UNIT-1

INTRODUCTION

Algorithm:-

Algorithm is a finite set of instructions. That can be used to perform a particular task.

Characteristics of Algorithm:

An algorithm should possess following characteristics.

(i) **input**: For each and every algorithm zero or more number of quantities should be given to the algorithm.

Ex: Display message is the algorithm which takes zero input.

(ii) **output**: For each and every algorithm atleast one quantity of output must be produced. Even in the case of display message program, it produce one output if it takes zero input.

(iii) **Definiteness**: Each and Every step of algorithm should be specific, clear and unambiguous.

Ex: Adding 6+7 is possible but adding 6 or 7 to x is not possible.

Likewise 5/0.

(iv) **Finiteness**: The algorithm should be finite. That means if we trace any algorithm, then it has to terminate after finite number of steps for every legal input.

(v) **Effectiveness**: Every step of algorithm should be feasible. That means, each and every step of algorithm should be written by person using pen or pencil on paper.

Ex: Performing arithematic operations between integers is effective but performing arithematic operations between imaginary numbers is not effective.

Advantages of Analysis of algorithms

1. We can make the algorithm error free.
2. We can make the algorithm reliable.
3. We can understand the algorithm clearly.
4. We can pick the best algorithm among the several algorithms.

There are various issues in the study of algorithms and those are:

→ How to devise an algorithms?

→ How to validate an algorithms?

→ How to analyze an algorithm?

→ How to test a program.

How to device an algorithm:

(invent or plan)

creating an algorithm is a logical activity and one cannot automate it. But to create an algorithm, one can follow some design strategies. And these design strategies are used to implement many useful algorithms. So, mastering of these design strategies is an important activity in creating algorithms.

How to validate an algorithm:

The process of checking whether an algorithm computes the correct answer for all possible legal input is called algorithm validation. The purpose of validation of algorithm is to find whether algorithm works properly without being dependent upon programming languages. once validation of algorithm is done a program can be written using corresponding algorithm.

How to analyze algorithms?

Main purpose of analysis of algorithm is measuring time and space complexities. This is also called performance analysis. It is needed when two algorithms gets compared.

Analysis of algorithm is needed in Best, average and worst cases.

How to test a program

After performing above steps, the next step is writing a program for best algorithm; and testing of program can be done in two ways.

Those are

(i) Debugging

(ii) profiling

Debugging :- The process of checking whether the program gives faulty results for valid set of input is called Debugging.

once Debugging is completed, that means program is corrected.

profiling :- The process of calculating time & space complexities is called profiling.

Pseudo-code for Expressing Algorithms

Algorithm is basically sequence of instructions written in simple English language.

In general algorithm can be represented in two ways.

(i) flowchart

(ii) pseudo code.

→ flowchart is graphical representation of an algorithm.

→ pseudo code is sequence of instructions based on some programming constructs.

let us first understand the conventions used for writing an algorithm using pseudo-code.

1. Algorithm is a procedure consisting of heading and body. The heading consists of name of the procedure and parameter list.

Syntax :-

1. Algorithm name-of-procedure (parameter1, parameter2 .... parametern)

2. The begining and end of block should be indicated by '{' and '}' respectively. The compound statements should be enclosed within '{' and '}' brackets.

3. The delimiters ; are used at end of each statement.

4. Single line comments are written using // as begining of comment

5. The identifier should begin by letter and not by digit. An identifier can be a combination of alphanumeric string.

6. Using assignment operator := an assignment statement can be given.

For instance:

variable := expression;

7. There are other types of operators such as boolean operators such as true or false. Logical operators such as and, or, not and relational operators such as $<, <=, >, >=, /=$.

8. The array indices are stored with in '[' and ']' brackets. The index of array usually start at zero. The multidimensional arrays can also be used in algorithm.

9. The inputting and outputting can be done using read and write.

10. The conditional statements such as if-then or if-then-else are written in following form:

```
if (condition) then statement
if (condition) then statement
  else
  statement.
```

If the if-then statement is of compound type then '{' and '}' should be used for enclosing block.

11. While statement can be written as

```
While (condition) do
{
    Statement 1
    Statement 2
      |
      |
    Statement n
}
```

While the condition is true the block enclosed with { } gets executed otherwise statement after '}' will be executed.

12. The general form for writing for loop is:

```
for variable := value1 to valuen step do
{
    statement 1
    statement 2
      :
    statement n
}
```

Here value1 is initialization condition and value n is a terminating condition. The step indicates the increments or decrements in value1 for executing the for loop

13. The repeat-until statement can be written as:

```
repeat
    statement 1
    statement 2
      :
    statement n
until (condition)
```

14. The break statement is used to exit from inner loop. The return statement is used to return control from one point to another.

Examples

(1) Write an algorithm to count the sum of n numbers.

```
Algorithm Sum(1,n)
{
    result := 0;
    for i:=1 to n do i:=i+1
    result := result + i;
}
```

(2) Write an algorithm to check whether given number is even or odd

```
Alogorithm eventest (val)
{
    if( val %2 == 0) then
        Write ("Given number is even");
        else
        Write ("Given number is odd");
}
```

(3) Write an algorithm for sorting the elements.

```
Algorithm Sort(a,n)
{
  for i:=1 to n do
    for j:=i+1 to n+1 do
    {
      if (a[i]>a[j]) then
      {
        temp:=a[i];
        a[i]:=a[j];
        a[j]:=temp;
      }
    }
  Write ("List is sorted");
}
```

(4) Write an algorithm to find factorial of n numbar.

```
Algorithm fact (n)
{
  if n:=1 then
    return 1;
  else
    return n*fact(n-1);
}
```

(or)

```
Algorithm fact (n)
{
  fact := 1;
  for i:=1 to n do
    fact := fact * i;
  Write ("factorial is" fact);
}
```

(5) Write an algorithm to perform multiplication of two matrices.

```
Algorithm Mul(A,B,n)
{
  for i:=1 to n do
    for j:=1 to n do
      c[i,j]:=0;
    for K:=1 to n do
      c[i,j] := c[i,j] + A[i K] * B[K,j];
}
```

Performance Analysis:-

The efficiency of an algorithm can be decided by measuring the performance of an algorithm. We can measure the performance of an algorithm by computing amount of time and storage requirement.

performance analysis is calculated by 2 ways.

Those are
(1) Space complexity
(2) Time complexity

Space complexity

The amount of memory required by an algorithm to run is called 'space complexity'.

To compute the space complexity we use two factors

(i) constant variable &

(ii) Instant variable.

The space requirement S(p) can be given as

$$S(p) = c + sp$$

Where 'c' is a constant i.e., fixed part and it denotes the space of inputs and outputs. This space is an amount of space ~~in an amount of space~~ taken by instruction, variables and identifiers.

'sp' is a space dependent upon instance characteristics. This is a variable part whose space requirement depends on particular problem instance.

Ex:

```
Algorithm Add(a,b,c)
{
   return a+b+c;
}
```

In the above algorithm we have 3 constant variables, a,b,c so, c=3 & There is no instance variables.

So, sp=0

$S(p) \geq c+sp$

$S(p) \geq 3$.

[∵ space complexity have instance variables, so there is no fixed memory for sp. so, memory size will go up and down. ∴ we always declare S(p) ≥ space we can't use]

Ex:-2

```
Algorithm Add (a,n)
{
   Sum := 0.0;
   for i := 1 to n do
      Sum := Sum + a[i];
   return Sum;
}
```

In the above algorithm, we have 3 constant variables Sum, n, ~~a[]~~ i and one instance variable (i.e n for a[])

$\therefore$ $Sp = n$, $C = 3$

$$S(p) \geq C + Sp$$
$$3 + n$$
$$S(p) \geq 3 + n$$

EX:-3

```
Algorithm Add (x, n)
{
    return (Add (x, n-1) + x [n]);
}
```

In the above algorithm, 3 constant variables x, n, return address are called (n+1) times from n to 0.

So, $S(p) \geq 3(n+1)$

Time complexity

- The time complexity of an algorithm is the amount of computer time required by an algorithm to run to completion.
- There are two types of computing time.
  - compile time and
  - run time

The time complexity is generally computed using runtime or execution time.

- It is difficult to compute the time complexity in terms of physically clocked time. For instance in multiuser system, executing time depends on many factors such as system load, number of other programs running, instruction set used.
- The time complexity is therefore given in terms of "frequency count".

Frequency count :-

The number of times the statement will be executed by the compiler.

Time complexities are of many types.

(1) constant
(2) Linear
(3) Quadratic

1. constant :-

```
Stmt;
```

This statement will be executed by the compiler only once. Such type of complexities are called as constant type.

2. Linear :-

```
for i:=1 to n do  i:=i+1
Stmt;
```

This statement will be executed n times by the compiler. Such types are called as linear type.

3. Quadratic :-

```
for i:=1 to n do i:=i+1
for j:=1 to n do j:=j+1
Stmt;
```

This statement will be executed n^2 times by the compiler.

4. cubic :-

```
for i:=1 to n do i:=i+1
for j:=1 to n do j=j+1
for k:=1 to n do k:=k+1
 Stmt;
```

This statement will be executed n^3 times by the compiler. This is called cubic type.

