DESIGN AND ANALYSIS OF ALGORITHMS.

SYLLABUS.

Unit-1:

Introduction: Algorithm, Pseudo code for expressing algorithms, Performance analysis - Space complexity, Time complexity, Asymptotic notation - Big oh notation, Omega notation, Theta notation and Little oh notation, Probabilistic analysis, Amortized complexity.

Divide and conquer: General method, applications - Binary Search, quick sort, merge sort, Strassen's matrix multiplication

Unit-2:

Searching and traversal technique: Efficient non-recursive binary tree traversal algorithms, disjoint set operations, union and find algorithms, Spanning trees, Graph traversals - Breadth first search and Depth first search, AND/OR graphs, game trees, Connected components, Bi-connected components.

Unit-3:

Greedy method: General method, applications - Job Sequencing with deadlines, 0/1 knapsack problem, Minimum cost Spanning trees, Single source Shortest path problem.

Dynamic programming: General method, applications - Multistage graphs, Optimal binary search trees, 0/1 knapsack problem, All pairs shortest path problem, travelling sales person problem, Reliability methodor design.

Unit-4:

Backtracking: General method, applications: n-queen proble sum of subsets problem, graph coloring, Hamiltonian cycles.

Brands and Bound: General method, applications-Traveling sales person problem, 0/1 knapsack problem-LC Branch and bound solution, FIFO branch and bound solution

Unit-5:

NP-Hard and NP- Complete problems: Basic concepts, Non-deterministic algorithms, NP-Hard and NP-complete classes, NP-Hard problems, Cook's theorem.

UNIT-I

ALGORITHM INTRODUCTION.

* Algorithm:

An algorithm is a finite set of instructions that performs a particular task. The word 'algorithm' comes from the name of a persian author Abu Ja'far Mohammed ibn mosa al khowarizmi (c. 825 A.D)

* Attributes or properties or characteristics of algorithm:

All algorithms must satisfy the following criteria

1. Input:

An algorithm may take zero or more number of inputs.

2. Output:

An algorithm must produce atleast one output.

3. Definiteness:

Each instruction must be defined clearly and unambiguous.

4. Finiteness:

An algorithm must be terminated successfully

after a finite number of steps.

5. Effectiveness:

Performing airthmetic operations on integers is an example of effective operation. But, performing arithemetic operations with real numbers is not effective.

Example for definiteness:

```
int a=5, b=10, c;
c=a+b; // defined clearly
add 6 or 7 to b // not defined clearly.
```

* Areas to study an algorithm:

→ There are four areas to study an algorithm

1. How to devise algorithms.
2. How to validate algorithms.
3. How to analyse algorithms
4. How to test a program.

1. How to devise algorithms:

→ Devise an algorithm means designing an algorithm or writing or creating or developing an algorithm

→ There are different design techniques to

develop an algorithm.

1. Divide and conquer.
2. Greedy method.
3. Dynamic programming
4. Back tracking.
5. Branch and bound.

2. How to validate algorithm:

→ Once an algorithm is designed it is necessary to show that it computes the correct answer for all legal inputs. This process is known as algorithm validation.

→ Once the validity of a algorithm has been shown a program can be written and a second phase begins. This phase is called as program verification or program proving.

3. How to analyze algorithm:

→ This field of study is also known as analysis of algorithms or performance analysis.

Performance analysis:

→ performance analysis refers to a task of determining how much computing time and storage an algorithm requires.

4. How to test a program:

Testing a program consists of two phases.

1. Debugging
2. Profiling/performance measurement.

1. Debugging:

→ Debugging is the process of executing process on sample data sets.

→ To determine whether faulty results occur and, if so to correct them.

2. Profiling/performance measurement:

→ Performance measurement is the process of executi a correct program on data sets and measuring the time and space, it takes to compute the results.

* pseudo code:

→ Pseudo code is one of the way to describe an algorithm.

→ Pseudo code uses both natural and programming language to describe an algorithm.

→ Pictorial representation of an algorithm is know

as flowchart.

→ Textual representation of an algorithm is known as pseudo code.

* Pseudo code specifications:

1. Comments begins with // and continue until the end of the line.
2. Blocks are indicated with matching braces { , }. A compound statement (collection of statements) can be represented as a block. The body or procedure or function also forms a block.
3. An identifier begins with a letter.
4. Assignment of values to variables is done using assignment statement.

   Assignment of

   (variable) := (expression) ;

5. Two boolean values true and false. In order to produce these values there are the logical operators and, or, and not and relational operations <, >, ≤, ≥, ≠ are provides
6. Elements of multi-dimensional arrays are accessed using '[' and ']'

7. The following loop statements are used for, whi
do-while (repeat-until).

→ The while loop takes the following form

```
while (condition) do
{
 statement 1;
    :
    :
 statement n;
}
```

→ The for loop takes the following form

```
for var:= value1 to value2 step step do
{
 statement 1;
    :
    :
 statement n;
}
```

→ The repeat-until statement takes the following f

```
repeat
{
 statement 1
   :
 statement 2
} until (condition);
```

8. A conditional statement has the following forms

Ex: 1. if (condition) then ~~statement;~~
statement;

2. if (condition) then

Statement 1

else

Statement 2

9. Input and output are done using the instructions. read and write.

10. There is only one type of procedure Algorithm.

An algorithm consists of a heading and body. The heading takes the form

Algorithm Name (parameter list)

{

.....

}

* Recursive algorithms:

→ There are two types of recursive algorithms.

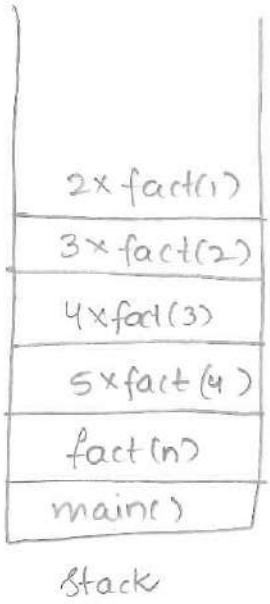
1. Direct recursive algorithms
2. Indirect recursive algorithms.

1. Direct recursive algorithms:

An algorithm called by itself known as direct

recursive algorithms.

```
void main()
{
  ....
f = fact(n);   → actual arguments
  ....
}
int fact (int n)   → formal arguments
{
if(n==1)
return 1;
else
 return (n*fact(n-1));
}
```



Stack

2. Indirect recursive algorithms:

```
void main()
 {
  A();
 }
 A()
 {
  B();
 }
```

```
B( )
{
 A( );
}
```

* Performance Analysis:

→ Determining time and space an algorithm requires is known as performance analysis.

→ Performance analysis contains two components.

1. Time complexity
2. Space complexity.

1. Time Complexity:

→ Time required for an algorithm to perform a given task is known as time complexity.

(or)

→ Number of steps required for an algorithm to ~~performance~~ perform a given task is known as time complexity.

→ Time complexity = compile time + Run time

→ Compile time does not depends on ~~instance~~ characteristics ~~statistics~~ (input values).

→ Compile time is constant or fixed

→ Execution time depends on instance characteristics

(input values).

→ Run time is not fixed that means it varies.

→ We can determine time complexity of an algorithm in two ways.

1. Table method [using frequency]
2. Count variable method

1. Table method (using frequency):

→ First, determine the number of steps for execution (s/e) of the statement.

→ Determine the total number of times each statement is executed.

