Sorting

"Sorting refers to the operation of rearranging data in either ascending or descending order."

* The data may be numerical data or character data.

The sorting methods are classified into two types: 'internal sorting' and 'external sorting'.

In Internal Sorting, all the data elements to be sorted are present in the main memory.

External sorting techniques are applied to large data sets which reside on secondary storage devices and can not be completely fit in main memory. (now we restrict our to only internal sorting methods).

There are many sorting methods available. But no method for sorting is best in all cases. The factors to be considered while choosing a 'sorting technique are,

- Programming time of the sorting technique
- Execution " " " " "
- Number of comparisons required for sorting the list
- Main or Secondary memory space needed for sorting

That means different applications requires different sorting methods

Various sorting techniques are:

- Bubble Sort (or) Exchange Sort
- Selection Sort
- Quicksort (or) partition Exchange Sort
- Insertion Sort
- Merge Sort.

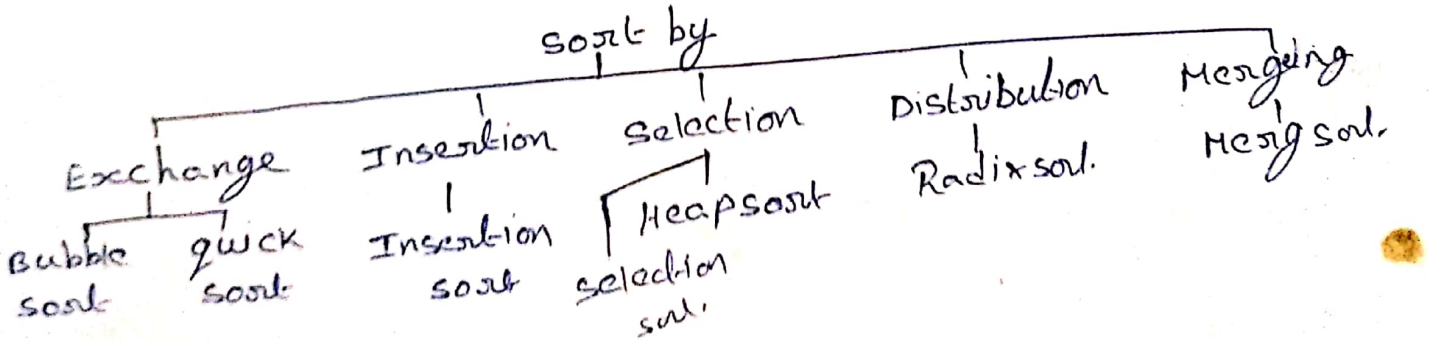
Note: A sorting method is said to be stable when it has minimum no. of swaps.

Stable sorting techniques are -

Bubble Sort, Insertion Sort, Selection Sort

Unstable Sorting techniques are -

Quick Sort, Merge Sort.



Bubble Sort (or) Exchange Sort

The simplest and the most widely used sorting technique is "Bubble Sort".

This method compares the two adjacent (consecutive) elements of the list. If they are not in order, the two elements will be interchange. If they are in order, the two elements will not be interexchanged. This process continues (n-1) times for sorting an array of size 'n'. These ends one pass. During the first pass the largest/smallest element will be moved to Nth position. The second pass will be continued with first (n-1) elements. Repeat this process till all the elements are in sorted order.

"Since, each places an element into its proper position, a list of 'n' elements requires (n-1) passes."

The procedure for bubble sort to sort 'n' elements in ascending order is,

Step 1: first compare two elements at time, starting with the first two elements.

Step 2: If the ~~top~~ first element is larger than second element then exchange the two elements.

Step 3: Go down one element and compare that element to the element that follows it.

Step 4: Continue this process till the end of list.

[During this process largest element moved to n^{th} position]

~~Step 5: leave~~ the last element of ~~sorting~~ list

~~can~~ then repeat ~~step~~ steps 1 thru 4.

Step 5: Repeat steps 1 thru 4 $(n-1)$ times by leaving the last element of sorting list each time.

Analysis (or) time complexity for bubble sort:

To sort 'n' elements bubble sort requires $(n-1)$ passes

(first) pass 1 requires $(n-1)$ comparisons

pass 2 performs $(n-2)$ "

$(n-k)$ "

pass K "

pass $(n-1)$ " 1 comparison

The total no of comparisons $= 1+2+3+\cdots+n-1$

$= \frac{n(n-1)}{2} = \frac{n^2-n}{2} \approx \frac{n^2}{2}$

$\therefore O(n^2)$

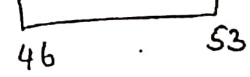
For example, Consider the list of 5 numbers.

a[0]	a[1]	a[2]	a[3]	a[4]
25,	53,	46,	37,	14

The following comparisons are made on the first pass.

