Unit-1

- Design and analysis of algorithms includes designing or developing of algorithms and analyzing algorithms.
- An algorithm is a step by step procedure for solving a problem. It contains sequence of steps which indicate how to solve a problem.
- The basic reason for writing algorithms is to write or implement programs

  easily. To design or develop algorithms, the following methods are used

  1. Divide and Conquer
  2. Greedy
  3. Dynamic Programming
  4. Backtracking
  5. Branch and Bound
- Analysis of algorithms is measuring performance of algorithms in terms of space complexity and time complexity and making some decisions.

Consider sorting problem as an example. For sorting a list of values, number of techniques exist

1. Bubble sort
2. Selection sort
3. Insertion sort
4. Quick sort
5. Merge sort
6. Heap sort
7. Radix sort

- The time and space complexity of all algorithms are calculated in order to decide the best sorting method for sorting a list of values.
- If there are number of methods to solve a problem and if we need to identity the best method for solving the problem then performance analysis is used.

I. Criteria (or) characteristics (or) properties of an algorithm

The following are characteristics of an algorithm

- Input
- Output
- Definiteness
- Finiteness
- Effectiveness

1.Input

There are zero (or) more inputs to an algorithm.

2. Output

Every algorithm or its equivalent program generates one or more outputs.

3. Definiteness

Each step of algorithm should be clear and unambiguous.

Ex: add 5 or 6 to 7 is am ambiguous statement.

4. Finiteness

The algorithm should terminate after a finite number of steps. The algorithm should not enter into an infinite loop.

5. Effectiveness

Each step of the algorithm should be such that it can be easily converted to equivalent statement of the program.

Specification of algorithm

Two methods are generally used to specify an algorithm

1. Flow chart
2. Pseudo code

- In flow chart representation, the steps of algorithm are represented using graphical notations. Flow chart representation is effective when the algorithm is simple and small.
- If the algorithm is large and complex then pseudo code is used to represent the algorithm.

2.Pseudo code representation of algorithms

The syntax rules for specifying an algorithm in pseudo code are as follows

Delimiter: ; is used as delimiter of statements.

Comments:
// notation is used to indicate comments.

Block of statements:
{} are used to indicate block of statements.

Variables:
Any variable name should start with a letter. No need to specify data type and scope for the variables. Variables can be used at any place in the algorithm without declaring them.

Operators:

Relational operators: $<, \leq, >, \geq, =, \neq$
Logical operators: and, or, not
Assignment operator: :=

The symbols for remaining operators are same as in „c" language.

Arrays:
Single dimensional arrays are used with the notation- arrayname[index]
Multi dimensional arrays are used with the notation- arrayname[index of first dimension, index

of second dimension, ……]

Ex: a[i]
a[i,j]
a[i,j,k]

Conditional Statements:
The conditional statements if and case are used in pseudo code.

if:
if statement is used to check one condition. The syntax of if statement is

```
if condition then
{
        Block of statements
}

if condition then
{

            Block of statements

        }

    else
{

    Block of statements

}
```

case:
case statement is similar to switch statement. It is used to check number of conditions. The syntax of case statement is

```
case
{
        :condition1: statements
        :condition2: statements
        :condition3: statements
                        .
                        .
        :conditionn: statements
}
```

The conditions are checked one after another and when any condition becomes true then the corresponding statements are executed and then control comes out of case statement.

Loop statements:

while
The syntax of while statement is

```
while condition do
{
        Block of statements
}
```

repeat
The syntax of repeat statement is

```
repeat
        Block of statements
until condition
```

for
The syntax of for statement is

```
for variable := value1 to value2 step step do
{
        Block of statements
}
```

Variable is any variable name. Value1 is starting value of variable. Value2 is ending value of variable.

step is either a +ve or –ve value. After each iteration, the value of variable is incremented by step value if step value is +ve or decremented by step value if step value is –ve.

step is optional.

Default value of step is +1

Input & Output:
„read" statement is used to read input. „write" statement is used to display output.

Heading of the algorithm:
Each algorithm should start with the heading

```
Algorithm name(list of parameters)
{
}
```

„name" is user defined name and parameter list is optional. No need to specify data type for parameters.

Ex1: Write an algorithm in pseudo code format to calculate sum of values in an array

```
Algorithm sum(a, n)
//a is an array containing list of values
//n is size of the array
{
        sum := 0;
        for i := 1 to n do
        {
                sum := sum + a[i];
        }
        write “sum”;
}
```

Ex2: Write an algorithm in pseudo code format to find maximum value in a list of values

```
Algorithm max(a, n)
//a is an array and n is size of the array
{
        max := a[1];
        for i := 2 to n do
        {
                if a[i] > max then
                        max := a[i];
        }
        write “max”;
}
```

Ex3: Write an algorithm in pseudo code format to check whether a number is Armstrong number or not

```
Algorithm Armstrong(n)
// n is a positive integer
{
        sum := 0;
        m := n;
        while n > 0 do
        {
                r := n % 10;
                sum := sum + (r * r * r);
                n := n / 10;
        }
        if sum = m then
                write “the given number is Armstrong”;
        else
                write “the given number is not Armstrong”;
}
```

Ex4: Write an algorithm to check whether the given number is strong number or not

A number is said to be strong number if sum of the factorial values of digits of the given number is same as the number

```
Algorithm strong(n)
// n is a positive integer
{
        sum := 0;
        m := n;
        while n > 0 do
        {
                r := n % 10;
                f := 1;
                for i := 1 to r do
                {
                        f := f * i;
                }
                sum := sum + f;
                n := n / 10;
        }
        if sum = m then
                write "the given number is strong";
        else

          write "the given number is not strong";

       }
```

3.Recursive algorithm

- An algorithm which calls itself is said to be recursive algorithm.
- Recursive algorithms are used to solve complex problems in an easy manner.
- Recursion can be used as replacement of loops.
- There are two types of recursive algorithms: 1) direct recursive algorithms, 2) indirect recursive algorithms.

Direct recursive algorithm
An algorithm which calls itself is direct recursive algorithm.

Ex:

```
Algorithm A()
 {
        .
        .
        .
        A();
        .
        .
 }
```

Indirect recursive algorithm
If A and B are two algorithms and if algorithm A calls algorithm B and if algorithm B calls algorithm A then algorithm A is called as indirect recursive algorithm.

