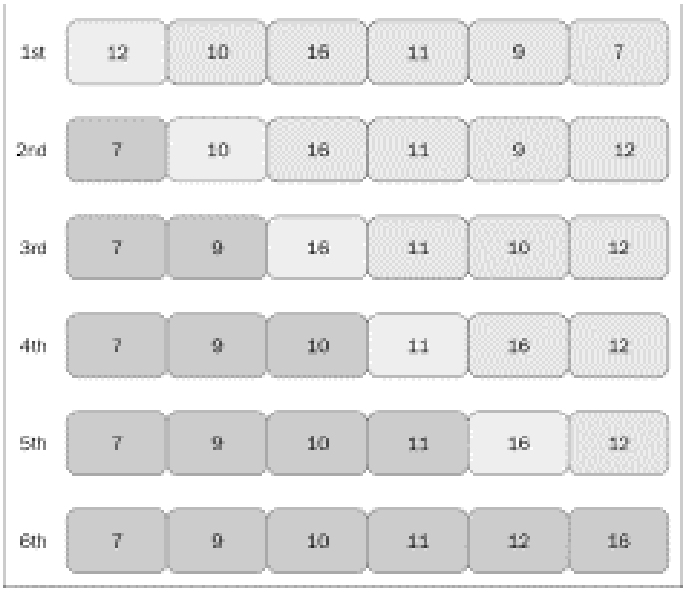
Step 4 – Increment MIN to point to next element

Step 5 – Repeat until list is sorted

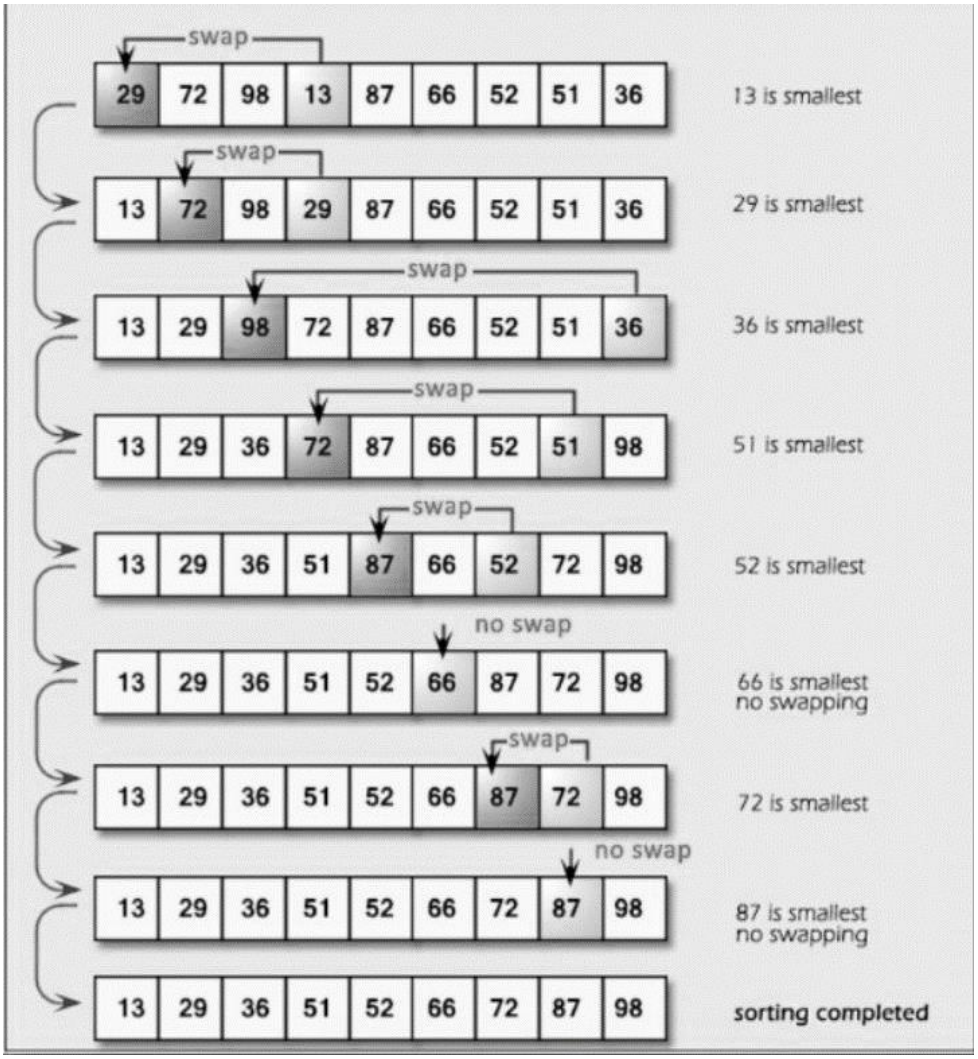
Worst Case : $O(n^2)$
Best Case : $\Omega(n^2)$
Average Case : $\Theta(n^2)$

```
//Selection sort logic

  for(i=0; i<size-1; i++){
    min =i;
      for(j=i+1; j<size; j++){
            if(list[j] < list[min])
            {
                min=j;

            }
      }
      if (min != i)
      {
        swap (list[i], list[min]);
      }
  }
```