a[0] with a[1] - 25 with 53 - (25 < 53) - No interchang

a[1] with a[2] - 53 with 46 - (53 > 46) - Interchange



a[2] with a[3] - 53 with 37 - (53 > 37) - Interchange

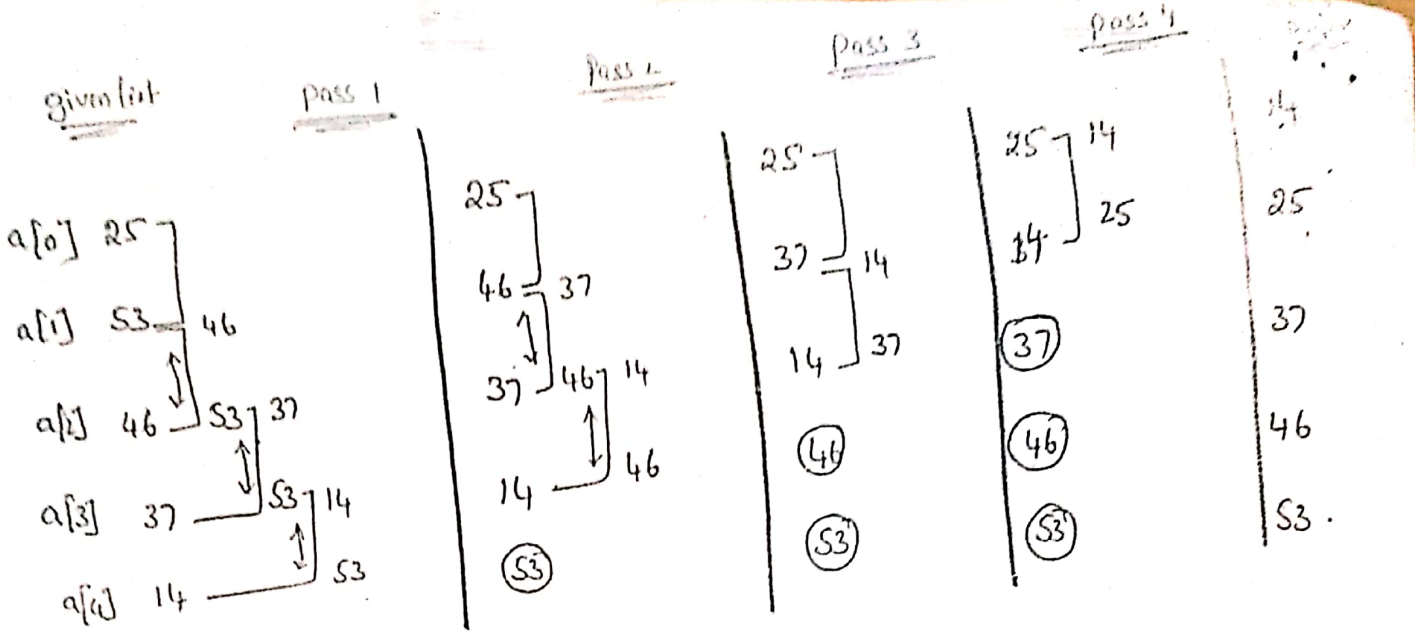
a[3] with a[4] - 53 with 14 - (53 > 14) - Inter change

After the first pass, the list is in the following order.

a[0]	a[1]	a[2]	a[3]	a[4]
25	46	37	14	53

" The first largest element '53' is in proper position i.e. at A[n-1] after the first pass." In general " k^{th} largest element is in its proper place i.e. at A[n-k] after pass 'k'."

The complete set of passes in bubble sort method is, ×



Algorithm: Bubble Sort (A, n)

/* A is an array of n elements. */

Step 1: Input n elements of an array A

" 2: $i \leftarrow 1$

" 3: Repeat through step 6 while ($i < n$)

" 4: $j \leftarrow 0$

" 5: Repeat through step 6 while ($j < n-i$)

" 6: if ($A[j] > A[j+1]$)

6.1: temp = A[j]

6.2: A[j] = A[j+1]

6.3: A[j+1] = temp

" 7: Display the sorted elements of array A.

" 8: Exit.

Note: The time complexity for bubble sort is $O(n^2)$.

Selection Sort is the easiest method to sort a list of elements.

In Selection Sort method first find the smallest element in the list and swap it with the first element position. Then find the second smallest element in the list and it is swapped with the second element in the list. The process of searching the next smallest is repeated, until all the elements in the list have been sorted in ascending order.

i.e. The selection sort searches all of the elements in a list until it finds the smallest element. It swaps this with the first element in the list. Next it finds the smallest of the remaining elements and swaps it with the second element and so on.