Ex:

```
Algorithm A()
 {
```

```
        .
        .
        .
        B();
        .
        .
}

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

Ex: Write a recursive algorithm to find factorial of a number

```
Algorithm factorial(n)
//n is a positive integer
{
        if n = 1 then
                return 1;
        else
        return n * factorial(n-1);
}
```

Ex: Write an algorithm to find factorial of a number

```
Algorithm factorial(n)
//n is a positive integer
{
        f := 1;
        for i := 1 to n step 1 do
                f := f * i;
        write "f";
}
```

Ex: Write a recursive algorithm to calculate sum of values in an array

```
Algorithm rsum(a, n)
// a is an array containing list of values and n is size of array
{
        if n = 0 then
                return 0;
        else
                return a[n] + rsum(a, n-1);
}
```

Ex: Write a recursive algorithm to calculate sum of digits in a number

```
Algorithm rsdigits(n)
// n is a positive number
{
        if n = 0 then
                return 0;

         else
          return n % 10 + rsdigits(n / 10);
 }
```

4.Performance analysis

Performance of any algorithm is measured in terms of space and time complexity.

1. Space complexity of an algorithm indicates the memory requirement of the algorithm.
2. Time complexity of an algorithm indicates the total CPU time required to execute the algorithm.

4.1 Space complexity for an algorithm

- Space complexity of an algorithm is sum of space required for fixed part of algorithm and space required for variable part of algorithm.
- Under fixed part, the space for the following is considered
  1) Code of algorithm
  2) Simple variables or local variables
  3) Defined constants
- Under variable part, the space for the following is considered
  1) Variables whose size varies from one instance of the problem to another instance (arrays, structures and so on)
  2) Global or referenced variables
  3) Recursion stack
- Recursion stack space is considered only for recursive algorithms. For each call of recursive algorithm, the following information is stored in recursion stack
  1) Values of formal parameters
  2) Values of local variables
  3) Return value

Ex1: Calculate space complexity of the following algorithm

```
Algorithm Add(a, b)
{
        c := a+b;
        write c;
}
```

Space complexity=space for fixed part + space for variable part

Space for fixed part:
Space for code=c words
Space for simple variables=3 (a, b, c) words
Space for defined constants=0 words

Space for variable part:
Space for arrays=0 words
Space for global variables=0 words
Space for recursion stack=0 words

Space complexity=c+3 +0+0+0+0=(c+3) words

Ex2: Calculate space complexity of the following algorithm

```
Algorithm Sum(a, n)
{
        sum := 0;
        for i := 1 to n do
                sum := sum + a[i];
        write „sum“;
}
```

Space for fixed part:
Space for code=c words
Space for simple variables=3 (n, sum, i) words
Space for defined constants=0 words

Space for variable part:
Space for arrays=n (a) words
Space for global variables=0 words
Space for recursion stack=0 words

Space complexity=c+3+0+n+0+0=(c+n+3) words

Ex3: Calculate space complexity for the following algorithm

```
Algorithm Armstrong(n)
// n is a positive integer
{
        sum := 0;
        m := n;
        while n > 0 do
        {
                r := n % 10;
                sum := sum + (r * r * r);
                n := n / 10;
        }
        if sum = m then
                write “the given number is Armstrong”;
        else
                write “the given number is not Armstrong”;
}
```

Space for fixed part:
Space for code=c words
Space for simple variables=4 (n, sum, m, r) words
Space for defined constants=0 words

Space for variable part:
Space for arrays=0 words
Space for global variables=0 words
Space for recursion stack=0 words

Space complexity=c+4+0+0+0+0=(c+4) words

Ex4: calculate space complexity for the following algorithm

```
Algorithm MatAdd(a, b, m, n)
// a, b are matrices of size mxn
{
        for i := 1 to m do
        {
                for j := 1 to n do
                {
                        c[i, j] := a[i, j] + b[i, j];
                        write c[i, j];
                }
        }
}
```

Space for fixed part:
Space for code=c words
Space for simple variables=4 (m, n, i, j) words
Space for defined constants=0 words

Space for variable part:
Space for arrays=3mn (a, b, c) words
Space for global variables=0 words
Space for recursion stack=0 words

Space complexity=c+4+0+3mn+0+0=(c+3mn+4) words

Ex5: calculate space complexity for the following algorithm

```
Algorithm MatMul(a, b, m, n)
// a, b are matrices of size mxn
{
        for i := 1 to m do
        {
                for j := 1 to n do
                {
                        c[i, j] := 0;
                        for k := 1 to m do
                        {
```

```
                                        c[i, j] := c[i, j] + a[i, k] * b[k, j];
                                }
                        }
                        write c[i, j];
                }
        }
```

Space for fixed part:
Space for code=c words
Space for simple variables=5 (m, n, i, j, k) words
Space for defined constants=0 words
Space for variable part:
Space for arrays=3mn (a, b, c) words
Space for global variables=0 words
Space for recursion stack=0 words

Space complexity=c+5+0+3mn+0+0=(c+3mn+5) words
Ex6: calculate space complexity for the following recursive algorithm

```
Algorithm factorial(n)
// n is a positive integer
{
        if n = 1 then
                return 1;
        else
                return  n*factorial(n-1);
}
```

Space for fixed part:Space for code=c words
Space for simple variables=1 (n) word

Space for defined constants=0 words

Space for variable part:
Space for arrays=0 words
Space for global variables=0 words
Space for recursion stack=2n words

For each call of factorial algorithm, two values are stored in recursion stack (formal parameter n and return value). The factorial algorithm is called for n times. Total space required by the recursion stack is n*2 words.

Space complexity=c+1+0+0+0+2n=(c+2n+1) words

Ex7: calculate space complexity for the following recursive algorithm

```
Algorithm Rsum(a, n)
// a is an array of size n
{
        if n = 0 then
                return 0;
                else
                   return a[n] + Rsum(a, n-1);    }
```

Space for fixed part:
Space for code=c words
Space for simple variables=1 (n) word
Space for defined constants=0 words

Space for variable part:
Space for arrays=n words
Space for global variables=0 words Space
for recursion stack=3(n+1) words

For each call of the algorithm, three values are stored in recursion stack (formal parameters: n, starting address of array and return value). The algorithm is called for n+1 times. Total space required by the recursion stack is (n+1)*3 words.

Space complexity = c+1+0+n+0+(n+1)3=(c+4n+4) words

4.2 Time complexity

Time complexity of an algorithm is the total time required for completing the execution of the algorithm. Two methods are used to calculate time complexity of the algorithm

1) Step count
2) Frequency count

4.2.1 Step count method

In this method, a global variable called count with initial value 0 is used.

The value of count variable is incremented by 1 after each executable statement in the algorithm. At the end of algorithm, the value of count variable indicates the time complexity of the algorithm.

The computer executes a program in steps and each step has a time cost associated with it. It means that a step could be done in finite amount of time.

Program Step	**Step Value**	**Description**
Comments	0	comments are not executed
Assignments	1	can be done in constant time
Arithmetic Operations	1	con be done in constant time
Loops	Count Steps	if loop runs n times, count step as n+1 and any assignment inside loop is counted as n step.
Flow Control	1 step	only take account of part that was executed.

Ex1: calculate time complexity of the algorithm

