Chapter 3 Sorting

Sorting is arranging the elements in increasing or decreasing order. Several algorithms exists for performing sorting. Some of them are discussed in this chapter.

3.1 Bubble sort

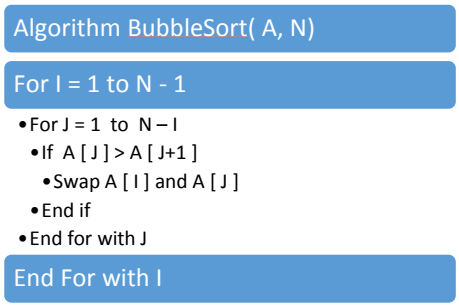
Bubble sort is the most primitive sorting algorithm. The complexity of bubble sort is n^2. It compares the adjacent elements in the array and brings largest element in the end (if sorting in ascending order).

Take the example below. To sort 5,3,2,7,1, first 5 and 3 are compared. Since 5 is large so they are swapped. Then the array becomes 3,5,2,7,1. Now 5 is compared with 2. Again 2 is small so swap is performed between 2 and 5. Array becomes 3,2,5,7,1. Now 5 and 7 are compared. Since 7 is large so no swap is performed. Now 1 is compared with 7. 1 is small so 1 and 7 are swapped. Array becomes 3,5,2,1,7. Again from the beginning adjacent items will be compared. The working is shown in table below.

5	3	2	7	1	Let this be the array to sort. The size of the array is N=5. Take two iterative variables I and J.
I =1, J=2 to (N-I=4) , compare A[I] with A[J] where J will increase by one each time.					
5	3	2	7	1	Compare A[1] > A[2] = 5> 3, which is true so swap A[1] with A[2]
3	5	2	7	1	Compare A[2] > A[3]> = 5> 2, true so swap A[2] with A[3]
3	2	5	7	1	Compare A[3] > A[4] = 5> 7, false so do not do anything
3	2	5	7	1	Compare A[4] > A[5] = 7> 1, true so swap.
3	2	5	1	7	Now J becomes 5 but loop is till 5-I=4 So J loop ends
Now I =2 and J iterates from 1 to (N-I = 5-2 = 3)					
3	2	5	1	7	Compare 3 > 2, true so swap. For J=1
2	3	5	1	7	Compare 3> 5 false, for J=2
2	3	5	1	7	Compare 5>1 true swap, for J=3

2	3	1	5	7	Now J tries to become 4 which makes the J loop end.
Now I =3 and J = 1 to 5-3=2					
2	3	1	5	7	Compare 2>3 false for J=1
2	3	1	5	7	Compare 3>1 true so swap for J=2
2	1	3	5	7	Now J loop ends
Now I=4 and J = 1 to 5-4=1					
2	1	3	5	7	Compare 2>1, true so swap for J=1
1	2	3	5	7	Sorted

The algorithm for bubble sort is given below. There are two loops the first I loop controls the number of total iterations. The inner loop J is used to compare adjacent elements. So adjacent elements are compared J times and J loop runs I times.



3.2 Insertion sort

Complexity of Insertion sort is also N^2 like bubble sort. It assumes that the first element is sorted and then tries to put second element in place with respect to first. Then first two are sorted and third element is placed either before first or in middle of first and second or after second element depending on the relation that a[3] is more or less than a[2] and a[1]. And so on for all the elements.

Insertion sort is so called because we insert the element in its proper position in the first part of the array.

The algorithm for insertion sort is given below. The item which is to be inserted in its position in first part of the list is stored in target. Then target is compared with all the elements in J loop. Where ever the element is larger than a[J] we copy it at J+1 location

after shifting all elements towards right. So the algorithm can be divided into two parts: Selection of target, comparison and shifting to put the element in correct order.

```
Algorithm InsertionSort ( A, N)
For I = 2 to N
  •J= I - 1
  •Target = A [ I ]
  •While( A [ J ] > Target) and J >=1
    •A [ J+1 ] = A [ J ]
    •J =J-1
  •End while
  •A [ J + 1] = target
End for
```

Example: The list to be sorted is 5,3,2,7,1. The element 5 is assumed to be sorted. First, 3 is target. Now, 3 is compared with 5. 5 is more than 3 so shift 5 to location J+1 and insert target=3 at j location. Similarly Go on with target as 2. Now 2 is compared with 5. 5 is large so shift it to j+1. Then compare 2 with 3, 3 is large so shift it to j+1. Now no more elements are left to compare so copy target at j. So on.