The procedure to sort 'n' elements in ascending order is,

Pass 1: Find the position (minpos) of the smallest element in the list of 'n' elements.

Then interchange A[minpos] & A[0].

i.e A[0] is sorted means it is in proper place.

Pass 2: Find position (minpos) of smallest in the sublist of (n-1) elements Then interchange A[minpas] & A[1]. Now A[0] & A[1] are sorted i.e. A[0] ≤ A[1].

:

Pass (n-1): Find the position (minpos) of the smallest element in A[n-2], A[n-1]. Then interchange A[minpos] & A[n-2]. Now, A[0], A[1], ... A[n-2] are sorted means A[n-1] is also in proper place.

Thus "the list of 'n' elements are sorted after (n-1) passes."

for example, Suppose an array 'A' contains 5 elemts,

65 32 45 12 26.

In first pass, the smallest element is searched by making of 4 Comparisons and the smallest element is 12. Then it is swapped with the first element i.e- 65. After the first pass the elements are

12 32 45 65 26.

```
#include <stdio.h>
void ssort(int[], int);
main()
{
  int a[100], n, i;
  printf("enter the no. of elements:");
  scanf("%d", &n);
  printf("enter the list of elements:");
  for (i=0; i<n; i++)
    scanf("%d", &a[i]);
  ssort(a, n);
  printf("\nThe sorted list is:");
  for (i=0; i<n; i++)
    printf("%d", a[i]);
}
```

```
void ssort(int A[], int n)
{
  int i, j, min, minpos, t;
  for (i=0; i<n-1; i++)
  {
    min = A[i];
    minpos = i;
    for(j=i+1; j<n; j++)
    {
      if (A[j] < min)
      {
        min = A[j];
        minpos = j;
      }
    }
    t = A[i];
    A[i] = A[minpos];
    A[minpos] = t;
  }
}
```

Analysis:

To sort 'n' elements selection sort requires (n-1) passes.

In pass1, process to select first minimum element requires (n-1) comparis

In pass2, process & select second minimum element " (n-2) "

In pass3, " " " " " (n-3) "

:

In passk, " " " " " 1 "

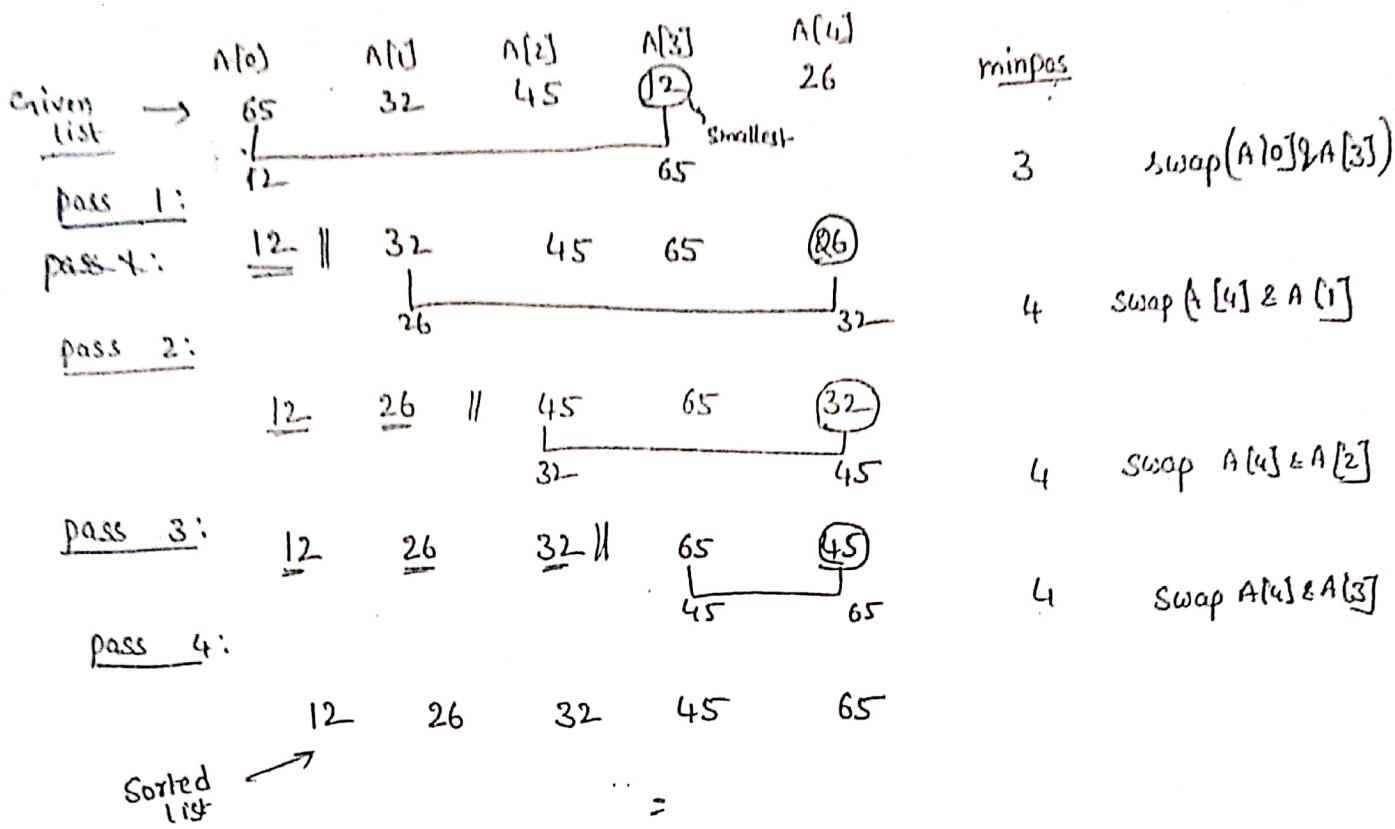
:

In pass(n-1), " " " " " 1

The total no. of comparisons = $(n-1)+(n-2)+\cdots+3+2+1$

$= \frac{n(n-1)}{2} = \frac{n^2-n}{2} = O(n^2)$

The complete set of passes in selection



Algorithm: Selection - Sort

Step 1: Input n elements of on array A

" 2: $i \leftarrow 0$

" 3: Repeat through step 7 while $(i < n-1)$

" 4: $j \leftarrow i+1$, $min \leftarrow A[i]$, $minpos \leftarrow i$:

" 5: Repeat through step 6 while $(j < n)$

" 6: if $(A[j] < min)$

6.1 : $min = A[j]$

6.2 : $minpos = j$

" 7: Interchange $A[i]$ & $A[minpos]$

" 8: Display the sorted element of array A

" 9. Exit

Insertion sort

This is a (searching) method in which insertion of an element is done at appropriate place in sorted list.

The main idea of Insertion sort is to consider each element at a time, into the appropriate position relative to the sequence of previously ordered elements, such that the resulting sequence is also ordered.

In each pass an item is compared with its predecessors and if it is not in the right position, it is placed in the right place among the elements being compared.

Suppose an array 'A' with 'n' elements A[0], A[1], ..., A[n-1] is in memory. The insertion sort algorithm scans A from A[0] to A[n-1], inserting each element A[k] into proper position in the previously sorted subarray A[0], A[1], ... A[k-1].

The procedure is, consider the first element, for only one element no sort is required because it is trivially sorted so the procedure starts with second element.

pass1 – A[1] is inserted either before or after A[0] so that A[0], A[1] is sorted

pass2 – A[2] is inserted into position by comparing with A[0], A[1] so that A[0], A[1], A[2] are sorted.

passn-1 – A[n-1] is inserted into its proper position in A[0], A[1], .. A[n-2] so that all the n elements are sorted.

* Simply, in ~~each~~ pass k - A[K] is Compared with previous elements [A[0], A[1], .. A[K-1]] and inserted after the element which is less than or equal to the A[K].

Example: Consider the list of element-s

5, 3, 7, 2, 8, 1, 4

To sort this list in ascending order using insertion method, the following steps [passes] are followed one by one.

Pass		Data		Output
1	–	5, 3	–	3, 5
2	–	3, 5, 7	–	3, 5, 7
3	–	3, 5, 7, 2	–	2, 3, 5, 7
4	–	2, 3, 5, 7, 8	–	2, 3, 5, 7, 8
5	–	2, 3, 5, 7, 8, 1	–	1, 2, 3, 5, 7, 8
6	–	1, 2, 3, 5, 7, 8, 4	–	1, 2, 3, 4, 5, 7, 8.

↳ key.

Algorithm: Insertion – Sort

Step 1: Input 'n' elements of an array A

2: I ← 1

3: Repeat through step 8 while (I < n)

4: Key ← A[I], J ← I − 1

5: Repeat through step 7 while ((J >= 0) and A(J) > key

6: A[J+1] ← A[J]

7: J ← J − 1

8: A[J+1] ← Key

9: Display the sorted elements of array A

10: STOP

14 Program for Insertion Sort

```
#include<stdio.h>
void Isort(int[], int);
main()
{
    int a[100], i, n;
    printf("enter no. of elements:");
    scanf("%d", &n);
    printf("enter the list of elements:");
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    Isort(a, n); /* function call */
    printf("The sorted list is:");
    for(i=0; i<n; i++)
        printf("%d", a[i]);
}
```

```
void Isort(int a[], int n)
{
    int i, j, key;
    for(i=1; i<n; i++)
    {
        key = a[i];
        j = i-1;
        while((j>=0) && a[j]>key)
        {
            a[j+1] = a[j];
            j--;
        } /* end of while */
        a[j+1] = key;
    } /* end of for */
} /* function end */
```