Example 1:



Example 2:



Bubble Sort/ Exchange sort:

- a simple sorting algorithm which
  - repeatedly steps through the list to be sorted
  - compares each pair of adjacent items
  - swaps them if they are in the wrong order
  - The passing through the list is continued until the swapping is not required (i.e. the list sorted)
- it is a comparison sort
- It is called Bubble Sort because the data gradually bubbles up in its proper position
- In each pass at least one data is bubbled up in its proper position

```
for (i = 0; i<= n-1; i++)
{
    for (j = 0; j <= n-1-i; j++)
    {
        if (a[j] > a[j+1])
        {
            temp = a[j];
            a[j] = a[j +1];
            a[j +1] =temp;
        }

    }
}
```

Example: 6 1 3 2 7

First Pass:

$(\mathbf{6\ 1}\ 3\ 2\ 7)(\mathbf{1\ 6}\ 3\ 2\ 7), Swap\ since\ 6 > 1.$

$(1\ \mathbf{6\ 3}\ 2\ 7)(1\ \mathbf{3\ 6}\ 2\ 7), Swap\ since\ 6 > 3$

$(1\ 3\ \mathbf{6\ 2}\ 7)(1\ 3\ \mathbf{2\ 6}\ 7), Swap\ since\ 6 > 2$

$(1\ 3\ 2\ \mathbf{6\ 7}), does\ not\ swap$

Second Pass:

$(\mathbf{1\ 3}\ 2\ 6\ 7)$

$(1\ \mathbf{3\ 2}\ 6\ 7)(1\ \mathbf{2\ 3}\ 6\ 7), Swap\ since\ 3 > 2$

$(1\ 2\ \mathbf{3\ 6}\ 7)(1\ 2\ \mathbf{3\ 6}\ 7)$

$(1\ 2\ 3\ \mathbf{6\ 7})$

- List is already sorted, but our algorithm does not know. Hence one more pass to see if further swapping has to be done

Third Pass:

- (**1 2 3 6 7**), No swap up to the last comparison, hence the list is sorted

Algorithm:

The basic methodology of the working of bubble sort is given as follows:

$(a) In\ Pass\ 1, A[0] and\ A[1] are\ compared, then\ A[1] is\ compared\ with\ A[2],$

$A[2] is\ compared\ with\ A[3], and\ so\ on. Finally, A[N-2] is\ compared\ with\ A[N-1].$

$Pass\ 1\ involves\ n-1\ comparisons\ and\ places\ the\ biggest\ element\ at\ the$

$highest\ index\ of\ the\ array.$

$(b) In\ Pass\ 2, A[0] and\ A[1] are\ compared, then\ A[1] is\ compared\ with\ A[2],$

$A[2] is\ compared\ with\ A[3], and\ so\ on. Finally, A[N-3] is\ compared\ with\ A[N-2].$

$Pass\ 2\ involves\ n-2\ comparisons\ and\ places\ the\ second\ biggest\ element\ at\ the$

$second\ highest\ index\ of\ the\ array.$

$(c)\ In\ Pass\ n-1, A[0]\ and\ A[1]\ are\ compared\ so\ that\ A[0].$

Efficiency:

