UNIT – I

Introduction: Algorithm Definition, Algorithm Specification, Performance Analysis, Performance Measurement, Asymptotic notations.
Divide and Conquer: General Method, Binary Search, Finding the Maximum and Minimum, Quick Sort.

Algorithm Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the input into the output.

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms should satisfy the following criteria.

Characteristics of an Algorithm:

1. INPUT → Zero or more quantities are externally supplied.
2. OUTPUT → At least one quantity is produced.
3. DEFINITENESS → Each instruction is clear and unambiguous.
4. FINITENESS → If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. EFFECTIVENESS → Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Differences between algorithm and program:

Algorithm	Program
1. It is a step by step procedure for solving the given problem.	1. A program is nothing but a set of instructions or executable code.
2. An algorithm is designed by designer	2. The program can be implemented by a programmer.
3.An algorithm is done at design phase	3. A program is implemented at implementation phase.
4. An algorithm can be expressed by using English, flowchart and pseudo code.	4. A program can be written by using languages like C, C++, Java etc…
5. After writing the algorithm, we have to analyze it using space and time complexities.	5.After writing a program, we have to test them

Algorithm Specification: Algorithm can be described in three ways.

1. Natural language like English: When this way is choose care should be taken, we should ensure that each & every statement is definite.
2. Graphic representation called flowchart: This method will work well when the algorithm is small& simple.
3. Pseudo-code Method: In this method, we should typically describe algorithms as program, which resembles language like PASCAL & ALGOL

Pseudo-Code Conventions: The following are set of rules need to be followed while writing algorithms

1. Comments begin with **//** and continue until the end of line.
2. Blocks are indicated with matching braces { and }. A compound statement can be represented as a block. The body of a procedure also forms a block. Statements are delimited by **;**.
3. An identifier begins with a letter. The data types of variables are not explicitly declared. Whether a variable is local or global to a procedure will also be evident from the context.
4. Compound data types can be formed with records. Here is an example,

```
Node= Record
{
  data type – 1   data-1;
        .
        .
        .
  data type – n  data – n;
  node * link;
}
```

   Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

5. Assignment of values to variables is done using the assignment statement.

   <Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.

   → Logical Operators AND, OR, NOT

   →Relational Operators <, <=,>,>=, =, !=

7. The following looping statements are employed.

   For, while and repeat-until

While Loop:

```
While < condition > do
{
      <statement-1>
            .
            .
```

.

<statement-n>
}

As long as condition is TRUE, the statements get executed. When condition becomes FALSE, the loop is exited. The value of condition is evaluated at top of the loop. The general form of For loop is

For Loop:

for variable: = value 1 to value 2 **step** step do
{
<statement-1>
.
.
.
<statement-n>
}

Here value 1, value 2 and step are arithmetic expressions. A variable of type integer or real or a numerical constant is a simple form of an arithmetic expression. The clause "**step** step" is optional and taken as +1 if it does not occur. Step could either be positive or negative. Variable is tested for termination at the start of each iteration. The repeat-until loop is constructed as follows.

repeat-until:

repeat
<statement-1>
.
.
.
<statement-n>
until<condition>

The statements are executed as long as condition is false. The value of condition is computed after executing the statements. The instruction **break;** can be used within any of the above looping instructions to force exit. In case of nested loops, **break;** results in the exit of the innermost loop that it is a part of. A **return** statement within any of the above also will result in exiting the loops. A return statement results in the exit of the function itself.

8. A conditional statement has the following forms.

→ If <condition> then <statement>

→ If <condition> then <statement-1>
else <statement-1>

Here condition is the Boolean expression and statements are arbitrary statements.

Case statement:

```
Case
{
        : <condition-1> : <statement-1>
                          .
                          .
                          .
        : <condition-n> : <statement-n>
        : else : <statement-n+1>
}
```

Here statement 1, statement 2 etc. could be either simple statement or compound statements. A case statement is interpreted as follows. If condition 1 is true, statement 1 gets executed and case statement is exited. If statement 1 is false, condition 2 is evaluated. If condition 2 is true, statement 2 gets executed and the case statement exited and so on. If none of the conditions are true, statements n+1 is executed and the case statement is exited. The else clause is optional.

9. Elements of multidimensional arrays are accessed using [and]. For example, if A is a two dimensional array, the <i,j>th element of an array is denoted as A[i,j].
10. Input and output are done using the instructions read & write.
11. There is only one type of procedure:
    Algorithm, the heading takes the form,
        Algorithm Name (Parameter lists)
    Where Name is the name of the procedure and parameter list is a listing of the procedure parameters. The body has one or more statements enclosed with braces { and }. An algorithm may or may not return values. Simple variables to procedures are passed by value. Arrays and records are passed by reference. An array name or record name is treated as a pointer to the respective data type.

→ As an example, the following algorithm fields & returns the maximum of 'n' given numbers:

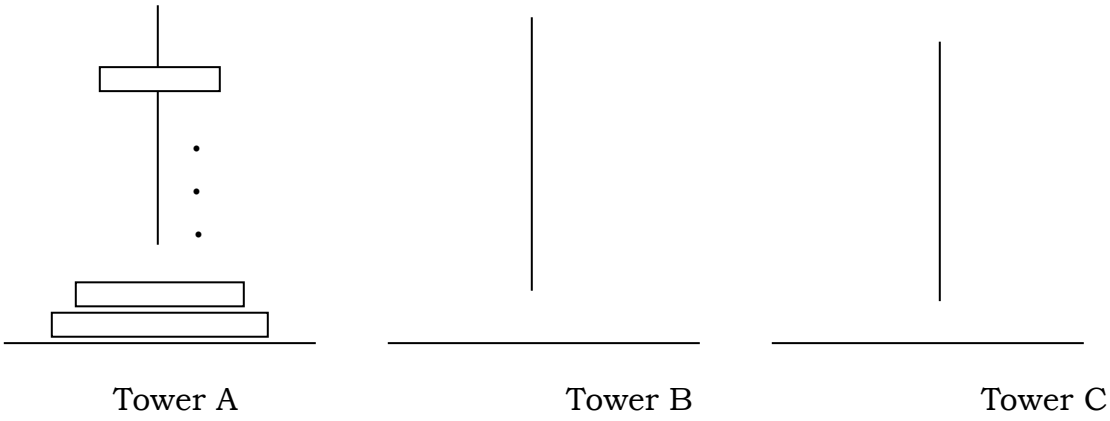
```
Algorithm Max(A,n)
// A is an array of size n
{
    Result := A[1];
    for I:= 1 to n do
      if A[I] > Result then
           Result :=A[I];
     return Result;
}
```

In this algorithm (named Max), A & n are procedure parameters. Result & I are Local variables.

Recursive Algorithms:

- A Recursive function is a function that is defined in terms of itself.
- Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself is Direct Recursive.
- Algorithm 'A' is said to be Indirect Recursive if it calls another algorithm which in turns calls 'A'.
- The following 2 examples show how to develop recursive algorithms.
- In the first, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.

1. Towers of Hanoi:



- According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks.
- The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top.
- Besides these tower there were two other diamond towers(labeled B & C)
- Goal is to move the disks from tower A to tower B using tower C, for intermediate storage.
- As the disks are very heavy, they can be moved only one at a time.
- In addition, at no time can a disk be on top of a smaller disk.
- According to legend, the world will come to an end when the priest have completed this task.
- A very elegant solution results from the use of recursion.
- Assume that the number of disks is 'n'.
- To get the largest disk to the bottom of tower B, we move the remaining 'n-1' disks to tower C and then move the largest to tower B.

- Now we are left with the tasks of moving the disks from tower C to B.
- To do this, we have tower A and B available.
- The fact, that towers B has a disk on it can be ignored as the disks larger than the disks being moved from tower C and so any disk scan be placed on top of it.

Algorithm:

```
Algorithm TowersofHanoi(n, s, d, a)
//Move the top 'n' disks from tower s to tower d.
{
    if n = 1 then
            write("Move disk from s to d");
    else
            TowersofHanoi(n-1, s, a, d);
                    Write("Move disk from s to d);
            Towersofhanoi(n-1, a, d, s);
    }
}
```

2. **Recursive algorithm for Factorial Of Given Number:**

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

3. **Recursive algorithm for GCD of two numbers:**

```
Algorithm rgcd(a,b)
{
    if a!=b then
    {
            if a > b then
            {
                    a := a – b;
                    rgcd(a,b);
            }
            else
            {
                    b := b- a;
                    rgcd(a,b);
            }
```

```
    }
    return a;
}
```

Analysis: Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size 'n' as a function of n and the running time on inputs of smaller sizes.