```
Algorithm sum(a, n)
// a is an array of size n
{
        sum := 0;                               1
        for i := 1 to n do                      n+1
                sum := sum + a[i];              n
        write „sum“;                            1
}
```

Time complexity=1+n+1+n+1=2n+3

Ex2: calculate time complexity of the algorithm

```
Algorithm Max(a, n)
//
{
        max := a[1];                            1
        for i := 2 to n do                      n
        {
                if max < a[i] then              n-1
                        max := a[i];            n
        }
        write „max“;                            1
}
```

Time complexity=1+n+n-1+n+1=3n+1

Ex3: calculate time complexity for the algorithm

```
Algorithm MatAdd(a, b, m, n)
// a, b are matrices of size mxn
{
        for i := 1 to m do                              m+1
```

```
        {
                for j := 1 to n do                              m(n+1)
                {
                        c[i, j] := a[i, j] + b[i, j];           mn
                        write c[i, j];                          mn
                }
        }
}
```

Time complexity=m+1+m(n+1)+mn+mn=3mn+2m+1

Ex4: calculate time complexity for the algorithm

```
Algorithm MatMul(a, b, m, n)
// a, b are matrices of size mxn
{
        for i := 1 to m do                                      m+1
        {
                for j := 1 to n do                              m(n+1)
                {
                        c[i, j] := 0;                           mn
                        for k := 1 to m do                      mn(m+1)
                        {
                                c[i, j] := c[i, j] + a[i, k] * b[k, j];     mn(m)
                        }
                }
                write c[i, j];                                  mn
        }
}
```

Time complexity=m+1+m(n+1)+mn+mn(m+1)+mn(m)+mn=$2m^2n+4mn+m+1$

Ex5: Calculate time complexity for the following algorithm

```
Algorithm Armstrong(n)
// n is a positive integer
{
        sum := 0;                                               1
        m := n;                                                 1
        while n > 0 do                                          k+1
        {
                r := n % 10;                                    k
                sum := sum + (r * r * r);                       k
                n := n / 10;                                    k
        }
        if sum = m then                                         1
                write “the given number is Armstrong”;          1
        else                                                    1
                write “the given number is not Armstrong”;      1
}
```

Time complexity=1+1+k+1+k+k+k+1+1=4k+5

Where „k" is number of digits in „n".

Ex6: calculate time complexity of the algorithm

```
Algorithm factorial(n)
// n is a positive integer
{
        if n = 1 then                                   1
                return 1;                               1
        else                                            1
                return n*factorial(n-1);                1
}
```

Time complexity:
Case1: when n=1
In this case, if and return statements are executed and the algorithm terminates. So, the time complexity is

$$T(1)=2$$

Case2: when n>1
In this case, else and return statements are executed and the algorithm is called with (n-1). So, the time complexity is

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

Solving the above equation

$T(n)=2+T(n-1)$
$=2+2+T(n-2)$
$=2+2+2+T(n-3)$
.
.
After (n-1) times
$=2+2+2+2+\ldots\ldots+T(1)$
$=2+2+2+2+\ldots\ldots$ n times
$=2n$

$$T(n)=2n$$

Ex7: Calculate time complexity for the following recursive algorithm

```
Algorithm Rsum(a, n)
// a is an array containing n number of values
{
        if n=0 then                                     1
                return 0;                               1
        else                                            1
                return a[n]+Rsum(a,n-1);                1
}
```

Time complexity
Case1: when n=0

In this case, if and return statements are executed and the algorithm terminates. So, the time complexity is

$$T(1)=2$$

Case2: when n>1
In this case, else and return statements are executed and the algorithm is called with (n-1). So, the time complexity is

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

Solving the above equation

$T(n)=2+T(n-1)$
$=2+2+T(n-2)$
$=2+2+2+T(n-3)$
.
.
After n times
$=2+2+2+2+\ldots\ldots+T(0)$
$=2+2+2+2+\ldots\ldots$ n+1 times
$=2(n+1)$

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

$=2n$

$$T(n)=2n$$

Ex7: Calculate time complexity for the following recursive algorithm

```
Algorithm Rsum(a, n)
// a is an array containing n number of values
{
        if n=0 then                                  1
                return 0;                            1
        else                                         1
                return a[n]+Rsum(a,n-1);             1
}
```

Time complexity
Case1: when n=0
In this case, if and return statements are executed and the algorithm terminates. So, the time complexity is

$$T(1)=2$$

Case2: when n>1
In this case, else and return statements are executed and the algorithm is called with (n-1). So, the time complexity is

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

Solving the above equation

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

.

After n times

=2+2+2+2+……+T(0)

=2+2+2+2+…… n+1 times

=2(n+1)

T(n)=2(n+1)

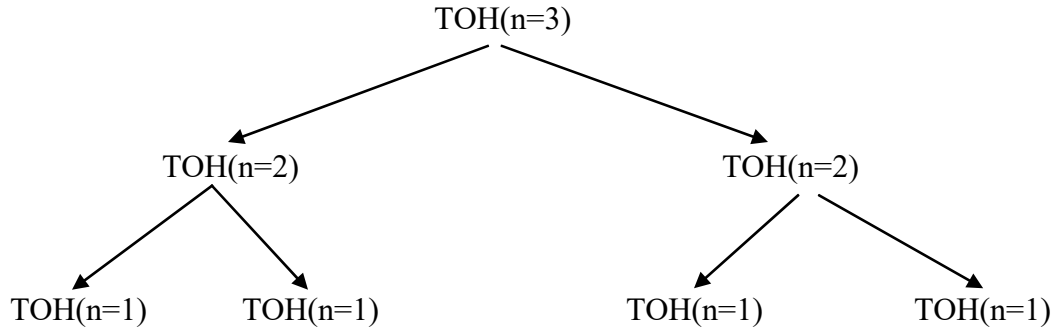
Ex8: Write recursive algorithm for Towers of Hanoi. Calculate space and time complexity.

