SUBJECT: DATA STRUCTURE AND PROGRAM DESIGN

SUBJECT CODE: BECSE402T

SEMESTER: 4TH (CSE)

SUBJECT INCHARGE: DR. SHRIKANT ZADE

UNIT 1 NOTES

UNIT I: **(08 Hrs)**

Introduction to algorithm: General concepts of data structures, Types of Data Structure with its properties and Operations, Time and space analysis of algorithms, Big oh, theta, and omega notations, Average, best and worst case analysis.

Sorting and Searching Techniques: Selection sort, insertion sort, heap sort, shell sort, linear and binary search.

Definition: - An algorithm is a ***Step By Step*** process to solve a problem, where each step indicates an intermediate task. Algorithm contains finite number of steps that leads to the solution of the problem.

Properties /Characteristics of an Algorithm:-

Algorithm has the following basic properties

- Input-Output:- Algorithm takes '0' or more input and produces the required output. This is the basic characteristic of an algorithm.
- Finiteness:- An algorithm must terminate in countable number of steps.
- Definiteness: Each step of an algorithm must be stated clearly and unambiguously.
- Effectiveness: Each and every step in an algorithm can be converted in to programming language statement.
- Generality: Algorithm is generalized one. It works on all set of inputs and provides the required output. In other words it is not restricted to a single input value.

Categories of Algorithm:

Based on the different types of steps in an Algorithm, it can be divided into three categories, namely

- Sequence
- Selection and
- Iteration

Sequence: The steps described in an algorithm are performed successively one by one without skipping any step. The sequence of steps defined in an algorithm should be simple and easy to understand. Each instruction of such an algorithm is executed, because no selection procedure or conditional branching exists in a sequence algorithm.

Example:

// adding two numbers
Step 1: start
Step 2: read a,b
Step 3: Sum=a+b
Step 4: write Sum
Step 5: stop

Selection: The sequence type of algorithms are not sufficient to solve the problems, which involves decision and conditions. In order to solve the problem which involve decision making or option selection, we go for Selection type of algorithm. The general format of Selection type of statement is as shown below:

```
if(condition)
  Statement-1;
else
  Statement-2;
```

The above syntax specifies that if the condition is true, statement-1 will be executed otherwise statement-2 will be executed. In case the operation is unsuccessful. Then sequence of algorithm should be changed/ corrected in such a way that the system will re-execute until the operation is successful.

Example1:

```
// Person eligibility for vote
Step 1 : start
Step 2 : read age
Step 3 :  if age > = 18 then step_4 else step_5
Step 4 : write "person is eligible for vote"
Step 5 : write " person is not eligible for vote"
Step 6 : stop
```

Example2:

```
// biggest among two numbers
Step 1 : start
Step 2 : read a,b
Step 3 :  if a > b then
Step 4 : write "a is greater than b"
Step 5 : else
Step 6 : write "b is greater than a"
Step 7 : stop
```

Iteration: Iteration type algorithms are used in solving the problems which involves repetition of statement. In this type of algorithms, a particular number of statements are repeated 'n' no. of times.

Example1:

Step 1 : start
Step 2 : read n
Step 3 : repeat step 4 until n>0
Step 4 : (a) r=n mod 10
(b) s=s+r
(c) n=n/10
Step 5 : write s
Step 6 : stop

Performance Analysis an Algorithm:

The Efficiency of an Algorithm can be measured by the following metrics.

i. Time Complexity and
ii. Space Complexity.

i.Time Complexity:

The amount of time required for an algorithm to complete its execution is its time complexity. An algorithm is said to be efficient if it takes the minimum (reasonable) amount of time to complete its execution.

ii. Space Complexity:

The amount of space occupied by an algorithm is known as Space Complexity. An algorithm is said to be efficient if it occupies less space and required the minimum amount of time to complete its execution.

1.**Write an algorithm for roots of a Quadratic Equation?**

// Roots of a quadratic Equation
Step 1 : start
Step 2 : read a,b,c
Step 3 : if (a= 0) then step 4 else step 5
Step 4 : Write " Given equation is a linear equation "
Step 5 : d=(b * b) _ (4 *a *c)
Step 6 : if (d>0) then step 7 else step8
Step 7 : Write " Roots are real and Distinct"
Step 8: if(d=0) then step 9 else step 10
Step 9: Write "Roots are real and equal"
Step 10: Write " Roots are Imaginary"
Step 11: stop

2. Write an algorithm to find the largest among three different numbers entered by user

Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4: If a>b
    If a>c
      Display a is the largest number.
    Else
      Display c is the largest number.
  Else
    If b>c
      Display b is the largest number.
    Else
      Display c is the greatest number.
Step 5: Stop

3.Write an algorithm to find the factorial of a number entered by user.

Step 1: Start
Step 2: Declare variables n,factorial and i.
Step 3: Initialize variables
    factorial←1
    i←1
Step 4: Read value of n
Step 5: Repeat the steps until i=n
  5.1: factorial←factorial*i
  5.2: i←i+1
Step 6: Display factorial
Step 7: Stop

4.Write an algorithm to find the Simple Interest for given Time and Rate of Interest .

Step 1: Start
Step 2: Read P,R,S,T.
Step 3: Calculate S=(PTR)/100
Step 4: Print S
Step 5: Stop

ASYMPTOTIC NOTATIONS

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

The time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

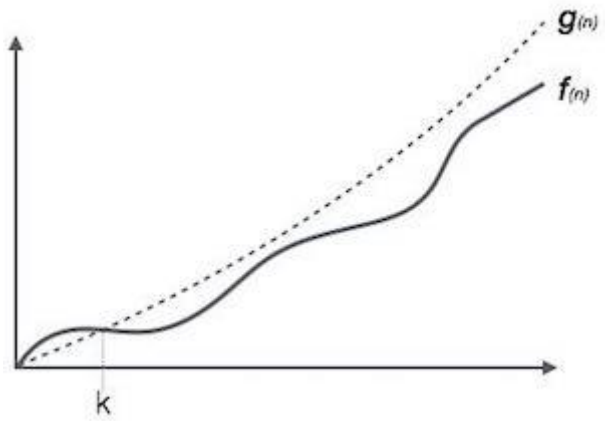
Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