Performance Analysis: The efficiency of an algorithm is declared by measuring the performance of an algorithm. Performance of an algorithm can be computed using Space and Time complexities. Given algorithm can be analyzed in two ways:

1. **Space Complexity:** The space complexity of an algorithm is the amount of space or memory it needs to run to compilation.
2. **Time Complexity: T**he time complexity of an algorithm is the amount of computer time it needs to run to compilation.

1. **Space Complexity:** The Space needed by each of these algorithms is seen to be the sum of the following component.

   a. A **fixed part** that is independent of the characteristics (eg: number, size) of the inputs and outputs.

   The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

   b. A **variable part** that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space.

   The space requirement s(p) of any algorithm p may therefore be written as,

   $S(P) = c + S_p$ (Instance Variable)

   Where 'c' is a constant variable.

   Example 1: Compute Space complexity for the following examples:

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

   Here, the above algorithm contains three fixed part variables (which requires 3 words of memory), and no variable part (hence 0). Hence S(P) = 3

Example 2:

```
Algorithm sum(a,n)
{
        s=0.0;
        for I=1 to n do
        s= s+a[I];
        return s;
}
```

- The problem instances for this algorithm are characterized by n, the number of elements to be summed. The space needed d by 'n' is one word, since it is of type integer.
- The space needed by 'a'a is the space needed by variables of type array of floating point numbers.
- This is at least 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed.
- So, we obtain S(P) >= (n + 3)
  [n for a[ ], one each for n, i and s]

2. **Time Complexity:** The time T(p) taken by a program P is the sum of the compile time and the run time(execution time). The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by T_p (instance characteristics). Time complexity is done by using Frequency count method i.e. the number of times a statement is executed by the compiler.

   → The number of steps any problem statement is assigned depends on the kind of statement.

   For example,

   Comments → 0 steps.

   Assignment statements → 1 steps.

   Interactive statement such as for, while & repeat-until → Control part of the statement.

Time complexity is classified in 5 types based on frequency count method:

- Constant: This statement will be executed by the compiler only once. For example, c:=a+b;
- Linear: This statement will be executed by the compiler n number of times.

```
for i := 1 to n step do   ----------n+1 times
    Statement; ----------n times
```

- Quadratic: This statement will be executed by the compiler n*n times that is n^2 times.

```
for i := 1 to n step do  ------------------- n+1 times
    for j := 1 to  n step do ----------------n(n+1) times
        Statement; ------------------------n^2 times
```

- Cubic: This statement will be executed by the compiler n*n*n times that is n^3 times.

```
for i := 1 to n step do  ------------------- n+1 times
    for j := 1 to  n step do ---------------- n(n+1) times
       for k := 1 to n step do-------------- n^2(n+1) time
        Statement; ------------------------n^3 times
```

- Logarithmic: For each and every time the work area will be sliced to half. In such cases time complexity will be **log n**.

Time complexity can be expressed in three ways: **Best case, Worst case and Average case.**

If an algorithm takes minimum amount of time to complete for a set of specific inputs it is the Best case. For example, 'key' element is found at beginning of an array in linear search.

If an algorithm takes maximum amount of time to complete for a specific set of inputs it is worst case. For example, 'key' element is found at end of an array or element not found.

If an algorithm takes average amount of time to complete for set of specific inputs it is average case. For example, 'key' element found at middle of an array in linear search.

➔ Compute Space and Time complexity to find Sum of individual digits in a number.

Algorithm	**Time Complexity**	**Space Complexity**

Algorithm Sumofindiviual(n)
{
while n > 0 do
{
k:= n % 10;
n:= n / 10;
s:= s + k;
}
return s;
}
	-
	-
	m+1
	-
	m
	m
	m
	-
	1
		
		1 for n
		1 for k
		1 for s

Total = *4m+2 (where m indicates number of digits in the given number)*		S(P) = 3

➔ Compute Space and Time complexity to check given number is Palindrome or not.

Algorithm	**Time Complexity**	**Space Complexity**

Algorithm Palindrome(n)
{
m:= n;
while n > 0 do
{
k := n % 10;
n := n / 10;
s := (s * 10) + k;
}
If s = m then
Write “given number is Palindrome”
Else
Write “Given number is not Palindrome”
}
	-
	-
	1
	m+1
	-
	m
	m
	m
	-
	1
	1
	
	0
		
		1 for m
		1 for n
		1 for k
		1 for s

Total = *4m+4 (where m indicates number of digits in the given number)*		S(P) = 4

➔ Compute Space and Time complexity to check given number is Armstrong or not.

Algorithm	**Time Complexity**	**Space Complexity**

Algorithm armstrong(n)
{
m:=n; ----------
sum := 0; ----------
while(n>0) do ----------
{
n := n / 10;----------
k := n % 10;----------
sum := sum + (k * k* k);----------
}
if(m = sum)----------
write “Given number is Armstrong”;----------
else
write “Given number is Not Armstrong”;----
}
	
	1
	1
	m+1
	
	m
	m
	m
	
	1
	1
	
	0
		
		m - - 1
		n - - 1
		sum - 1
		k - - 1

Total	**4m + 5**	**S(P) = 4**

➔ Compute Space and Time complexity to check given number is perfect or not

Algorithm	**Time Complexity**	**Space Complexity**

Algorithm perfect(n)
{
f := 0;----------
for i := 1 to n-1 do----------
if (n%i=0) then----------
f := f + i;----------
if (f = n) then----------
write “Given number is perfect number”;-----
else
write “Given number is not perfect number”;
}
	
	1
	n
	n - 1
	n - 1
	1
	1
	
	0
		
		f -------1
		i -------1
		n ------1

Total	**3n + 1**	**S(P) = 3**

➔ Compute Space and Time complexity to check given number is prime or not

Algorithm	**Time Complexity**	**Space Complexity**

Algorithm Prime(n)
{
for i := 1 to n do ----------
if (n%i=0) then ----------
c++; ----------
if c = 2 then ----------
write “Given number is Prime”; ----------
else
write “Given number is not Prime number”;
}
	
	n + 1
	n
	n
	1
	1
	
	0
		
		i -------1
		n ------1
		c ------1

Total	**3n + 3**	**S(P) = 3 + 0 = 3**

➔ Compute Space and Time complexity to find Fibonacci sequence up to given number.

Algorithm	**Time Complexity**	**Space Complexity**
Algorithm fibonacci(n) { a := 0;---------- b := 1;---------- write a, b;----------	 1 1 1	 m - - 1 n - - 1 sum - 1

c := a + b;----------	1	k - - 1
while (c<=n) do----------	n+1	
{		
write c;----------	n	
a := b;----------	n	
b := c;----------	n	
c := a + b;----------	n	
}		
}		
Total	**5n+5**	**S(P) = 4 + 0 = 4**

➔ Compute Space and Time complexity to find GCD of two numbers.

Algorithm	**Time Complexity**	**Space Complexity**

Algorithm GCD(a, b)
{
While a != b do
{
If a > b then
a := a – b;
else
b := b – a;
}
Return a;
}
	
	n+1
	
	n
	n
	
	0
	
	1
		
		1 for a
		1 for b

Total	**3n + 2**	**S(P) = 2 + 0 = 2**

➔ Compute Space and Time complexity to find factorial of a given number

Statement	**Time Complexity**	**Space Complexity**
Algorithm factorial(n)	-	
{	-	
f=1.0; ----------	1	f – 1
for i=1 to n do ----------	n+1	i – 1
f:=f * i;----------	n	n – 1
return f;----------	1	
}	-	
Total	**2n + 2**	**S(P) = 3 + 0**

➔ Compute Space and Time complexity to find sum of elements present in an array

Statement	**Time Complexity**	**Space Complexity**

Algorithm Sum(a,n)
{
S=0.0; ----------
for i=1 to n do ----------
s=s+a[i];----------
return s;----------
}
	-
	-
	1
	n+1
	n
	1
	-
		
		S – 1
		i – 1
		a[] – n

Total	**2n + 2**	**S(P) = 2 + n**

➔ Compute Space and Time complexity to perform matrix addition

Algorithm	**Time Complexity**	**Space Complexity**

Algorithm mat-add(a,b,c,n)
{
c[i,j] := 0; ----------
for i := 1 to n do ----------
for j := 1 to n do ----------
c[i,j] := a[i,j] + b[i,j] ----------
return c[i,j]; ----------
}
	