5. Logarithmic :-

```
While (low<=high)
{
  mid:= low+high/2;
  if(key < a[mid])
  high:= mid-1;
  if(key > a[mid])
  low:= mid+1;
  else
  return mid;
}
```

In the above algorithm, every time the working area is divided into half of the working area. Such type of algorithms

Ex:-1 find the frequency count for display message.

Algorithm msg()	0
{	0
Write ("Hellow DAA");	1
}	0
Total	1

The frequency count fore above code is 1.

Ex:-2

```
Algorithm Sub (A, B, C, m, n)
{
    for i:=1 to m do
        for j=1 to n do
            C[i,j] := A[i,j] - B[i,j];
}
```

		F.C
1	Algorithm Sub (A, B, C, m, n)	0
2	{	0
3	for i=1 to m do	m+1
4	for j=1 to n do	m(n+1)
5	C[i,j] := A[i,j] - B[i,j];	mn
6	}	0
	Total	2mn+2m+1

The frequency count for above algorithm is 2mn+2m+1

Ex:-3

		F.C
1	Algorithm msg(n)	0
2	{	0
3	for i=1 to n do i: = i+1	n+1
4	Write ("Hello DAA");	n
5	}	0
	Total	2n+1

Ex-4 Find the frequency count for addition of two matrices which have different rows & columns.

		F.C
1	Algorithm Sum(a,b,c,r,co)	0
2	{	0
3	for i:=1 to r do i:=i+1	r+1
4	for j:=1 to co do j:=j+1	r(co+1)
5	c[i][j] := a[i][j] + b[i][j];	rxco
6	}	0
7	Total	2rco+2r+1

Ex-5 Find frequency count for Subtraction of two matrices which has same rows & columns.

		F.C
1	Algorithm Sub(a,b,c,r)	0
2	{	0
3	for i:=1 to r do i:=i+1	r+1
4	for j:=1 to r do j:=j+1	r(r+1)
5	c[i][j] := a[i][j] - b[i][j];	r^2
6	}	0
7	Total	$2r^2+2r+1$

Ex-6 Find frequency count for multiplication of two matrices.

1	Algorithm mul(a,b,c,n)	0
2	{	0
3	for i:=1 to n do i:=i+1	n+1
4	for j:=1 to n do j:=j+1	n(n+1)
5	{	0
6	c[i][j] := 0	n^2
7	for K:=1 to n do K:=K+1	$n^2(n+1)$
8	c[i][j] := c[i][j] + A[i][K] * B[K][j];	n^3
9	}	0
10	Write(C[i][j]);	1
11	}	0
	Total	$2n^3+3n^2+2n+3$

Time complexity can be represented in three ways

(i) Best case
(ii) Worst case
(iii) Average case

Best case : It an algorithm takes minimum amount of time to run an algorithm for specific set of inputs is called best case time complexity.

Ex:- While searching an element in liner search, If we get desired element in first place itself. Then the time complexity of linear search is Best case time complexity.

Worst case : It is the maximum time required to run an algorithm for specific set of inputs.

Ex:- While searching an element in linear search, If we get desired element in end of the list, then the time complexity of linear search is worst case.

Average case : It is the average time required to run an algorithm for specific set of inputs.

Asymptotic Notations :-

To choose the best algorithm, we need to find efficiency of an algorithm. The efficiency can be measured by time complexity of an algorithm, Asymptotic notations are used to represent time complexity and they give best, average and worst case time complexities.

Bigo(O), omega(Ω), Theta(Θ) are various notations used to represent time complexities.

Big Oh notation :-

Big oh notation is denoted by "O" and it gives longest amount of time required to run an algorithm.
It is used to represent worst case time complexity of algorithm.

Definition:-

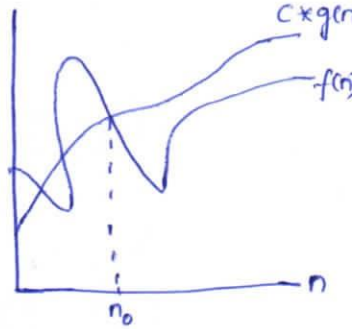
Let, $f(n)$ and $g(n)$ are two non-negative functions And if there exists an integer number and constant c such that $c>0$ and for all integers $n>n_0$, $f(n) \leq c * g(n)$, then $f(n)$ is big oh of $g(n)$

It is also denoted as "$f(n) = O(g(n))$".

- Big-oh notation represents that $g(n)$ value is an upperbound value on $f(n)$.

Ex: Let $f(n) = 2n+2$, $g(n) = n^2$ prove $f(n) = O(n^2)$ for all n, $n \geq n_0$

Big oh condition is $f(n) \leq c * g(n)$



Let us assume constant $c=1$

$f(n) = 2n+2 \quad g(n) = n^2$

for $n=1$ $f(n) = 2(1)+2$

$= 4$

$c * g(n) = 1 \times n^2$

$= 1 \times 1^2 = 1$

$4 \leq 1$ False

for $n=2$ $f(n) = 2(2)+2$

$= 4+2 = 6$

$c * g(n) = 1 \times n^2$

$= 1 \times 4 = 4$

$6 \leq 4$ False

for $n=3$ $f(n) = 2(3)+2$

$= 6+2 = 8$

$c * g(n) = 1 \times n^2 = 1 \times 3^2$

$= 1 \times 9 = 9$

$8 \leq 9$ True

$f(n) \leq c * g(n)$ is true for $n > 2$ (→ n_0)

$n_0 = 2$ and $c = 1$

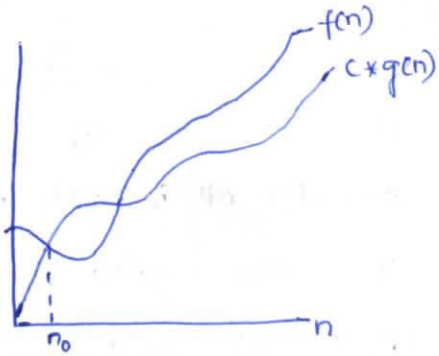
Omega notation:-

Omega notation is denoted by "Ω" and it gives lowest amount of time required to run an algorithm.

It is used to represent best case time complexity of algorithm

Definition:- Let $f(n)$, and $g(n)$ are two non-negative functions and if there exists constant c and integer number such that $c>0$ and n_0 such

Ex: Let $f(n) = 3n+2$ prove $f(n) = \Omega(n)$.



$f(n) = 3n+2$

$g(n) = n$ Let $c=3$

for $n=1$ $f(n) = 5$

$g(n) = 1$

$\Rightarrow f(n) > c * g(n)$

$5 > 3*1$

$5>3$ True for $n>1$

$\therefore n_0 = 1$ and $c=3$.

$\therefore f(n) = \Omega(n)$.

Hence proved

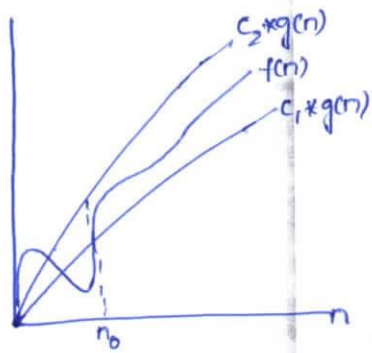
Theta Notation :- Theta notation is denoted by "Θ" and it gives average amount of time required to run an algorithm. It is used to represent average case time complexity of algorithm.

Definition: Let, $f(n)$ and $g(n)$ be two non negative functions. There exists positive constants c_1 and c_2 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ and $f(n)$ is theta of g of n and it is denoted by $\Theta(g(n))$.

Ex: Let $f(n) = 3n+2$ prove $f(n) = \Theta(n)$.

$f(n) = 3n+2$ $g(n) = n$

Theta condition is $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$



let us assume $c_1 = 3$

for $n=1$ $f(n) = 3+2 = 5$

$c_1 * g(n) = 3 \times 1 = 3$

$3 \leq 5$ True

$c_1 * g(n) \leq f(n)$ is true for $n>1$

let us assume $c_2 = 4$

for $n=1$ $f(n) = 3+2 = 5$

$c_2 * g(n) = 4*1 = 4$

$5 \leq 4$ False

for $n=2$ $f(n) = 3(2)+2 = 6+2 = 8$

$c_2 * g(n) = 4*2 = 8$

$n=3$ $f(n) = 3*3+2 = 11$

$C_2 * g(n) = 4*3 = 12$

$11 \leq 12$ True

$f(n) \leq C_2 * g(n)$ is true for $n>2$

$C_1 * g(n) \leq f(n) \leq C_2 * g(n)$ is true for

$n_0 = 2$ and $C_1 = 3$, $C_2 = 4$.

$\therefore f(n) = \Theta(n)$

Hence proved

More Examples

(1) $f(n) = 3n^3$ (Big oh)

$g(n) = n^3$ $C=4$

for $n=1$ $f(n) = 3$

$g(n) = 1 \Rightarrow C*g(n) = 4*1 = 4$

$3 \leq 4$.

$\therefore f(n) \leq C*g(n)$ is true for $n=1$, $C=4$, $n_0=1$.

(2) $f(n) = 3n^3+2n^2+3$ (Big oh)

$g(n) = n^3$, $C=4$

for $n=1$ $f(n) = 3+2+3 = 8$

$g(n) = 1 \Rightarrow C*g(n) = 4*1 = 4$

$8 \leq 4$ False

for $n=2$ $f(n) = 35$ $24+8+3$

$C*g(n) = 32$

$35 \leq 32$ False

for $n=3$ $f(n) = 102$

$C*g(n) = 108$

$102 \leq 108$ True

Big oh is true for $n>2$, $C=4$, $n_0=2$.