Nested loops and for n items, n possible swaps

Worst Case : $O(n^2)$
Best Case : $\Omega(n^2)$
Average Case : $\Theta(n^2)$

Example 1:

```
A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:
 (a) Compare 30 and 52. Since 30 < 52, no swapping is done.
 (b) Compare 52 and 29. Since 52 > 29, swapping is done.
     30, 29, 52, 87, 63, 27, 19, 54
 (c) Compare 52 and 87. Since 52 < 87, no swapping is done.
 (d) Compare 87 and 63. Since 87 > 63, swapping is done.
     30, 29, 52, 63, 87, 27, 19, 54
 (e) Compare 87 and 27. Since 87 > 27, swapping is done.
     30, 29, 52, 63, 27, 87, 19, 54
 (f) Compare 87 and 19. Since 87 > 19, swapping is done.
     30, 29, 52, 63, 27, 19, 87, 54
 (g) Compare 87 and 54. Since 87 > 54, swapping is done.
     30, 29, 52, 63, 27, 19, 54, 87
```

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

```
Pass 2:
 (a) Compare 30 and 29. Since 30 > 29, swapping is done.
     29, 30, 52, 63, 27, 19, 54, 87
 (b) Compare 30 and 52. Since 30 < 52, no swapping is done.
 (c) Compare 52 and 63. Since 52 < 63, no swapping is done.
 (d) Compare 63 and 27. Since 63 > 27, swapping is done.
     29, 30, 52, 27, 63, 19, 54, 87
 (e) Compare 63 and 19. Since 63 > 19, swapping is done.
     29, 30, 52, 27, 19, 63, 54, 87
 (f) Compare 63 and 54. Since 63 > 54, swapping is done.
     29, 30, 52, 27, 19, 54, 63, 87
```

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

```
Pass 3:
 (a) Compare 29 and 30. Since 29 < 30, no swapping is done.
 (b) Compare 30 and 52. Since 30 < 52, no swapping is done.
 (c) Compare 52 and 27. Since 52 > 27, swapping is done.
     29, 30, 27, 52, 19, 54, 63, 87
 (d) Compare 52 and 19. Since 52 > 19, swapping is done.
     29, 30, 27, 19, 52, 54, 63, 87
 (e) Compare 52 and 54. Since 52 < 54, no swapping is done.
```

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

```
Pass 4:
 (a) Compare 29 and 30. Since 29 < 30, no swapping is done.
 (b) Compare 30 and 27. Since 30 > 27, swapping is done.
     29, 27, 30, 19, 52, 54, 63, 87
 (c) Compare 30 and 19. Since 30 > 19, swapping is done.
     29, 27, 19, 30, 52, 54, 63, 87
 (d) Compare 30 and 52. Since 30 < 52, no swapping is done.
```

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

```
Pass 5:
 (a) Compare 29 and 27. Since 29 > 27, swapping is done.
     27, 29, 19, 30, 52, 54, 63, 87
 (b) Compare 29 and 19. Since 29 > 19, swapping is done.
     27, 19, 29, 30, 52, 54, 63, 87
 (c) Compare 29 and 30. Since 29 < 30, no swapping is done.
```

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

```
Pass 6:
 (a) Compare 27 and 19. Since 27 > 19, swapping is done.
     19, 27, 29, 30, 52, 54, 63, 87
 (b) Compare 27 and 29. Since 27 < 29, no swapping is done.
```

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

```
Pass 7:
 (a) Compare 19 and 27. Since 19 < 27, no swapping is done.
```

Merge sort:

- It is a divide and conquer algorithm
- At first we divide the given list of item
    - list is divided into two parts from middle
    - The process is repeated until each sub list contain exactly 1 item
- Now is the turn for sort and combine (conquer)
    - A list with a single element is considered sorted automatically
    - Pair of list is sorted and merged into one (i.e. approx. n/2 sub lists of size 2)
    - The sort and merge is keep on repeated until a single list of size n is found
- The overall dividing and conquering is done recursively
- To sort A[p .... r]: (p=starting index , r=ending index)

Divide Step:

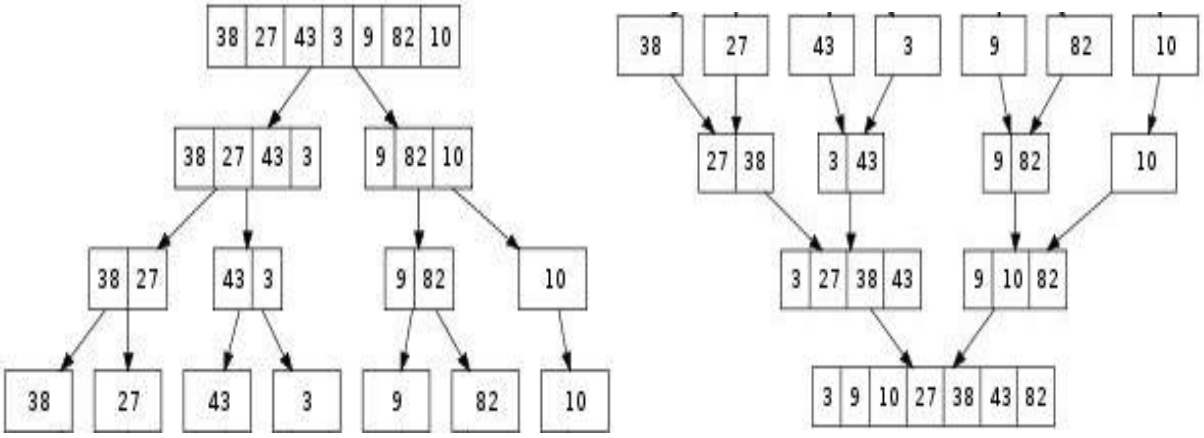
- If a given array A has zero or one element, simply return; it is already sorted.
- Otherwise, split A [p ... r] into two sub arrays A [p ... q] and A [q + 1 … r], each containing about half of the elements of A [p ... r]. That is,
- q is the halfway point of A[p .. r].

Conquer Step

- Conquer by recursively sorting the two sub arrays A [p ... q] and A [q + 1 … r].

Combine Step

- Combine the elements back in A [p … r] by merging the two sorted sub arrays A [p ... q] and A [q + 1 ... r] into a sorted sequence.
- To accomplish this step, we will define a procedure MERGE (A, p, q, r).



Divide Conquer

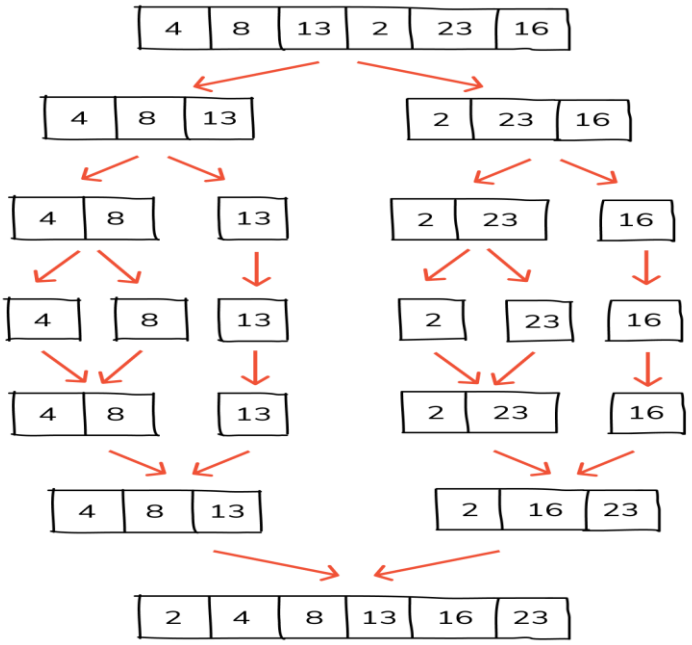
```
Algorithm MergeSort(Arr, start, end)
1. If start < end Then
2.   mid = (start + end)/2
3.   MergeSort(Arr, start, mid)
4.   MergeSort(Arr, mid + 1, end)
5.   Merge(Arr, start, mid, end)
```

```
Algorithm Merge(Arr, start, mid, end)
1.  temp = Create array temp of same size Arr
2.  i = start, j = mid + 1, k = 0
3.  While i <= mid and j <= end
4.    If Arr[i] > Arr[j] Then
5.      temp[k++] = Arr[j++]
6.    Else
7.       temp[k++] = Arr[i++]
8.  While i <= mid
9.     temp[k++] = Arr[i++]
10. While j <= end
11.    temp[k++] = Arr[j++]
12. Loop from p = start to end
13.   Arr[p] = temp[p]
```

Efficiency:
Divide and conquer (tree structure) : log n
However n sub-lists need to be sorted

$\boldsymbol{Worst\ Case}: \boldsymbol{O}(n * logn)$
$\boldsymbol{Best\ Case}: O(n * logn)$
$\boldsymbol{Average\ Case}: O(n * logn)$

Example 1:



Redix sort:

Radix sort is the linear sorting algorithm that is used for integers. In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.

The process of radix sort works similar to the sorting of students names, according to the alphabetical order.

Algorithm:

Step 1 – Find largest element in the given array and number of digits in the largest element.

Step 2 - Define 10 queues each representing a bucket for each digit from 0 to 9.