	1
	(n+1)
	n(n+1)
	n^2
	1
		
		i ---- 1
		j ---- 1
		n ----1
		a - - n^2
		b - - n^2
		c - - n^2

Total	**$2n^2 + 2n + 3$**	**S(P) = $3 + 3n^2$**

➔ Compute Space and Time complexity to perform matrix multiplication

Algorithm	**Time Complexity**	**Space Complexity**

Algorithm mat-mul(a,b,c,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]) ----------
return c[i,j]; ----------
}
	
	n + 1
	n(n+1)
	n^2
	$n^2(n+1)$
	n^3
	1
		
		i ---- 1
		j ---- 1
		k --- 1
		n ----1
		a - - n^2
		b - - n^2
		c - - n^2

Total	**$2n^3+3n^2+2n+2$**	**S(P) = $4 + 3n^2$**

➔ Compute Space and Time complexity to perform transpose of a matrix

Algorithm	**Time Complexity**	**Space Complexity**

Algorithm mat-transpose(a,c)
{
c[i,j] := 0; ----------
for i := 1 to n do ----------
for j := 1 to n do ----------
c[i,j] := a[j,i] ----------
return c[i,j]; ----------
}
	
	1
	(n+1)
	n(n+1)
	n^2
	1
		
		i ---- 1
		j ---- 1
		n ----1
		a - - n^2
		c - - n^2

Total	**$2n^2 + 2n + 3$**	**S(P) = $3 + 2n^2$**

➔ Compute Space and Time complexity to perform Linear Search

Algorithm	**Time Complexity**	**Space Complexity**

Algorithm LS(a, key)
{
flag:=0;
for i := 1 to n step do
{
if a[i] = key then
flag:=1;
break;
}
If flag:=1 then
write “successful search”
else
write “unsuccessful search”
}
	
	1
	n + 1
	
	n
	1
	1
	
	1
	1
	
	0
		
		1 for i
		1 for key
		n for a[n]

Total	**2n + 3**	**S(P) = 2 + n**

➔ Compute Space and Time complexity to perform Binary Search

Algorithm	**Time Complexity**	**Space Complexity**
Algorithm BS(a, key) {		1 for low
low:=1;	1	1 for high
high:=n	1	1 for mid
while low<=high do	n+1	1 for n
{		1 for key
mid:=(low + high)/2;	n	n for a[n]
if a[mid] < key then	n	
low:=mid+1;	n	
else if a[mid] > key then	0	
high:=mid – 1;	0	
else		
return mid;	1	
}		
return 0;	0	
}		
Total	**4n + 4**	**S(P) = 5 + n**

How to validate Algorithms: Algorithm validation consists of two phases**: Debugging and Profiling**.

Debugging is the process of executing programs on sample data sets to check whether faulty results occur, and if so correct them.

In case, verifying correction of output on sample data fails, the following strategy can be used: Let more than one programmer develop programs for the same problem, and compare outputs produced by those programs. If the outputs match, then there is a good chance that they are correct.

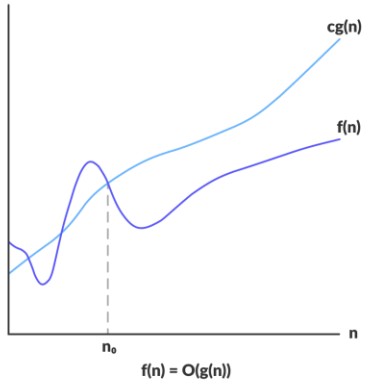
Profiling or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to complete the results.

Asymptotic Notations: Asymptotic notations are used to express time complexities of algorithms in worst, best and average cases. The following are different types of asymptotic notations which are used.

1. Big – Oh Notation
2. Omega Notation
3. Theta Notation
4. Small Oh Notation
5. Small Omega Notation

1. **Big – Oh Notation: (O)** Big – Oh Notation gives upper bound of an algorithm. This notation describes the Worst case scenario.

Definition: Let f(n), g(n) are two non-negative functions and there exists positive constants c, n_0 such that f(n) = O(g(n)) iff f(n)≤c*g(n) for all n, n ≥ n_0. It is represented as follows.



Examples:

a) Compute Big-Oh notation for f(n) = 3n+2

Ans: Given f(n) = 3n+2

f(n) ≤ c * g(n)

3n+2 ≤ 3n + n for n ≥ 2

3n+2 ≤ 4n where c = 4, g(n) = n and n_0=2

Hence f(n) = O(n)

b) Compute Big-Oh notation for f(n) = $10n^2+4n+2$

Ans: Given f(n)= $10n^2+4n+2$

f(n) ≤ c * g(n)

$10n^2+4n+2 \le 10n^2+4n+n$ for n ≥ 2

$10n^2+4n+2 \le 10n^2+5n$

$10n^2+4n+2 \le 10n^2+n^2$ for n ≥ 5

$10n^2+4n+2 \le 11n^2$ where c = 11, g(n) = n^2 and n_0=5

Hence f(n) = $O(n^2)$

c) Compute Big-Oh notation for f(n) = $1000n^2+100n-6$

Ans: Given f(n) = $1000n^2+100n-6$

f(n) ≤ c * g(n)

$1000n^2+100n-6 \le 1000\ n^2+100n$ for all values of n

$1000n^2+100n-6 \le 1000\ n^2+n^2$ for n ≥ 100

$1000n^2+100n-6 \le 1001\ n^2$ where c =1001, g(n) = n^2 and n_0=100

Hence f(n)=$O(n^2)$

d) Compute Big-Oh notation for f(n) = $6*2^n+n^2$

Ans: Given f(n) = $6*2^n+n^2$

f(n) ≤ c * g(n)

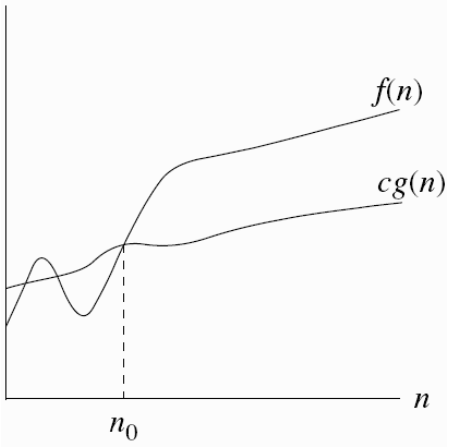
$6*2^n+n^2 \le 6*2^n+ 2^n$ for n ≥ 4

$6*2^n+n^2 \le 7*2^n$ where c = 7, g(n)=2^n and n_0=4

Hence f(n)=$O(2^n)$

2. **Omega Notation (Ω):** Omega Notation gives lower bound of an algorithm. This notation describes best case scenario.

   Definition: Let f(n), g(n) are two non-negative functions and there exists positive constants c, n_0 such that f(n) = Ω(g(n)) iff f(n) ≥c*g(n) for all n, n ≥ n_0. It is represented as follows.



   Examples:

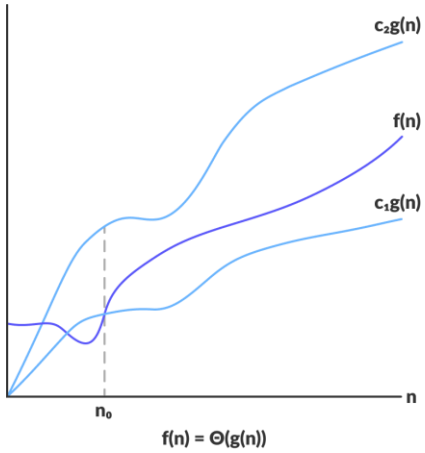
   a) Compute omega notation for f(n)=3n+2
      Ans: Given f(n)=3n+2
      f(n) ≥ c * g(n)
      3n+2≥3n for all values of n (n≥0)
      Where c =3, g(n)=n and n_0=0
      Hence f(n) = Ω(n)

   b) Compute omega notation for f(n)= $10n^2+4n+2$
      Ans: Given f(n)= $10n^2+4n+2$
      f(n) ≥ c * g(n)
      $10n^2+4n+2 \geq 10n^2$ for all values of n (n≥0)
      Where c=10, g(n)=n^2 and n_0=0
      Hence f(n) = Ω(n^2)

   c) Compute omega notation for f(n)= $4n^3+2n+3$
      Ans: Given f(n)= $4n^3+2n+3$
      f(n) ≥ c * g(n)
      $4n^3+2n+3 \geq 4n^3$ for all values of n (n≥0)
      Where c=4, g(n)=n^3 and n_0=0

3. **Theta Notation (θ):** Theta Notation gives the complexity between lower bound and upper bound. This notation describes the average case scenario.