(3) $f(n) = 9n^2$ (Big oh notation)

$g(n) = n^2$ $c = 10$

for $n=1$ $f(n) = 9$

$g(n) = 1$ $c * g(n) = 10$

$9 \leq 10$ true.

for $n=2$ $f(n) = 36$

$c * g(n) = 40$

$36 \leq 40$ True

Big oh $\Rightarrow f(n) \leq c * g(n)$ is true for $n=1$ & $c=10$, $n_0=1$.

Hence proved

(4) $f(n) = n^4 + 3n^3 + 2n + 1$ (Ω notation)

$g(n) = n^3$ $c=1$

for $n=1$ $f(n) = 7$

$g(n) = 1$

$7 > 1$ True

$f(n) > c * g(n)$ for $n_0 = 1$ & $c=1$.

Hence proved

(5) $f(n) = 6n^2 + 7n + 8$ (Ω notation)

$g(n) = n^2$ $c=6$

$n=1$ $f(n) = 21$

$c * g(n) = 6$

$21 > 6$ True

$n=2$ $f(n) = 46$

$c * g(n) = 24$

$46 > 24$ ~~false~~ True

Ω notation $f(n) > c * g(n)$, $c = 1, 6$

(6) $f(n) = n^4 + 3n^3 + 5n + 1$ (Θ notation)

Θ condition is $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

let $c_1 = 1$ $c_2 = 2$ $g(n) = n^4$

for $n=1$ $f(n) = 10$

$c_1 * g(n) = 1$ $10 \geq 1$ True

$c_1 * g(n) \leq f(n)$ is true for $n=1$, $c_1 = 1$, $n_0 = 1$.

for $n=1$ $f(n) = 10$

$c_2 * g(n) = 2*(1)^4 = 2$

$10 \leq 2$ false

for $n=2$ $f(n) = 51$

$c_2 * g(n) = 2*(2)^4 = 32$

$51 \leq 32$ false

for $n=3$ $f(n) = 178$

$c_2 * g(n) = 2*(3)^4 = 162$

$178 \leq 162$ false

for $n=4$ $f(n) = 469$

$c_2 * g(n) = 2*(4)^4 = 512$

$469 \leq 512$ True

$\therefore c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ is true for $n > 4$

$c_1 = 1, c_2 = 2, n_0 = 4$.

Hence proved

Little 'o' notation :-

Little oh notation is denoted by 'o'. It is used very rarely and it is used instead of Big oh (O) notation for lower 'c' values.

Ex: $c = 0.0001$.

Definition : Let $f(n)$, $g(n)$ are two non-negative functions "$f(n) = o(g(n))$" if and only if (iff) $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

Ex:- $f(n) = 7n+6$ Show that $f(n)$ is $\Omega(n^2)$

$f(n) = 7n+6$

$g(n) = n^2$

$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \lim_{n \to \infty} \frac{n}{7n+6} = \frac{\infty}{\infty} \neq 0$. (We can't define).

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{7n+6}{n^2}$

$= \lim_{n \to \infty} \frac{n(7+6/n)}{n^2}$

$= \lim_{n \to \infty} \frac{(7+6/n)}{n} = 0 \left(\because \frac{1}{\infty} = 0\right)$

$= \frac{1}{7+6/\infty}$

Problems in previous Question paper

(1) Find frequency count for given algorithm.

$$t_{sum}(n) = \begin{cases} 2 & \text{if } n=0 \\ 2+t_{sum}(n-1) & \text{if } n>0 \end{cases}$$

Sol:- if $n>0$ $t_{sum}(n) = 2+t_{sum}(n-1)$

$= 2+2+t_{sum}(n-2)$

$= 2+2+2+t_{sum}(n-3)$

$= 3(2)+t_{sum}(n-3)$

$= 2n + t_{sum}(0)$

frequency count $= 2n+2$

(2) prove $f(n)=O(n\log n)$ if $f(n) = 16\log n!$ prove $16\log n! = O(n\log n)$

Sol:- $f(n) = 16\log n!$ $g(n) = n\log n$.

$\log n! = \log[n(n-1)(n-2)\cdots(n-(n-1))]$

$= \log n + \log(n-1) + \log(n-2) + \cdots + \log 1$ [$\because \log ab = \log a + \log b$]

$\leq \log n + \log n + \log n + \cdots + \log n$

$\leq n\log n$

$16 \Rightarrow O(1)$ [$\because$ 16 is constant time complexity of 16 is '1'.]

$\log n! = O(n\log n)$.

$\therefore 16\log n! = O(1)\cdot O(n\log n)$

$= O(n\log n)$

Hence proved.

probabilistic Analysis :-

Algorithms which make use of randomised are called Randomised algorithms / probabilistic analysis of algorithms. Analysis of these algorithms is called probabilistic analysis.

Randomiser is a function which is used to generate random numbers. The output of randomiser is different from run to run. So, that the output of randomised algorithms is also depends on output of randomiser.

Randomised algorithms are of two types.

(1) Los vegas algorithm

(2) Monte carlo algorithm.

Los vegas algorithm:- It is an algorithm which gives same or correct output for same input. The output of los vegas algorithm depends on randomiser. If we are lucky, the algorithm will terminate fastly, otherwise it will go lengthy. So, the execution time of these algorithms is also vary from run to run. The time complexity of los vegas algorithms is also a randomiser.

Monte carlo algorithm:- In this algorithm which gives different output for same input. But the time complexity of Monte carlo algorithm is same for run to run.

Ex:-
```
Algorithm repeater(a,n)
{
  While(true)
  {
    i:= Random() mod n+1;
    j:= Random() mod n+1;
    if((i≠j) and (a[i] = a[j]))
    return i;
  }
}
```

Amortized Analysis :-

Amortized analysis means finding the average running time per operation over a worst case sequence of operations. On the other hand amortised analysis guarantees the time per operation over the worst case performance.

- Amortized analysis assumes Worst-case input and typically does not allow random choices.
- The average case analysis and amortized analysis are different. In average-case analysis, we are averaging over all possible inputs where as in amortized analysis we are averaging over a sequence of operations.
- The amortized analysis does not allow random selection of input.

There are several techniques used in amortized analysis.

- **Aggregate analysis** : In this type of analysis the upper bound T(n) on the total cost of a sequence of n operations is decided, then the average cost is calculated as T(n)/n.
- **Accounting Method** : In this method the individual cost of each operation is determined, by combining immediate execution time and its influence on the running time of future operations.
- **potential Method** : is like the accounting method, but overcharges operations early to compensate for undercharges later.

Review Questions

1. What do you mean by algorithm?
2. Give the characteristics of an algorithm?
3. Discuss various issues in algorithmic design
4. Write an algorithm to compute Fibonacci Series.
5. What do you meant by performance analysis of an algorithm? Explain.
6. Explain the concept of 1. Space complexity
   2. Time complexity
7. What are different asymptotic notations used? Explain.
8. Write a short note on Amortized analysis.
9. prove that (i) $3n^3 + 2n^2 = O(n^3)$ (April 2014)
   (ii) $3^n + 1 = O(2^n)$ (April-2014)

22 Various meanings associated with big-oh are

$O(1)$	constant computing time
$O(n)$	Linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential
$O(\log n)$	logarithmic

The relationship among these computing time is

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$$

The time complexities for growth of various functionalities

$\log n$	n	$n \log n$	n^2	2^n
0	1	0	1	2
1	2	2	4	4
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1024	4,294,967,296

[∵: Answer for order of Growth with time function]

Q: Write the non-recursive algorithm for finding the fibonacci sequence and derive it's time complexity. [April 1, 2005 – 16M]
April - 2006
NOV - 2004

Ans:

```
1.  Algorithm Fib(n)
2.  {
3.      if(n≤1) then                    — 1
4.          write n
5.      else
6.          {
7.              f1:=0;
8.              f2:=1;
9.              for i:=2 to n do        n+1
10.             {
11.                 f:=f1+f2;           n-1
12.                 f2=f1;              n-1
13.                 f1=f;               n-1
14.             }
```

There are two cases
(i) When $n \leq 1$ and
(ii) $n > 1$
The total step count for stmt 3 & 4 is 2
When $n > 1$ step 9 will be executed for n times. Steps 11, 12, 13 will be executed for $n-1$ times.
Hence total step count for $n > 1$ is $4n+1$.

Q: What do you meant by input size of a problem? Explain it's significance.

[May-2005, set-3 (8 M)]

Ans: The input size of any instance of a problem is defined as the number of words required to describe that instance of problem.

Input size is the instance characteristics used for time complexity

For example: compute the input size of following program segment.

```
Algorithm Sub (A, B, C, m, n)
{
    for i:=1 to m do
        for j:=1 to n do
            C[i,j]:=A[i,j] - B[i,j];
}
```

for computing input size of above segment we will compute input size of each statement.

	Algorithm Sub (A, B, C, m, n)	0
	{	0
	for i:=1 to m do	m+1
	for j:=1 to n do	m(n+1)
	C[i][j] := A[i,j] - B[i][j];	mn
	}	0

2mn+2m+1

Thus the input size turns to 2mn+2m+1. Also frequency count for above code is 2mn+2m+1. The input size gives the number of accesses made to programming statements.