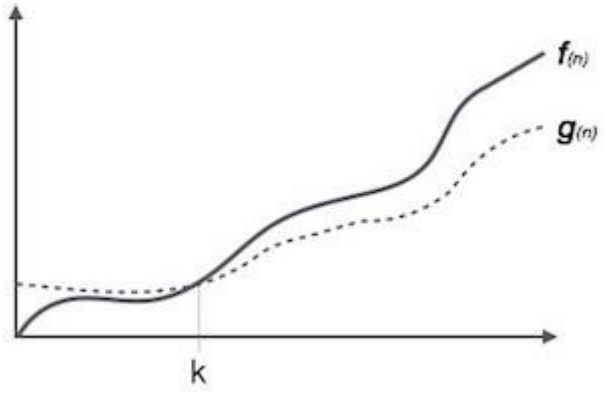


For example, for a function ***f*(n)**

O(*f*(n)) = { *g*(n) : there exists c > 0 and n_0 such that *f*(n) ≤ c.*g*(n) for all n > n_0. }

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

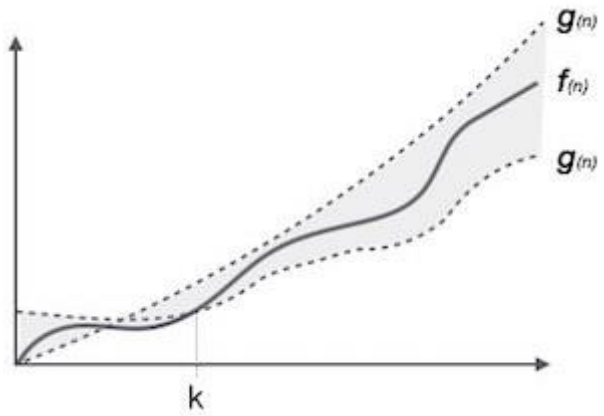


For example, for a function ***f*(n)**

Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n_0 such that *g*(n) ≤ c.*f*(n) for all n > n_0. }

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



θ(*f*(n)) = { *g*(n) if and only if *g*(n) = O(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n_0. }

DATA STRUCTURES

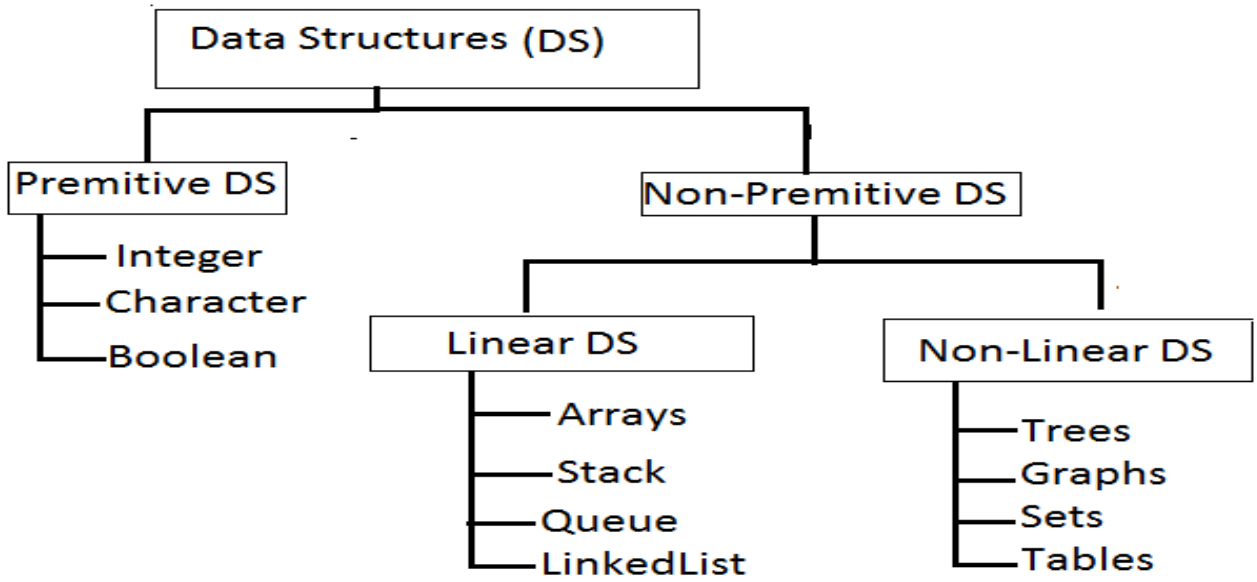
Data may be organized in many different ways logical or mathematical model of a program particularly organization of data. This organized data is called "Data Structure".

Or

The organized collection of data is called a 'Data Structure'.

Data Structure=Organized data +Allowed operations

Data Structure involves two complementary goals. The first goal is to identify and develop useful, mathematical entities and operations and to determine what class of problems can be solved by using these entities and operations. The second goal is to determine representation for those abstract entities to implement abstract operations on this concrete representation.



Primitive Data structures are directly supported by the language ie; any operation is directly performed in these data items.

Ex: integer, Character, Real numbers etc.

Non-primitive data types are not defined by the programming language, but are instead created by the programmer.

Linear data structures organize their data elements in a linear fashion, where data elements are attached one after the other. Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion. Some commonly used linear data structures are arrays, linked lists, stacks and queues.

In nonlinear data structures, data elements are not organized in a sequential fashion. Data structures like multidimensional arrays, trees, graphs, tables and sets are some examples of widely used nonlinear data structures.

Operations on the Data Structures:

Following operations can be performed on the data structures:

1. Traversing
2. Searching
3. Inserting
4. Deleting
5. Sorting
6. Merging

1. Traversing- It is used to access each data item exactly once so that it can be processed.

2. Searching- It is used to find out the location of the data item if it exists in the given collection of data items.

3. Inserting- It is used to add a new data item in the given collection of data items.

4. Deleting- It is used to delete an existing data item from the given collection of data items.

5. Sorting- It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

6. Merging- It is used to combine the data items of two sorted files into single file in the sorted form.

SORTING-INTRODUCTION

Sorting is a technique of organizing the data. It is a process of arranging the records, either in ascending or descending order i.e. bringing some order lines in the data. Sort methods are very important in Data structures.