   Definition: Let f(n), g(n) are two non-negative functions and there exists positive constants c_1, c_2, n_0 such that f(n) = **θ** (g(n)) iff c_1* g(n) ≤ f(n) ≤ c_2*g(n) for all n, n ≥ n_0



Examples:

a) Compute theta notation for f(n)=3n+2

Ans: Given f(n)=3n+2

c_1* g(n) ≤ f(n) ≤ c_2*g(n)

Compute f(n) ≤ c_2*g(n)

3n+2≤3n+n for n≥2

3n+2≤4n where c_2=2 and g(n)=n

Compute c_1* g(n) ≤ f(n)

3n ≤ 3n+2 for all values of n

Where c_1=3, g(n)=n

Hence

f(n)= ɵ(n)

b) Compute theta notation for f(n)=$10n^2$+4n+2

Ans: Given f(n)=$10n^2$+4n+2

c_1* g(n) ≤ f(n) ≤ c_2*g(n)

Compute f(n) ≤ c_2*g(n)

$10n^2$+4n+2≤$10n^2$+4n+n for n≥2

$10n^2$+4n+2≤$10n^2$+5n

$10n^2$+4n+2≤$10n^2$+n^2 for n≥5

$10n^2$+4n+2≤$11n^2$

where c_2=11 and g(n)=n^2

Compute c_1* g(n) ≤ f(n)

$10n^2$ ≤ $10n^2$+4n+2 for all values of n

Where c_1=10, g(n)=n^2

Hence

f(n)= ɵ(n^2)

4. **Small Oh Notation (o):** Let f(n), g(n) are two non-negative functions, then we can say that f(n) = o(g(n) iff

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

Examples:

a) Compute theta notation for f(n)=3n+2

Ans: Given f(n)=3n+2

Let g(n)=1

Then **Lt** **f(n)/g(n) = 3n+2/1=3n+2=** ∞

n $\rightarrow \infty$

Let g(n)=n

Then **Lt** **f(n)/g(n) = 3n+2/n=3+2/n=3**

n $\rightarrow \infty$

Let g(n)=n^2

Then **Lt** **f(n)/g(n) = 3n+2/n^2=3/n+2/n^2=0**

n $\rightarrow \infty$

Hence f(n) = o(n^2)

5. **Small Omega Notation (ω):** Let f(n), g(n) are two non-negative functions, then we can say that f(n) = ω(g(n) iff

Lt **f(n)/g(n) =** ∞

n $\rightarrow \infty$

Examples:

a) Compute Small Omega notation for f(n)=3n+2

Ans: Given f(n)=3n+2

Let g(n)=1

Then **Lt** **f(n)/g(n) = 3n+2/1=3n+2=** ∞

n $\rightarrow \infty$

Hence, f(n)= **ω(1)**

Frequently Asked Questions

1. Define an algorithm. What are the different criteria that satisfy the algorithm?
2. Explain pseudo code conventions for writing an algorithm.
3. Explain how algorithms performance is analyzed? Describe asymptotic notation?
4. What are the different techniques to represent an algorithm? Explain.
5. Explain recursive algorithms with examples.
6. Distinguish between Algorithm and Pseudo code.
7. Give an algorithm to solve the towers of Hanoi problem.
8. Write an algorithm to find the sum of individual digits of a given number
9. Explain the different looping statements used in pseudo code conventions.
10. What is meant by recursion? Explain with example, the direct and indirect recursive algorithms.

11. What is meant by time complexity? What is its need? Explain different time complexity notations. Give examples one for each.
12. Describe the performance analysis in detail
13. Discuss about space complexity in detail.
14. Define Theta notation. Explain the terms involved in it. Give an example
15. Explain about two methods for calculating time complexity.
16. Show that $f(n) = 4n^2 - 64n + 288 = \mathbf{\Omega}(n^2)$.
17. Present an algorithm for finding Fibonacci sequence of a given number.
18. Write the non-recursive algorithm for finding the Fibonacci sequence and derive its time complexity.
19. Compare the two functions n^2 and $2n/4$ for various values of n. Determine when the second becomes larger than the first.
20. Determine the frequency counts for all statements in the following algorithms.

```
i) for i:=1 to n do
for i:=1 to i do
for k:=1 to j do
x:= x+1;
ii) i := 1;
while (i<=n) do
{
 x : = x + 1;
i : = i + 1;
}
```

21. Calculate the time complexity for matrix multiplication algorithm.
22. Calculate the time complexity for Armstrong number algorithm
23. Explain about different Asymptotic Notations with two examples
24. Find the time complexity for calculating sum of given array elements.
25. Calculate space and time complexity for matrix multiplication algorithm
26. Write an algorithm for Armstrong number and also calculate space and time complexity?
27. Write an algorithm for strong number and also calculate space and time Complexity?
28. Describe the Algorithm Analysis of Binary Search.
29. Differentiate between Big-oh and Omega notation with example.

Divide and Conquer.

General Method: In Divide and Conquer method, a given problem is

- → Divided into smaller sub problems.
- → these subproblems are solved independently.
- → Combine all the solutions of sub problems into a single solution.
- → If the sub problems are large enough then again Divide and Conquer is applied.

Algorithm for Divide and Conquer: A Control abstraction for Divide and Conquer general method is as shown below

```
Algorithm DAndC(P).
{
   if Small(P) then return S(P);
   else
   {
      divide P into smaller instances P1, P2 ... Pk, k≥1;
      Apply DAndC to each of these subproblems.
      return Combine(DAndC(P1), DAndC(P2) ... DAndC(Pk));
   }
}.
```

Here Small(P) is a Boolean valued function which determines whether the input size is small enough. If it is small, then solution is returned.

Otherwise, the problem P is divided into smaller subproblems $P_1, P_2 \cdots P_k$. these are solved by recursive applications of DAndC. Combine is a function that determines the solution 'P' using the solutions to the 'k' subproblems.

(*) Recurrence Relation: the Recurrence Relation is an equation that defines a sequence recursively. The Recurrence Relation can be solved by the following me

① Substitution Method.

② Master's Method.

① Substitution Method: there are two types of substitut methods.

ⓐ Forward Substitution: This method makes use of initia condition and value for the next term is generated. for example, Consider a recurrence relation.

$$T(n) = T(n-1)+n \quad \text{with } T(0)=0.$$

If $n=1 \Rightarrow T(1) = T(0)+1. = 1. \quad = 1$

$n=2 \Rightarrow T(2) = T(1)+2 = 1+2. = 3$

$n=3 \Rightarrow T(3) = T(2)+3 = 1+2+3 = 6$

$\vdots \qquad\qquad\qquad\qquad \vdots = 1+2+\cdots+n.$

$n=n \Rightarrow T(n) = .$

By observing the above generated equation we can deri a formula

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

ⓑ Backward Substitution: In this method backward values are substituted recursively in order to derive some formula. For example, Conside a recurrence relat

$$T(n) = T(n-1)+n. \quad \text{with } T(0)=0.$$

$\longrightarrow$ ①

$$\Rightarrow T(n-1) = T(n-1-1)+n-1 = T(n-2)+(n-1) \sim ②$$

Substitute ~② in ~①

$\Rightarrow T(n) = T(n-2) + (n-1) + n \sim (3)$.

Let $T(n-2) = T(n-2-1) + (n-2) \sim (4)$.

Sub $\sim(4)$ in $\sim(3)$

$\Rightarrow T(n) = T(n-3) + (n-2) + (n-1) + n$.

$\vdots$

$T(n) = T(0) + n - (n-1) + \cdots + n$.

$= 0 + 1 + 2 + \cdots + n$.

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

(2) Master's Method: Consider the following recurrence relation:

$$T(n) = aT(n/b) + F(n).$$

then Master's theorem can be stated as -

If $F(n)$ is $O(n^d)$ where $d \geq 0$ in the RR then.

(i) $T(n) = O(n^d)$ if $a < b^d$.

(ii) $T(n) = O(n^d \log n)$ if $a = b^d$.

(iii) $T(n) = O(n^{\log_b a})$ if $a > b^d$.

For example, $T(n) = 4T(n/2) + n$. Compare with

$T(n) = aT(n/b) + f(n)$.

Here $a = 4, b = 2, f(n) = n^1$. where $d = 1$.

$\because a > b^d$. $T(n) = O(n^{\log_b^a})$

$= O(n^{\log_2^4}) = O(n^2)$.

Another variation of Master theorem is. for

$$T(n) = aT(n/b) + f(n).$$

(i). If $f(n)$ is $O(n^{\log_b a})$ then $T(n) = O(n^{\log_b a})$.

(ii) If $f(n)$ is $O(n^{\log_b a} \log^k n)$ then $T(n) = O(n^{\log_b a} \log^{k+1} n)$.

~~(iii) If $f(n)$ is~~ For example, Consider

$$T(n) = 2T(n/2) + n\log n.$$

Here $f(n) = n\log n$. where $a=2, b=2., K=1$

According to case (ii)

$$T(n) = O(n^{\log_b a} \log^{k+1} n).$$

$$= O(n^{\log_2 2} \log^2 n)$$

$$= O(n \log^2 n).$$

For more examples refer class notes

(*) **Analysis of DandC General Method:** the computing time of Divide and Conquer is given by the recurrence relation.