UNIT - II

Divide and Conquer

Syllabus: General Method, Applications - Binary Search, QuickSort, Merge Sort

Divide and Conquer - General Method:

In Divide & conquer method, a given problem is solved using following 3 steps:

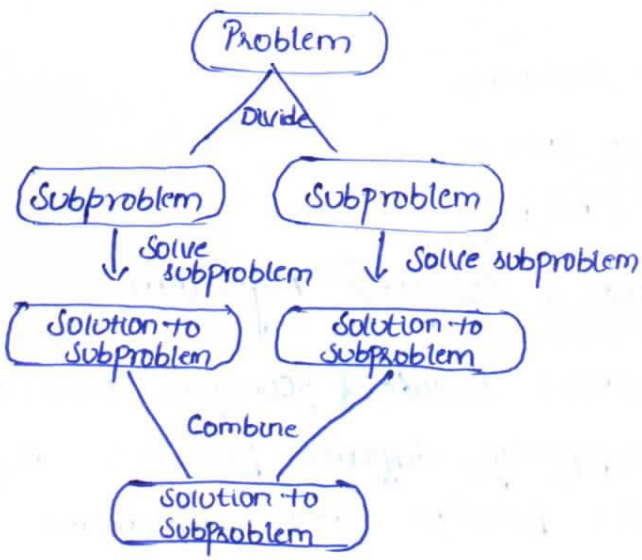
(i) Divide : A given problem is divided into subproblems.

(ii) Conquer : These subproblems are solved independently

(iii) Combine : Combining solutions of all the subproblems into a solution of the whole.

- If the subproblems are large again divide & conquer is reapplied.
- Recursive algorithms are used on divide & conquer strategy

Eg:



Procedure:

```
Algorithm Dc(P)
{
  if P is too small then
  return solution of P
  else
  {
    Divide (P) and obtain P1, P2, P3 ... Pn
    where n >= 1
    Apply Dc to each subproblem
    return Combine ( Dc(P1), Dc(P2) ..... Dc(Pn));
  }
}
```

If the size of P is n & the sizes of the k subproblems are $n_1, n_2 \cdots n_k$ respectively then the computing time of DC is described by the recurrence relation,

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots T(n_k) + f(n) & \text{otherwise} \end{cases}$$

where

$T(n)$ is the time for DC on any i/p of size n

$g(n)$ is the time to compute the answer directly for small inputs

$f(n)$ is the time for dividing P & combining solutions to subproblems

The complexity of many divide & conquer algorithms is given by the recurrences of the form

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

where

a & b are known constants.

we assume $T(1)$ is known

n is power of b ($n = b^k$)

Advantages of Divide & Conquer Algorithm:

1. Solving Difficult Problems: Divide & conquer (D&C) is a powerful tool for solving conceptually difficult problems. All it requires is a way of breaking the problem into subproblems & combining subproblems to the original problem.
2. Parallelism: D&C algorithms are naturally adapted for execution in multiprocessor machines, because distinct subproblems can be executed on different processors.
3. Memory Access: once a subproblem is small enough, it & all its subproblems can be solved within the cache without accessing the slower main memory.
4. Roundoff Control: In computations with rounded arithmetic, a D&C algorithm may yield more accurate results than a equivalent iterative method

Disadvantages

1. Recursion is slow.
2. Very simple problem may be more complicated than an iterative approach.
   Eg: Adding n numbers.

Applications

1. Binary Search
2. Quick Sort
3. Merge Sort

Binary Search:

Procedure: In the Binary Search method, the elements are sorted & stored in ascending order. If we want to search for the element say x. Then we first divide the last at middle, so that two sublists are get created. If x > middle then right sublists are created. If x < middle then considered & x is searched in right sublist otherwise i.e., if x < middle then x is searched in left sublist.

To search the element using Binary search method, D & C strategy is used. If we consider that P is the no of elements & if there is only one element then return 1. If P has more no of elements then we divide P into two sublists & solve each sublist seperately. If the desired element is found, the search terminates successfully.

Non-Recursive Algorithm:

```
Algorithm BinarySearch (A, n, key)
{
    low := 1;
    high := n;
    while (low ≤ high) do
    {
        mid := (low + high)/2;
        if (key < a[mid]) then
            high := mid - 1;
```

```
        else if (key > a[mid]) then
            low := mid + 1;
        else
            return mid;
    }   }
```

Recursive Algorithm:

```
Algorithm Binarysearch (a, key, low, high)
{   while (low ≤ high)
    {   mid := (low + high)/2;
        if (key == a[mid]) then
        return mid;
        else if (key < a[mid]) then
          Binarysearch (a, key, low, mid-1);
        else
          Binarysearch (a, key, mid+1, high);
    }   }
```

Eg-1: Let us select the 14 entries Key = 151

1	2	3	4	5	6	7	8	9	10	11	12	13	14
-15	-6	0	7	9	23	54	82	101	112	125	131	142	151

low = 1

high = 14

$mid = \frac{low + high}{2} = \frac{1+14}{2} = \frac{15}{2} = 7$

a[7] = 54

Split the array into two sublists

-15	-6	0	7	9	23	54	82	101	112	125	131	142	151

Sublist 1 | mid | Sublist 2

key == a[mid] ⇒ 151 = 54 (F)

key < a[mid] ⇒ 151 < 54 (F)

key > a[mid] ⇒ 151 > 54 (T)

Binarysearch (a, key, mid+1, high)

low = mid+1 = 7+1 = 8

high = 14

mid = $\frac{low+high}{2}$ = $\frac{8+14}{2}$ = $\frac{22}{2}$ = 11

8	9	10	11	12	13	14
82	101	112	125	131	142	151

↑ mid

key == a[mid] ⇒ 151 = a[11] ⇒ 151 == 125 (F)

key < a[mid] ⇒ 151 < 125 (F)

key > a[mid] ⇒ 151 > 125 (T)

Binarysearch (a, key, mid+1, high)

low = mid+1 = 11+1 = 12

high = 14

12	13	14
131	142	151

mid = $\frac{low+high}{2}$ = $\frac{12+14}{2}$ = $\frac{26}{2}$ = 13

a[mid] = a[13] = 142

key == a[mid] ⇒ 151 == 142 (F)

key < a[mid] ⇒ 151 < 142 (F)

key > a[mid] ⇒ 151 > 142 (T)

Binarysearch (a, key, mid+1, high)

low = mid+1 = 13+1 = 14

high = 14

13	14
142	151

a[mid] = a[14] = 151

key == a[mid] ⇒ 151 == 151 (T)

found

Eg-2:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
-15	-6	0	7	9	23	54	82	101	112	125	131	142	151

key = 9

low = 1

high = 14

mid = $\frac{low+high}{2}$ = $\frac{1+14}{2}$ = $\frac{15}{2}$ = 7

a[mid] = a[7] = 54

key == a[mid] ⇒ 9 == 54 (F)

key < a[mid] ⇒ 9 < 54 (T)

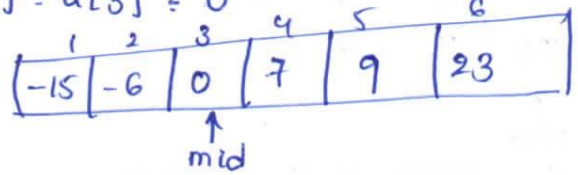
Binarysearch (a, key, low, mid)

low = 1

high = mid-1 = 7-1 = 6

mid = $\frac{low+high}{2}$ = $\frac{1+6}{2}$ = $7/2$ = 3

a[mid] = a[3] = 0



key == a[mid] ⇒ 9 == 0 (F)

key < a[mid] ⇒ 9 < 0 (F)

key > a[mid] ⇒ 9 > 0 (T)

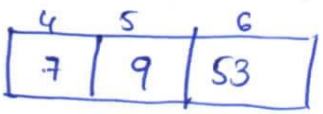
Binarysearch (a, key, mid+1, high)

low = mid+1 = 3+1 = 4

high = 6

mid = $\frac{low+high}{2}$ = $\frac{4+6}{2}$ = $10/2$ = 5

a[mid] = a[5] = 9



key == a[mid] ⇒ 9 == 9 (T)

return mid;

return 9

Found

Eg-3

1	2	3	4	5	6	7	8	9	10	11	12	13	14
-15	-6	0	7	9	23	54	82	101	112	125	131	142	151

Key = -14 low = 1 mid = $\frac{low+high}{2}$ = $\frac{1+14}{2}$ = $15/2$ = 7

high = 14

a[mid] = a[7] = 54

key == a[mid] ⇒ -14 == 54 (F)

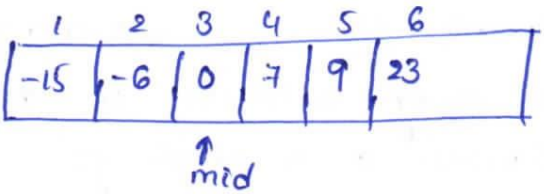
key < a[mid] ⇒ -14 < 54 (T)

- RBinarysearch (a, key, low, mid-1)

low = 1

high = mid-1 = 7-1 = 6

mid = $\frac{low+high}{2}$ = $\frac{1+6}{2}$ = 3



a[mid] = a[3] = 0

key == a[mid] ⇒ -9 == 0 (F)

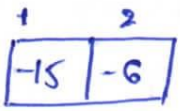
key < a[mid] ⇒ -9 < 0 (T)

- Binarysearch (a, key, low, mid-1)

low = 1

high = mid-1 = 3-1 = 2

mid = $\frac{low+high}{2}$ = $\frac{1+2}{2}$ = 3/2 = 1



a[mid] = a[1] = -15

key == a[mid] ⇒ -9 == -15 (F)

key < a[mid] ⇒ -9 < -15 (F)

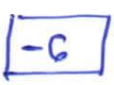
key > a[mid] ⇒ -9 > -15 (T)

- Binarysearch (a, key, mid+1, high)

low = mid+1 = 1+1 = 2

high = 2

mid = $\frac{low+high}{2}$ = $\frac{2+2}{2}$ = 4/2 = 2



key == a[mid] ⇒ -9 = a[2] = -9 == -6 (F)

key < a[mid] ⇒ -9 < -6 (T)

Binarysearch (a, key, low, mid-1)

low = 2

high = mid-1 = 2-1 = 1

low ≤ high ⇒ 2 ≤ 1 (F)

So element not found

Time Complexity of Binarysearch:

Best case: The best case of Binarysearch occurs when the element you are searching for is the middle element of the array because in that case you will get the desired element in a single go.