→ Number of times a statement executes is known as frequency of that statement.

→ By combining this two quantities (i) Step for execution ((s/e), frequency) (ii) frequency, the total continuation of each statement is obtained.

→ By adding the contributions of all the statements, step count for entire algorithm is obtained.

Ex:1. Find ~~table count~~ time complexity for the given algorithm using table method.

L.No	Statement	s/e	frequency	total steps
1	Algorithm Sum(a,n)	0	-	0
2	{	0	-	0
3	s:=0.0;	1	1	1
4	for i:=1 to n do	1	n+1	n+1
5	s:=s+a[i];	1	n	n
6	Return s;	1	1	1
7	}	0	-	0
				2n+3

→ Time complexity of this algorithm is 2n+3.

→ 2n+3 is number of steps required for this algorithm to perform the task.

→ If n=5 then 2n+3 = 13

if n=10 then 2n+3 = 23

where n is instance characteristics (input value)

→ when 'n' value increases, time complexity also increases (run time).

2) Determine the time complexity for the given algorithm addition of two matrices.

Sol:-

L.No	Statement	s/e	frequency	total steps
1.	Algorithm add(a,b,c,m,n)	0	-	0
2.	{	0	-	0
3.	for i=1 to m do	1	m+1	m+1
4.	for j=1 to n do	1	m(n+1)	mn+m
5.	c[i,j]=a[i,j]+ b[i,j];	1	mn	mn
6.	}	0	-	0
				2m+2mn+1

3) Write an algorithm to find n^{th} fibonacci number and find time complexity.

Sol:-

	Statement	s/e	frequency n≤1	frequency n>1	total step n≤1	
1	Algorithm Fibonacci(n)	0	-	-	-	-
2	{	0	-	-	-	-
3	if (n≤1) then	1	1	1	1	1
4	write(n);	1	1	0	1	0
5	else	0	-	-	0	0
6	{	0	-	-	0	0
7	f2:=0; f1:=1;	2	0	1	0	1
8	for i:=2 to n do	1	0	n	0	n
9	{	0	-	-	0	0
10	fn =f1+f2;	1	0	n-1	0	n-1
11	f2=f1; f1=fn;	2	0	n-1	0	2(n-1)
12	}	0	-	-	0	0
13	write (fn);	1	0	1	0	1
14	}	0	-	-	0	0
15	}	0	-	-	0	0
					2	4n

L.No	Statement	s/e	frequency	total steps
1.	Algorithm add(a,b,c,m,n)	0	-	0
2.	{	0	-	0
3.	for i=1 to m do	1	m+1	m+1
4.	for j=1 to n do	1	m(n+1)	mn+m
5.	c[i,j]=a[i,j]+ b[i,j];	1	mn	mn
6.	}	0	-	0
				2m+2mn+1

3) Write an algorithm to find n^{th} fibonacci number and find time complexity.

Sol:-

	Statement	s/e	frequency n≤1	frequency n>1	total step n≤1	
1	Algorithm Fibonacci(n)	0	-	-	-	-
2	{	0	-	-	-	-
3	if (n≤1) then	1	1	1	1	1
4	write(n);	1	1	0	1	0
5	else	0	-	-	0	0
6	{	0	-	-	0	0
7	f2:=0; f1:=1;	2	0	1	0	2
8	for i:=2 to n do	1	0	n	0	n
9	{	0	-	-	0	0
10	fn =f1+f2;	1	0	n-1	0	n
11	f2=f1; f1=fn;	2	0	n-1	0	2n
12	}	0	-	-	0	0
13	write (fn);	1	0	1	0	1
14	}	0	-	-	0	0
15	}	0	-	-	0	0
					2	4n

→ To analyse the time complexity of an algorithm we need to consider two cases $n \leq 1$ and $n > 1$

case i : $n \leq 1$

when $n \leq 1$, lines 3 and 4 get executed once each. Since each line has an steps for execution of one(1) so total step count for this case is 2 that means time complexity is 2 (constant).

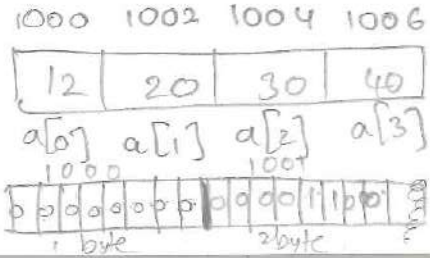
```
2  for(i=1; i<=n; i++)   | n+1 | n
                         | n   | n-1
   Pf("%d", i);
```

case ii : $n > 1$

The total steps for this case is $4n+1$.

4. Write an algorithm to find sum of elements of an array using recursion and find time complexity?

S.No	Statement	s/e	frequency $n \leq 0$	frequency $n > 0$	total steps $n \leq 0$	total steps $n > 0$
1	Algorithm Rsum(a,n) (a → array of elements, n → size)	0	-	-	0	0
2	{	-	-	-	-	-
3	if (n ≤ 0) then	1	1	1	1	1
4	return 0;	1	1	0	1	0
5	else	-	-	-	-	-
6	return (Rsum(a,n-1)+a[n]);	1+x	0	1	0	1+x
7	}	0	-	-	0	0
					2	2+x



$x = t_{Rsum(n-1)}$

To analyze the time complexity of the algorithm a need to consider 2 cases.

case (i): [$n \leq 0$]:

when $n \leq 0$ lines 3, 4 executed once each. Since each lin has s/e of 1. Total step = 2 i.e Time complexity = 2.

case (ii) $n > 0$

The total step count for this case is $x+2$

where $x = t_{Rsum}(n-1)$

2. Count variable method:

→ In this method, count variable is used in the program.

→ Count variable is a global variable initialized with a value 'zero.'

→ Statements to increment count variable by the appropriate amount are introduced into the program. This

→ This is done so that each time a statement original program is executed. Value of count variable is incremented by the step count of that statement.

Ex:

```
Algorithm Sum(a,n)
{
s:=0.0;
count=count+1;  // count is global, it is intially zero
for i:=1 to n do
{
count:=count+1;  // for loop
s:=s+a[i];
count:=count+1;
}
Count:=count+1; // last time of for loop
count:=count+1; // for the return statement
return s;
}
```

Output:
n=5
s=13

* Space Complexity:

→ Space complexity is the amount of memory it means to run.

→ Space needed by an algorithm is the sum of the following components.

1. Fixed part
2. Variable part.

1. Fixed part:

→ A fixed part that consisting of the space needed by instructions, variables, constants etc

2. Variable part:

→ A variable part that consists of the space need by reference variables and the recursion stack space and instance characteristics.

→ The space requirement $S(P)$ of any algorithm 'P' may be written as $S(P) = C + S_P(\text{instance characteristics})$

where C is constant.

→ When analyzing the space complexity of an algorithm we concentrate on estimating instance characteristics (S_P).

→ For any given problem we need to determine which instance characteristics to use to measure the space requirements.

Ex: 1. Find space complexity for given algorithm.

```
Iterative function for sum of array of elements
Algorithm Sum(a,n)
{
 s:=0.0
 for i:=1 to n do
 s:=s+a[i];
 return s;
}
```

→ The given algorithm consisting of three variables (n, i, s) and one array 'a'.

→ The space needed by each variable is 1 word since it is of integer type. So, 3 words of space required for three variables.

→ The space needed by an array 'a' is n words. Since it contains 'n' elements.

