**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.

**Criteria (or) characteristics (or) properties of an algorithm**

The following are characteristics of an algorithm

* Input
* Output
* Definiteness
* Finiteness
* Effectiveness

**Input**

There are zero (or) more inputs to an algorithm or its equivalent program

Ex:

main()

{

printf(“Welcome to VVIT”);

}

The above program has zero inputs.

main( )

{

int a;

printf(“%d”,a);

}

The above program has one input.

**Output**

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

**Definiteness**

Each step of algorithm should be clear and unambiguous.

Example for an ambiguous statement

add 5 or 6 to 7

**Finiteness**

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

**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.

**Pseudo code representation of algorithms**

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

**Delimiters:**

Two types of delimiters are used:

Space as separator

; as end of statements

**Comments:**

// notation is used to indicate both single line as well as multi line 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: <, ≤, >, ≥, =, ≠

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]

Array indexing can be started from 0 or 1.

**Conditional Statements:**

The conditional statements **if** and **case** are used in pseudo code.

**if:**

if statement is used to check one condition. There are three versions of if statement

1. Simple if statement
2. If –else statement
3. Nested if statement

The syntax of simple if statement is

if condition then

{

Block of statements

}

The syntax of if-else statement is

if condition then

{

Block of statements

}

else

{

Block of statements

}

The syntax of nested if statement is

if condition then

{

.

.

if condition then

{

.

.

}

}

**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

.

.

:default: 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:**

Three loop statements are used:

1. While
2. Repeat-until
3. For

while

The syntax of while statement is

while condition do

{

Block of statements

}

Repeat-until

The syntax of repeat-until 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.

Ex: read n;

write n;

write “VVIT”;

**Heading of the algorithm:**

Each algorithm should start with the heading

Algorithm name(list of parameters)

{

}

‘name’ is user defined and parameter list is optional. No need to specify data type for parameters. Parameters are used to pass input to the algorithm.

The statements break, continue and return are used with the same syntax and for the same purpose as in programming languages.

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

Algorithm sum(a, n)

//a is an array of size n

{

s := 0;

for i := 1 to n do

{

s := s + a[i];

}

write s;

}

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 of size n

{

m := a[1];

for i := 2 to n do

{

if a[i] > m then

m := a[i];

}

write m;

}

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

{

s := 0;

m := n;

while n > 0 do

{

r := n % 10;

s := s + (r \* r \* r);

n := n / 10;

}

if s = 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

{

s := 0;

m := n;

while n > 0 do

{

r := n % 10;

f := 1;

for i := 1 to r do

{

f := f \* i;

}

s := s + f;

n := n / 10;

}

if s = m then

write “the given number is strong”;

else

write “the given number is not strong”;

}

**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 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);

}

**Performance analysis**

Performance of any algorithm is measured in terms of space and time complexity. Space complexity of an algorithm indicates the memory requirement of the algorithm. Time complexity of an algorithm indicates the total CPU time required to execute the algorithm.

**Calculation of space complexity of 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)

// a and b are two numbers

{

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)

// a is an array of size 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 of 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 of 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 of 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 of 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 of 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

**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

**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.

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)

// a is an array of size n

{

max := a[1]; 1

for i := 2 to n do n

{

if max < a[i] then n-1

max := a[i]; n-1

}

write max; 1

}

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

Ex3: calculate time complexity of 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 of 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=2m2n+4mn+m+1

Ex5: Calculate time complexity of 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 following recursive 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 is terminated. 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+……+T(1)

=2+2+2+2+…… n times

=2n

T(n)=2n

Ex7: Calculate time complexity of 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 is terminated. 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=1 (n) word

Space for defined constants=0 words

Space for variable part:

Space for arrays=3n (A, B, C are arrays of size n) words

Space for global variables=0 words

Space for recursion stack=4(2n-1) words

This algorithm is called for (2n-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(2n-1) words.

TOH(n=3)

TOH(n=2) TOH(n=2)

TOH(n=1) TOH(n=1) TOH(n=1) TOH(n=1)

Space complexity = c+1+0+3n+0+4(2n-1) = c+1+3n+4(2n-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+22+22T(n-2)

T(n)=2+22+22[2+2T(n-3)]=2+22+23+23[2+2T(n-4)]

.