In this the time complexity of the algorithm will be,

$$T(n) = O(1)$$

Average case & worst case:

Each time we remove half elements in the binarysearch method if the element is not found in the middle. Then time complexity will be

$$\begin{aligned}
T(n) &= T(n/2) + c \\
&= T(n/4) + c + c \\
&= T(n/4) + 2c \\
&= T(n/8) + c + 2c \\
&= T(n/8) + 3c \\
&\vdots \\
&= T(n/2^k) + kc \qquad (\because 2^k = n) \\
&= T(n/n) + (\log n)c \qquad k = \log_2 n \\
&= T(1) + (\log n)c \\
&= c_1 + c(\log n) \\
&= O(\log n)
\end{aligned}$$

Disadvantages:

1. Sorted list is required.
2. It is only suitable for static lists, because any change requires restoring of the list.

Applications:

1. Given A a sorted array find out how many times x occur in A.
2. Given two sorted arrays of length n & m, find out the kth element of their sorted union.

QUICKSORT : Also called Partition Exchange Sort

QUICKSORT is a sorting technique in which divide & conquer strategy is used. QUICKSORT contains following 3 steps.

1. Divide :
   (i) Divide the given list into two sublists based on pivot.
   (ii) one list contains elements which are lessthan pivot.
   (iii) Second list contains elements which are greaterthan pivot.
2. Conquer : Recursively sort the given two sublists.
3. Combine : Combine all the sorted elements in a group to form a list of sorted elements.

Procedure :

Pivot = a[low]
i = low
j = high

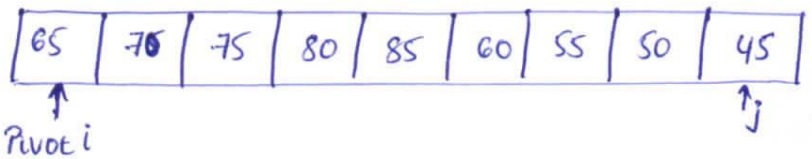
i has to incremented when a[i] < pivot
j has to be decremented when a[j] > pivot
otherwise no changes to be made
If i & j has stop changing, then swap a[i] & a[j]
then again continue
If j crosses i, then swap pivot & a[j]

Algorithm :

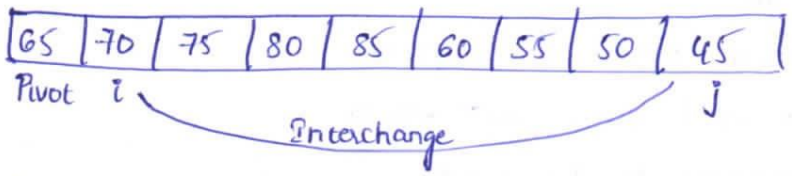
```
Algorithm quicksort (A, low, high)
{
  if (i<j) then
  {
    m := partition (A, low, high);
    quicksort(A, low, m-1);
    quicksort (A, m+1, high);
  }

Algorithm Partition (A, low, high)
{
    Pivot := A[low];
    i := low;
    j := high;
```

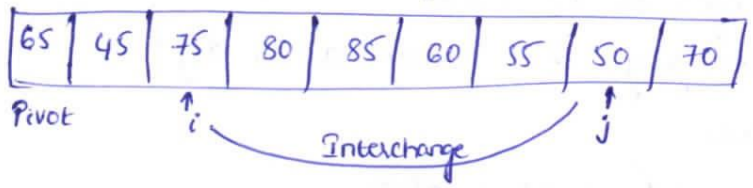
```
while (i<=j) do
{
  while (A[i] <= pivot) do
    i:=i+1;
  while (A[j] >= pivot) do
    j:=j-1;
  if (i<j) then
   swap(A[i], A[j]);
  else
   swap(pivot, A[j]);
} }
```

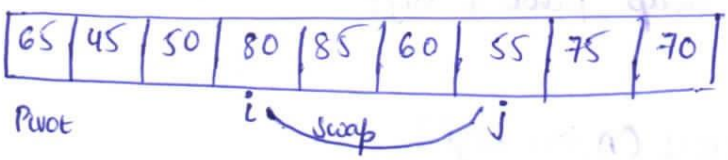
Eg:

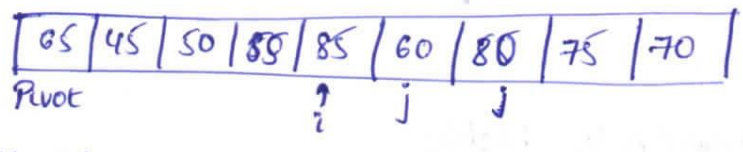


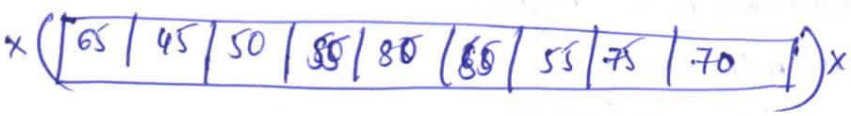
Pivot = A[low] = A[i] = 65
i = low = 1
j = high = 9
A[i] ≤ Pivot ⇒ A[i] ≤ 65 ⇒ 65 ≤ 65 (T) i=i+
A[j] ≥ Pivot ⇒ 45 ≥ 65 (F)
A[i] ≤ Pivot ⇒ 70 ≤ 65 (F)