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

Where $T(n)$ is the time for DAndC on any input of size n and $g(n)$ is the time to compute the solution directly for small inputs. the function $f(n)$ is the time for dividing P and combining the solutions to subproblems.

The complexity of many divide and conquer algorithms is given by recurrence relation.

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

Where a and b are constants.

Let $n = b^k$

$\therefore T(b^k) = aT(b^k/b) + f(b^k)$

$T(b^k) = aT(b^{k-1}) + f(b^k) \sim (1)$.

from $\sim(1)$

$T(b^{k-1}) = aT(b^{k-2}) + f(b^{k-1}) \sim (2)$

Sub stitute $\sim(2)$ in $\sim(1)$ we get

$T(b^k) = a[aT(b^{k-2}) + f(b^{k-1})] + f(b^k)$.

$= a^2T(b^{k-2}) + af(b^{k-1}) + f(b^k) \sim (3)$

By substituting $T(b^{k-2}) = aT(b^{k-3}) + f(b^{k-2})$ in $\sim(3)$

$$T(b^k) = a^3 T(b^{k-3}) + a^2 f(b^{k-2}) + a f(b^{k-1}) + f(b^k)$$

$$\vdots$$

$$= a^k T(b^{k-k}) + a^{k-1} f(b^1) + a^{k-2} f(b^2) + \cdots + a^0 f(b^k)$$

$$= a^k T(1) + \frac{a^k}{a} f(b) + \frac{a^k}{a^2} f(b^2) + \cdots + \frac{a^k}{a^k} f(b^k)$$

$$= a^k \left[T(1) + \frac{f(b)}{a} + \frac{f(b^2)}{a^2} + \cdots + \frac{f(b^k)}{a^k} \right]$$

$$= a^k \left[1 + \sum_{j=1}^{k} \frac{f(b^j)}{a^j} \right] \qquad \because b^k = n.$$

$$\therefore k = \log_b n.$$

$$= a^{\log_b n} \left[1 + \sum_{j=1}^{\log_b n} \frac{f(b^j)}{a^j} \right]$$

$$= n^{\log_b a} \left[1 + \sum_{j=1}^{\log_b n} f(b^j)/a^j \right] //.$$

(*) **Applications of Divide and Conquer:** Various applications of Divide and conquer strategy are

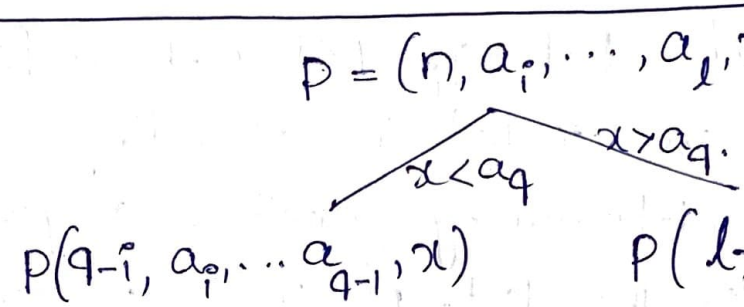
1. Binary Search. ✓
2. Merge Sort
3. Quick Sort ✓
4. Strassen's Matrix Multiplication.
5. Finding Max and Min element in an array ✓
6. Defective chessboard.

① Binary Search: Let a_i, $1 \le i \le n$, be a list of elements that are sorted in ascending order. Consider the problem of determining whether a given element x is present in the list. If x is present, determine a value j such that $a_j = x$. If x is not in the list, then j is set to be zero.

Let $P = (n, a_1, \ldots, a_l, x)$ ~~dento~~ denote an arbitary instance of this search problem, where n is the number of elements in the list, $a_1, \ldots a_l$ is the list of elements, and x is the searching element.

According to Divide and Conquer, if the problem is small ($n=1$) then take the value i if $x = a_i$; otherwise it will take the value 0.

If P has more than one element, it can be divided into sub problems. Pick an index q in the range $[i, l]$, compare x with a_q. If $x = a_q$, the problem P is solved. If $x < a_q$, then search for x in the sublist $a_i, a_{i+1}, \ldots, a_{q-1}$. If $x > a_q$, then search for x in the sublist $a_{q+1}, \ldots, a_l$. Now, the given problem P is divided into the following two sub problems.



To obtain solution, repeatedly apply DAnd C on each subproblems.

⊛ Algorithm for Recursive Binary Search:

```
Algorithm BinSearch(a, i, l, x)
// Given an array a[i:l] of elements in ascending order,
// 1≤i≤l, determine whether x is present, and if so,
// return j such that x = a[j]; else return 0.
{
  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);
  }
}//
```

(*) Algorithm for Iterative & Non-Recursive Binary Search:

```
Algorithm BinSearch(a,n,x)
// Given an array a[i:l] of elements in ascending order,
// n≥0, determine whether x is present, and if so, return
// j such that x = a[j]; else return 0.
{
   low := i; 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;
   }
   return 0;
}//.
```

Ex: Apply Divide and Conquer strategy for the following input values for searching 112, -14 by showing the values of low, mid, high for each search.

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.

Sol: For x = 112.

low	high	mid.
1.	14	7
8	14	11
8	10	9.
10	10	10
		found.

For x = -14.

low	high	mid
1	14	7
1	6	3
1	2	1
		not found.

Ex: ② : Explain the method for searching the element 94 following set of elements using Binary Search. Also draw binary decision tree for the same.

10, 12, 14, 16, 18, 20, 25, 30, 35, 38, 40, 45, 50, 55, 60, 70, 80, 90

Sol: Given that $i=1$, $l=18$, $x=94$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
10	12	14	16	18	20	25	30	35	38	40	45	50	55	60	70	80	90

Now pick the index $q = \left\lfloor \frac{(i+l)}{2} \right\rfloor = \left\lfloor \frac{1+18}{2} \right\rfloor = 9$

Compare x and $a[9]$. Since $x > a[9]$ divide the given array into two sub arrays and continue search in second sub array.

Now $i = 9+1 = 10$, $l=18$, $x=94$.

Now pick the index $q = \left\lfloor \frac{(i+l)}{2} \right\rfloor = \left\lfloor \frac{(10+18)}{2} \right\rfloor = 14$.

Compare x and $a[9]$. Since $x > a[9]$ divide the array into two sub arrays and continue search in second sub array.

Now $i = 9+1 = 15$, $l=18$, $x=94$.

Now pick the index $q = \left\lfloor \frac{(i+l)}{2} \right\rfloor = \left\lfloor \frac{(15+18)}{2} \right\rfloor = 16$

$\because x > a[9]$. continue search in second sub array.

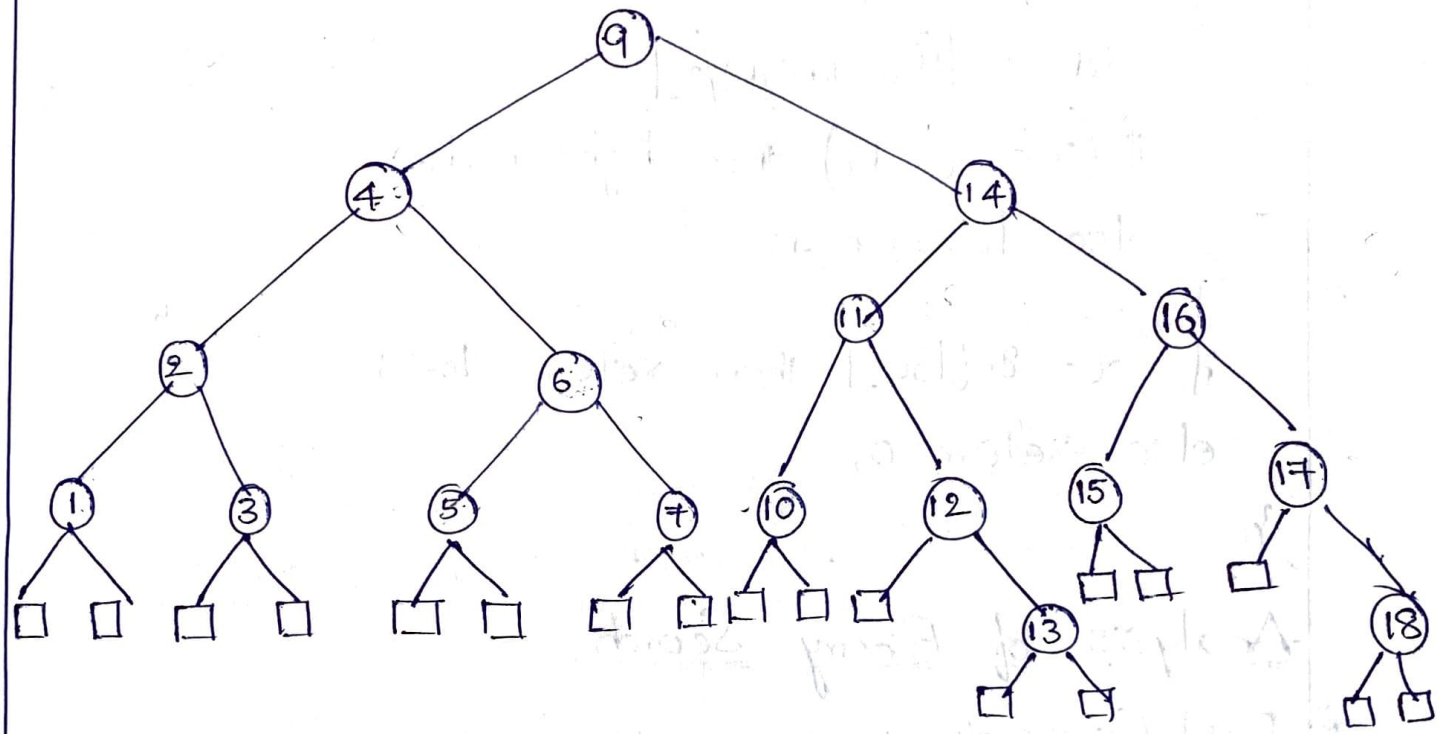
$\therefore i = 9+1 = 17$, $l=18$, $x=94$. $\therefore q = \left\lfloor \frac{(17+18)}{2} \right\rfloor = 17$