.

After n times

T(n)=2+22+23+….+2n

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;

write(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

**Frequency Count**

This is another method for calculating time complexity of an algorithm. In this method, frequency count is calculated for each executable statement in the algorithm. 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

**Asymptotic Notations**

Asymptotic notations are symbols used to represent complexity (space and time) of algorithms.

There are five asymptotic notations

1. Big Oh (O)
2. Omega (Ω)
3. Theta (θ)
4. Small Oh (o)
5. Small Omega (w)

If the space complexity of an algorithm is **2n+3** then it can be represented as **O(n).**

if the time complexity of an algorithm is **10n2+20** then it can be represented as **Ω(n2).**

**Big Oh (O) Notation**

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 exist two positive constants **c** and **n0** such that **f(n) ≤ c\*g(n)**, for all **n ≥ n0**

Here, f(n) is the complexity of the algorithm.

g(n) is selected based on the largest term in f(n).

Ex: if f(n)=n2+4n+15 then g(n)=n2

if f(n)=2n+3 then g(n)=n

if f(n)=10n4+5n3+10 then g(n)=n4

The values of c and n0 are identified based on the condition f(n)≤c\*g(n)

Ex: Represent the complexity **2n+3** using Big Oh notation

f(n)=2n+3

g(n)=n

f(n) ≤ c\*g(n)

2n+3 ≤ c\*n

c should be minimum 3 for the above condition to be satisfied (c=3).

2n+3 ≤ 3n

The above condition is satisfied for all values of n starting from 3 (n0 = 3).

So, f(n)=O(g(n))

**2n+3=O(n)**

2n+3 can also be represented as O(n2), O(n3), O(n4), O(n5), ……

2n+3 ≤ cn2

c=3 and n0= 2

2n+3=O(n2)

2n+3 ≤ cn3

c=3 and n0= 2

2n+3=O(n3)

But, out of all possibilities, the least upper bound should be used. So, the correct representation for **2n+3** is **O(n)**.

Ex: Represent the complexity **100n+6** using Big Oh notation

f(n)=100n+6

g(n)=n

f(n) ≤ c\*g(n)

100n+6 ≤ c\*n

c should be minimum 101 for the above condition to be satisfied (c=101).

100n+6 ≤ 101n

The above condition is satisfied for all values of n starting from 6 (n0 = 6).

So, f(n)=O(g(n))

**100n+6=O(n)**

Ex: Represent the complexity **10n2+4n+6** using Big Oh notation

f(n)=10n2+4n+6

g(n)=n2

f(n) ≤ c\*g(n)

10n2+4n+6 ≤ c\*n2

c should be minimum 11 for the above condition to be satisfied (c=11).

10n2+4n+6 ≤ 11n2

The above condition is satisfied for all values of n starting from 6 (n0 = 6).

So, f(n)=O(g(n))

**10n2+4n+6 = O(n2)**

Ex: Represent the complexity **6\*2n+n2** using Big Oh notation

f(n)=6\*2n+n2

g(n)=2n

f(n) ≤ c\*g(n)

6\*2n+n2 ≤ c\*2n

c should be minimum 7 for the above condition to be satisfied (c=7).

6\*2n+n2 ≤ 7\*2n

The above condition is satisfied for all values of n starting from 4 (n0 = 4).

So, f(n)=O(g(n))

**6\*2n+n2 = O(2n)**

**Omega (Ω) Notation**

If **f(n)** and **g(n)** are two functions defined in terms of **n**, then **f(n)=Ω(g(n))** if and only if there exist two positive constants **c** and **n0** such that **f(n) ≥ c\*g(n)**, for all **n ≥ n0**

Ex: Represent the complexity **2n+3** using Omega notation

f(n)=2n+3

g(n)=n

f(n) ≥ c\*g(n)

2n+3 ≥ c\*n

for the above condition to be satisfied, the maximum allowed value for c is 2 (c=2).

2n+3 ≥ 2n

The above condition is satisfied for all values of n starting from 1 (n0 = 1).

So, f(n)=Ω(g(n))

**2n+3= Ω(n**)

2n+3 can also be represented as Ω(1)

2n+3 ≥ c\*1

c=2 and n0= 1

2n+3= Ω(1)

But, out of all possibilities, the highest lower bound should be used. So, the correct representation for **2n+3** is **Ω(n)**.