Analysis: It is easiest Sorting method. If list is already sorted, only one comparison is made on each pass so that for (n-1) passes (n-1) comparisons Hence the complexity is O(n) i.e. best case

In worst case i.e. if list is sorted in reverse order then the total no. of comparisons are

$$1+2+3+\cdots+n-1 = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = O(n^2)$$

Worst Case Time complexity . $O(n^2)$

Best " " " $O(n)$

Quick Sort

The most powerful sorting algorithm is QuickSort. It uses the policy of 'DIVIDE AND CONQUER'. It is also known as "partition exchange Sort". This algorithm was developed by 'CA.R. Hoare'. It is a sorting algorithm, which performs very well on larger lists than any other sorting methods.

In Quick Sort, at each step ~~select~~ a particular element to be selected let it be 'pivot' and place that 'pivot' element in its proper position so that the list is divided into two halves in such a way that all the elements in left sublist are less than than pivot and all the elements in right sublist are greater than pivot. The procedure is then repeated recursively for both the sublists

Suppose there are 'n' elements in the list which are to be sorted Quick Sort method identifies an element say 'pivot' within the

list and moves all the elements which are less than 'pivot' to the left of pivot and moves all the elements which are greater than pivot to the right of pivot. Thus the list is divided into two sublists so that 'pivot' element is placed into its proper position. We then identify new 'pivot' in sublist-1 and sublist-2. Again these sublists are partitioned into further sublists and so on. This continues until the entire list is exhausted.

The ~~Post~~ procedure to sort 'n' elements in ascending order is -

Step I: select first element of array A (or subarray) as pivot.

II: Initialise i and j to first and last element positions of the array respectively

III: Increment i, until (A[i] > pivot)

IV: Decrement j, until A[j] < pivot

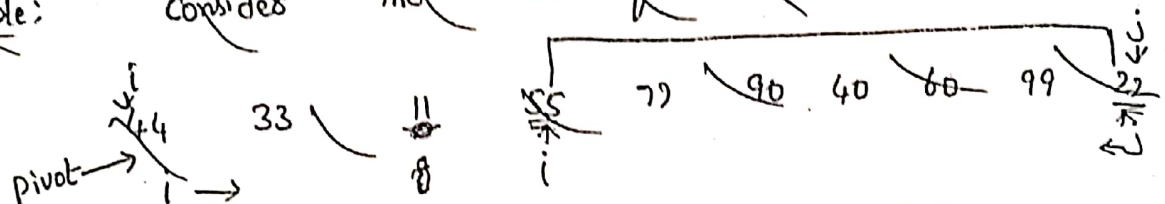
v. if (i<j) exchange A[i] and A[j]

vi. Repeat steps iii, iv, v until i>j i.e. when i and j cross each other

vii. Exchang pivot element (i.e. and first element) and A[j] which is correct place for pivot.

viii. Apply all the above steps recursively on two sublists of the array on either side of the pivot in the same way.

Example: Consider the list of elements



pass 1:

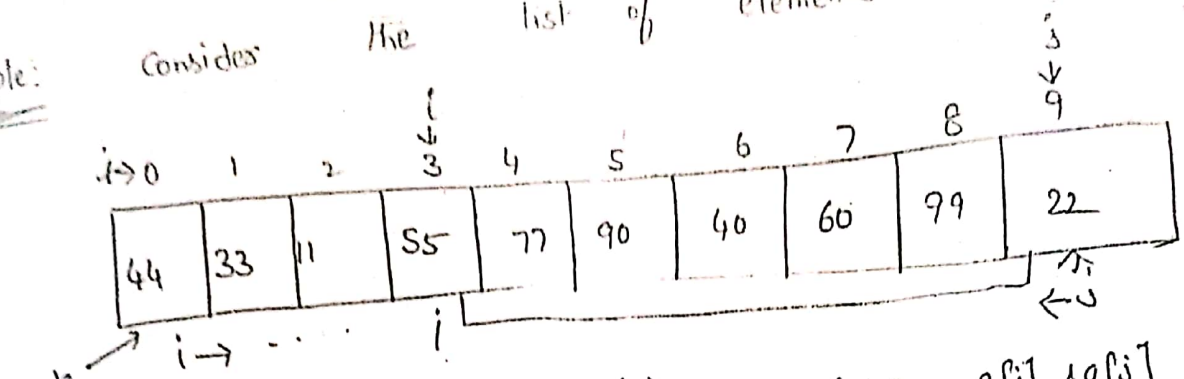
(i<j) swap a[i], a[j]