Sorting can be performed on any one or combination of one or more attributes present in each record. It is very easy and efficient to perform searching, if data is stored in sorting order. The sorting is performed according to the key value of each record. Depending up on the makeup of key, records can be stored either numerically or alphanumerically. In numerical sorting, the records arranged in ascending or descending order according to the numeric value of the key.

Let A be a list of n elements A_1, A_2, A_3 ……A_n in memory. Sorting A refers to the operation of rearranging the contents of A so that they are increasing in order, that is, so that $A_1 <= A_2 <= A_3 <= \ldots\ldots\ldots\ldots <= A_n$. Since A has n elements, there are n! Ways that the contents can appear in A. these ways corresponding precisely to the n! Permutations of 1,2,3,……n. accordingly each sorting algorithm must take care of these n! Possibilities.

Ex: suppose an array DATA contains 8elements as follows:

DATA: 70, 30,40,10,80,20,60,50.

After sorting DATA must appear in memory as follows:

DATA: 10 20 30 40 50 60 70 80

Since DATA consists of 8 elements, there are 8!=40320 ways that the numbers 10,20,30,40,50,60,70,80 can appear in DATA.

The factors to be considered while choosing sorting techniques are:

- Programming Time
- Execution Time
- Number of Comparisons
- Memory Utilization
- Computational Complexity

Types of Sorting Techniques:

Sorting techniques are categorized into 2 types. They are Internal Sorting and External Sorting.

Internal Sorting: Internal sorting method is used when small amount of data has to be sorted. In this method , the data to be sorted is stored in the main memory (RAM).Internal sorting method can access records randomly. EX: Bubble Sort, Insertion Sort, Selection Sort, Shell sort, Quick Sort, Radix Sort, Heap Sort etc.

External Sorting: Extern al sorting method is used when large amount of data has to be sorted. In this method, the data to be sorted is stored in the main memory as well as in the secondary memory such as disk. External sorting methods an access records only in a sequential order. Ex: Merge Sort, Multi way Mage Sort.

Complexity of sorting Algorithms: The complexity of sorting algorithm measures the running time as a function of the number n of items to be stored. Each sorting algorithm S will be made up of the following operations, where A_1, A_2, A_3 ……A_n contain the items to be sorted and B is an auxiliary location.

- Comparisons, which test whether $A_i < A_j$ **or test whether** $A_i < B$.
- Interchanges which switch the contents of A_i **and** A_j **or of** A_i and B.
- Assignment which set B: A_i **and then set** A_j **:= B or** $A_j := A_i$

Normally, the complexity function measures only the number of comparisons, since the number of other operations is at most a constant factor of the number of comparisons.

SELECTION SORT

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array. The array with n elements is sorted by using n-1 pass of selection sort algorithm.

- In 1st pass, smallest element of the array is to be found along with its index pos. then, swap A[0] and A[pos]. Thus A[0] is sorted, we now have n -1 elements which are to be sorted.
- In 2nd pas, position pos of the smallest element present in the sub-array A[n-1] is found. Then, swap, A[1] and A[pos]. Thus A[0] and A[1] are sorted, we now left with n-2 unsorted elements.
- In n-1th pass, position pos of the smaller element between A[n-1] and A[n-2] is to be found. Then, swap, A[pos] and A[n-1].

Therefore, by following the above explained process, the elements A[0], A[1], A[2], ... , A[n-1] are sorted.

Example: Consider the following array with 6 elements. Sort the elements of the array by using selection sort.

A = {10, 2, 3, 90, 43, 56}.

Pass	Pos	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
1	1	2	10	3	90	43	56
2	2	2	3	10	90	43	56
3	3	2	3	10	90	43	56
4	4	2	3	10	43	90	56
5	5	2	3	10	43	56	90

Sorted A = {2, 3, 10, 43, 56, 90}

Complexity

Complexity	Best Case	Average Case	Worst Case
Time	$\Omega(n)$	$\theta(n^2)$	$o(n^2)$
Space			o(1)

Algorithm

SELECTION SORT (ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1 to N-1

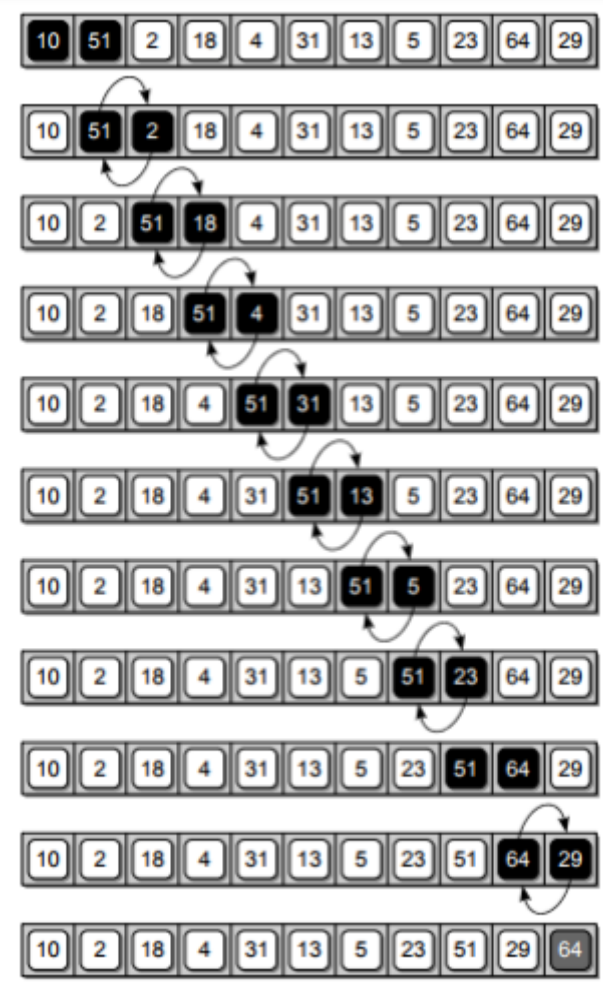
Step 2: CALL SMALLEST(A, K, N, POS)

Step 3: SWAP A[K] with
A[POS] [END OF LOOP]
Step 4: EXIT

BUBBLE SORT

Bubble Sort: This sorting technique is also known as exchange sort, which arranges values by iterating over the list several times and in each iteration the larger value gets bubble up to the end of the list. This algorithm uses multiple passes and in each pass the first and second data items are compared. if the first data item is bigger than the second, then the two items are swapped. Next the items in second and third position are compared and if the first one is larger than the second, then they are swapped, otherwise no change in their order. This process continues for each successive pair of data items until all items are sorted.