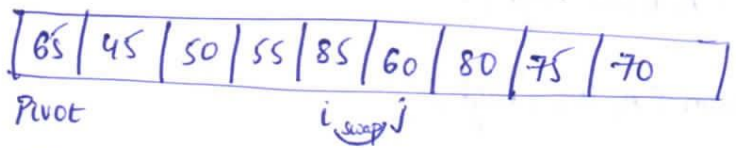


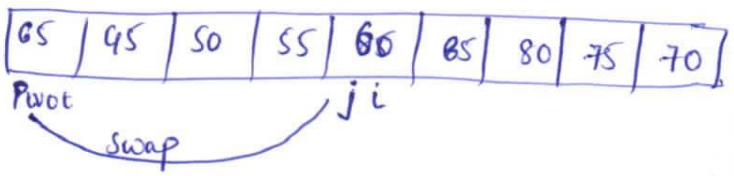


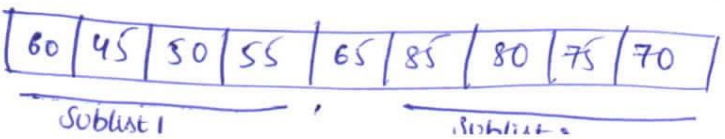


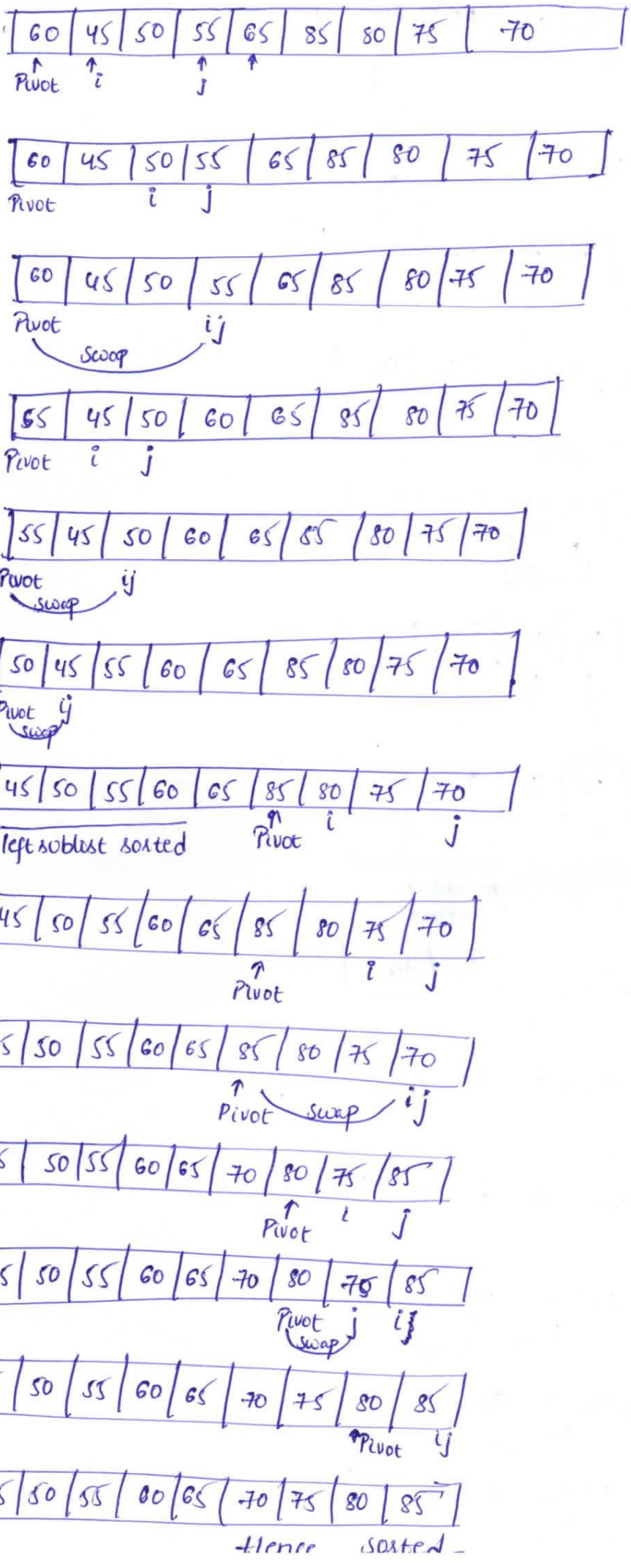




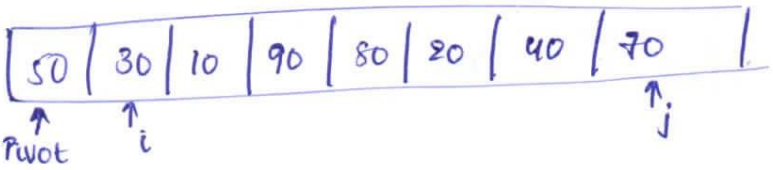


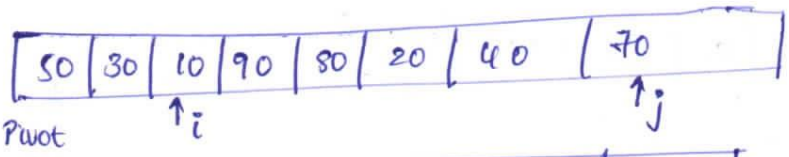


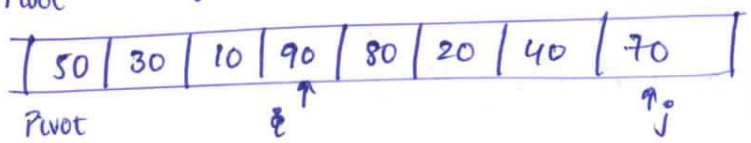


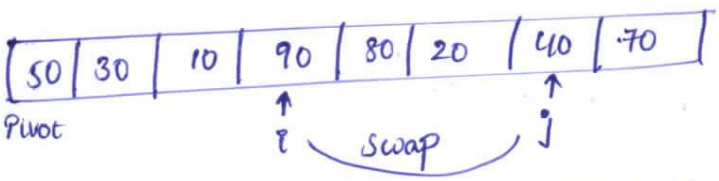

60 45 50 55 65 85 80 75 70
Pivot i j
60 45 50 55 65 85 80 75 70
Pivot i j
60 45 50 55 65 85 80 75 70
Pivot ij
swap
65 45 50 60 65 85 80 75 70
Pivot i j
55 45 50 60 65 85 80 75 70
Pivot ij
swap
50 45 55 60 65 85 80 75 70
Pivot ij
swap
45 50 55 60 65 85 80 75 70
left sublist sorted
Pivot i j
45 50 55 60 65 85 80 75 70
Pivot i j
45 50 55 60 65 85 80 75 70
Pivot swap ij
45 50 55 60 65 70 80 75 85
Pivot i j
45 50 55 60 65 70 80 75 85
Pivot j ij
swap
45 50 55 60 65 70 75 80 85
Pivot ij
45 50 55 60 65 70 75 80 85
Hence sorted.