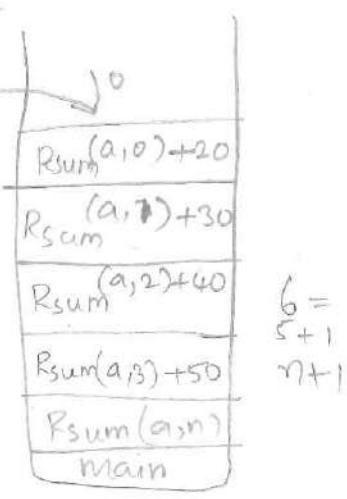
→ The space complexity of this algorithm is

$$S(P) = C + S_p$$

$$S_{Sum(n)} \geq 3 + n$$

2. Find the space complexity for given algorithm.
[Sum of array of elements using recursive function].

```
Algorithm Rsum (a,n)
{
 if (n ≤ 0) then
 return 0;
 else
 return (Rsum (a,n-1) + a[n]);
}
```



→ This algorithm consists of recursive function. So recursive function uses stack concept.

→ The recursion stack space includes space for the formal parameters, local variables and the return address.

→ Assume that the return address requires only one word of memory.

→ Each call to R_{sum} requires atleast three words [value of n, return address and pointer to array 'a'].

→ Since the depth of recursion is n+1. So recursion Stack space needed is $\geq 3(n+1)$

* → Performance evaluation can be divided into two major phases.

1. Priori estimates
2. Posteriori testing

1. Priori estimates:

→ These are also known as performance analysis. Estimating time and space required for an algorithm to perform a task is known as performance analysis. This can be performed before execution.

2. Posteriori testing:

→ It is also known as performance measurement. Estimating time and space required for a program to perform a task is known as performance measurement. This can be performed during execution of program.

* Best case, worst case, Average case in time complexity:

Best case:

Minimum number of steps executed on a given parameter (n) is known as best case

Ex: 1. Time complexity of linear search in best case is $O(1)$

2. Time complexity of binary search in best case is $O(1)$

Average case:

Average number of steps executed on a given parameter is known as average case.

Ex: 1. Time complexity of linear search in average case is $O(n)$

2. Time complexity of binary search in average case is $O(\log n)$

Worst Case:

Maximum number of steps executed on a given parameter is known as worst case.

Ex: 1. Time complexity of linear search in worst case

is O(n).

2. The time complexity of binary search in worst case is O(logn).

* Order of growth:

Measuring performance of an algorithm in relation with input size(n) is known as order of growth.

n	logn	n logn	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65536
32	5	160	1024	4294967296.

Example for best case, worst case and average case

1. Here n=5 10 20 30 40 50

Linear Search:

→ Given input is 10. One comparison is required to search an element 10. So time complexity of linear search in best case is O(1).

→ Given input is 50. '5' comparisons is required to search the element 50. Here input size n=5. So time complexity of linear search is O(n).

→ Given input values are 10, 20, 30, 40, 50 .. 1, 2, 3, 4, 5 comparisons are required to search the elements 10, 20, 30, 40, 50 respectively.

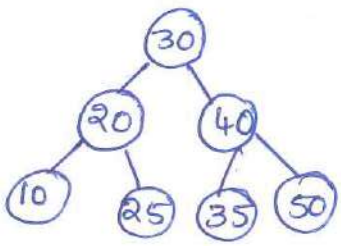
$1+2+3+4+5 = 15$

$Avg = 15/5 = 3$

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

(neglected, neglected)

2. Consider the elements 10, 20, 25, 30, 35, 50, 40.



Binary Search:

→ Given input is 30. '1' comparison is required to search this element. So time complexity of binary search is O(1).

→ Given input is 10. '3' comparisons required to search element 10. So time complexity of binary

Search in worst case is $O\log n$.

$\log_2 7 = 2.81$

$= 3$ (approximately)

→ 1,2,2,3,3,3,3 comparisons are required to search the elements 30,20,40,10,25,35,50 ~~10,20,25,30,35,40,50~~. respectively.

$1+2+2+3+3+3+3 = 17$ total comparisons

$Avg = \frac{17}{7} = 2.42 = 3$ (Approximately)

* Asymptotic Notation (O, Ω, Θ):

Time complexity can be expressed in different notations known as asymptotic notations. They are

1. Big "oh" (O)
2. Omega (Ω)
3. Theta (Θ)
4. Little "oh" (o)
5. Little Omega

1. Big "Oh" Notation:

→ Big "oh" notation, the function $f(n) = O(g(n))$ [read as $f(n)$ is big oh of $g(n)$] if and only if there exists positive constants c and n_0. Such that

$f(n) \leq c * g(n)$ for all n, $n \geq n_0$.

Problem:

1. Convert the given function $f(n) = 2n+3$ into big O notation.

$f(n) = 2n+3$

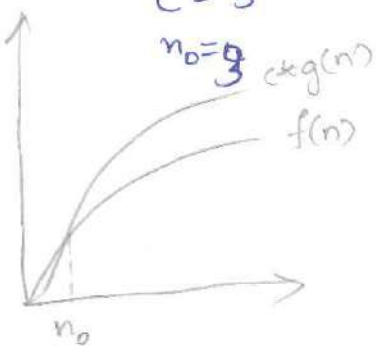
$f(n) = O(g(n))$

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

$2n+3 = O(n)$

$C = 3$

$n_0 = 3$



$f(n) \leq C * g(n)$

$2n+3 \leq C \cdot n$

$2n+3 \leq 3n$

$5 \leq 3$ (F) $n=1$

$7 \leq 6$ (F) $n=2$

$9 \leq 9$ (T) $n=3$

$11 \leq 12$ (T) $n=4$.

→ This statement $f(n) = O(g(n))$ states only that $g(n)$ is an upper bound value of $f(n)$ for all $n \geq n_0$.

→ Value of 'c' must be least as possible.

→ The function $2n+3 = O(n)$ iff there exists positive constants $c=3$ and $n_0 = 3$ such that $2n+3 \leq 3n$ for all n, $n \geq 3$.

notation?

$f(n) = O(g(n))$ iff .....

$f(n) = 3n+2$

$f(n) = O(g(n))$

$3n+2 = O(n)$

$c = 4$

$n_0 = 2$

$f(n) \leq c * g(n)$

$3n+2 \leq c \cdot n$

$3n+2 \leq 4n$

$5 \leq 4$ (F) $n=1$

$8 \leq 8$ (T) $n=2$

$11 \leq 12$ (T) $n=3$

→ The function $3n+2 = O(n)$ iff there exists positive constants c and n_0 $f(n) \leq c * g(n)$ for all n, $n \geq 2$.

3. Convert the function $f(n) = 100n+6$ into big "Oh" notation.

$f(n) = O(g(n))$

$f(n) = 100n+6$

$f(n) = O(g(n))$

$100n+6 = O(n)$

$c = 101$

$n_0 = 6$ $[n \geq n_0, n \geq 6]$

$f(n) \leq c * g(n)$

$100n+6 \leq c \cdot n$

$100n+6 \leq 101 \cdot n$

$n = 6.$

4. Convert the function $f(n) = 10n^2 + 4n + 2$ into big "oh" notation.

$f(n) = 10n^2 + 4n + 2$

$f(n) = O(g(n))$

$10n^2 + 4n + 2 = O(n^2)$

$c = 11$

$n_0 = 5$

$f(n) \leq c * g(n)$

$10n^2 + 4n + 2 \leq c \cdot n^2$

$10n^2 + 4n + 2 \leq 11n^2$

$16 \leq 11$ (F) $n = 1$

$50 \leq 44$ (F) $n = 2$

$n = 3$

$n = 4$

$272 \leq 275$ $n = 5$

5. $f(n) = 6 \cdot 2^n + n^2$ into big "oh" notation.