```
Algorithm TOH(n, A, B, C)
// n is number of disks
// A, B, C are towers. A is source and C is destination
{
        if n>0 then
        {
                TOH(n-1, A, C, B);
                Move nth disk from tower A to tower C;
                TOH(n-1, B, A, C);
        }
}
```

Space for fixed part:
Space for code=c words
Space for simple variables=4 (n,A,B,C) word
Space for defined constants=0 words

Space for variable part:
Space for arrays=0
Space for global variables=0 words
Space for recursion stack=$4(2^n-1)$ words

This algorithm is called for (2^n-1) times. The recursive calls of the algorithm for n=3 are shown below. For each call of the algorithm, the values of formal parameters (n, starting address of A, starting address of B and starting address of C) are stored in the recursion stack. These formal parameters require 4 words of memory. So, total space required by recursion stack is $4(2^n-1)$ words.



Space complexity = $c+4+0+0+0+4(2^n-1) = c+4+4(2^n-1)$ words

Time Complexity:
Case1: when n=0
In this case, only if statement is executed. The time complexity is

$$T(0)=1$$

Case2: when n>0
In this case, time complexity is

$$T(n)=1+T(n-1)+1+T(n-1)$$
$$T(n)=2+2T(n-1)$$
$$T(n)=2+2[2+2T(n-2)]=2+2^2+2^2T(n-2)$$
$$T(n)=2+2^2+2^2[2+2T(n-3)]=2+2^2+2^3+2^3[2+2T(n-4)]$$

.
.
After n times

$$T(n)=2+2^2+2^3+\ldots.+2^n$$

Ex9: Write recursive algorithm for displaying Fibonacci numbers. Calculate space and time complexity.