1	2	3	4	5	Index
5	3	2	7	1	Let this be the array to sort. I =2, J=1 , target = 3
3	5	2	7	1	Compare 5 > 3, true. Shift 5 and insert 3
3	5		7	1	Now I =3 , J=2, target = 2
3		5	7	1	Compare 5> 2 true so shift 5 to location 3, for J=2
	3	5	7	1	Compare 3> 2 true so shift 3 to location 2, for J =1
2	3	5	7	1	J becomes 0 while loop is quit Insert A[J+1] = target => A[0+1] = target
					Now I = 4, J =3, target = 7
2	3	5	7	1	Compare 5>7 false so do not enter while loop
2	3	5	7		Now I = 5, J = 4, target=1
2	3	5		7	Compare 7>1, shift 7
2	3		5	7	Compare 5>1, shift 5

2		**3**	**5**	**7**	**Compare 3>1, shift 3**
	2	**3**	**5**	**7**	**Compare 2 >1, shift 2**
1	**2**	**3**	**5**	**7**	**J becomes zero so while loop is quit Copy target at A[J+1]**

3.3 Selection sort

In this algorithm we select the smallest element from the whole list and place it in its proper position. Like this one by one the next smallest element in the list is selected and copied to its proper position.

The algorithm for selection sort is as follows. The LOC variable is the location of net smallest element in the list. The loop I moves from 1 to N-1 because N elements are to be moved to its proper location. The J loop moves from I to N because first I elements will be the already sorted elements. Initially the LOC points to first element. Now LOC is compared with each of Jth element till the end of list. In the end LOC will have the location of smallest element in the list. Then simply exchange the Ith element with LOC element. Repeat this N times for each element.

```
Algorithm Selection Sort (A, N)
For I=1 to N-1
  •Loc = I
  •For J= I+1 to N
    •If A[Loc] > A[J]
      •Loc =J
  •End for
  •Temp= A[loc], A[loc]= A[I] , A[I] = Temp
End for
```

As can be seen in the following example, the list to be sorted is 5,3,2,7,1. LOC is 1 and I is also 1. Compare 5 with 3. 3 is smaller so LOC becomes 2. The compare 3 with 2. 2 is smaller so LOC becomes 3. Then 2 is compared with 7. 7 is large so LOC remains 3. Compare 2 with 1. 1 is smaller so LOC becomes 5. Now Interchange A[LOC] with A[I]. So 5 is exchanged with 1. Now the smallest element 1 is at its proper position. And so on for the next smallest element 2.

1	2	3	4	5	Index
5	3	2	7	1	Let this be the array to sort. I =1, J=2 , Loc=1
	3<5 ,T	2<3, T	7< 2, F	1< 2, T	
	Loc=2	Loc=3		Loc=5	Swap A[I] with A[Loc]
1	3	2	7	5	I =2, J=3, Loc=2
		2<3, T	7<2, F	5<2, F	
		Loc=3			Swap A[I] with A[Loc]
1	2	3	7	5	I= 3, J=4, Loc=3
			7<3, F	5<3, F	No change in array
1	2	3	7	5	I=4, J =5, Loc=4
				5<7, T	
				Loc=5	Swap A[I] with A[Loc]
1	2	3	5	7	

3.4 Merge Sort

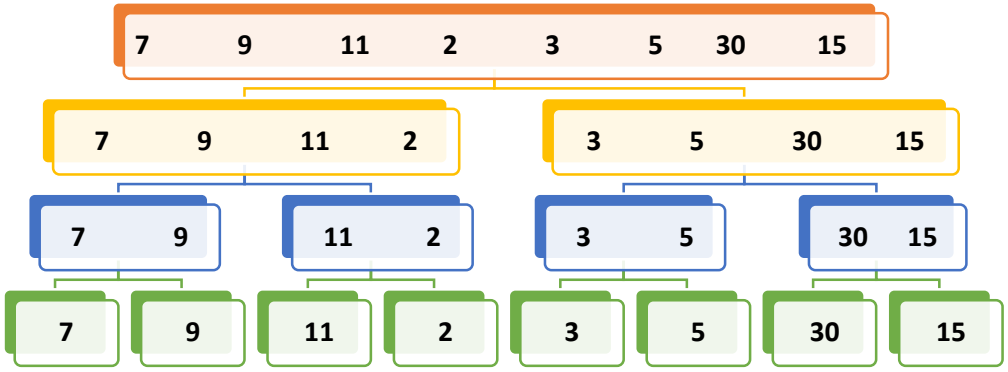
Merge sort splits the array into parts. The goes on partitioning the array till single elements are left in all the partitions. Then it merges the adjacent element in sorted order.