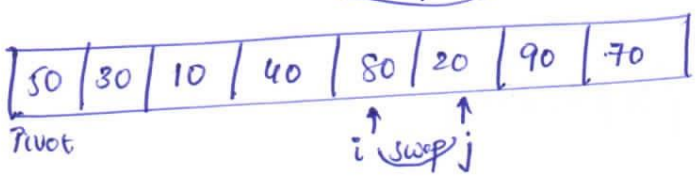
Eg-2

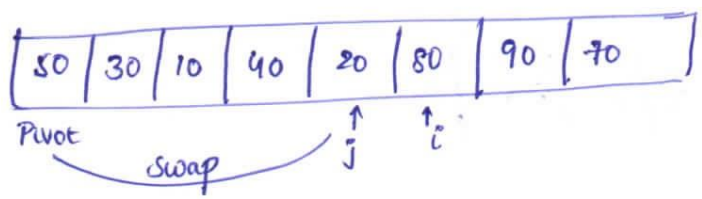


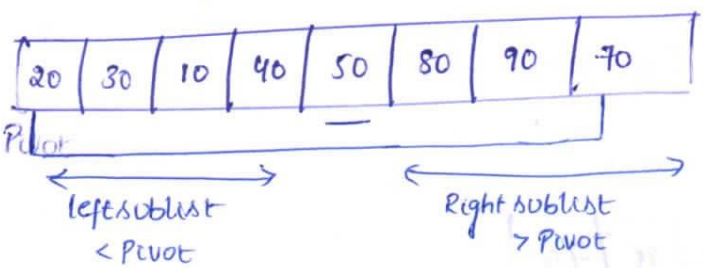


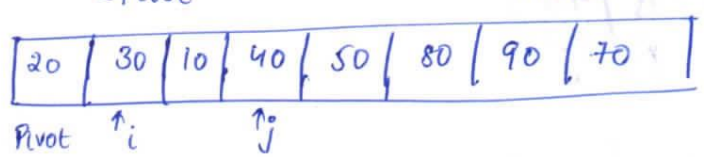


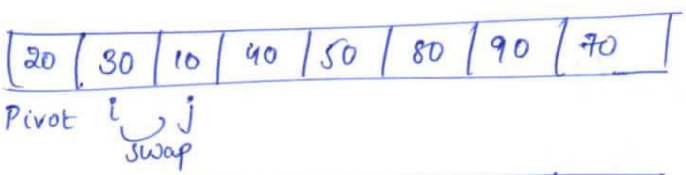


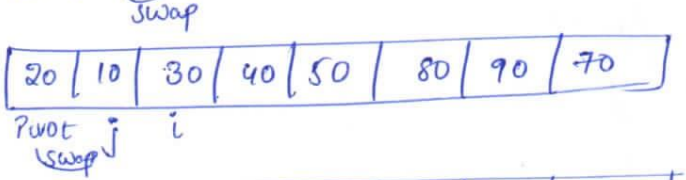


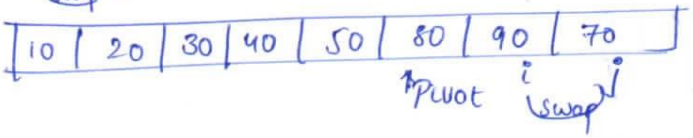


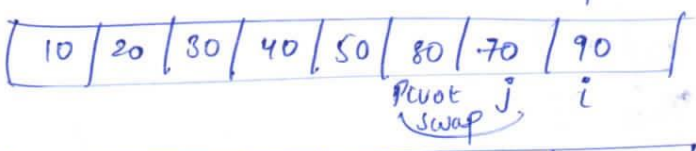


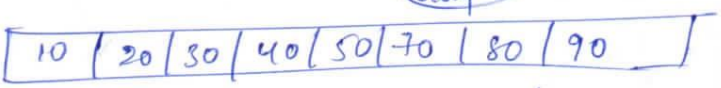






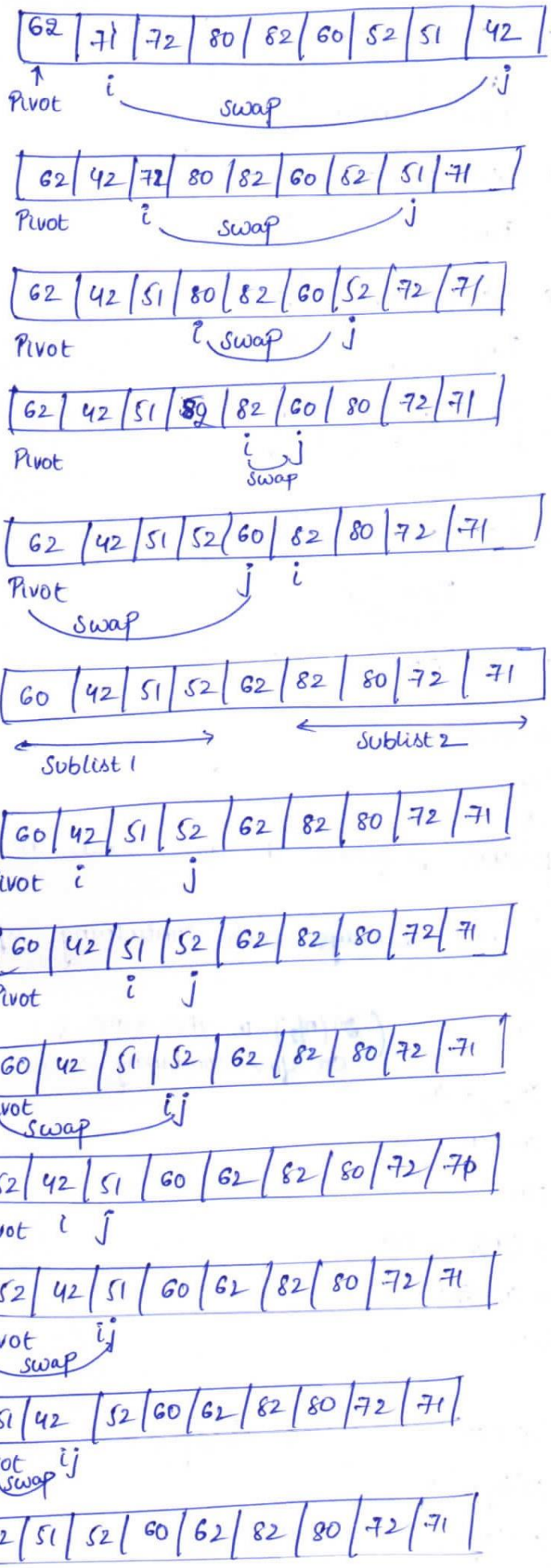


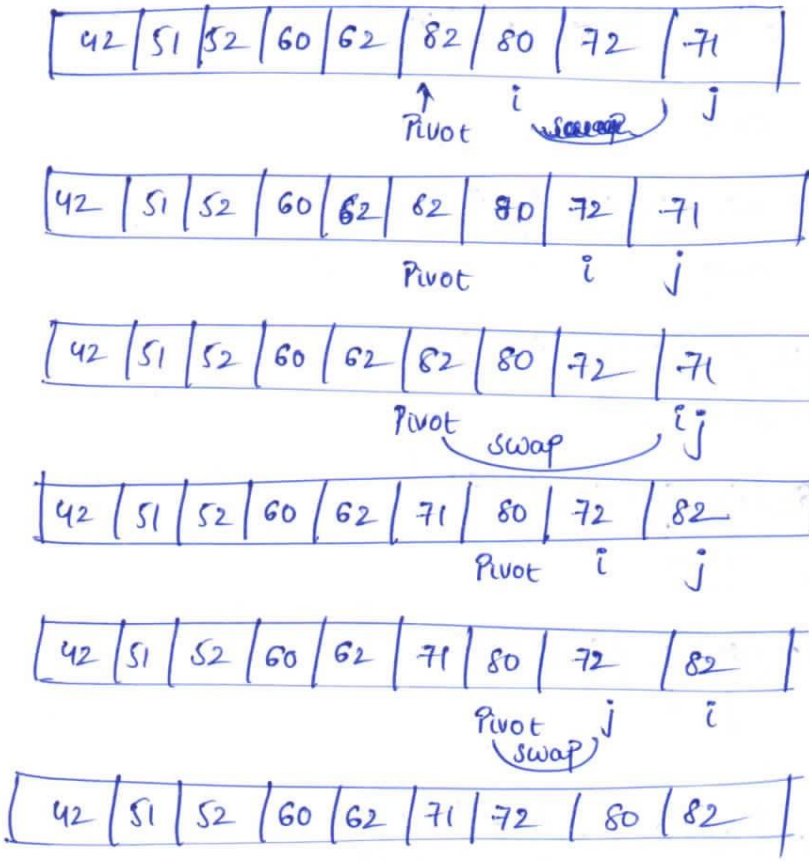




Hence sorted

Eg-3


62 71 72 80 82 60 52 51 42
Pivot
i
j
swap
62 42 72 80 82 60 52 51 71
Pivot
i
j
swap
62 42 51 80 82 60 52 72 71
Pivot
i
j
swap
62 42 51 89 82 60 80 72 71
Pivot
i
j
swap
62 42 51 52 60 82 80 72 71
Pivot
j
i
swap
60 42 51 52 62 82 80 72 71
Sublist 1
Sublist 2
60 42 51 52 62 82 80 72 71
Pivot
i
j
60 42 51 52 62 82 80 72 71
Pivot
i
j
60 42 51 52 62 82 80 72 71
Pivot
i j
swap
52 42 51 60 62 82 80 72 71
Pivot
i
j
52 42 51 60 62 82 80 72 71
Pivot
i j
swap
51 42 52 60 62 82 80 72 71
Pivot
i j
swap
42 51 52 60 62 82 80 72 71



Hence sorted

Quicksort - Time Complexity :

Bestcase : quicksort algorithm performs the best when the pivot is the middle value in the array.

- The ip array is divided into 2 arrays each containing approximately $n/2$ elements

$T(n) = 2T(n/2) + cn$ ($2T(n/2)$ is the time for sorting two subarrays, cn for combining solution for subarray)

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

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

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

$= 4(2T(n/8) + c(n/4) + 2cn$

$= 8T(n/8) + cn + 2cn$

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

$\vdots$

$= 2^k T(n/2^k) + kcn$ ($\because 2^k = n$, $k = \log_2 n$)

$= nT(n/n) + (\log n)cn$

$= nT(1) + cn(\log n)$ ($T(1) = 1$)

$= n + cn(\log n)$

$= n(1 + c\log n)$

$= n\log n$

$= O(n\log n)$

Average case: To sort an array of n distinct elements, quicksort takes $O(n \log n)$ time in expectation, averaged overall $n!$ permutations of n elements with equal probability.

$$T(n) = T(0) + T(n-1) + cn$$
$$T(n) = T(1) + T(n-2) + cn$$
$$T(n) = T(2) + T(n-3) + cn$$
$$\vdots$$
$$T(n) = T(n-2) + T(1) + cn$$
$$T(n) = T(n-1) + T(0) + cn$$

Adding above all equations

$$nT(n) = 2(T(1) + \dots T(n-1)) + cn^2 \quad \text{——} \quad (1)$$

Substituting $n-1$ in above eq (1)

$$(n-1)T(n-1) = 2(T(1) + \dots T(n-2)) + c(n-1)^2 \quad \text{——} \quad (2)$$

(1) − (2) $\Rightarrow$ $nT(n) - (n-1)T(n-1) = 2T(n-1) + c(n^2 - (n-1)^2)$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + c(n^2 - n^2 + 2n - 1)$$
$$nT(n) = 2T(n-1) + (n-1)T(n-1) + 2nc - c$$
$$nT(n) = T(n-1)(2 + n - 1) + 2nc - c$$
$$nT(n) = T(n-1)(n+1) + 2nc - c$$

Divide above eq by $n(n+1)$ & drop significant term c

$$\frac{nT(n)}{n(n+1)} = \frac{(n+1)T(n-1)}{n(n+1)} + \frac{2nc}{n(n+1)}$$
$$\frac{T(n)}{(n+1)} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

$\Rightarrow$ $$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$
$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1}$$
$$\vdots$$
$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

Adding above eq's & cancelling equal terms on both sides

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c(1/3 + 1/4 + \dots 1/n+1)$$
$$T(n) = (n+1)(1/2 + 2c(1/3 + 1/4 \dots 1/n+1)) \qquad (T(1) = 1)$$

$$= (n+1)\left(\tfrac{1}{2} + \sum_{j=3}^{n+1} \tfrac{1}{j}\right)$$

$$= (n+1)\left(\tfrac{1}{2} + \log n\right)$$

$$= (n+1)(\log n)$$

$$= O(n \log n)$$

Worst-case: This happens when the pivot is the smallest (or the largest) element.

- Then one of the partitions is empty & we repeat recursively the procedure for n-1 elements

$$T(n) = T(n-1) + cn$$

Place n-1 in above eqn

$$T(n-1) = T(n-2) + c(n-1)$$

$$T(n-2) = T(n-3) + c(n-2)$$

$$\vdots$$

$$T(2) = T(1) + c2$$

Adding above all eq's

$$T(n) + T(n-1) + T(n-2) + \cdots T(2) = T(n-1) + T(n-2) + \cdots T(2) + T(1) + cn + c(n-1) + \cdots c2$$

$$T(n) = T(1) + c(2+3+\cdots n)$$

$$= 1 + c((1+2+3\cdots n) - 1)$$

$$= 1 + c\left(\frac{n(n+1)}{2} - 1\right)$$

$$= 1 + c\left(\frac{n^2+n}{2} - 1\right)$$

$$T(n) = O(n^2)$$

	Best case	Average	Worst case
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Why quicksort is the best sorting algorithm?

- Its cache performance is higher than other sorting algorithms
- Quicksort is very faster mainly because that the operations in the innermost loop are simple
- No extra memory.
- In Java, Arrays.sort() uses quicksort for sorting primitives

Mergesort: Mergesort is a sorting technique in which Divide & conquer strategy is used in the following 3 steps:

1. **Divide:** Divide the given list into two sublists.
2. **Conquer:** Recursively sort the given two sublists
3. **Combine:** Combine the two sorted sublists to obtain entire sorted list.