$f(n) = 6 2^n + n^2$

$f(n) = O(g(n))$

$6 \cdot 2^n + n^2 = O(2^n)$

$c = 7$

$n_0 = 4$

$f(n) \leq c * g(n)$

$6 \cdot 2^n + n^2 \leq c \cdot 2^n$

$6 \cdot 2^n + n^2 \leq 7 \cdot 2^n$

$13 \leq 14$ (T) $n = 1$

$28 \leq 28$ (T) $n = 2$

$57 \leq 56$ (F) $n = 3$

$112 \leq 112$ (T) $n = 4$

6. $f(n) = 2n^2+n^3$ into 'O' notation. (8)

$f(n) = O(g(n))$

$2n^2+n^3 = O(n^3)$

$c=2$

$n_0=2$

$f(n) \leq c * g(n)$

$2n^2+n^3 \leq 2n^3$

$3 \leq 2(F)$ $n=1$

$16 \leq 16(T)$ $n=2$

$45 \leq 54(T)$ $n=3$

27
18

* Time complexities names:

Time Complexity	Names
$O(1)$	constant
$O(n)$	linear
$O(n^2)$	Quadratic
$O(n^3)$	cubic
$O(\log n)$	logn
$O(n \log n)$	nlogn
$O(2^n)$	Exponent

Order of time complexities:

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

2. Omega Notation (Ω):

A function $f(n) = \Omega g(n)$ iff there exists positive constants c and n_0 such that $f(n) \geq c * g(n)$ $\forall n, n \geq n_0$.

Problems:

1. Convert the given function $f(n) = 3n+2$ into Omega.

$f(n) = 3n+2$

$f(n) = \Omega(g(n))$

$3n+2 = \Omega(n)$

$c = 3$

$f(n) \geq c * g(n)$

$f(n) \geq c.n$

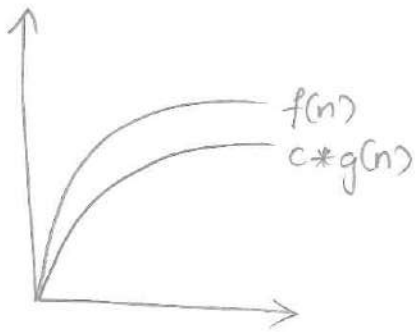
$3n+2 \geq 3n$

$5 \geq 3 \quad n=1$ T

$8 \geq 6 \quad n=2$ T

$11 \geq 9 \quad n=3$ T

$n_0 = 1 \quad (n \geq 1, \; n \geq n_0)$



→ The function $3n+2 = \Omega(n)$ iff there exist positive constants $c=3, n_0=1$ such that $3n+2 \geq 3n \; \forall n, n \geq 1$.

→ The function $g(n)$ is only a lower bond on $f(n)$

↔ value of c must be maximum as possible.

2. Convert the function $2n+3$ into Ω.

$f(n) = 2n+3$

$f(n) = \Omega(g(n))$

$2n+3 = \Omega(n)$

$c = 2, \; n_0 = 1 \quad (n \geq 1, \; n \geq n_0)$

$f(n) \geq c * g(n)$

$2n+3 \geq 2n \quad [\because c=2]$

$2n+3 \geq 2n$

$5 \geq 2 \quad n=1$ T

$8 \geq 4 \quad n=2$ T

$9 \geq 6 \quad n=3$ T

3. $10n^2+4n+2$.

$f(n) = 10n^2+4n+2$

$10n^2+4n+2 = \Omega(n^2)$

$C=10$ $\left(\begin{matrix} n \geq 1 \\ n \geq n_0 \end{matrix}\right)$

$n_0 = 1$

$f(n) \geq C * g(n)$

$10n^2+4n+2 \geq Cn^2$ $[\because C=10]$

$16 \geq 10$ $n=1$ T

$50 \geq 40$ $n=2$ T

$104 \geq 90$ $n=3$

4. $6 \cdot 2^n + n^2$

$f(n) = 6 \cdot 2^n + n^2 = \Omega(g(n))$

$6 \cdot 2^n + n^2 = \Omega(2^n)$

$f(n) \geq C * g(n)$

$6 \cdot 2^n + n^2 \geq C \cdot 2^n$

$6 \cdot 2^n + n^2 \geq 6 \cdot 2^n$

$13 \geq 12$ $n=1$

$28 \geq 24$ $n=2$

$57 \geq 48$ $n=3$

$C=6, n_0=1$ $\left(\begin{matrix} n \geq 1 \\ n \geq n_0 \end{matrix}\right)$

3. Theta (Θ) Notation:

The function $f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2, and n_0 such that

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \quad \forall\, n_1, n \geq n_0$$

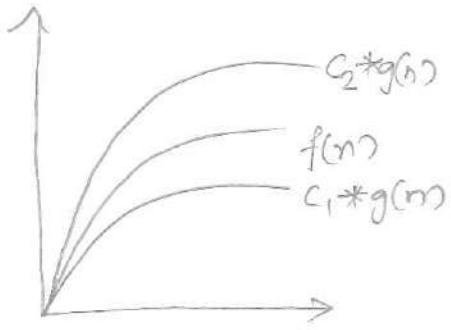
1. Convert the function $3n+2$ into 'θ' notation.

Sol: Given

$f(n) = 3n+2$

$f(n) = \theta(g(n))$

$3n+2 = \theta(n)$



$c_1 * g(n) \le f(n) \le c_2 * g(n)$

$f(n) \le c_2 * g(n)$

$3n+2 \le c_2 * n$

$3n+2 \le 4n$

$5 \le 4$ F $n=1$

$8 = 8$ T $n=2$

$11 \le 12$ T $n=3$

$c_2 = 4, n_0 = 2.$

$[n \ge 2, n \ge n_0]$

$c_1 * g(n) \le f(n)$

$c_1 * n \le 3n+2$

$3n \le 3n+2$

$3 \le 5$ $n=1$ T

$6 \le 8$ $n=2$ T

$c_1 = 3, n_0 = 1$

$[n \ge 1, n \ge n_0]$

$c_1 = 3,\ c_2 = 4$

$n_0 = 1, 2, 3, \ldots\ldots, n_0 = 2, 3, 4 \ldots\ldots$

→ The function $3n+2 = \theta(n)$ iff there exists positive constants $c_1 = 3$, $c_2 = 4$ & $n_0 = 2$ such that

$3n \le 3n+2 \le 4n \ \forall n,\ n \ge 2$

→ Note: The 'θ' notation is more precious than both O & 'Ω' notations.

→ The $f(n)$ is $\Theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.