Step 3 - Consider the least significant digit of each number in the list which is to be sorted.

Step 4 - Insert each number into their respective queue based on the least significant digit.

Step 5 - Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.

Step 6 - Repeat from step 4 based on the next least significant digit.

Step 7 - Repeat from step 3 until all the numbers are grouped based on the most significant digit.

Example 1:

$$\boldsymbol{Worst\ Case}: \boldsymbol{O}(n)$$
$$\boldsymbol{Best\ Case}: \boldsymbol{O}(n)$$
$$\boldsymbol{Average\ Case}: \boldsymbol{O}(n)$$

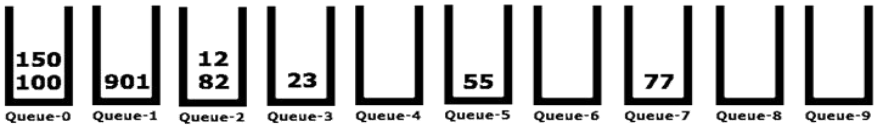
Consider the following list of unsorted integer numbers

82, 901, 100, 12, 150, 77, 55 & 23

Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.

Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

8**2**, 90**1**, 10**0**, 1**2**, 15**0**, 7**7**, 5**5** & 2**3**

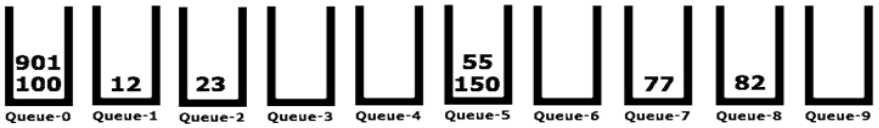


Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

1**0**0, 1**5**0, 9**0**1, **8**2, **1**2, **2**3, **5**5 & **7**7

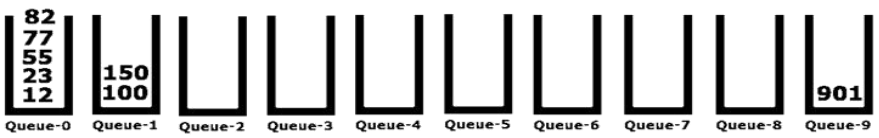


Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundres placed digit) of every number.

100, **9**01, 12, 23, **1**50, 55, 77 & 82



Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the incresing order.

Chapter 7 Sorting

Shell sort:

Shell sort is the generalization of insertion sort, which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

It first sorts elements that are far apart from each other by swapping and successively reduces the gap between the elements to be sorted. This gap is called as interval. The interval between the elements is reduced based on the sequence used. Some of the optimal sequences that can be used in the shell sort algorithm are:

- Shell's original sequence: $N/2$, $N/4$, … , 1
- Knuth's Formula $= h * 3 + 1$ →where $-h$ is interval with initial value 1
- Hibbard's increments: $1, 3, 7, 15, 31, 63, 127, 255, 511$ …
- Pratt: $1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, 36, 54, 81\ldots.$

Best Case: O(n*log n)
Average Case: O(n*log n)
Worst Case: $O(n^2)$

Algorithm:

Step 1 – for the size of array 'N'.

Step 2 – Divide the list into smaller sub-list of interval N/2.

Step 3 – Sort these sub-lists using **insertion sort.**

Step 3 – Repeat until complete list is sorted.

Shell Sort(a, n) // 'a' is the given array, 'n' is the size of array

- $for\ (interval\ =\ n/2;\ interval\ >=\ 1;\ interval\ /=\ 2)$
- $for\ (\ j\ =\ interval;\ j\ <\ n;\ j++)$
- $for\ (i\ =\ j - interval;\ i\ >=\ 0;\ i -=\ interval)$
- $if\ (\ a[i\ +\ interval]\ >\ a[i]\ )$
    - $break$
- $otherwise$
    - $swap\ (\ a[i\ +\ interval], a[i]\ )$
- $End\ Shell\ Sort$

Example:

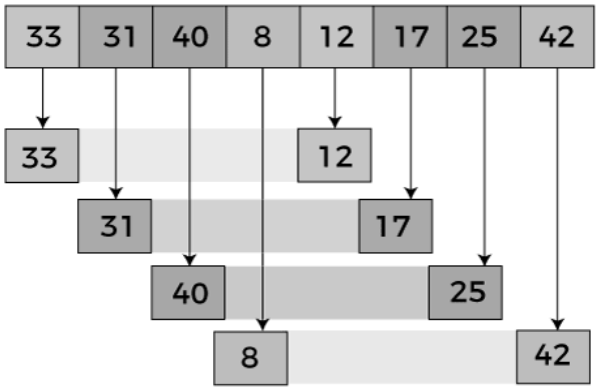
Let the elements of array are –

33	31	40	8	12	17	25	42

We will use the original sequence of shell sort, i.e., N/2, N/4...1 as the intervals.