Given a sequence of n elements (also called keys) a[1]……a[n], the general idea is to imagine them to split into two sets:

a[1]….a[n/2] and a[n/2 +1].…….a[n].

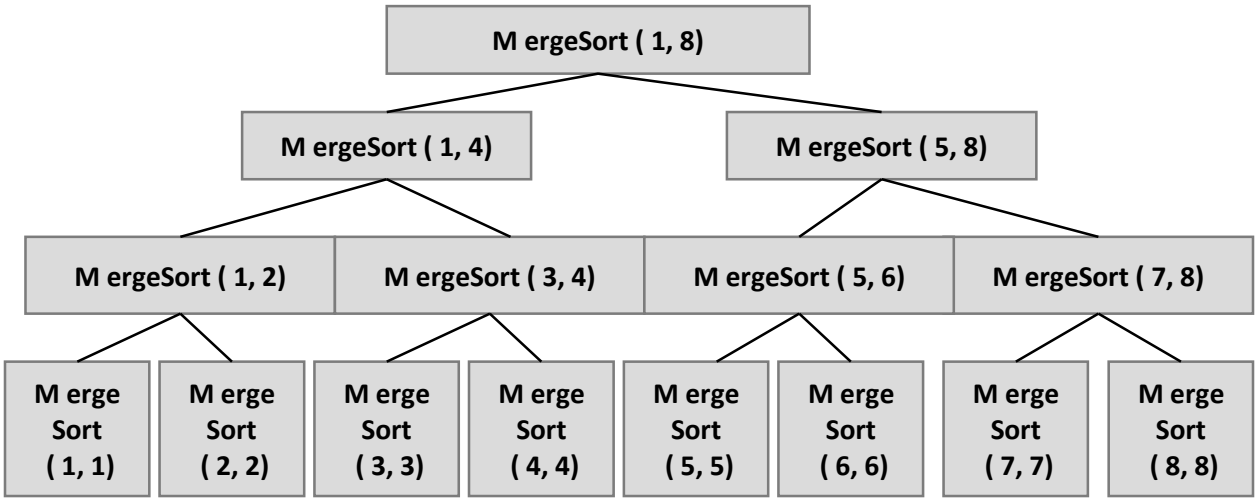
Each set is **individually sorted** and resulting sorted sequences are **merged** to produce single sorted sequence of n elements.

Given the list below 7,9,11,2,3,5,30,15. The beginning of the list is 1 and end is 8. So calculate mid = (beg+end)/2= (1+8)/2= 4.5. Drop the decimal part. So mid is 4. So split the list after 4. So 7,9,11,2 are in part 1 and 3,5,30,15 in part 2. Not split part 1 into two parts with mid=(1+4)/2. Split part 2 in two parts with mid=(5+8)/2. So on keep dividing till single elements are left.



Once the split is complete now start merging. Merge 7 and 9 because they are parts of same list. & is smaller so keep it first. So new list is 7,9. Now merge 11 and 2. 2 is smaller so after merge we het 2,11. Now merge 3,5. Then 30,15 becomes 15,30. Now merge 7,9 with 2,11. Compare 2 with 7. 2 is small so it becomes first element of new list. The compare 7 with 11. 7 is smaller so it becomes second element of new list. Now compare 9 with 11. 9 is small so it becomes third element of new list. Then 11 becomes fourth element of new list. So the new list is 2,7,9,11. Now merge 3,5 with 30,15. The list after merging is 3,5,15,30. Then merge 2,7,9,11 with 3,5,15,30. This will generate the sorted list.

The algorithm for merge sort has two parts. Merge sort function that splits the list recursively into two parts. And function merge that merges the adjacent lists to generate the final list. The call tree of merge sort is given below. After the last calls the merging starts from 1 with 2, 3 with 4, 5 with 6 and 7 with 8. Then 1,2 with 3,4 and 5,6 with 7,8. Then 1,2,3,4 with 5,6,7,8 to generate the final list.