Bubble Sort Algorithm:
Step 1: Repeat Steps 2 and 3 for i=1 to 10
Step 2: Set j=1
Step 3: Repeat while j<=n
(A)
if a[i] < a[j] Then
interchange a[i] and a[j]
[End of if]
(B) Set j = j+1
[End of Inner Loop]
[End of Step 1 Outer Loop]
Step 4: Exit



Various Passes of Bubble Sort

INSERTION SORT

Insertion sort is one of the best sorting techniques. It is twice as fast as Bubble sort. In Insertion sort the elements comparisons are as less as compared to bubble sort. In this comparison the value until all prior elements are less than the compared values is not found. This means that all the previous values are lesser than compared value. Insertion sort is good choice for small values and for nearly sorted values.

Working of Insertion sort:

The Insertion sort algorithm selects each element and inserts it at its proper position in a sub list sorted earlier. In a first pass the elements A_1 is compared with A_0 and if A[1] and A[0] are not sorted they are swapped.

In the second pass the element[2] is compared with A[0] and A[1]. And it is inserted at its proper position in the sorted sub list containing the elements A[0] and A[1]. Similarly doing i^{th} iteration the element A[i] is placed at its proper position in the sorted sub list, containing the elements A[0],A[1],A[2],……A[i-1].

To understand the insertion sort consider the unsorted Array A={7,33,20,11,6}.

The steps to sort the values stored in the array in ascending order using Insertion sort are given below:

7	33	20	11	6

Step 1: The first value i.e; 7 is trivially sorted by itself.
Step 2: the second value 33 is compared with the first value 7. Since 33 is greater than 7, so no changes are made.
Step 3: Next the third element 20 is compared with its previous element (towards left).Here 20 is less than 33.but 20 is greater than 7. So it is inserted at second position. For this 33 is shifted towards right and 20 is placed at its appropriate position.

7	33	20	11	6

7	20	33	11	6

Step 4: Then the fourth element 11 is compared with its previous elements. Since 11 is less than 33 and 20 ; and greater than 7. So it is placed in between 7 and 20. For this the elements 20 and 33 are shifted one position towards the right.

7	20	33	11	6

7	11	20	33	6

Step5: Finally the last element 6 is compared with all the elements preceding it. Since it is smaller than all other elements, so they are shifted one position towards right and 6 is inserted at the first position in the array. After this pass, the Array is sorted.

7	11	20	33	6

6	7	11	20	33

Step 6: Finally the sorted Array is as follows:

6	7	11	20	33

ALGORITHM:
Insertion_sort(ARR,SIZE)
Step 1: Set i=1;
Step 2: while(i<SIZE)
Set temp=ARR[i]
J=i=1;
While(Temp<=ARR[j] and j>=0)
Set ARR[j+1]=ARR[i]
Set j=j-1
End While
SET ARR(j+1)=Temp;

Print ARR after i^{th} pass
Set i=i+1
End while

Step 3: print no.of passes i-1

Step 4: end

Advantages of Insertion Sort:

- It is simple sorting algorithm, in which the elements are sorted by considering one item at a time. The implementation is simple.
- It is efficient for smaller data set and for data set that has been substantially sorted before.
- It does not change the relative order of elements with equal keys
- It reduces unnecessary travels through the array
- It requires constant amount of extra memory space.

Disadvantages:-

- It is less efficient on list containing more number of elements.
- As the number of elements increases the performance of program would be slow

Complexity of Insertion Sort:

BEST CASE:-

Only one comparison is made in each pass.

The Time complexity is $O(n^2)$.

WORST CASE:- In the worst case i.e; if the list is arranged in descending order, the number of comparisons required by the insertion sort is given by:

1+2+3+…….+(n-2)+(n-1)= (n*(n-1))/2;

= $(n^2-n)/2$.

The number of Comparisons are **$O(n^2)$.**

AVERAGE CASE:- In average case the numer of comparisons is given by

$\frac{1}{2}+\frac{2}{2}+\frac{3}{3}+\cdots+\frac{(n-2)}{2}+\frac{(n-1)}{2}=\frac{n*(n-1)}{2*2}$ =$(n^2-n)/4$ = **$O(n^2)$.**

Program:

```
/* Program to implement insertion sort*/
#include<iostream.h>
#include<conio.h>
main()
{
int a[10],i,j,n,t;
clrscr();
cout<<"\n Enter number of elements to be Sort:";
cin>>n;
cout<<"\n Enter the elements to be Sorted:";
for(i=0;i<n;i++)
cin>>a[i];
for(i=0;i<n;i++)
{  t=a[i];
   J=I;
while((j>0)&&(a[j-1]>t))
{  a[j]=a[j-1];
```

```
   J=j-1;
}
a[j]=t;
}
cout<<"Array after Insertion sort:";
for(i=0;i<n;i++)
cout<"\n a[i]";
getch();
}
```

OUTPUT:

Enter number of elements to sot:5
Enter number of elements to sorted: 7 33 20 11 6
Array after Insertion sort: 6 7 11 20 33.

QUICK SORT

The Quick Sort algorithm follows the principal of divide and Conquer. It first picks up the partition element called 'Pivot', which divides the list into two sub lists such that all the elements in the left sub list are smaller than pivot and all the elements in the right sub list are greater than the pivot. The same process is applied on the left and right sub lists separately. This process is repeated recursively until each sub list containing more than one element.