$\because x > a[9]$ continue search in second sub array.

$\therefore i = 9+1 = 18$, $l=18$, $x=94$ $\therefore q = \frac{(18+18)}{2} = 18$.

$\because x > a[9]$. element is not found.

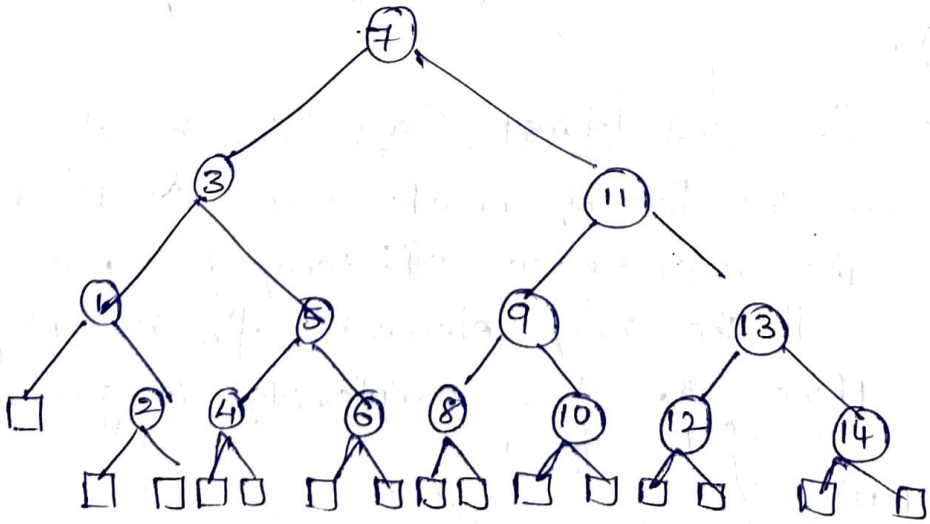
Binary Decision Tree: Binary Decision tree contains: circular nodes and square nodes. Every successful search ends at circular node and unsuccessful search ends at square node. Binary Decision tree for the above problem n=18 is as shown below.



Ex: Draw Binary Decision Tree for the following elements:

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

Sol: Here n=14.



(*) Present an Algorithm for Binary Search using one comparison per cycle.

```
Algorithm BinSearch1(a,n,x)
{
  low:=1; high:=n+1;
  while (low<high-1) do
  {
    mid:= ⌊(low+high)/2⌋;
    if (x<a[mid]) then high:=mid;
    else low:=mid;
  }
  if x=a[low] then return low;
  else return 0;
}
```

Analysis of Binary Search:

(*) Best Case: the basic operation in binary search is comparison of search key with array element. If the search key is found at middle of the array, total no: of comparisons required is 1. Hence, Analysis of binary search in best case is O(1).

(*) Average Case and Worst Case: In the algorithm after one comparision the list of n elements is divided into n/2 sublists. the worst-case efficiency is that the algorithm compares all the array elements for searching the desired element. Hence the time complexity is given by

$$T(n)=\begin{cases}1 & \text{if } n=1\\ T(n/2)+1 & \text{if } n>1\end{cases}$$

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

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

Substitute $\sim (2)$ in $\sim (1)$ we get

$T(n) = T(n/4) + 1 + 1 = T(n/4) + 2 \sim (3)$

$T(n/4) = T(n/8) + 1$

By substituting $T(n/4)$ value, we will get

$T(n) = T(n/8) + 1 + 2$

$\vdots$

$T(n) = T(n/16) + 4$.

Let $2^4 = n$.

$4 = \log_2 n$.

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

$= 1 + \log_2 n$.

$\therefore O(\log n)$.

⊛ Quick Sort: Quick Sort is a sorting algorithm that uses the divide and conquer strategy. the steps for quick sort are as follows:

- Divide: Split the array into two arrays such that each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element. the splitting of the array into two sub arrays is based on pivot element. All the elements that are less than pivot should be in left sub array and all the elements that are more than pivot should be in right sub array.
- Conquer: Recursively sort the two sub arrays.
- Combine: Combine all the sorted elements in a grop to form a list of sorted elements.

Algorithm for Quick Sort:

```
Algorithm QuickSort(p,q)
{
  if (p<q) then
  {
    j:= Partition (a,p,q+1);
    QuickSort(p,j-1);
    QuickSort(j+1, q);
  }
}
Algorithm Partition(a,m,p)
{
  v:=a[m]; i:=m; j:=p;
  repeat
  {
    repeat
      i:=i+1;
    until (a[i]≥v);
    repeat
      j:=j-1;
    until (a[j]≤v);
    if (i<j) then Interchange(a,i,j);
  }until(i≥j);
  a[m]:=a[j]; a[j]:=v; return j;
}
Algorithm Interchange(a,i,j)
{
  p:=a[i];
  a[i]:=a[j];
  a[j]:=p;
}//
```

(*) Sort the following values in ascending order using QS algorithm: 20, 30, 80, 50, 40, 70, 60, 90, 10.

Ans:

i=1	2	3	4	5	6	7	8	j=9
20	30	80	50	40	70	60	90	10
pivot								

Now split the array into two arrays based on pivot

Since a[i] ≤ pivot increment i: = 1+1 = 2.

Since a[i] ≥ pivot stop incrementing i

Since a[j] ≤ pivot stop decrementing j

Now swap a[2] and a[9]. Hence.

	i							j
20	10	80	50	40	70	60	90	30
p								

Since a[i] ≤ pivot increment i. i.e i=3.

Since a[i] ≥ pivot stop incrementing i

Since a[j] ≥ pivot decrement j. i.e j = 8

Since a[j] ≥ pivot decrement j. i.e j = 7

Since a[j] ≥ pivot decrement j. i.e j = 6

Since a[j] ≥ pivot decrement j. i.e j = 5

Since " " j. i.e j = 4.

Since " " j. i.e j = 3.

Since a[j] ≥ pivot " j. i.e j = 2.

Since a[j] ≤ pivot stop decrementing j

~~Now~~ ~~swap~~. Since i > j swap a[j] and pivot, then elements in the array are

(10) 20, (80, 50, 40, 70, 60, 90, 30).

(10), 20, (50, 40, 70, 60, 30) 80 (90)

10, 20, (40, 30) 50 (70, 60) 80 90

10, 20, 30, 40, 50, 60, 70, 80, 90.

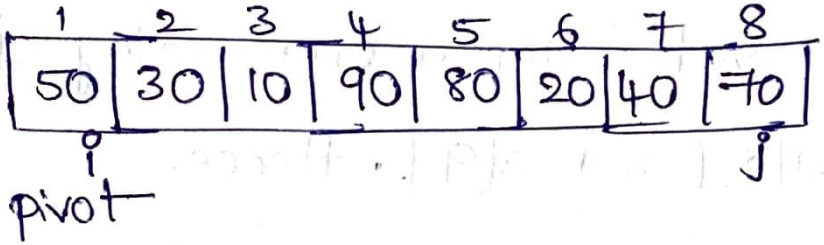
(*) Is quick sort a stable sorting method?