4. Little 'oh(o):

The function $f(n) = o(g(n))$ iff there exists

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

1. Convert $3n+2$ into little 'oh':

$f(n) = 3n+2$

$f(n) = o(g(n))$

$3n+2 = o(n^2)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$\lim_{n\to\infty} \frac{3n+2}{n^2} = \lim_{n\to\infty} \frac{3n}{n^2} + \frac{2}{n^2}$$

$$= \lim_{n\to\infty} \frac{3\cdot n}{n\cdot n} + \frac{2}{n\cdot n}$$

$$= \lim_{n\to\infty} 3\times\frac{1}{n} + 2\times\frac{1}{n}\times\frac{1}{n}$$

$$= 0+0=0.$$

5. Little 'Omega' (ω):

The function $f(n) = \omega(g(n))$ iff $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

Problem:

1. Write an algorithm for selection sort and find time complexity.

		s/e	Step	Total
1.	Algorithm Selectionsort (a, n)	0	-	-
2	{	0	-	-
3	for i=1 to n do	1	n+1	n+1
4	{	0	-	-
5	j:=i;	1	n	n
6	for k:=i+1 to n do	1	n×n	n^2
7	if (a[k] <a[j]) then	1	n(n-1)	n^2-n
8	j:=k;	1	n(n-1)	n^2-n
9	t:=a[i]; a[i]:=a[j]; a[j]:=t;	3	n	3n
10	}	0	-	-
11	}	0	-	-
				$3n^2+3n+1$

→ $3n^2+3n+1$ is number of steps required for Selection Sort algorithm to perform the given task.

→ Here $f(n) = 3n^2+3n+1$ convert this function into big 'oh' notation

$f(n) = O(g(n))$ $\quad f(n) \leq c * g(n)$

$f(n) = O(n^2)$ $\quad 3n^2+3n+1 \leq c \cdot n^2$

$c = 4$

$n_0 = 4$

$7 \leq 4$	$n=1$
$19 \leq 16$	$n=2$
$37 \leq 36$	$n=3$
$61 \leq 64$	$n=4$

→ Time complexity of selection sort is big 'oh' notation.

$3n^2+3n+1 = O(n^2)$

* Towers of Hanoi problem is an example of recursive algorithm:

→ There was a diamond tower (labelled 'A') with 64 golden disks. The disks were of decreasing size and were stacked on the tower in decreasing order of size from bottom to top.

→ Besides this tower there were two other diamond towers [labelled 'B' & 'C'].

→ Since the time of creation Brahman priests have been attempting to move the disk from tower 'A' to tower 'B' using tower 'C' were intermediate stories.

→ As the disks were very heavy they can moved only one at a time.

→ In addition, at no time can a disk be on a top of smaller disc

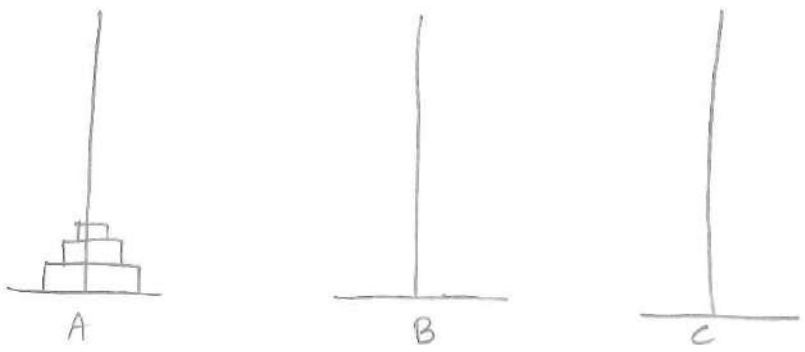
```
Algorithm Towers of Hanoi (n, x, y, z)
{
if (n ≥ 1) then
{
 Towers of Hanoi (n-1, x, z, y);
 write ("move top disk from tower", x, " to top of
                    tower", y);
```

Towers of Hanoi(n-1, z, y, x);

}
}

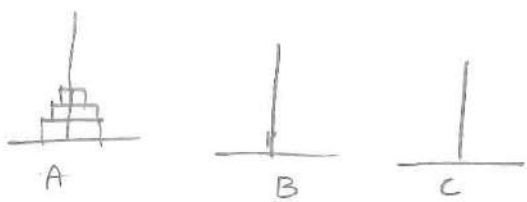
Recurrence Relation of towers of Hanoi is $2T(n-1)+1$

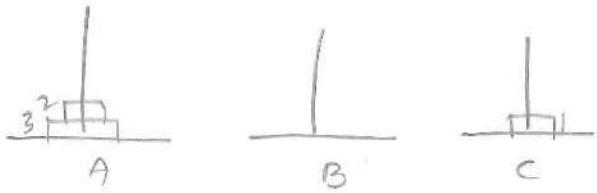
→ Assume that tower 'A' consists of 3 discs and move discs from 'A' to 'B' by using 'c' as intermediate

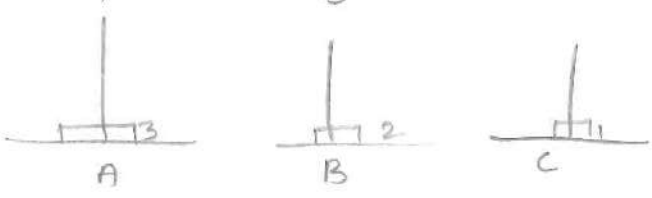


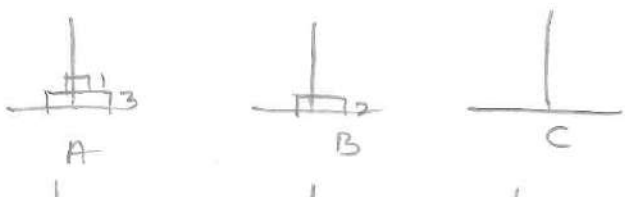
one-move
Heavy - Small

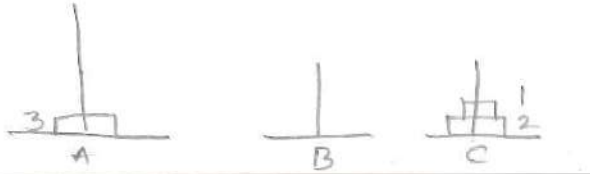
Sol:-

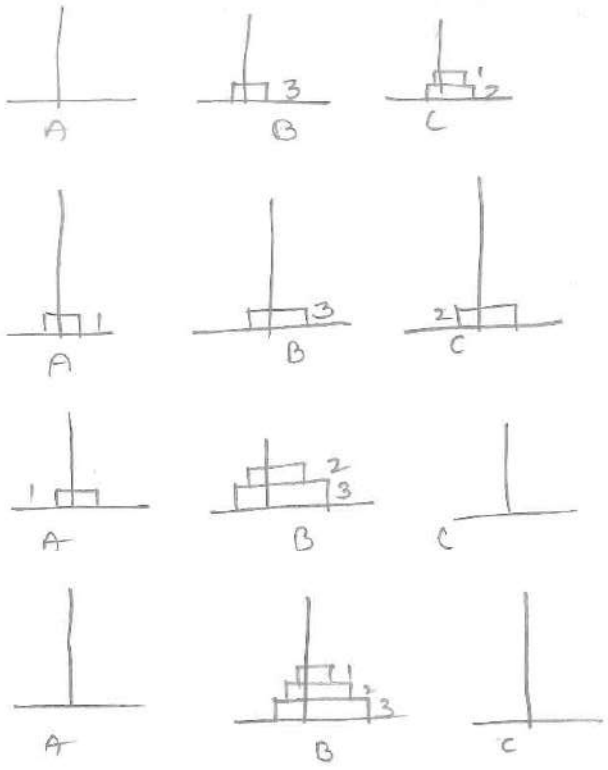












*Divide and conquer:

→ Divide and conquer is a method to develop an algorithm. In this method, if the problem is large, then it divides the problem into k-subproblems.