Working of Quick Sort:

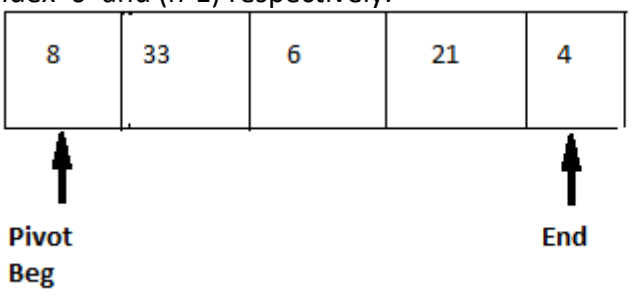
The main task in Quick Sort is to find the pivot that partitions the given list into two halves, so that the pivot is placed at its appropriate position in the array. The choice of pivot as a significant effect on the efficiency of Quick Sort algorithm. The simplest way is to choose the first element as the Pivot. However the first element is not good choice, especially if the given list is ordered or nearly ordered .For better efficiency the middle element can be chosen as Pivot.

Initially three elements Pivot, Beg and End are taken, such that both Pivot and Beg refers to 0^{th} position and End refers to the $(n-1)^{th}$ position in the list. The first pass terminates when Pivot, Beg and End all refers to the same array element. This indicates that the Pivot element is placed at its final position. The elements to the left of Pivot are smaller than this element and the elements to it right are greater.

To understand the Quick Sort algorithm, consider an unsorted array as follows. The steps to sort the values stored in the array in the ascending order using Quick Sort are given below.

8	33	6	21	4

Step 1: Initially the index '0' in the list is chosen as Pivot and the index variable Beg and End are initiated with index '0' and (n-1) respectively.

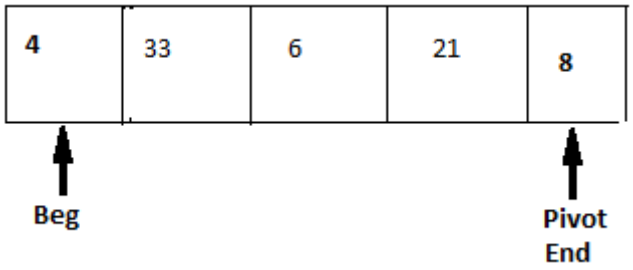


Step 2: The scanning of the element starts from the end of the list.

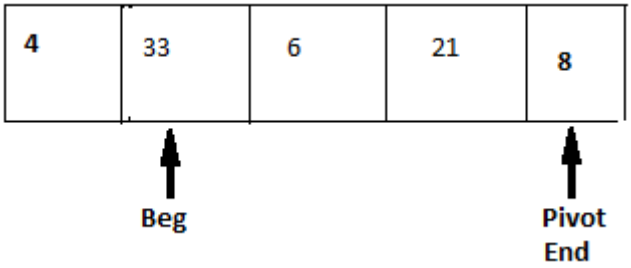
A[Pivot]>A[End]

i.e; 8>4

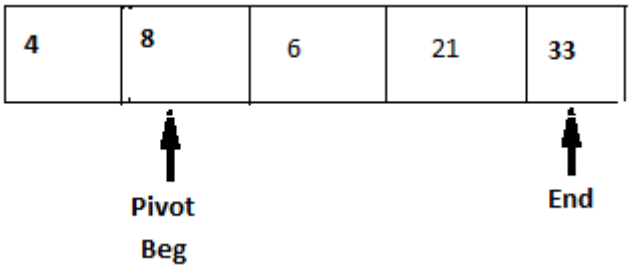
so they are swapped.



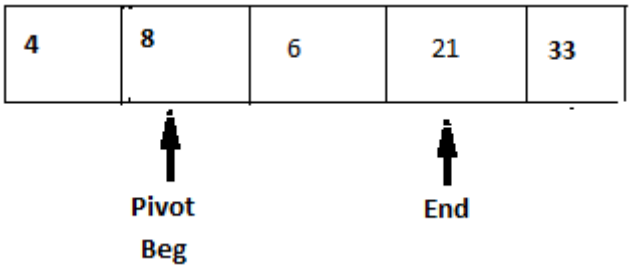
Step 3: Now the scanning of the elements starts from the beginning of the list. Since A[Pivot]>A[Beg]. So Beg is incremented by one and the list remains unchanged.



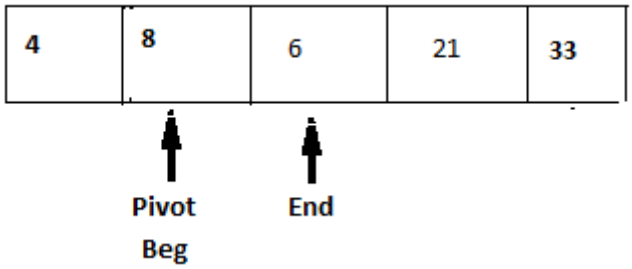
Step 4: The element A[Pivot] is smaller than A[Beg].So they are swapped.



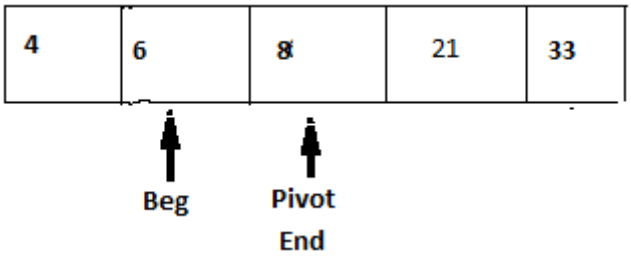
Step 5: Again the list is scanned form right to left. Since A[Pivot] is smaller than A[End], so the value of End is decreased by one and the list remains unchanged.



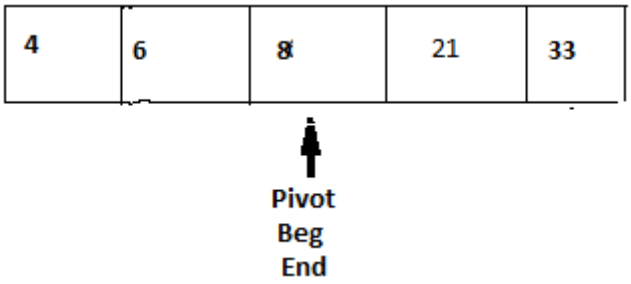
Step 6: Next the element A[Pivot] is smaller than A[End], the value of End is increased by one. and the list remains unchanged.