```
Algorithm Fibonacci(n, a, b)
// n is number of Fibonacci numbers
// a, b are previous two Fibonacci numbers
{
        if n>0 then
        {
        c:=a+b;
        w ri t e ( c);
        a:=b;
        b := c; Fibonacci( n-1, a, b);
        }
}
```

Space complexity:
Space for fixed part:
Space for code=c words
Space for simple variables=4 (n, a, b,

c) words Space for defined
constants=0 words

Space for variable part:
Space for arrays=0 words
Space for global
variables=0 words Space
for recursion stack=4n
words

This algorithm is called for n times. For each call of the algorithm, the values of formal parameters (n, a, b) and the value of local variable (c) are stored in the recursion stack. 4 words of memory are required for storing information of each call of the algorithm. So, total space required by recursion stack is 4n words.

Space complexity=c+4+0+0+0+4n= c+4n+4

words Time complexity:
Case1: when n=0
In this case, only if statement is executed. The time
complexity is T(0)=1

Case2: when n>0
In this case, time complexity is
T(n)=1+1+1+1+1+T(n-1)
=5+T(n-1)
T(n)=5+5+T(n-2)
T(n)=5+5+5+T(n-3)
. .
.
After n
times
T(n)=5+5+5+….
n times+1
T(n)=5n+1

4.2.2 Frequency Count or Tabulation Method

- This is another method for calculating time complexity of an algorithm.
- In this method, steps per execution and frequency count is calculated for each executable statement in the algorithm.
- Steps per executions is the no of steps needed by the statement.
- Frequency count of a statement indicates the number of times that statement is executed.
- The frequency counts of all executable statements are added to get time complexity of the algorithm.

Ex1:

Statements	Step count	Frequency	Total steps

Algorithm Sum(a, n)
{
s := 0;
for i := 1 to n do
{
s := s + a[i];
}
write s;
}
	
	1
	1
	
	1
	
	1
		
		1
		n+1
		
		n
		
		1
			
			1
			n+1
			
			n
			
			1

		Total: 2n+3	

Time complexity=2n+3

- Steps needed per statement = steps per execution*frequency
- Sum of these steps gives the total step count.

4.3 Asymptotic notations

- Asymptotic notations are used to represent space and time complexity of algorithms.
- Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

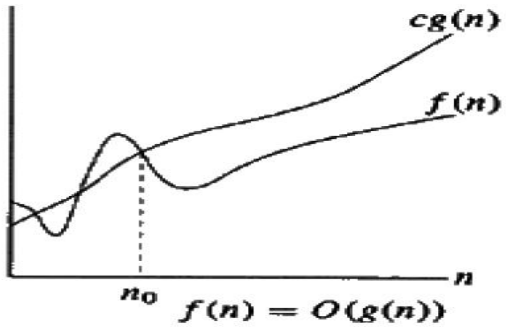
Commonly used asymptotic notations are

1) Big oh (O)
2) Omega (Ω)
3) Theta (θ)
4) Small oh (o)
5) Small omega (ω)

1.Big oh notation (O)

- The Big-O notation describes the worst-case running time of an algorithm.
- Compute the Big-O of an algorithm by counting how many iterations an algorithm will take in the worst-case scenario with an input of n.
- Denote upper bound.
  - Definition:

If f(n) and g(n) are two functions defined in terms of n then f(n)=O(g(n)) if and only if there exists two positive constants c and n_o such that $f(n) \leq c*g(n)$, for all values of n where $n \geq n_0$.



Ex1: if the complexity of an algorithm is 3n+2 then

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

3n +2 ≤ 4n, n≥2

c=4
and
n_0=2
So,
3n+2
=O(n
)
Ex2: if the complexity of an algorithm is 100n+6 then 100n+6=O(n)

f(n)=100n+6
g(n)=n

100n+6 ≤

101n, n≥6

c=101 and

n_0=6
So, 100n+6=O(n)

Ex3: if the complexity of an algorithm is $10n^2+4n+6$ then

$10n^2+4n+6=O(n^2)$ f(n)= $10n^2+4n+6$
$g(n)=n^2$

$10n^2+4n+6$ ≤

$11n^2$, n≥6

c=11 and

n_0=6
So, $10n^2+4n+6=O(n^2)$

Ex4: if the complexity of an algorithm is $6*2^n + n^2$ then $6*2^n + n^2=O(2^n)$

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

$6*2^n$ +

$n^2 \leq 7*2^n$, $n \geq 1$

c=7 and n_0=1
So, $6*2^n + n^2=O(2^n)$

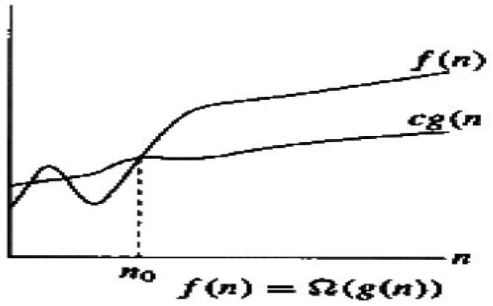
Actually, 3n+2 can be represented as
$3n+2=O(n)$ because $3n+2 \leq 4n$ where
c=4 and n_0=2
or as $3n+2=O(n^2)$
because $3n+2 \leq 4n^2$ where c=4
and n_0=2 or as $3n+2=O(n^3)$
because $3n+2 \leq 4n^3$ where c=4 and n_0=2

In Big Oh notation, the least upper bound has to be used. So, $3n+2=O(n)$

2.Omega Notation (Ω)

- Omega notation represents the lower bound of the running time of an algorithm.
- describe the best case complexity of an algorithm.

definition
If f(n) and g(n) are two functions defined in terms of n then $f(n)=\Omega(g(n))$ if and only if there exists two positive constants c and n_o such that $f(n) \geq c*g(n)$, for all values of n where $n \geq n_0$.



Ex1: if the complexity of an algorithm is 3n+2 then

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

$3n+2 \geq 3n$, $n \geq 1$

$c=3$
and
$n_0=1$
So,
$3n+2$
$=\Omega(n$
)

Ex2: if the complexity of an algorithm is $100n+6$ then

$100n+6=\Omega(n)$ $f(n)=100n+6$
$g(n)=n$

$100n+6 \geq$

$100n$, $n \geq 1$

$c=100$ and

$n_0=1$
So, $100n+6=\Omega(n)$

Ex3: if the complexity of an algorithm is $10n^2+4n+6$ then

$10n^2+4n+6=\Omega(n^2)$ $f(n)= 10n^2+4n+6$
$g(n)=n^2$

$10n^2+4n+6 \geq$

$10n^2$, $n \geq 1$

$c=10$ and

$n_0=1$
So, $10n^2+4n+6=\Omega(n^2)$

Ex4: if the complexity of an algorithm is $6*2^n + n^2$ then $6*2^n + n^2=\Omega(2^n)$

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

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

$6*2^n$, $n \geq 1$

c=6 and $n_0=1$
So, $6*2^n + n^2 = \Omega(2^n)$

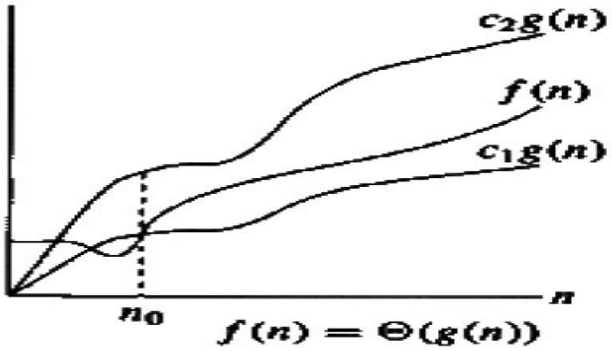
Actually, 3n+2 can be represented as
$3n+2=\Omega(n)$ because $3n+2 \geq 3n$ where
c=3 and $n_0=1$
or as $3n+2=\Omega(1)$
because $3n+2 \geq 3$ where c=3 and $n_0=1$

In Omega notation, the highest lower bound has to be used. So, $3n+2=\Omega(n)$

3. Theta notation (θ)

- theta notation encloses the function from above and below.
- Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.
- definition

If f(n) and g(n) are two functions defined in terms of n then $f(n)=\theta(g(n))$ if and only if there exists three positive constants c_1, c_2 and n_o such that $c_1*g(n) \leq f(n) \leq c_2*g(n)$, for all values of n where $n \geq n_0$.



Ex1: if the complexity of an algorithm is 3n+2 then

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

$3n \leq 3n+2 \leq 4n$, $n \geq 2$
$c_1=3$, $c_2=4$
and $n_0=2$
So,
$3n+2=\theta(n$

)

Ex2: if the complexity of an algorithm is 100n+6 then

$100n+6=\theta(n)$ $f(n)=100n+6$
$g(n)=n$

$100n \leq 100n+6 \leq 101n$, $n \geq 6$

$c_1=100$, $c_2=101$
and $n_0=6$ So,
$100n+6=\theta(n)$

Ex3: if the complexity of an algorithm is $10n^2+4n+6$ then

$10n^2+4n+6=\theta(n^2)$ $f(n)= 10n^2+4n+6$
$g(n)=n^2$

$10n^2 \leq 10n^2+4n+6 \leq$

$11n^2$, $n \geq 6$ $c_1=10$,

$c_2=11$ and $n_0=6$
So, $10n^2+4n+6=\theta(n^2)$

Ex4: if the complexity of an algorithm is $6*2^n + n^2$ then $6*2^n +$

$n^2=\theta(2^n)$ $f(n)=6*2^n + n^2$
$g(n)=2^n$

$6*2^n \leq 6*2^n + n^2 \leq$

$7*2^n$, $n \geq 1$ $c_1=6$,

$c_2=7$ and $n_0=1$
So, $6*2^n + n^2=\theta(2^n)$

4. Small Oh notation (o)

If f(n) and g(n) are two functions defined in terms of n then f(n)=o(g(n)) if and only if

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

Ex1: if the complexity of an algorithm is 3n+2 then $3n+2=o(n^2)$ as

$\lim_{n^2 \to \infty} \frac{3n+2}{n^2} = 0$

Ex2: if the complexity of an algorithm is $10n^2+4n+6$ then $10n^2+4n+6=o(n^3)$ as

$\lim_{n\to\infty} \frac{10n2+4n+6}{n^3} = 0$

5. Small Omega notation

If f(n) and g(n) are two functions defined in terms of n then $f(n)=\omega(g(n))$ if and only if

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

Ex1: if the complexity of an algorithm is 3n+2 then $3n+2=\omega(1)$ as

$\lim_{n\to\infty} \frac{1}{3n+2} = 0$

Out of the five notations, the frequently used notations are O, Ω and θ. The θ notation accurately represents the complexity of algorithms.

Ex1: Show that

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

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

$3n^3+2n^2 \leq 4n^3, n\geq 2$
c

=4,

$n_0=2$

$3n^3+2n$

$^2=O(n^3$

)

Ex2: Show that $3^n \neq O(2^n)$

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

It is not possible to identify c and n_0 such that $3^n \leq c2^n$ is satisfied. So,

$3^n \neq O(2^n)$ Ex3: Show that $3n^3+2n^2=\Omega(n^3)$

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

$3n^3+2n^2$
$\geq 3n^3$,
$n\geq 1$ c=3,
$n_0=1$

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

Ex4: Show that $3n^3+2n^2=\theta(n^3)$

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

$g(n)=n^3$

$3n^3 \leq 3n^3+2n^2 \leq 4n^3$, $n\geq 2$

$c_1=3$, $c_2=4$,$n_0=2$

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

4.4 Performance Measurement

- Performance measurement concerned with obtaining the space and time requirements of a particular algorithm.
- These quantities depend on the compiler and options used as well as on the computer on which program is executed.
- Here focus is on the measuring the computing time of a program.
- Clocking procedure is used to compute the time needed.
- This procedure assumes GetTime() that returns current time in milliseconds
- Worst case performance measured by taking different vales for input size of the program.

Ex:measuring performance of linear search.