⇒

ex: 78 81 14 97 87 62 74 85 76 45 84 22

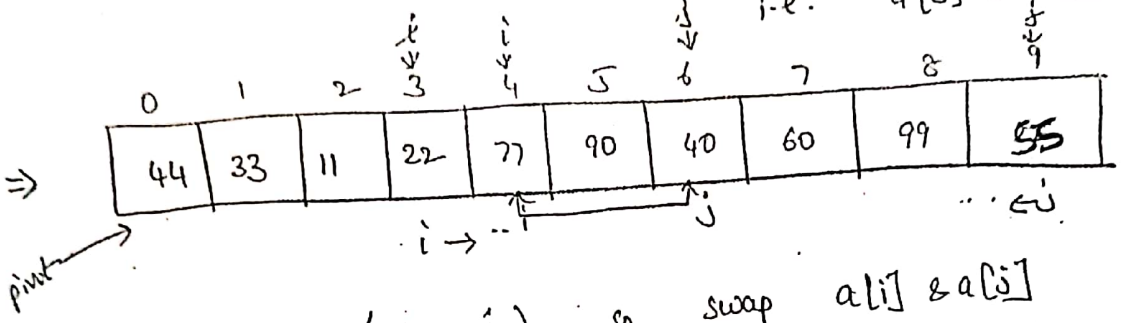
ex: 34 26 1 45 18 78 12 89 27

Example: Consider the list of elements



(i < j) so swap a[i] & a[j]

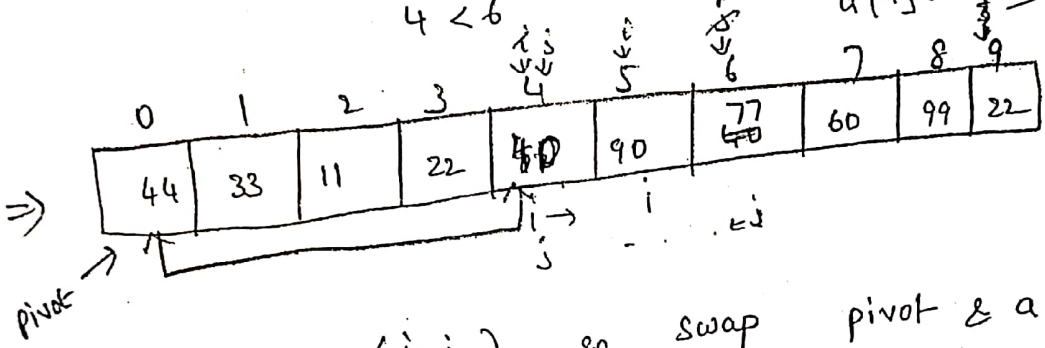
i.e. a[3] & a[9]



(i < j) so swap a[i] & a[j]

4 < 6

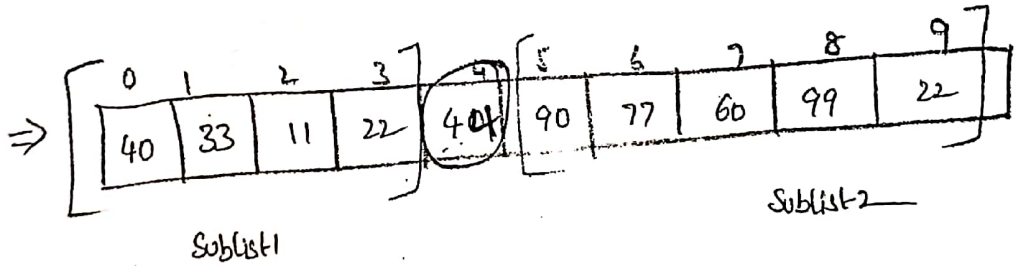
a[4] & a[6]



(i > j) so swap pivot & a[j]

a[0] & a[4]

5 > 4



Again the quicksort procedure has to be applied on sublist1 i.e. [40 33 11 22] and sublist2 i.e. [90 77 60 99 22]

10. Program for quick sort #1

```
#include<stdio.h>
void qsort(int[], int, int);
void swap(int *, int *);
main()
{
    int a[100], i, n;
    printf("enter no. of elements:");
    scanf("%d", &n);
    printf("enter the list of elements:");
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    printf("The sorted list is:");
    qsort(a, 0, n-1); /* function call */
    printf("The sorted list is:");
    for(i=0; i<n; i++)
        printf("%d", a[i]);
}
void swap(int *x, int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
```

```
void qsort(int a[], int lb, int ub)
{
    int i, j, pivot;
    if(lb<ub)
    {
        pivot = a[lb];
        i = lb; j = ub;
        while(i<=j)
        {
            while((i<=ub)&&(a[i]<=pivot))
                i++;
            while((j>lb)&&(a[j]>=pivot))
                j--;
            if(i<j)
                swap(&a[i], &a[j]);
        } /* end of while */
        swap(&a[lb], &a[j]);
        qsort(a, lb, j-1);
        qsort(a, j+1, ub);
    } /* end of if */
} /* function end */
```

Analysis: The running time of sorting algorithm is measured by the no of comparisons required to sort 'n' elements.