Step 7: A[Pivot>>A[End] so they are swapped.



Step 8: Now the list is scanned from left to right. Since A[Pivot]>A[Beg],value of Beg is increased by one and the list remains unchanged.



At this point the variable Pivot, Beg, End all refers to same element, the first pass is terminated and the value 8 is placed at its appropriate position. The elements to its left are smaller than 8 and the elements to its right are greater than 8.The same process is applied on left and right sub lists.

ALGORITHM

Step 1: Select first element of array as Pivot
Step 2: Initialize i and j to Beg and End elements respectively
Step 3: Increment i until A[i]>Pivot.
Stop
Step 4: Decrement j until A[j]>Pivot
Stop
Step 5: if i<j interchange A[i] with A[j].
Step 6: Repeat steps 3,4,5 until i>j i.e: i crossed j.
Step 7: Exchange the Pivot element with element placed at j, which is correct place for Pivot.

Advantages of Quick Sort:

- This is fastest sorting technique among all.
- It efficiency is also relatively good.
- It requires small amount of memory

Disadvantages:

- It is somewhat complex method for sorting.
- It is little hard to implement than other sorting methods
- It does not perform well in the case of small group of elements.

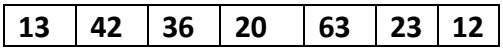
Complexities of Quick Sort:

Average Case: The running time complexity is O(n log n).
Worst Case : Input array is not evenly divided. So the running time complexity is $O(n^2)$.
Best Case: Input array is evenly divided. So the running time complexity is O(n logn).

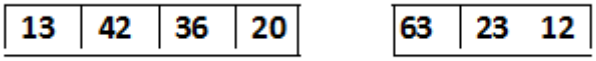
MERGE SORT

The Merge Sort algorithm is based on the fact that it is easier and faster to sort two smaller arrays than one large array. It follows the principle of "Divide and Conquered". In this sorting the list is first divided into two halves. The left and right sub lists obtained are recursively divided into two sub lists until each sub list contains not more than one element. The sub list containing only one element do not require any sorting. After that merge the two sorted sub lists to form a combined list and recursively applies the merging process till the sorted array is achieved.

Let us apply the Merge Sort to sort the following list:



Step 1: First divide the combined list into two sub lists as follows.

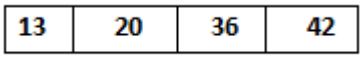


Step 2: Now Divide the left sub list into smaller sub list

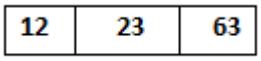
Step 3: Similarly divide the sub lists till one element is left in the sub list.

Step 4: Next sort the elements in their appropriate positions and then combined the sub lists.

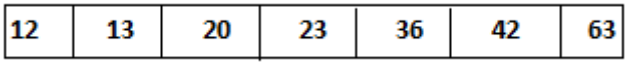
Step 5: Now these two sub lists are again merged to give the following sorted sub list of size 4.

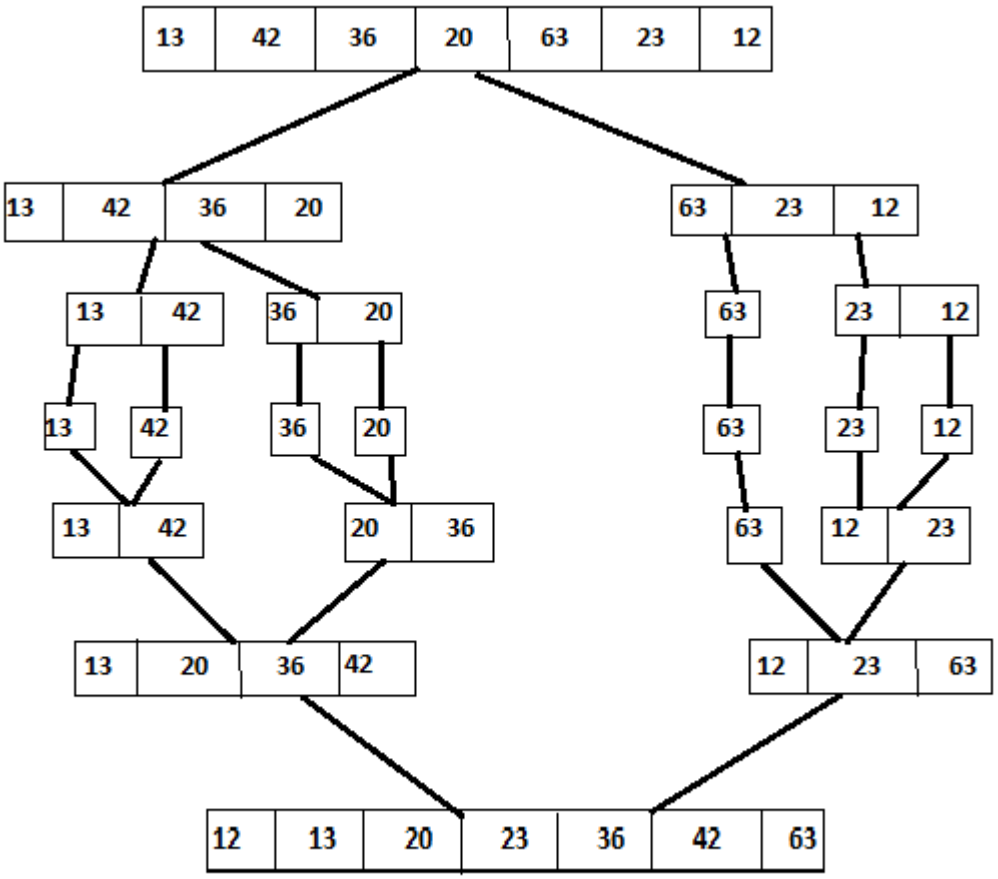


Step 6: After sorting the left half of the array, containing the same process for the right sub list also. Then the sorted array of right half of the list is as follows.