```
Algorithm SeqSearch(a, x, n)
// Search for x in a[1 : n]. a[0] is used as additional space.
{
    i := n; a[0] := x;
    while (a[i] ≠ x) do i := i − 1;
    return i;
}
```

- The following algorithm generates input data for input size and array.
- Timesearch algorithm marks the starting and end times of executing the code related to linear search using GetTime() and computes time needed as difference between the start and end time.

```
Algorithm TimeSearch()
{
    for j := 1 to 1000 do  a[j] := j;
    for j := 1 to 10 do
    {
        n[j] := 10 * (j − 1); n[j + 10] := 100 * j;
    }
    for j := 1 to 20 do
    {
        h := GetTime();
        k := SeqSearch(a, 0, n[j]);
        h1 := GetTime();
        t := h1 − h;
        write (n[j], t);
    }
}
```

Running time search produces the following output.

n	time	n	time
0	0	100	0
10	0	200	0
20	0	300	1
30	0	400	0
40	0	500	1
50	0	600	0
60	0	700	0
70	0	800	1
80	0	900	0
90	0	1000	0

- Time is not computed for dome input sizes as program can be executed bofore clock can change time.
- The process of running linear search can be repeated for r no of times and the time obtained is divided by r to get time needed for running linear search one time.
- In below algorithm r is the array that contains repetition factors.

```
Algorithm TimeSearch()
{
    // Repetition factors
    r[21] := {0, 200000, 200000, 150000, 100000, 100000, 100000,
        50000, 50000, 50000, 50000, 50000, 50000, 50000, 50000,
        50000, 50000, 25000, 25000, 25000, 25000};
    for j := 1 to 1000 do a[j] := j;
    for j := 1 to 10 do
    {
        n[j] := 10 * (j - 1); n[j + 10] := 100 * j;
    }
    for j := 1 to 20 do
    {
        h := GetTime();
        for i := 1 to r[j] do k := SeqSearch(a, 0, n[j]);
        h1 := GetTime();
        t1 := h1 - h;
        t := t1; t := t/r[j];
        write (n[j], t1, t);
    }
}
```

With the repetition time is computed.

n	$t1$	t	n	$t1$	t
0	308	0.002	100	1683	0.034
10	923	0.005	200	3359	0.067
20	1181	0.008	300	4693	0.094
30	1087	0.011	400	6323	0.126
40	1384	0.014	500	7799	0.156
50	1691	0.017	600	9310	0.186
60	999	0.020	700	5419	0.217
70	1156	0.023	800	6201	0.248
80	1306	0.026	900	6994	0.280
90	1460	0.029	1000	7725	0.309

2.Divide and Conquer

1.General Method:

Divide and Conquer is one of the best-known general algorithm design technique.

- Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, 1<k<=n, yielding 'k' sub problems.
- These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
- If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem. For those cases the reapplication of the divide-and- conquer principle is naturally expressed by a recursive algorithm.
- Control Abstraction for divide and conquer:

```
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));
    }
}
```

In the above specification,

- ✓ Initially ***DAndC(P)*** is invoked, where 'P' is the problem to be solved.
- ✓ ***Small (P)*** is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this so, the function '***S'*** is invoked. Otherwise, the problem P is divided into smaller sub problems. These sub problems P_1, P_2 …P_k are solved by recursive application of ***DAndC***.
- ✓ ***Combine*** is a function that determines the solution to P using the solutions to the 'k' sub problems.

Recurrence equation for divide and conquer:

If the size of problem 'p' is n and the sizes of the 'k' sub problems are n_1, n_2….n_k, respectively, then the computing time of divide and conquer is described by the recurrence relation

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

- T(n) is the time for divide and conquer method on any input of size n and
- g(n) is the time to compute answer directly for small inputs.
- The function f(n) is the time for dividing the problem 'p' and combining the solutions to sub problems.

- More generally, an instance of size **n** can be divided into **b** instances of size **n/b**, with **a** of them needing to be solved. (Here, a and b are constants; **a>=1 and b > 1**.). Assuming that size **n** is a power of **b**(i.e. $n=b^k$), to simplify our analysis, we get the following recurrence for the running time T(n):

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

- where f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.

2. Binary Search

Problem definition: Let a_i, $1 \le i \le n$ be a list of elements that are sorted in non-decreasing order. The problem is to find whether a given element x is present in the list or not. If x is present we have to determine a value j (element's position) such that a_j=x. If x is not in the list, then j is set to zero.

Solution: Let **P = (n, a_i…a_l , x)** denote an arbitrary instance of search problem where ***n*** is the number of elements in the list, **a_i…a_l** is the list of elements and **x** is the key element to be searched for in the given list. **Binary search** on the list is done as follows:

Step1: Pick an index *q* in the middle range [i, *l*] i.e. *q*= [(n + 1)/2] and compare x with a_q.
Step 2: if x = a_q i.e key element is equal to mid element, the problem is immediately solved.
Step 3: if x <$a_q$in this case x has to be searched for only in the sub-list a_i, a_{i+1}, ..., a_{q-}

Therefore, problem reduces to **(q-i, a_i…a_{q-1}, x).**

Step 4: if x >a_q,x has to be searched for only in the sub-list a_{q+1}, ..., a_l .

Therefore problem reduces to **(*l*-i, a_{q+1}…a_l, x).**

For the above solution procedure, the Algorithm can be implemented as recursive or non- recursive algorithm.