→ If the sub-problem is still large then divide the sub-problem into smaller sub problems.

→ Apply divide and conquer strategy on sub-problems until sub problem becomes small enough to find the answer and find solutions of each and every sub-problem. Atlast combine solutions of the sub-problem to get the solution for given problem.

* Control abstraction or general method for divide and conquer :

```
Algorithm  D and C(P)
{
if (small (P)) then
    return S(P);
else
{
divide P into smaller instances P1, P2, .... Pk, k ≥ 1;
Apply D and C to each of these subproblems.
return combine (D and C (P1), D and C (P2) ... D and C (Pk));
}
}
```

→ Control abstraction consists of three methods

1. Small (P)
2. S(P)
3. Combined function.

1. Small(P):

→ Small (P) is a boolean function that determines whether the problem is small or not. If the problem is small then boolean function returns true then the function S(P) is invoked to find the answer.

→ If the problem is large, boolean function returns false. Now the problem is divided into smaller sub problems. These sub problems $P, P_2, \ldots$ $\ldots P_k$ are solved by recursive applications of D And C.

→ Combine is a function that determines the solution to the given problem 'p' by using solutions of k-sub problems.

→ If the size of the problem 'p' is n and the size of the k-sub problems are $n, n_2, \ldots n_k$ respectively. Then computing time of D And C is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n\text{-small} \\ T(n_1) + T(n_2) + \ldots\ldots + T(n_k) + f(n) & n \geq 1 \end{cases}$$

where $T(n)$ = Time complexity of divide and conquer algorithm on any input size of 'n'.

$g(n)$ = Time complexity of the algorithm for smaller inputs

$f(n)$ = The function $f(n)$ is the time for dividing the problem 'p' and

combining these solutions to sub problems.

→ The time complexity of divide and conquer algorithms is given by recurrence relation

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

where a, b are known constants and

$n = b^k$

→ One of the methods for solving such recurrence relation is called the substitution method.

1) Consider the case $a=1, b=2, f(n)=1$ and solve the recurrence relation.

$$T(n) = 1 \cdot T(n/2) + 1$$

$$T(n) = T(n/2) + 1$$

$$T(n/2) = T((n/2/2) + 1) + 1$$

$$= (T(n/4) + 1) + 1 \quad = T(n/4) + 2$$

$$T(n/4) = ((T(n/4/2) + 1) + 1) + 1)$$

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

$$= T(n/2^3) + 3$$

$$= \vdots$$

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

$= T(n/n) + \log_2 n$

$= T(1) + \log_2 n$

$= 1 + \log_2 n$

$= O(\log n)$

$n = 2^k$

$k = \log_2 n$

2) Consider the case in which $a=2$ and $b=2$. let $T(1)=2$ and $f(n)=n$

Sol:- $T(n) = 2T(n/2) + n$

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

$= 2[2T(n/4) + n/2] + n$

$= 2[2[2T(n/8) + n/4] + n/2] + n$

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

$= 2^3 T(n/2^3) + 3n$

$\vdots$

$= 2^k T\left(\frac{n}{2^k}\right) + kn$

$= nT(n/n) + n\log_2 n$

$= nT(1) + n\log_2 n$

$= 2n + n\log_2 n.$

$b = 2$

$n = b^k$

$n = 2^k$

$k = \log_2 n$

3. Solve the recurrence relation for the following choices of a, b and $f(n)$; is a constant.

a) $a=1, b=2, f(n)=cn$

b) $a=5, b=4, f(n)=cn^2$

(a) $a=1, b=2, f(n)=nc$

$T(1)=2$

$$T(n) = a \cdot T(n/b) + f(n)$$

$$= 1 \cdot T(n/2) + cn$$

$$T(n/2) = (1 \cdot T(n/4) + c\,n/2) + cn$$

$$= T(n/4) + c\,n/2 + nc$$

$$T(n/8) \rightarrow [[[1 \cdot T(n/8) + c\,n/4] + cn/2] + nc]$$

$$= T(n/8) + \frac{nc}{4} + \frac{nc}{2} + nc$$

$$= 1 \cdot T(n/8) + \frac{7n}{4}$$

$b=2$
$n=b^k$
$n=2^k$
$k=\log_2 n$

$$= T(n/2^3) + \frac{7}{4}n$$

$$= T(1) + \frac{7}{4}n$$

$$= 1 + 7/4\,n$$

b) $a=5, b=4, f(n)=cn^2$

$$T(n) = a \cdot T(n/b) + f(n)$$

$$= 5 \cdot T(n/4) + cn^2$$

$$T(n) = 5 \cdot T(n/4) + cn^2$$

$$T(n/4) = 5\left[5 \cdot T(n/4/4) + c(n/4)^2\right] + cn^2$$

$$T(n/16) = 5\left[5\left[5T(n/16/4) + c(n/16)^2\right] + c(n/4)^2\right] + cn^2$$

$$= 5\left[5\left[5T(n/64) + c(n^2/256)\right] + \frac{n^2}{16}\right] + n^2$$

$$= 5^3 T(n/4^3) + \frac{n^2 + 16n^2 + 256n^2}{256}$$

$$=$$

* Divide and conquer Applications:

· This design technique is used in following applications

1. Binary search
2. Quick sort
3. Merge sort
4. Strassen's matrix multiplication

1. Binary search:

→ Binary search is a searching technique. It takes list of elements and searching element as inputs.

→ It compares an element to be searched with middle element. If searching element is less than

the middle element then searching process continues in the left part of the list.

→ If the searching element is greater than middle element the searching process continues in the right part of the list.

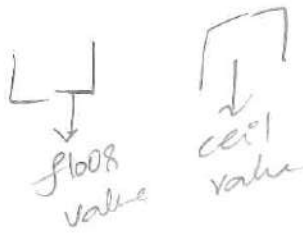
→ This process continues until the end of the list.

*Note:

List of elements divided into two parts based on middle element.

$$\text{middle element} = \frac{low + high}{2}$$

i) write an algorithm for binary search using Iterative method (Non-recursive method)




```
Algorithm Binsearch (a, n, x)
{
 low := 1;
 high := n;
 while (low ≤ high) do
 {
  mid := ⌊(low + high)/2⌋;
  if (x < a[mid]) then
     high = mid - 1;
  else if (x > a[mid]) then
   low = mid + 1;
  else
```

```
        return mid;
    }
}
```

Time complexity:

$$T(n) = 1 \cdot T(n/2) + 1$$

$$= T(n/2) + 1$$

$$= T(n/4) + 1 + 1$$

$$= T(n/8) + 1 + 1 + 1$$

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

$$= T(n/2^3) + 3$$

$$\vdots$$

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

$$= T(n/n) + \log_2 n \quad \left[\because n = b^k,\ n = 2^k,\ k = \log_2 n\right]$$

$$= T(1) + \log_2 n$$

$$= 1 + \log n$$

$$= O(\log n)$$

Binary search using recursive function:

```
Algorithm Binsearch (a, i, l, x)
{
 if (l == i) then
  if (x == a[i]) then
    return i;
```