The worst case occurs when the list is already sorted. Then the first element require 'n' comparisons to recognise that it remains in the first position. The first sublist will be empty, but the second sublist will have (n-1) elements. The second element will require (n-1) comparisons to recognise that it remains in the second position

∴ total no. of comparisons = $n + (n-1) + (n-2) + \cdots + 2 + 1$

$= \frac{n(n+1)}{2} = \frac{n^2+n}{2} = O(n^2)$.

The average case and best case time complexity is, $O(n \log n)$. Comes from the fact that, on the average each reduction step of the algorithm produces two sublists.

Accordingly,

step 4) Reducing the initial list places '1' element and produces two sublists

2) Reducing the two sublists places 2 elements and produces four sublists

step 3) Reducing the four sublists places 4 elements and produces eight sublists

Step 4: Reducing the eight sublists places 8 elements and produces sixteen sublists.

The reduction step in the k^{th} level finds the location of 2^{k-1} elements. Hence there will be approximately $\log_2 n$ levels of reduction steps. Each level uses at most 'n' comparisons

So, total no. of comparisons are = $O(n \log n)$

Simply

If the partition algorithm divides the given list into 2 equal halves, then it is the 'bestcase' for quick sort algorithm. In this case, the Complexity can be expressed using the following recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + Cn \quad \text{when } n > 1$$

$$= a \quad \text{when } n = 1$$

solving the recurrence relation by repeated substitution method we get -

$$T(n) = O(n \log_2 n)$$

Best case Analysis

Let $T(n)$ be the time complexity of quick sort.

$$T(n) = c \quad \text{if } n = 1$$

$$T(n) = T(K) + T(n-K) + \alpha n \quad \text{otherwise.}$$

K is the total number of elements in the first sub list.

Best case occures when the list is broken in to two equal halfs.

$$K = n/2 \qquad n-K = n/2$$

$$T(n) = T(n/2) + T(n/2) + \alpha n$$

$$\Rightarrow 2T(n/2) + \alpha n \quad -①$$

$$T(n/2) = 2T(n/2^2) + \frac{\alpha n}{2} \quad -②$$

sub ② in ①

$$T(n) = 2\left(2T(n/2^2) + \frac{\alpha n}{2}\right) + \alpha n$$

$$\Rightarrow 2^2 T(n/2^2) + 2\alpha n \qquad 2^{nd} \text{ step}$$

$$\Rightarrow 2^3 T(n/2^3) + 3\alpha n \qquad 3^{rd} \text{ step}$$

$$\vdots$$

$$\Rightarrow 2^K T(n/2^K) + K\alpha n \qquad K^{th} \text{ step}$$

Recursion will continue only while $(n = 2^K \text{ i.e. } n/2^K < 1)$

$$K = \log_2 n$$

$$T(n) = 2^{\log_2 n} T(n/2^{\log n}) + \alpha n \log K$$

$$\Rightarrow n\,T(1) + \alpha n \log K$$

$$\Rightarrow nc + \alpha n \log K$$

$$O(n \log n)$$

It is a sorting method using divide and conquer approach. This method is based on the concept of dividing given lists into two halves, sorting each one seperately by recursive division and finally both sorted lists are merged together.

The basic criteria of this method is 'the no. of elements in the list is not more than one'.

So by specifying lower and upper bound of lists, the merge sort, recursively, can be defined as

If lb < ub

→ Divide the lists in two halves

→ mergesort the left half.

→ mergesort the right half.

→ Merge two sorted halves into one sorted list.

Algorithm: For implementing merge sort method two procedures are required after partition of main list.

1. Sorting of subarray lists recursively
2. Merging of Sublists

Note: Partition of Sublists is done by middle element

mergesort (lb, ub)

Algorithm 1 : [Recursive algorithm for dividing lists and their sorting; lb, ub represents lower bound and upper bound of list]

Local : mid

Step 1 : if (lb >= ub)
return.

Step 2 : mid ← (lb + ub) / 2

step 3 : Call mergeSort (list, lb, mid)
[Recursive Call of algorithm for first sublist]

Step 4 : Call mergeSort (list, mid+1, ub)
[Recursive call of algorithm for second sublist]

step 5 : Call merge (list, lb, mid, mid+1, ub)
[for merging two sorted sublists]

step 6 : Return.