```
Algorithm BinSrch(a, i, l, x)
// Given an array a[i : l] of elements in nondecreasing
// 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 Small(P)
     {
          if (x = a[i]) then return i;
          else return 0;
     }
     else
     { // Reduce P into a smaller subproblem.
          mid := ⌊(i + l)/2⌋;
          if (x = a[mid]) then return mid;
          else if (x < a[mid]) then
                    return BinSrch(a, i, mid − 1, x);
               else return BinSrch(a, mid + 1, l, x);
     }
}
```

Analysis:

In binary search the basic operation is key comparison.

Binary Search can be analyzed with the best, worst, and average case number of comparisons.

Recursive Binary Search, count each pass through the if-then-else block as one comparison.

Best case –Θ(1) In the best case, the key is the middle in the array. A constant number of comparisons (actually just 1) are required.

Worst case - Θ(log_2 n) In the worst case, the key does not exist in the array at all. Through each recursion or iteration of Binary Search, the size of the admissible range is halved. This halving can be done ceiling(log_2n) times. Thus, [log_2 n] comparisons are required.

Sometimes, in case of the successful search, it may take maximum number of comparisons.
] log_2 n]. So worst case complexity of successful binary search is Θ (log_2 n).

Average case - Θ (log_2n) To find the average case, take the sum of the product of number of comparisons required to find each element and the probability of searching for that element. To simplify the analysis, assume that no item which is not in array will be searched for, and that the probabilities of searching for each element are uniform.

successful searches			unsuccessful searches
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
best,	average,	worst	best, average, worst

Space Complexity - The space requirements for the recursive and iterative versions of binary search are different. Iterative Binary Search requires only a constant amount of space, while Recursive Binary Search requires space proportional to the number of comparisons to maintain the recursion stack.

3.Finding the maximum and minimum

Problem statement: Given a list of n elements, the problem is to find the maximum and minimum items.

StraightMaxMin: A simple and straight forward algorithm to achieve this is given below.

```
Algorithm StraightMaxMin(a, n, max, min)
// Set max to the maximum and min to the minimum of a[1 : n].
{
     max := min := a[1];
     for i := 2 to n do
     {
          if (a[i] > max) then max := a[i];
          if (a[i] < min) then min := a[i];
     }
}
```

StraightMaxMin requires 2(n-1) comparisons in the best, average & worst cases.

Algorithm based on Divide and Conquer strategy

Let P = (n, a [i],……,a [j]) denote an arbitrary instance of the problem. Here 'n' is the no. of elements in the list (a[i],….,a[j]) and we are interested in finding the maximum and minimum of the list. If the list has more than 2 elements, P has to be divided into smaller instances.

For example, we might divide 'P' into the 2 instances,

P1= ([n/2],a[1], a[n/2])

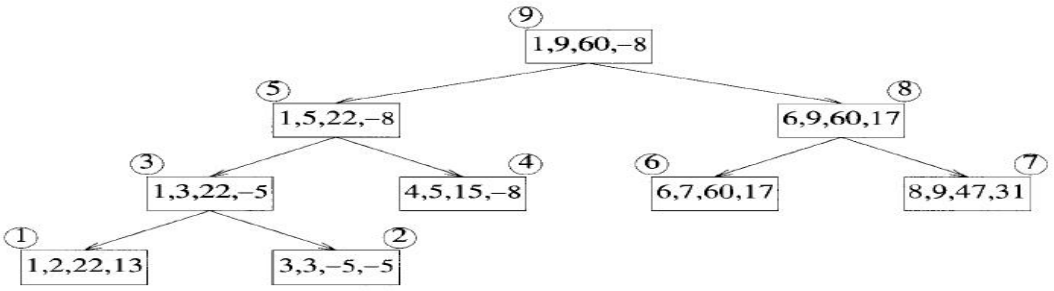
P2= (n-[n/2],[[n/2]+1],……., a[n])

After having divided 'P' into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