Step 7: Finally the left and right halves of the array are merged to give the sorted array as follows.





Merge Sort

Advantages:

- Merge sort is stable sort
- It is easy to understand
- It gives better performance.

Disadvantages:

- It requires extra memory space
- Copy of elements to temporary array
- It requires additional array
- It is slow process.

Complexity of Merge Sort: The merge sort algorithm passes over the entire list and requires at most log n passes and merges n elements in each pass. The total number of comparisons required by the merge sort is given by **O(n log n).**

External searching: When the records are stored in disk, tape, any secondary storage then that searching is known as 'External Searching'.
Internal Searching: When the records are to be searched or stored entirely within the computer memory then it is known as 'Internal Searching'.

LINEAR SEARCH

The Linear search or Sequential Search is most simple searching method. It does not expect the list to be sorted. The Key which to be searched is compared with each element of the list one by one. If a match exists, the search is terminated. If the end of the list is reached, it means that the search has failed and the Key has no matching element in the list.

Ex: consider the following Array A

23 15 18 17 42 96 103

Now let us search for 17 by Linear search. The searching starts from the first position.
Since A[0] ≠17.
The search proceeds to the next position i.e; second position A[1] ≠17.
The above process continuous until the search element is found such as A[3]=17.
Here the searching element is found in the position 4.

Algorithm: LINEAR(DATA, N,ITEM, LOC)

Here DATA is a linear Array with N elements. And ITEM is a given item of information. This algorithm finds the location LOC of an ITEM in DATA. LOC=-1 if the search is unsuccessful.

Step 1: Set DATA[N+1]=ITEM
Step 2: Set LOC=1
Step 3: Repeat while (DATA [LOC] != ITEM)
Set LOC=LOC+1
Step 4: if LOC=N+1 then
Set LOC= -1.
Step 5: Exit

Advantages:

- It is simplest known technique.
- The elements in the list can be in any order.

Disadvantages:

This method is in efficient when large numbers of elements are present in list because time taken for searching is more.

Complexity of Linear Search: The worst and average case complexity of Linear search is **O(n)**, where 'n' is the total number of elements present in the list.

BINARY SEARCH

Suppose DATA is an array which is stored in increasing order then there is an extremely efficient searching algorithm called "Binary Search". Binary Search can be used to find the location of the given ITEM of information in DATA.

Working of Binary Search Algorithm:

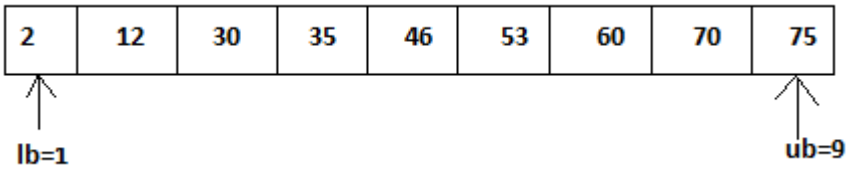
During each stage of algorithm search for ITEM is reduced to a segment of elements of DATA[BEG], DATA[BEG+1], DATA[BEG+2],…… DATA[END].

Here BEG and END denotes beginning and ending locations of the segment under considerations. The algorithm compares ITEM with middle element DATA[MID] of a segment, where MID=[BEG+END]/2. If DATA[MID]=ITEM then the search is successful. and we said that LOC=MID. Otherwise a new segment of data is obtained as follows:

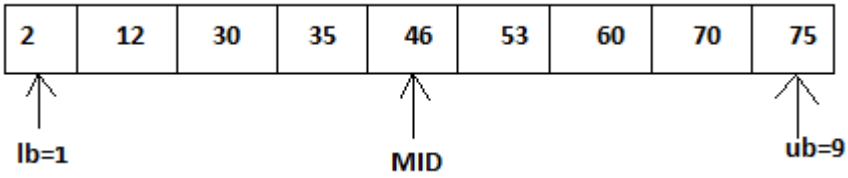
i. If ITEM<DATA[MID] then item can appear only in the left half of the segment.
DATA[BEG], DATA[BEG+1], DATA[BEG+2]
So we reset END=MID-1. And begin the search again.

ii. If ITEM>DATA[MID] then ITEM can appear only in right half of the segment i.e. DATA[MID+1], DATA[MID+2],……DATA[END].
So we reset BEG=MID+1. And begin the search again.

Initially we begin with the entire array DATA i.e. we begin with BEG=1 and END=n
Or
BEG=lb(Lower Bound)
END=ub(Upper Bound)
If ITEM is not in DATA then eventually we obtained END<BEG. This condition signals that the searching is Unsuccessful.
The precondition for using Binary Search is that the list must be sorted one.
Ex: consider a list of sorted elements stored in an Array A is

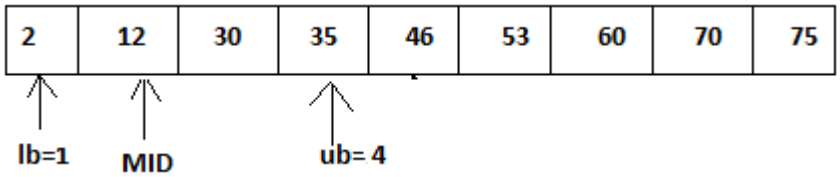


Let the key element which is to be searched is 35.
Key=35
The number of elements in the list n=9.
Step 1: MID= [lb+ub]/2
=(1+9)/2
=5



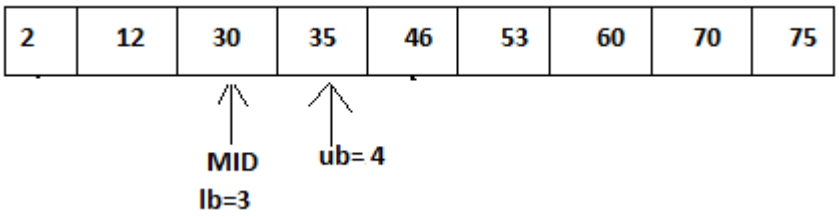
Key<A[MID]
i.e. 35<46.
So search continues at lower half of the array.
Ub=MID-1
=5-1
= 4.