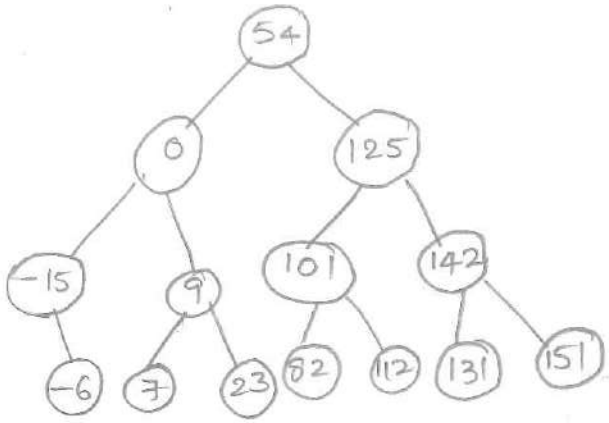
```
else
 return 0;
 }
else
{
mid= ⌊(i+l)/2⌋;
 if (x==a[mid]) then
   return mid;
 else if (x<a[mid]) then
   return Binsearch (a,i,mid-1,x);
 else
 return Binsearch (a,mid+1,l,x);
  }
}
```

2) Consider the following elements -15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151. Construct binary decision tree or binary search tree for given elements.

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

$$mid = \left\lfloor \frac{l+h}{2} \right\rfloor = \left\lfloor \frac{1+14}{2} \right\rfloor = \left\lfloor \frac{15}{2} \right\rfloor = \lfloor 7.5 \rfloor = 7.$$

Comparisons-	3	4	2	4	3	4	1	4	3	4	2	4	3	4



Best - O(1)

Worst - O(logn)

$n = 14$

$\log_2 14 = 3.21$

→ In the above binary search tree, minimum number of comparisons required to search an element 54 is 1. So time complexity of binary search in best case on successful search is O(1).

→ Maximum number of comparisons required to search the elements -6, 7, 23, 82, 112, 131, 151 is 4.

$n = 14$

$\log n = \log_2 14 = 3.8$ (Approximately $\simeq 4$)

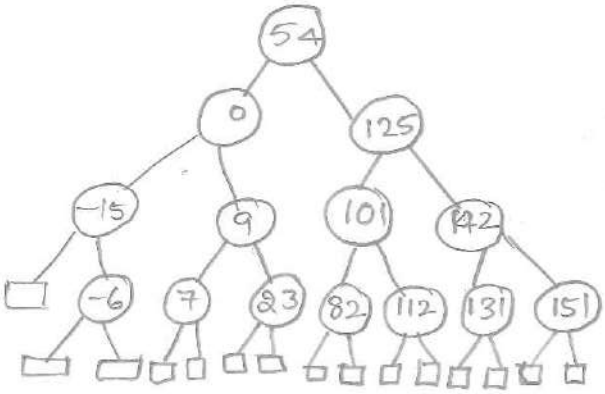
Time complexity of binary search in worst case on successful search is O(logn).

→ Average number of comparisons required to find all 14 elements is

$$\frac{3+4+2+4+3+4+1+4+3+4+2+4+3+4}{14} = 3.21 \text{ (Approx} \simeq 4)$$

Time complexity of binary search in average case on successful search is O(logn).

*Time complexity of binary search on unsuccessful search:



→ Circular nodes are called internal nodes.

→ Square nodes are refered as External nodes.

→ All successful searches end at a circular node.

→ All unsuccessful searches end at a square node.

→ There are 15 possible ways that an unsuccessful search may terminate depending on value of 'x'.

→ Average number of elements comparison for an unsuccessful search is $(3+14\times4)/15 = 59/15 = 3.93$

(Approxi ≅ 4)

→ Time complexity of binary search in best case on unsuccessful search is O(logn)

→ Time complexity of binary search in Avg case on unsuccessful search is O(logn)

→ Time complexity of binary search in worst case on unsuccessful search is O(logn).

* Merge Sort:

→ In this method given sequence of a elements divided into two sets and each set individually sorted and the resulting sorted sequence are merged to produce a single sorted sequence of n elements.

Ex: Consider the array of 10 elements.

a[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)

→ Algorithm merge sort begins by splitting a[1:10] into two subarrays each of size five (a[1:5] and a[6:10])

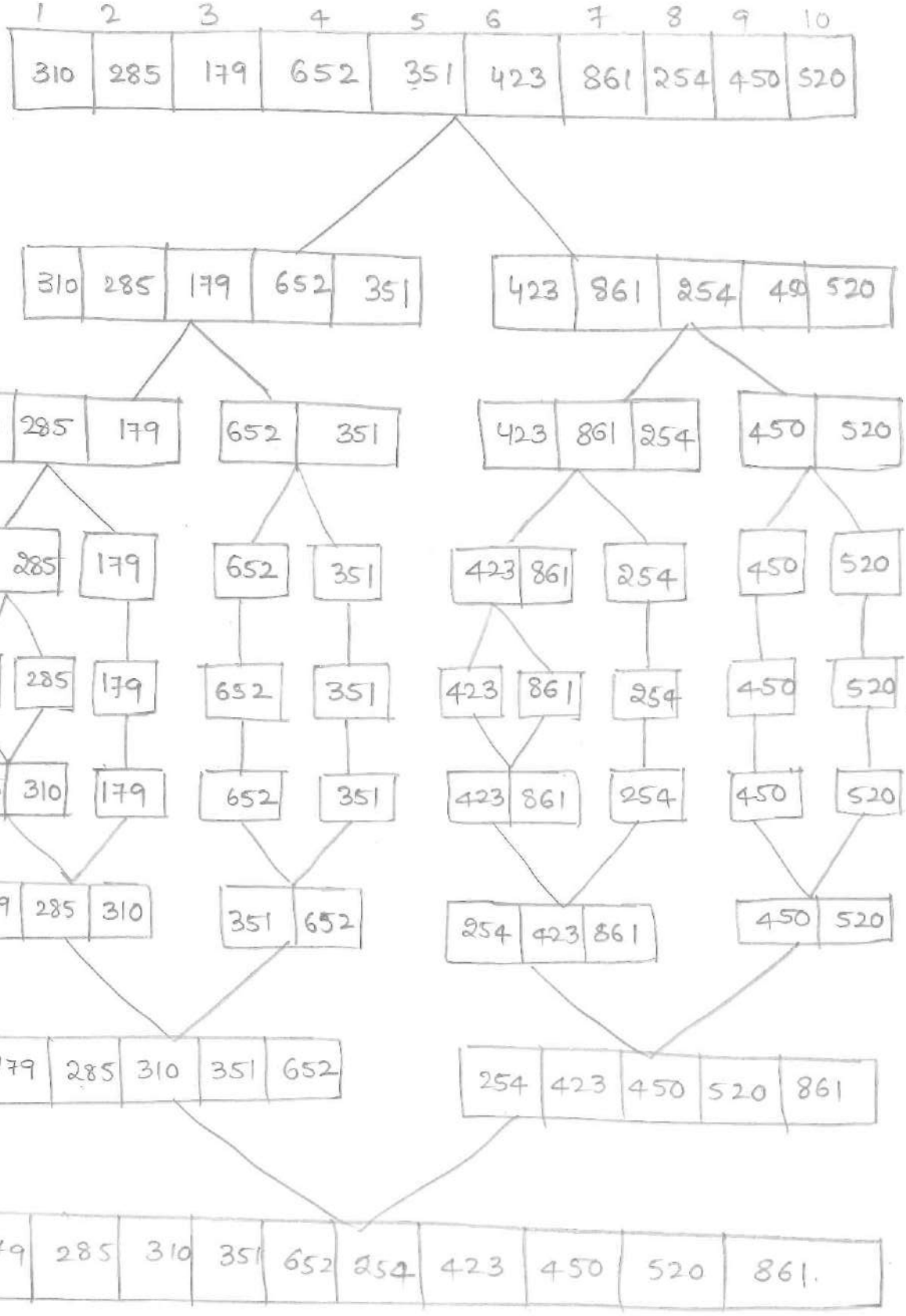
→ The elements in a[1:5] are then splitted into 2 subarrays of size 3 (a[1:3]) and two a[4:5]

→ Then elements in a[1:3] are splitted into subarrays of size two (a[1:2]) and one a[3:3].