Algorithm 2 : This algorithm simply merges two sorted sublists.

merge (list, lb1, ub1, lb2, ub2) : lb1, ub1 are lower and upper bounds of sublist1, lb2, ub2 are lower & upper bounds of sublist 2.

Local : i, j, k, slist [to store sorted elements]

step 1 : $i \leftarrow lb1$ $j \leftarrow lb2$ $k \leftarrow 0$

step 2 : Repeat while $((i \leq ub1)$ and $(j \leq ub2))$

if $(list[i] < list[j])$

$slist[k++] \leftarrow list[i++]$

else

$slist[k++] \leftarrow list[j++]$

step 3 : Repeat while $(i \leq ub1)$

$slist[k++] \leftarrow list[i++]$

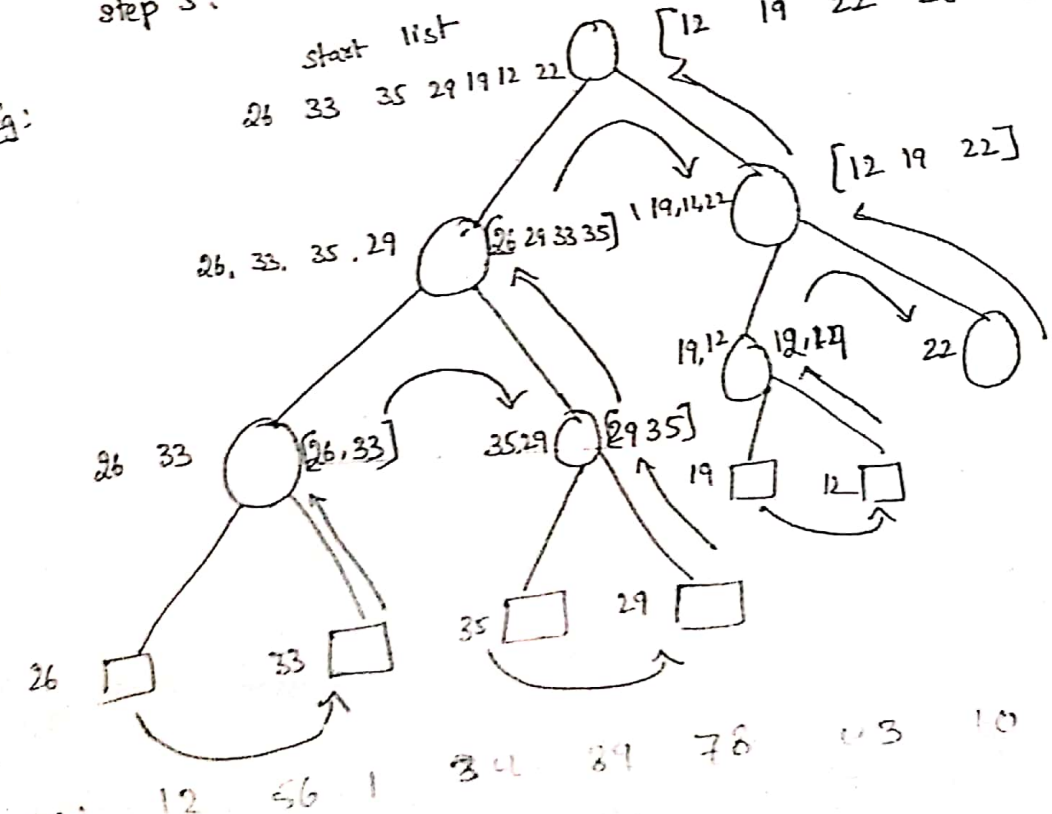
[copy remaining elements of sublist1 to sorted list]

step 4 : Repeat while $(j \leq ub2)$

$slist[k++] \leftarrow list[j++]$

step 5 : Return.

Eg :



Efficiency of merge sort:

As merge sort procedure works on dividing list into two halves and their sorting recursively. So, at each level the no. of segments doubles.

∴ The total no. of divisions are $\log_2 n$

The computing time for mergesort is described by the recurrence relation

$$T(n) = \begin{cases} a & n=1, \ a \text{ is a constant} \\ 2T(n/2) + cn & n>1, \ c \text{ is a constant.} \end{cases}$$

when n is a power of 2, $n = 2^k$, we can solve this equ. by successive substitutions:

$$T(n) = 2(2T(n/4) + cn/2) + cn$$

$$= 4T(n/4) + 2cn$$

$$= 8T(n/8) + 3cn$$

$$\vdots$$

$$= 2^k T(n/2^k) + kcn$$

$$= 2^k T(1) + kcn$$

$$= an + cn\log n$$

It is to see that if $2^k < n \leq 2^{k+1}$ then

$T(n) \in O(n \log n)$