Merge sort is called with low =1 and high = upper bound of the list. Mid is calculated from low and high. The mergesort is called recursively two times with 1,mid and mid+1,high. Then after the number of elements is one in each subarray then merge is called with low, mid and high.

```
Algorithm MergeSort(low,high)
If(low == high) then// small(p)
    •return
else
    •mid=[(low+high)/2]
    •MergeSort(low,mid)
    •MergeSort(mid+1, high)
    •Merge(low, mid, high)
end
```

```
Algorithm Merge(low, mid, high)
h=low; i=low; j=mid+1;
while((h<=mid) and (j<=high)) do
    •if (a[h]<a[j]) then  {b[i]=a[h]; h=h+1; }
    •else { b[i]=a[j]; j=j+1; }
    •i++;
If(h>mid) then        for k=j to high do   { b[i]=a[k]; i=i+1; }
else    for k=j to high do { b[i]=a[k]; i=i+1; }
return b;
```

In merge copy the smallest element from both lists into the new list and go on incrementing the pointer to point to next element. The first list is from low to mid and second list is from mid+1 to high. The loop h is for first list and loop j is for second list. If the element at h is smaller than element at j the copy element at h into new list. And increment h by 1. IF element at j is smaller then copy that element in new list and increment j by 1. In the end either the h or j list will contain some elements copy those left over elements in new list in same order.

3.5 Quick sort

In Mergesort the division of list into two parts is done statically at mid. In quicksort the division point is dynamically decided by a partition function. An element is selected as pivot element. This element is then compared with all other elements and while making the comparisons the pivot element is moved in such a manner that all the element smaller than pivot are moved to its left. And all the elements larger than pivot are moved to its right. Pivot then becomes the point of partition. Then new pivot is selected for both sides and sorting is done for the sublists independently.

Taking example of 5,3,2,7,1 array.

Select 5 as pivot. Compare 5 with all elements from left to right till we find an element larger than 5. In this case 5 is first compared with 3 then 2 and then 7. 7 is larger so we stop. Now compare 5 with all element from right to left till we find a element smaller than 5. 5 is compared with 1 and it is smaller than 5 so we stop. Now location of 7 is less than location of 1. So interchange 7 and 1. Now interchange 5 and 1. So we get – 1,3,2,5,7. Now 5 becomes the point of partition.

For sublist 1,3,2, Select 1 as pivot. Compare all elements from left to right till we find larger. 3 is large so we stop. Now try to find a element smaller than 1 from right to left. In this sublist 1 is the smallest element. So location of 1 is less than location of 3. So no swapping is performed. And 1 becomes the point of partition. So we get 1 and 3,2 as two lists. Now sort 3,2.

Now 3 is compares with all elements from left to right till a larger element is found. No element is larger so we stop. Now find a element smaller than 3 from right to left. 2 is smaller so we stop. Now interchange 3 with 2.

Now we look at the list the list is sorted. There is no merging required.

The algorithm for Quick sort is given below. It consists of two parts. Quicksort and partition. In quicksort we call partoition to find the partitioning place. Then quick sort is invoked for both sublists from p(low) to j-1 (place of partition -1) and j+1 to q(high).

```
Quicksort(A, p, q)
•// sorts the elements a[p]….a[q] for global array a[1:n]. A[n+1] = ∞.
if p >= q
• then return
else //divide p into two sub-problems
•j = Partition(A, p, q+1)
•//j is pivot element. Solve the below two sub-problems.
• Quicksort(A, p, j- 1)
• Quicksort(A, j + 1, q)
```

In partition the I loop is used to find a number larger than the pivot and j loop is used to find a number smaller than the pivot. If i< j then interchange element at I with element at j and continue finding larger and smaller elements respectively. When I becomes greater than j, then exchange jth element with pivot. And the location j becomes the partitioning value. The algorithm is given below.

```
Partition(A, m, last)
• // within a[m],a[m+1].....a[p-1] the elements are rearranged in such a manner that if initially
  t=a[m] then after completion a[q]=t for some q between m and p-1, a[k] <=t for m<=k<q  and
  a[k]>= t for q <k<p. q is returned.
v=a[m];   i=m;   j=last+1;
do
  • do  { i=i+1} while(a[i]< v);
  • do {j=j-1}  while (a[j] > v);
  • if(i<j)  then      Exchange A[i] and A[j]
while( i < j);
A[m]=A[j];  A[j]=v;
return j
```

3.6 Comparison of various Sorting algorithms

Criteria	Bubble	Insertion	Selection	Merge	Quick
Idea	Compare adjacent elements	Insert elements in later part at proper position in beginning of list.	Select the smallest element and place it in proper position.	Split the array into two parts and merge in sorted order.	Split the array in such a way that all elements in left sublist are smaller and right sublist are larger than the element at splitting position.
Complexity	N*n	N*N	N*N	Nlogn	Nlogn or N*N depending

					on where the array is split.