Here, in the first loop, the element at the 0^{th} position will be compared with the element at 4^{th} position. If the 0^{th} element is greater, it will be swapped with the element at 4^{th} position. Otherwise, it remains the same. This process will continue for the remaining elements.

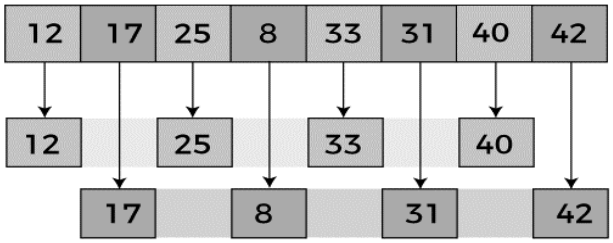
At the interval of 4, the sub lists are {33, 12}, {31, 17}, {40, 25}, {8, 42}.



After comparing and swapping, the updated array will look as follows –

12	17	25	8	33	31	40	42

In the second loop, elements are lying at the interval of 2 (n/4 = 2), where n = 8.



After comparing and swapping, the updated array will look as follows –

12	8	25	17	33	31	40	42

In the third loop, elements are lying at the interval of 1 (n/8 = 1), where n = 8.

12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

Chapter 7 Sorting

Heap sort as a priority queue:

Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heap sort algorithm uses one of the tree concepts called Heap Tree. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list. Heap sort is the in-place sorting algorithm.

Algorithm:

Step 1 - Construct a **Binary Tree** with given list of Elements.

Step 2 - Transform the Binary Tree into **Max Heap.**

Step 3 - Delete the root element from Max Heap using **Heapify** method.

Step 4 - Put the deleted element into the Sorted list.

Step 5 - Repeat the same until Max Heap becomes empty.

Step 6 - Display the sorted list.

Heapify (a, n) // 'a' is the given array, 'n' is the size of array

```
heapify (a , n)
{
    //To Build Max heap
    for (i = n/2; i>= 1 ; i--)
    {
        Maxheapify(a, n, i);
    }
    // To Delete
    for (i = n; i>= 1; i--)
    {
        swap (a[1],a[i]);
        Maxheapify(a, n, 1);
    }
}
```

```
Maxheapify(a, n,i)
{
    int largest = i;
    int l =(2*i);
    int r = (2*i) +1;
    while (l <= n && a[l] > a[largest])
    {
        largest = l;
    }
    while (r <= n && a[r] > a[largest])
    {
        largest = r;
    }
    if (largest != i)
    {
        swap(a[largest], a[i]);
        Maxheapify(a, n, largest);
    }
}
```

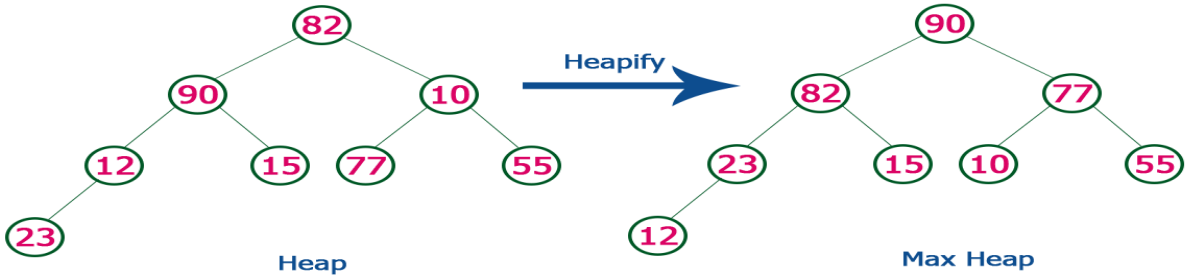
$$\boldsymbol{Worst\ Case}: \boldsymbol{O}(n \log n)$$
$$\boldsymbol{Best\ Case}: \boldsymbol{O}(n \log n)$$
$$\boldsymbol{Average\ Case}: \boldsymbol{O}(n \log n)$$

Example 1:

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

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

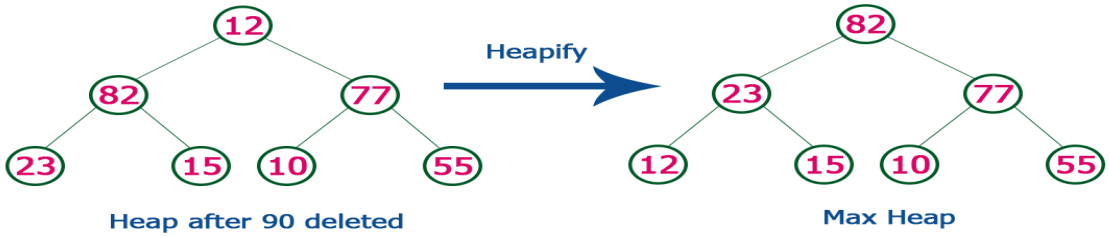
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

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

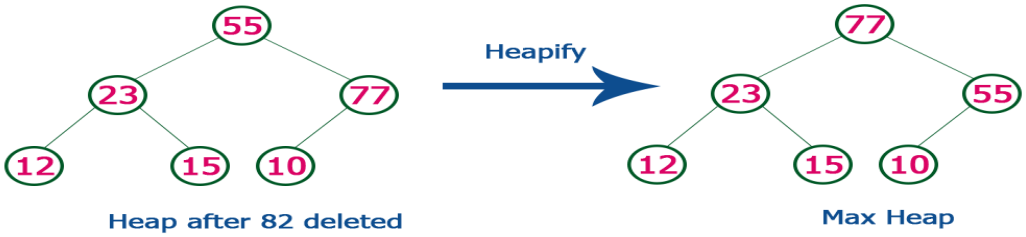
Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

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

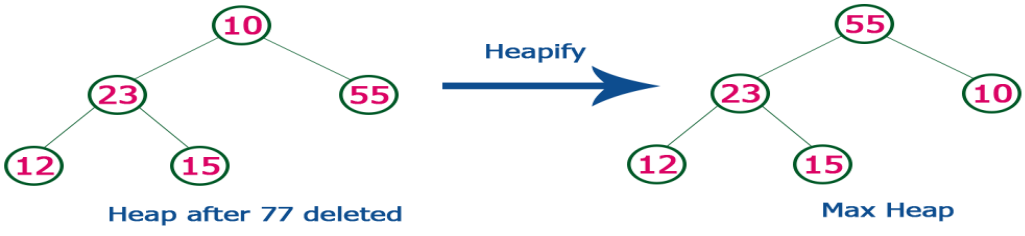
Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

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

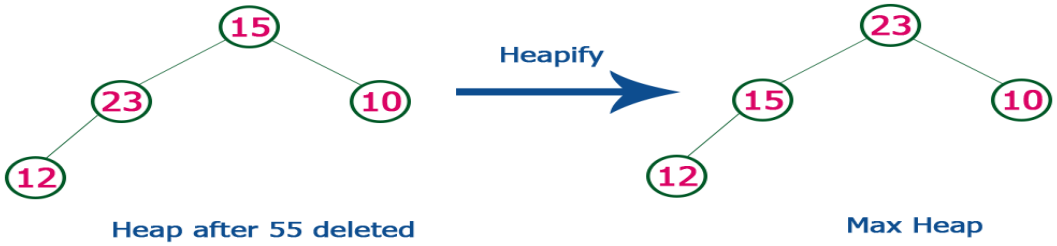
Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



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.

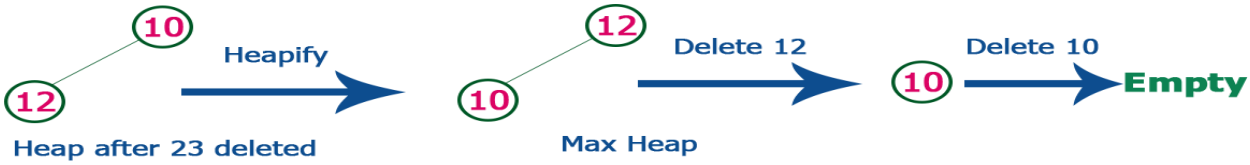
12, 15, 10, 23, **55, 77, 82, 90**

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

Big 'O' notation and Efficiency of sorting:

Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity. That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big-O Analysis of Algorithms:

We can express algorithmic complexity using the big-O notation. For a problem of size N:

- A constant-time function/method is "order 1" : O(1)
- A linear-time function/method is "order N" : O(N)
- A quadratic-time function/method is "order N squared" : O(N^2)

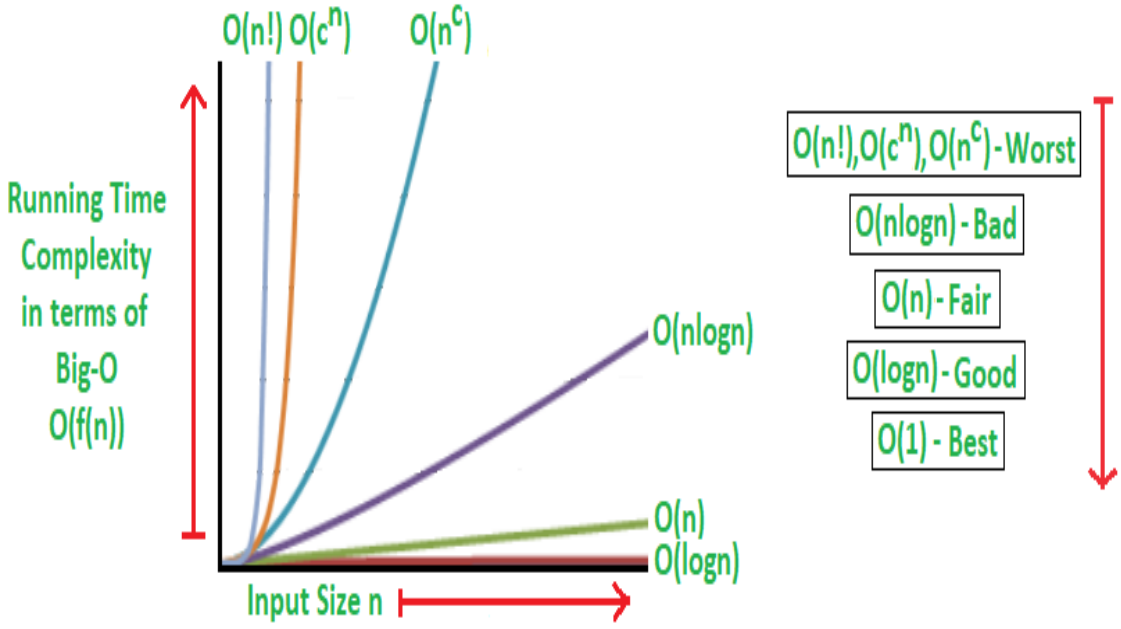
Chapter 7 Sorting

The general step wise procedure for Big-O runtime analysis is as follows:

1. Figure out what the input is and what n represents.
2. Express the maximum number of operations, the algorithm performs in terms of n.
3. Eliminate all excluding the highest order terms.
4. Remove all the constant factors.

Runtime Analysis of Algorithms:

- $A\ logarithmic\ algorithm - O(logn)$
  $Runtime\ grows\ logarithmically\ in\ proportion\ to\ n.$
- $A\ linear\ algorithm - O(n)$
  $Runtime\ grows\ directly\ in\ proportion\ to\ n.$
- $A\ superlinear\ algorithm - O(nlogn)$
  $Runtime\ grows\ in\ proportion\ to\ n.$
- $A\ polynomial\ algorithm - O(n^c)$
  $Runtime\ grows\ quicker\ than\ previous\ all\ based\ on\ n.$
- $A\ exponential\ algorithm - O(c^n)$
  $Runtime\ grows\ even\ faster\ than\ polynomial\ algorithm\ based\ on\ n.$
- $A\ factorial\ algorithm - O(n!)$
  $Runtime\ grows\ the\ fastest\ and\ becomes\ quickly\ unusable\ for\ even\ small\ values\ of\ n.$



Chapter 7 Sorting

Algorithmic Examples of Runtime Analysis:

- $Logarithmic\ algorithm - O(logn) - Binary\ Search.$
- $Linear\ algorithm - O(n) - Linear\ Search.$
- $Super\ linear\ algorithm - O(nlogn) - Heap\ Sort, Merge\ Sort.$
- $Polynomial\ algorithm - O(n\^c) - Bubble\ Sort, Selection\ Sort, Insertion\ Sort, Bucket\ Sort.$
- $Exponential\ algorithm - O(c\^n) - Tower\ of\ Hanoi.$
- $Factorial\ algorithm - O(n!) - Determinant\ Expansion\ by\ Minors.$