Ans: No, quick sort is not a stable sorting algorithm because after applying this sorting method the order of similar elements may not be preserved, since swapping is involved in this sorting with pivot element.

(*) Sor the following list of elements using Quick Sort:

50, 30, 10, 90, 80, 20, 40, 70.

Sol: G.T



Now split the array into two sub arrays based on pivot

Since a[i] ≤ pivot increment i. i.e i=2.

Since a[i] ≤ pivot increment i. i.e i=3.

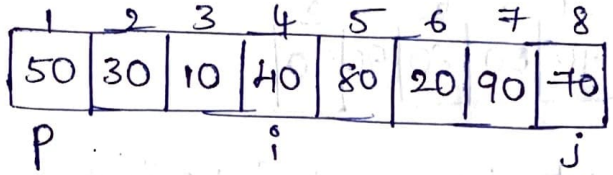
Since a[i] ≤ pivot increment i. i.e i=4.

Since a[i] ≥ pivot stop incrementing i.

Since a[j] ≥ pivot decrement j i.e j=7

Since a[j] ≤ pivot stop decrementing j.

∵ i<j swap a[i] and a[j]. then.



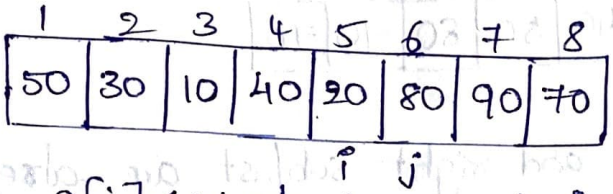
Since a[i] ≤ pivot increment i. i.e i=5

Since a[i] ≥ pivot stop incrementing.

Since a[j] ≥ pivot decrement j. i.e j=6

Since a[j] ≤ pivot stop decrementing.

∵ i<j swap a[i] and a[j] then.



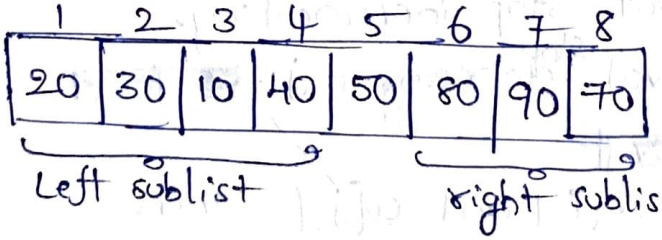
Since a[i] ≤ pivot increment i. i.e i=6

Since a[i] ≥ pivot stop incrementing.

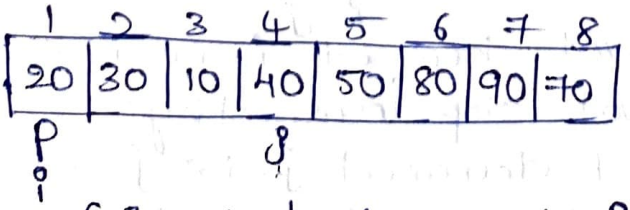
Since a[j] ≥ pivot decrement j. i.e j=5

Since a[j] ≤ pivot stop decrementing.

∵ i>j swap a[j] and pivot then



Now start sorting left sublist, by assuming first element as pivot element. then

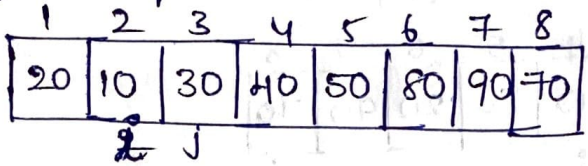


Since a[i] ≤ pivot increment i. i.e i=2.

Since a[i] ≥ pivot stop incrementing.

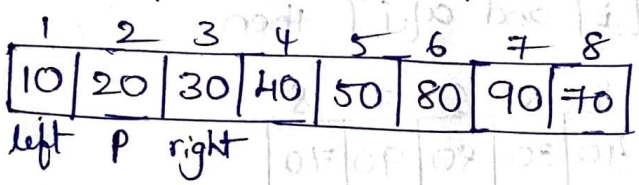
Since a[j] ≥ pivot decrement j. i.e j=3
Since a[j] ≤ pivot stop decrementing.

∵ i<j swap a[i] and a[j] then.

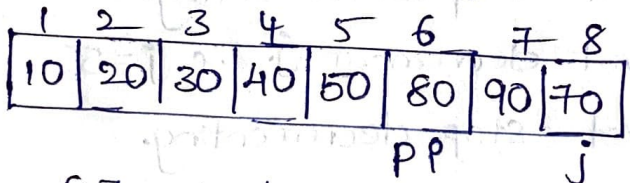


Since a[i] ≤ pivot increment i. i.e i=3
Since a[i] ≥ pivot stop incrementing.
Since a[j] ≥ pivot decrement j. i.e j=2.
Since a[j] ≤ pivot stop decrementing

∵ i>j swap a[j] and pivot element then.

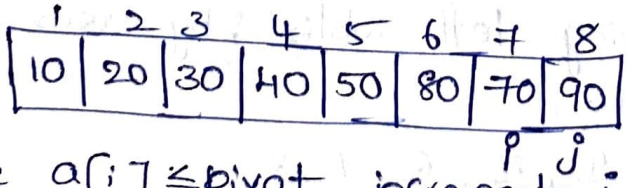


As left sublist and right sublist are already sorted assume first element of right sublist as pivot.



Since a[i] ≤ pivot increment i. i.e i=7
Since a[i] ≥ pivot stop incrementing.
Since a[j] ≤ pivot stop decrement j.

∵ i<j swap a[i] and a[j] then.



Since a[i] ≤ pivot increment i. i.e i=8
Since a[i] ≥ pivot stop incrementing
Since a[j] ≥ pivot decrement j i.e j=7
Since a[j] ≤ pivot stop decrementing

∵ i>j swap a[j] and pivot then.

10	20	30	40	50	70	80	90

This is the sorted list.

⊛ Analysis of Quick Sort:

⊛ **Best Case and Average Case:** If the array is always partitioned at the mid, then it brings the best case efficiency of an algorithm. then the recurrence relation is given by

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

= Time required to sort left sub array +
Time required to sort right sub array +
Time required for partitioning the sub array.

$$\therefore T(n) = \begin{cases} 1 & \text{if } n=1. \\ 2T(n/2)+n & \text{if } n>1 \end{cases}$$

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

$$\therefore T(n/2) = 2T(n/4) + (n/2) \sim (2)$$

By substituting ~(2) in ~(1) we will get

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

$$T(n) = 4T(n/4) + 2n. \sim (3)$$

By substituting $T(n/4)$ in ~(3) we will get

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

Let $2^3 = n$

$\therefore 3 = \log_2 n.$

$$T(n) = nT(n/n) + \log_2 n \times n.$$

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

Worst Case: the worst case for quick sort occurs when the pivot is a minimum or maximum of all the elements in the list. Hence recurrence relation is given as

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1)+n & \text{if } n>1. \end{cases}$$

$\therefore T(n) = T(n-1) + n \sim (1)$

$T(n-1) = T(n-2) + (n-1) \sim (2)$

By substituting $\sim(2)$ in $\sim(1)$ we get

$T(n) = T(n-2) + (n-1) + n \sim (3)$

By substing $T(n-2)$ in $\sim(3)$ we get

$T(n) = T(n-3) + (n-2) + (n-1) + n.$

$\vdots$

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

$= 0 + 1 + 2 + \cdots + n.$

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

$= \frac{n^2+n}{2} = \frac{n^2}{2} + \frac{n}{2}.$

$\therefore O(n^2).$

(*) Randomised Quick Sort: The worst case for quick sort depends upon the selected pivot element. If min or maximum element in the array is choosed as pivot results in worst case. the following are differe methods to choose pivot element which improves the performance of quick sort.

→ Use middle element of the array as pivot
→ Use a random element of the array as pivot
→ Take median of first, last and middle elements as a pivot.

Randomized Quick Sort Algorithm:

```
Algorithm RQuickSort(p,q).
{
  if (p<q) then
  {
    [redacted] then  i:=random(p,q);
                     swap(A[i], A[q]);
    [redacted]
    j:= Partition(a,p,q+1);
    RQuickSort(p, j-1);
    RQuickSort(j+1,q);
  }
}//.
```

Finding Maximum and Minimum element in an Array:

Let $P = (n, a[i], \ldots a[j])$ denote an arbitary instance of the problem. Here n is the number of elements in the list $a[i] \ldots a[j]$ and we have to find maximum and minimum of this list.

Let small (P) be true when $n \leq 2$. In this case, the maximum and minimum are $a[i]$ if $n=1$. If $n=2$, the problem can be solved by making one comparision.