Step 2: MID= [lb+ub]/2
=(1+4)/2
=2.



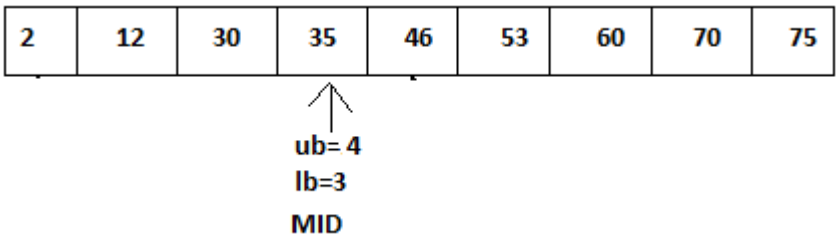
Key>A[MID]
i.e. 35>12.
So search continues at Upper Half of the array.
Lb=MID+1
=2+1
= 3.

Step 3: MID= [lb+ub]/2
=(3+4)/2
=3.



Key>A[MID]
i.e. 35>30.
So search continues at Upper Half of the array.
Lb=MID+1
=3+1
= 4.

Step 4: MID= [lb+ub]/2
=(4+4)/2
=4.



ALGORITHM:

BINARY SEARCH[A,N,KEY]

Step 1: begin

Step 2: [Initilization]
Lb=1; ub=n;

Step 3: [Search for the ITEM]

Repeat through step 4,while Lower bound is less than Upper Bound.

Step 4: [Obtain the index of middle value]
MID=(lb+ub)/2

Step 5: [Compare to search for ITEM]
If Key<A[MID] then
Ub=MID-1
Other wise if Key >A[MID] then
Lb=MID+1
Otherwise write “Match Found”
Return Middle.

Step 6: [Unsuccessful Search]
write “Match Not Found”

Step 7: Stop.

Advantages: When the number of elements in the list is large, Binary Search executed faster than linear search. Hence this method is efficient when number of elements is large.

Disadvantages: To implement Binary Search method the elements in the list must be in sorted order, otherwise it fails.

Define sorting? What is the difference between internal and external sorting methods?
Ans:- Sorting is a technique of organizing data. It is a process of arranging the elements either may be ascending or descending order, ie; bringing some order lines with data.

Internal sorting	External sorting
1. Internal Sorting takes place in the main memory of a computer.	1. External sorting is done with additional external memory like magnetic tape or hard disk
2. The internal sorting methods are applied to small collection of data.	2. The External sorting methods are applied only when the number of data elements to be sorted is too large.
3. Internal sorting takes small input	3. External sorting can take as much as large input.
4. It means that, the entire collection of data to be sorted in small enough that the sorting can take place within main memory.	4. External sorting typically uses a sort-merge strategy, and requires auxiliary storage.
5. For sorting larger datasets, it may be necessary to hold only a chunk of data in memory at a time, since it wont all fit.	5. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file.
6. Example of Internal Sorting algorithms are :- Bubble Sort, Internal Sort, Quick Sort, Heap Sort, Binary Sort, Radix Sort, Selection sort.	6. Example of External sorting algorithms are: - Merge Sort, Two-way merge sort.
7. Internal sorting does not make use of extra resources.	7. External sorting make use of extra resources.

Justify the fact that the efficiency of Quick sort is O(nlog n) under best case?
Ans:- Best Case:-
The best case in quick sort arises when the pivot element divides the lists into two exactly equal sub lists. Accordingly
i) Reducing the initial list places '1' element and produces two equal sub lists.
ii) Reducing the two sub lists places '2' elements and produces four equal sub lists and son on.

Observe that the reduction step in the k th level finals the location of $2^{(k-1)}$ elements, hence there will be approximately log n levels of reduction. Further, each level uses at most 'n' comparisons, So f(n) = O(n log n).Hence the efficiency of quick sort algorithm is O(nlog n) under the best case.

Mathematical Proof:- Hence from the above, the recurrence relation for quick sort under best case is given by

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

By using substitution method , we get

$$T(n)=2T(n/2)+Kn$$
$$=2\{ 2T(n/4)+k.n/2\}+kn$$
$$=4T(n/4) + 2kn$$

.
.
.

In general

$T(n)= 2^kT(n/2^k) + akn$ // after k substitutions

The above recurrence relation continues until $n=2^k$, $k=\log n$

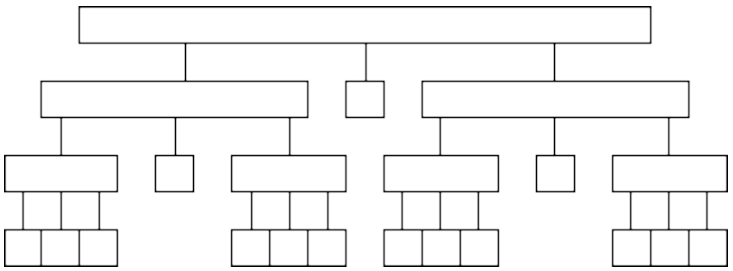
By substituting the above values , we get

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

Quick sort, or partition-exchange sort, is a sorting algorithm that, on average, makes $O(n \log n)$ comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons, though this behavior is rare. Quick sort is often faster in practice than other $O(n \log n)$ algorithms. Additionally, quick sort's sequential and localized memory references work well with a cache. Quick sort is a comparison sort and, in efficient implementations, is not a stable sort. Quick sort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only $O(\log n)$ additional space used by the stack during the recursion. Since each element ultimately ends up in the correct position, the algorithm correctly sorts. But how long does it take.

The best case for divide-and-conquer algorithms comes when we split the input as evenly as possible. Thus in the best case, each sub problem is of size n/2.The partition step on each sub problem is linear in its size. Thus the total effort in partitioning the 2^k problems of size $n/2^k$ is $O(n)$.

The recursion tree for the best case looks like this:



The total partitioning on each level is $O(n)$, and it take $\log n$ levels of perfect partitions to get to single element sub problems. When we are down to single elements, the problems are sorted. Thus the total time in the best case is $O(n\log n)$.