```
Algorithm MaxMin(i, j, max, min)
// a[1 : n] is a global array. Parameters i and j are integers,
// 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
// largest and smallest values in a[i : j], respectively.
{
     if (i = j) then max := min := a[i]; // Small(P)
     else if (i = j − 1) then  // Another case of Small(P)
          {
               if (a[i] < a[j]) then
               {
                    max := a[j]; min := a[i];
               }
               else
               {
                    max := a[i]; min := a[j];
               }
          }
          else
          {    // If P is not small, divide P into subproblems.
               // Find where to split the set.
                    mid := ⌊(i + j)/2⌋;
               // Solve the subproblems.
                    MaxMin(i, mid, max, min);
                    MaxMin(mid + 1, j, max1, min1);
               // Combine the solutions.
                    if (max < max1) then max := max1;
                    if (min > min1) then min := min1;
          }
}
```

Example:

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	−5	−8	15	60	17	31	47



Complexity Analysis

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

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

When n is a power of two, $n = 2^k$ for some positive integer k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \textstyle\sum_{1 \le i \le k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned}$$

4.Quick Sort

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides (or partitions) them according to their value.

A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are} \ge A[s]}$$

Obviously, after a partition is achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and the right of A[s] independently (e.g., by the same method).

In quick sort, the entire work happens in the division stage, with no work required to combine the solutions to the sub problems.

Example:

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

Partitioning:

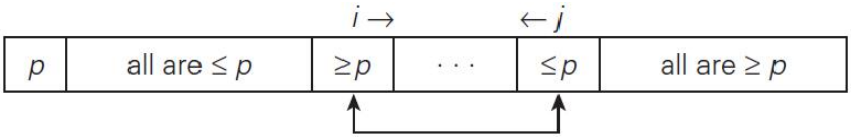
We start by selecting a pivot—an element with respect to whose value we are going to divide the subarray. There are several different strategies for selecting a pivot.We use the sophisticated method suggested by C.A.R. Hoare, the prominent British computer scientist who invented quicksort.

Select the subarray's first element: p = A[l].
Now scan the subarray from both ends,comparing the subarray's elements to the pivot.

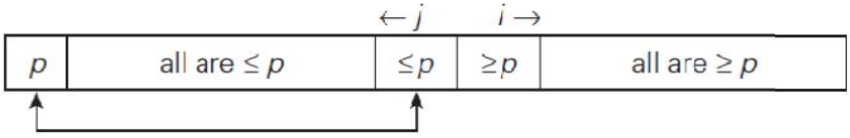
1. The left-to-right scan, denoted below by index pointer i, starts with the second element. Since we want elements smaller than the pivot to be in the left part of the subarray, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot.
2. The right-to-left scan, denoted below by index pointer j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the right part of the
   subarray, this scan skips over elements that are larger than the pivot and stops on encounteriAfter both scans stop,ng the first element smaller than or equal to the pivot.

three situations may arise, depending on whether or not the scanning indices have crossed.

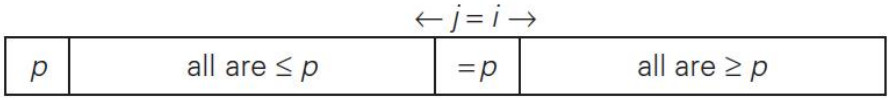
1. If scanning indices i and j have not crossed, i.e., i< j, we simply exchange A[i] and
   A[j] and resume the scans by incrementing I and decrementing j, respectively:



2. If the scanning indices have crossed over, i.e., i> j, we will have partitioned the subarray after exchanging the pivot with A[j]:



3. If the scanning indices stop while pointing to the same element, i.e., i = j, the value they are pointing to must be equal to p. Thus, we have the subarray partitioned, with the split position s = i = j :



We can combine this with the case-2 by exchanging the pivot with A[j] whenever i≥j

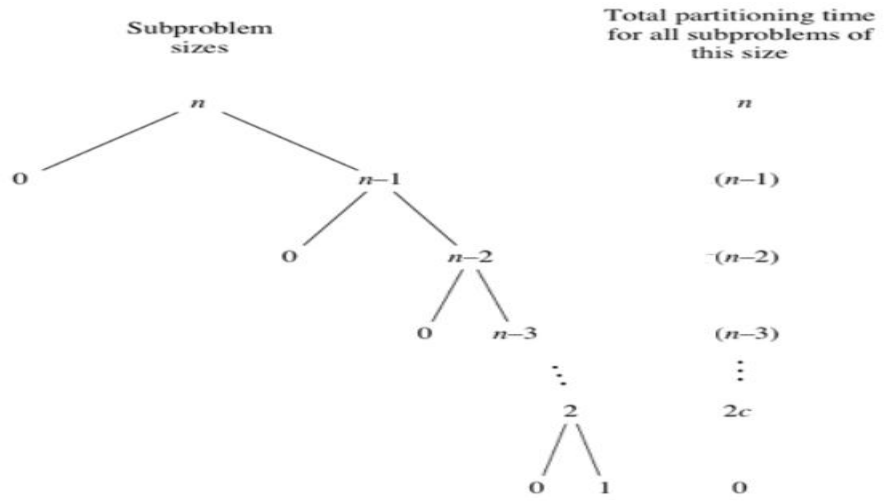
```
Algorithm Partition(a, m, p)
// Within a[m], a[m + 1], ..., a[p − 1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    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)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order; a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then  // If there are more than one element
    {
        // divide P into two subproblems.
            j := Partition(a, p, q + 1);
                 // j is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort(p, j − 1);
            QuickSort(j + 1, q);
        // There is no need for combining solutions.
    }
}
```

- Time complexity analysis

Worst case time complexity:

worst case occurs when the partition process always picks greatest or smallest element as pivot.



Worst case complexity is

T(n)=[n+(n-1)+(n-2)+..+2]+1-1

=n(n+1)/2-1

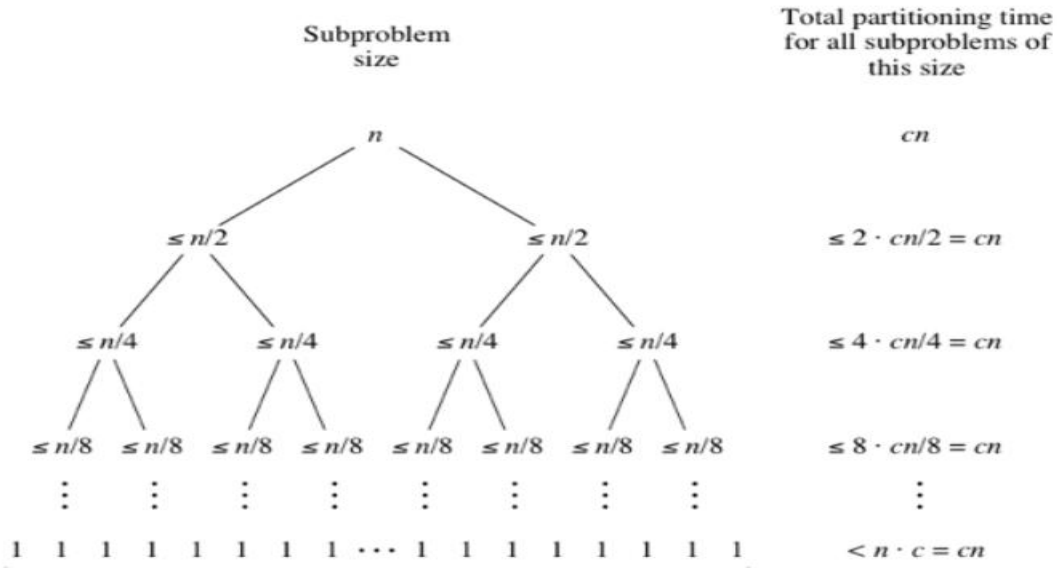
=(n^2+n)/2-1

T(n)=O(n^2)

Best case time complexity:

In the best case, the pivot is in the middle.

List is partitioned in to sub lists of equal size approximately.



Hence T (n) = 2 T (n/2) + C n

T(n)=2[2T(n /2)+cn/2]+c

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

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

=2^2 T(2T(n/8)+cn/4]+2cn

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

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

:

:

After k times

=2^kT(n/2^k)+kcn

Assuming that n=2^k then k= $\log_2 n$

T(n)=nT(n/n)+cn($\log_2 n$)=nT(1)+cn$\log_2 n$=n+cn$\log_2 n$=**O(n$\log_2 n$)**

Average case:

Comparisons needed when pivot is i^{th} smallest element are considered and average is taken to compute the average complexity.

T(n)=n+1+[T(0)+T(1)+…….+T(n)]/n+[T(n)+T(n-1)+……+0] /n

T(n)=cn+2/n[T(0)+T(1)+..... +T(n)]

After solving the above equation, the time complexity is T(n)=**O(n$\log_2 n$)**