Procedure:

In mergesort divide the given list into sublists until each sublist contains only one element.

Combine the sublists by sorting the given elements

Repeat the procedure until all elements are sorted.

Algorithm:

```
Algorithm Mergesort (A, low, high)
{
  mid := (low+high)/2 ;
  if (low <= high) then
  {
    Mergesort (A, low, high);
    Mergesort (A, mid+1, high);
} } Merge( A, low, mid, high);

Algorithm Merge (A, low, mid, high)
{
    i := low;
    k := low;
    j := mid+1;
  while (i <= mid && j <= high) do
  { if ( A[i] < A[j] ) then
    { i := i+1;
      k := k+1;
    }
  else
  { j := j+1;
} } k := k+1;
  while (i <= mid)
  { temp[k] := A[i];
    i := i+1;
  } k := k+1;
  while (j <= high)
  { temp[k] := A[j];
    j := j+1;
} } k := k+1;
```

Eg:

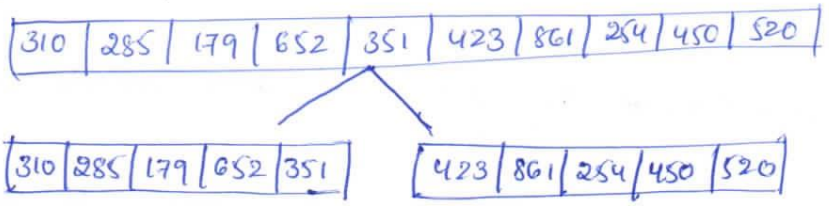
1	2	3	4	5	6	7	8	9	10
310	285	179	652	351	423	861	254	450	520

Mergesort begins by splitting a[] into two subarrays

low = 1
high = 10

$mid = \frac{low + high}{2} = \frac{1+10}{2} = 11/2 = 5$ ⇒ Mergesort (A, low, mid)
Mergesort (A, mid+1, high)

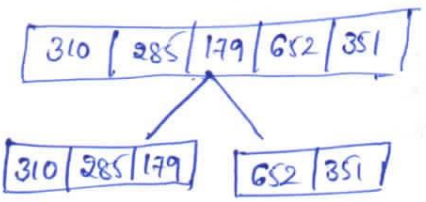
⇒ a[1:5] & a[6:10]



let us consider first subarray a[1:5]

$mid = \frac{low + high}{2} = \frac{1+5}{2} = 6/2 = 3$

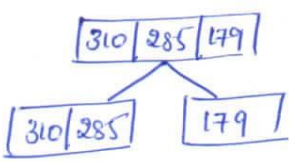
⇒ a[1:3] & a[4:5]



Consider array a[1:3]

$mid = \frac{low + high}{2} = \frac{1+3}{2} = 2$

⇒ a[1:2] & a[3:3]

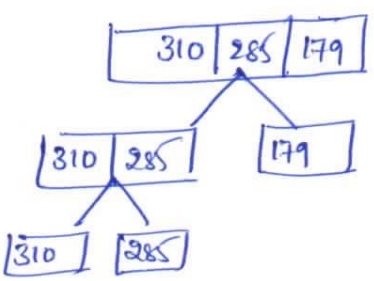


a[3:3] is single element

Consider array a[1:2]

$mid = \frac{low + high}{2} = \frac{1+2}{2} = 3/2 = 1$

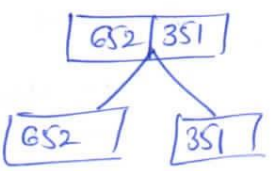
⇒ a[1:1] & a[2:2]

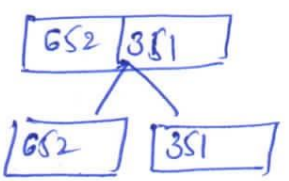


Consider array a[4:5]

$mid = \frac{low + high}{2} = \frac{4+5}{2} = 9/2 = 4$

⇒ a[4:4] & a[5:5]





Now merging begins,

Elements a[1] & a[2] are merged to yield

285	310

a[4] & a[5] are merged,

351	652

a[3] is merged with a[1:2] &

179	285	310

a[1:3] is | 179 | 285 | 310 |

a[4:5] is | 351 | 652 |

Now merge a[1:3] & a[4:5] then

179	285	310	351	652

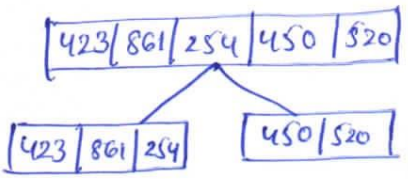
Thus the list [1:5] is now sorted we will consider

Second subarray.

6	7	8	9	10
423	861	254	450	520

mid = $\frac{low + high}{2} = \frac{6+10}{2} = 16/2 = 8$

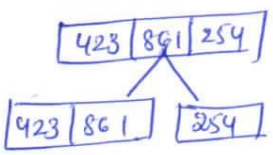
⇒ a[6:8] & a[9:10]



Consider subarray a[6:8]

mid = $\frac{low + high}{2} = \frac{6+8}{2} = 7$

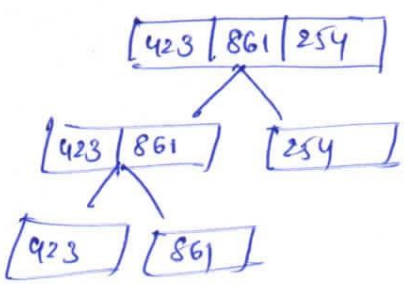
⇒ a[6:7] & a[8:8]



Consider subarray a[6:7]

mid = $\frac{low + high}{2} = \frac{6+7}{2} = 13/2 = 6$

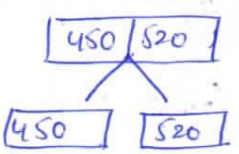
⇒ a[6:6] & a[7:7]



Consider subarray a[9:10]

mid = $\frac{low + high}{2} = \frac{9+10}{2} = 9$

⇒ a[9:9] & a[10:10]



Now merging begins

a[6] & a[7] are merged

423	861

a[9] & a[10] are merged

450	520

a[8] is merged with a[6:7]

254	423	861

merge a[6:8] & a[9:10]

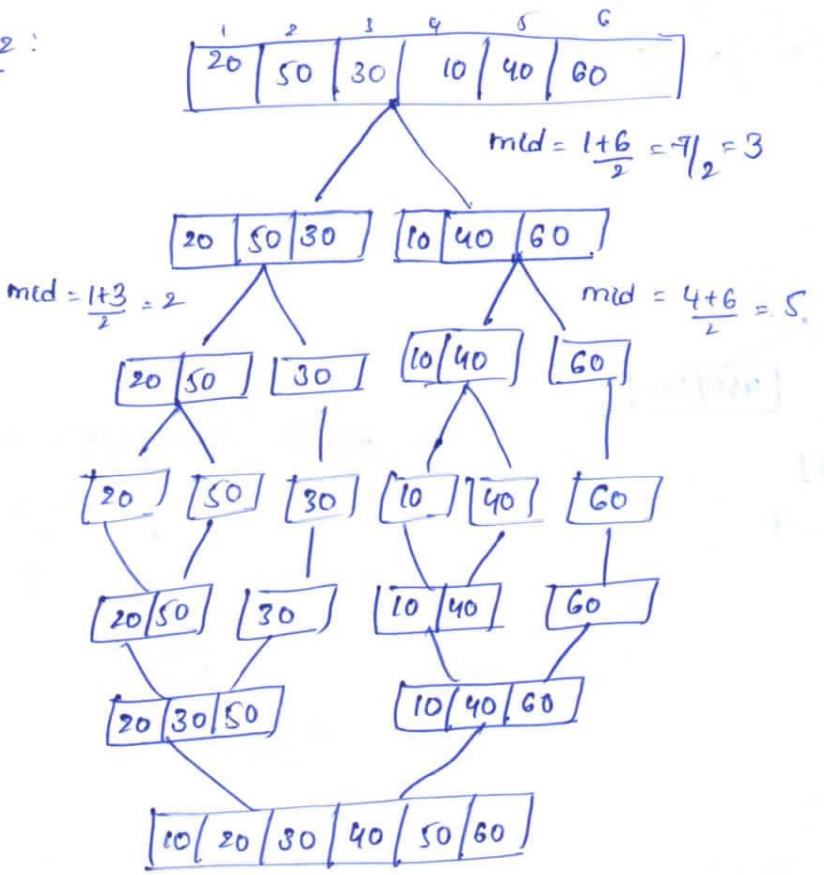
⇒

254	423	450	520	861

At this point there are two sorted subarrays & the final merge produces the fully sorted result.

179	254	285	310	351	423	450	520	652	861

Eg-2:



Hence sorted

Time complexity - Mergesort :

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

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

$$= 2\left(2T(n/4) + c(n/2)\right) + cn$$

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

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

$$\vdots$$

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

$$= 2^k T(n/2^k) + kcn \qquad (\because 2^k = n,\ k = \log n)$$

$$= n\,T(n/n) + (\log n)\,cn$$

$$= nT(1) + cn(\log n)$$

$$= n(1) + cn(\log n) \qquad (T(1) = 1)$$

$$= n(1 + c\log n)$$

$$= n\log n$$

$$= O(n\log n)$$

	Best	Average	Worst
Mergesort	O(nlogn)	O(nlogn)	O(nlogn)