→ The elements in a[1:2] are split into one element solution and now the meging begins.

→ Note that no moment of data has yet taken place

→ A record of the subarrays is implicity maintained by the reaursive mechanisms.


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

```
Algorithm Mergesort (low, high)
{
if (low < high)
{
  mid = (low + high)/2;
  merge sort (low, mid);
  merge sort (mid+1, high);
  merge sort (low, mid, high);
 }
}
Algorithm merge (low, high, mid)
{
i = low; j = mid+1; k = low;
while ((i ≤ mid) and (j ≤ high)); do
{
if (a[i] ≤ a[j]) then
{
 b[k] = a[i]; i = i+1;
}
else
{
 b[k] = a[j]; j = j+1;
 }
 k = k+1;
 }
 if (i > mid) then
 for m = j to high do
 {
```

```
b[k]=a[m];
k=k+1;
}
else
for m=i to mid do
{
b[k]=a[m];
k=k+1;
}
}
```

Time complexity:

$T(n) = a \cdot T(n/b) + f(n)$

$= 2 \cdot T(n/2) + c \cdot n$

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

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

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

$= 2^3 T(n/2^3) + 3n$

$= 2^5 T(n/2^3) + 3n$

$= 2^k T(n/n) + n \cdot k$

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

$= n + n \log n$

$= n(1 + \log n)$

$n = b^k$

$n = 2^k$

$k = \log_2 n$

Time complexity of merge sort is $O(n \log n)$

Time complexity of bubble sort, selection sort, insert sort is $O(n^2)$

*Quick sort:

Quick sort is an efficient sorting method. Divide and conquer method used in quicksort

Description:

→ First element is considered as pivot element, partioning element.

→ place the pivot element in correct position in such a way that left side elements should be less than pivot element and right side elements should be greater than pivot element.

→ Repeat this process for left part and right part of the pivot element until elements are sorted.

→ Consider the following example

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(I)	(P)
65	70	75	80	85	60	55	50	45	2	9
65	45	75	80	85	60	55	50	70	3	8
65	45	50	80	85	60	55	75	70	4	7
65	45	50	55	85	60	80	75	70	5	6

65	45	50	55	60	85	80	75	70	6	5
60	45	50	55	65	85	80	75	70		

→ First element 65 is considered as pivot element. place the pivot element 65 in the correct position in such a way that left side element should be less than the pivot element and the right side element should be greater than the pivot element. The list is divided into two parts left part contains elements 60, 45, 50, 55 and right part contains 85, 80, 75, 70

→ Each part is sorted separately.

Algorithm:

```
Algorithm Quicksort (p,q)
{
if (p<q) then
{
j := partition (a, p, p+1);
Quicksort (p, j-1);
Quicksort (j+1, q);
}
}
```

```
Algorithm partition (a, i, n)
{
pivot = a[i]; lb = i, ub = n+1;
```

```
repeat
{
repeat
 lb=lb+1;
until (a[lb] ≥ pivot);
repeat
 ub:=ub-1;
until (a[ub]≤ pivot);
if (lb< ub) then interchange (a, lb, ub);
}until (lb≥ ub);
 a[i]=a[ub]; a[ub]=pivot ; return ub;
}
Algorithm Interchange (a,lb,ub)
{
 temp=a[lb];
 a[lb]=a[ub]; a[ub]=temp;
 }
```

* Strassen's matrix multiplication:

Basic matrix multiplication : This requires 8 multiplications and 4 additions to find product of 2 matrices.

Consider the following ex:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$C_{11} = (A_{11} \cdot B_{11}) + (A_{12} \cdot B_{21})$

$C_{12} = (A_{11} \cdot B_{12}) + (A_{12} \cdot B_{22})$

$C_{21} = (A_{21} \cdot B_{11}) + (A_{22} \cdot B_{21})$

$C_{22} = (A_{21} \cdot B_{12}) + (A_{22} \cdot B_{22})$.

→ Note that no. of elements $n=8$

→ Matrix A contains $n/2$ elements (4)

→ Matrix B contains $n/2$ elements (4)

→ To find product of 2 matrices A and B we need to perform 8 multiplications and 4 additions.

→ Overall computing time $T(n)$ of the resulting of divide and conquer is given by the reference.

$$T(n) = \begin{cases} b & n \leq 2 \\ 8 \cdot T(n/2) + cn^2 ; & n > 2 \end{cases}$$

→ Two matrices can be added in time cn^2 for some constant 'c'.

→ This recurrence relation can be solved using substitution method to obtain $T(n) = O(n^3)$.

→ Volkers strazen has discovered a way to compute the product of two matrices using only 7 multiples and 18 additions / substractions.

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = (A_{21} + A_{22}) B_{11}$

$R = A_{11} (B_{12} - B_{22})$

$S = A_{22} (B_{21} - B_{11})$

$T = (A_{11} + A_{12}) B_{22}$

$U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$

$V = (A_{12} - A_{22}) (B_{21} + B_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

→ The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + cn^2 & n > 2 \end{cases}$$

$T(n) = 7T(n/2) + n^2$

$= 7\left[7 \cdot T(n/4) + (n/2)^2\right] + n^2$

$= 7\left[7\left[7T(n/8) + (n/4)^2\right] + (n/2)^2\right] + n^2$

$= 7^3 T(n/8) + 7^2 \frac{n^2}{4^2} + 7 \frac{n^2}{2^2} + n^2$

$= 7^3 T(n/2^3) + n^2\left[\left(\frac{7}{4}\right)^2 + \frac{7}{4} + 1\right]$

$= 7^K T(n/2^K) + n^2\left[\left(\frac{7}{4}\right)^{K-1} + \ldots\ldots + \left(\frac{7}{4}\right)^2 + \left(\frac{7}{4}\right)^1\right]$

$= 7^{\log_2 n} T(n/n) + n^2 \left[(7/4)^{\log_2 n}\right]$ $\quad [a^{\log_c b} = b^{\log_c a}]$

$= n^{\log_2 7} T(1) + n^2 \dfrac{7^{\log_2 n}}{4^{\log_2 n}}$

$= n^{\log_2 7} \cdot 1 + n^2 \dfrac{n^{\log_2 7}}{n^{\log_2 4}}$

$= n^{\log_2 7} + n^2 \dfrac{n^{\log_2 7}}{n^2}$

$= n^{\log_2 7} + n^{\log_2 7}$

$= 2n^{\log_2 7}$

$= O(n^{\log_2 7})$

$= O(n^{2.81})$