Divide: If the list has more than two elements, P has to be divided into smaller instances. For example, P might divide into two instances $P1 = (n/2, a[i], \ldots a[n/2])$ and $P2 = (n/2, a[n/2+1] \ldots a[j])$. After having divided P into two smaller sub problems, we can solve them by recursively invoking the same divide and conquer algorithm.

Conquer: Let Max (P) and Min (P) be the maximum and minimum elements of P, then Max (P) is the larger of Max (P1) and Max (P2), Min (P) is the smaller of Min (P1) and Min (P2).

Algorithm

```
Algorithm MaxMin (i, j, max, min)
{
  if i=j then max := min := a[i];
  else if i=j-1 then
  {
    if a[i] < a[j] then max := a[j]; min := a[i];
    else
        max := a[i]; min := a[j];
  }
  else
  {
    mid := (i+j)/2;
    MaxMin (i, mid, max, min);
    MaxMin (mid+1, j, max1, min1);
    if max < max1 then max := max1;
    if min > min1 then min := min1;
  }
}
```

Analysis: Computing time of Finding maximum and minimum element in the array is given the following Recurrence Relation.

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ T(n/2)+T(n/2)+2 & \text{if } n>2 \end{cases}$$

$T(n) = 2T(n/2) + 2 \sim ①$

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

Substitute $\sim ②$ in $\sim ①$

$\Rightarrow T(n) = 2[2T(n/4) + 2] + 2 = 4T(n/4) + 4 + 2 \sim ③$

From $\sim ①$ $T(n/4) = 2T(n/8) + 2 \text{ —— } ④$

Substitute $\sim ④$ in $\sim ③$

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

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

$\vdots$

$= 2^{k-1} T\left(\frac{2^k}{2^{k-1}}\right) + 2^k - 2$

$= 2^{k-1} T(2) + 2^k - 2$

$= 2^{k-1} + 2^k - 2$

$= \frac{n}{2} + n - 2 \quad \because n = 2^k.$

$= \frac{3n}{2} - 2.$

$= O(n) //.$

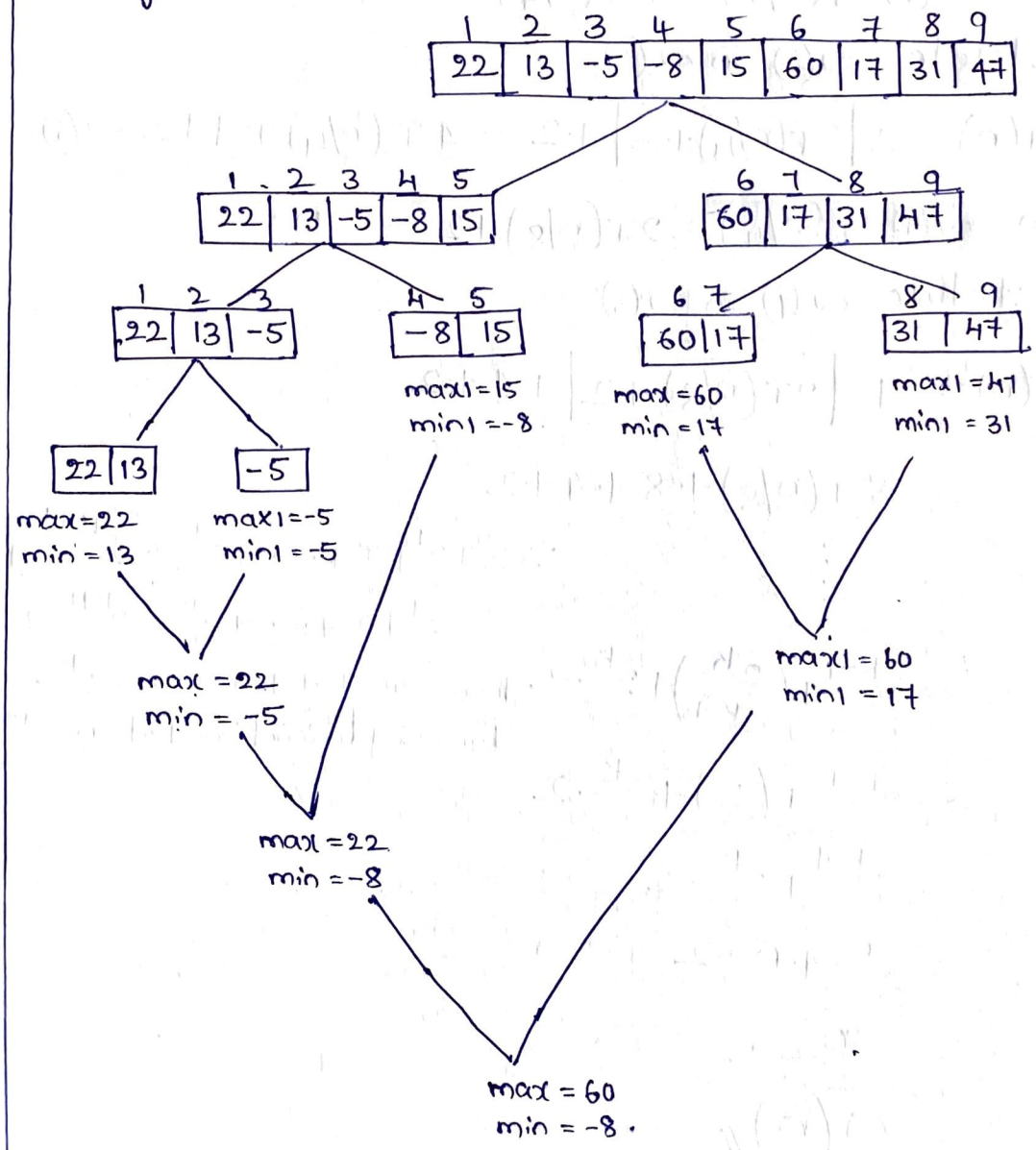
$\because 2^0 + 2^1 + 2^2 + \cdots + 2^k = 2^{k+1} - 1$

$2^1 + 2^2 + \cdots + 2^k = \underline{\underline{2^{k+1} - 2}}.$

there are k-1 terms so $= 2^k - 2$.

For simplification Let $n = 2^k$

Ex: Find maximum and minimum element in the array using D and C : 22, 13, -5, -8, 15, 60, 17, 31, 47



Frequently Asked Questions

1. What is Divide and Conquer strategy? Give its recurrence relation?
2. Write and explain the control abstraction for D and C ?
3. Explain the general method of Divide and Conquer?
4. What is Binary Search? How it can be implemented by divide and conquer strategy? Explain with example?
5. Write the iterative algorithm for searching an element by using Binary Search.
6. Present an algorithm for Binary Search using one comparison per cycle.
7. Write and explain Recursive Binary Search algorithm with an example?

8. Apply divide and Conquer strategy to the following input values for searching 112 and -14 by showing the values of low, mid, high for each search.

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

9. Explain the method for searching element 94 from the following set of elements. by using Binary Search.

   {10, 12, 14, 16, 18, 20, 25, 30, 35, 38, 40, 45, 50, 55, 60, 70, 80, 90}.

10. Search for an element -2 from the below set by using Binary Search.

    A = {-15, -6, 0, 7, 9, 23, 54, 82, 101, 112}

    Also draw Binary decision tree for the above.

11. Give an example for Binary Search. Draw binary decision tree.
12. Derive the time complexity of Binary Search?
13. Draw Binary Decision tree for the following set (3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 47).

(21) Explain the way divide and conquer works for quick sort with example.

(22) Write an algorithm of Quick Sort and explain in detail.

(23) Find the best, average and worst case complexity for Quick sort.

(24) Show how procedure QUICKSORT sorts the following set of keys:
(1, 1, 1, 1, 1, 1, 1) and (5, 5, 8, 3, 4, 3, 2).

(25) Sort the following elements using Quick Sort.
3-3
(i) 5, 1, 7, 3, 4, 9, 8, 2, 6.
(ii) 20, 30, 80, 50, 40, 70, 60, 90, 10
(iii) 25, 30, 36, 49, 58, 67, 69, 10.
(iv) 25, 20, 16, 49, 28, 17, 9, 10

(26) Discuss briefly about the randomized Quick Sort?

20. Explain the process of finding maximum and minimum element in an array using divide and conquer strategy with an example?

21. With a suitable algorithm, explain the problem of finding the maximum and minimum items in a set of n elements.

21. Apply divide and conquer strategy to find maximum and minimum elements from the following array of elements: 15, -1, 23, 6, 8, -4, 0, 3, 12, and 10.

22. Analyse running time to find maximum and minimum element in the array?