Ex: Represent the complexity **100n+6** using Omega notation

f(n)=100n+6

g(n)=n

f(n) ≥ c\*g(n)

100n+6 ≥ c\*n

for the above condition to be satisfied, the maximum allowed value for c is 100 (c=100).

100n+6 ≥ 100n

The above condition is satisfied for all values of n starting from 1 (n0 = 1).

So, f(n)=Ω(g(n))

**100n+6= Ω(n)**

Ex: Represent the complexity **10n2+4n+6** using Omega notation

f(n)=10n2+4n+6

g(n)=n2

f(n) ≥ c\*g(n)

10n2+4n+6 ≥ c\*n2

for the above condition to be satisfied, the maximum allowed value for c is 10 (c=10).

10n2+4n+6 ≥ 10n2

The above condition is satisfied for all values of n starting from 1 (n0 = 1).

So, f(n)=Ω(g(n))

**10n2+4n+6=Ω(n2)**

Ex: Represent the complexity 6\*2n+n2 using Omega notation

f(n)=6\*2n+n2

g(n)=2n

f(n) ≥ c\*g(n)

6\*2n+n2 ≥ c\*2n

for the above condition to be satisfied, the maximum allowed value for c is 6 (c=6).

6\*2n+n2 ≥ 6\*2n

The above condition is satisfied for all values of n starting from 1 (n0 = 1).

So, f(n)=Ω(g(n))

**6\*2n+n2 = Ω(2n)**

**Theta (θ) Notation**

If f(n) and g(n) are two functions defined in terms of n, then f(n)=θ(g(n)) if and only if there exist three positive constants c1, c2 and n0 such that c1\*g(n) ≤ f(n) ≤ c2\*g(n), for all n ≥ n0

Ex: Represent the complexity **2n+3** using Theta notation

f(n)=2n+3

g(n)=n

c1\*g(n) ≤ f(n) ≤ c2\*g(n)

c1\*n ≤ 2n+3 ≤ c2\*n

for the above condition to be satisfied, the maximum allowed value for c1 is 2 (c1=2) and c2 value should be minimum 3 (c2=3).

2n ≤ 2n+3 ≤ 3n

The above condition is satisfied for all values of n starting from 3 (n0 = 3).

So, f(n)=θ(g(n))

**2n+3= θ(n)**

Ex: Represent the complexity **100n+6** using Theta notation

f(n)=100n+6

g(n)=n

c1\*g(n) ≤ f(n) ≤ c2\*g(n)

c1\*n ≤ 100n+6 ≤ c2\*n

for the above condition to be satisfied, the maximum allowed value for c1 is 100 (c1=100) and c2 value should be minimum 101 (c2=101).

100n ≤ 100n+6 ≤ 101n

The above condition is satisfied for all values of n starting from 6 (n0 = 6).

So, f(n)=θ(g(n))

**100n+6 = θ(n**)

Ex: Represent the complexity **10n2+4n+6** using Theta notation

f(n)=10n2+4n+6

g(n)=n2

c1\*g(n) ≤ f(n) ≤ c2\*g(n)

c1\*n2 ≤ 10n2+4n+6 ≤ c2\*n2

for the above condition to be satisfied, the maximum allowed value for c1 is 10 (c1=10) and c2 value should be minimum 11 (c2=11).

10n2 ≤ 10n2+4n+6 ≤ 11n2

The above condition is satisfied for all values of n starting from 6 (n0 = 6).

So, f(n)= θ(g(n))

**10n2+4n+6=** **θ(n2)**

Ex: Represent the complexity **6\*2n+n2** using Theta notation

f(n)=6\*2n+n2

g(n)=2n

c1\*g(n) ≤ f(n) ≤ c2\*g(n)

c1\*2n ≤ 6\*2n+n2 ≤ c2\*2n

for the above condition to be satisfied, the maximum allowed value for c1 is 6 (c1=6) and c2 value should be minimum 7 (c2=7).

6\*2n ≤ 6\*2n+n2 ≤ 7\*2n

The above condition is satisfied for all values of n starting from 4 (n0 = 4).

So, f(n)=θ(g(n))

**6\*2n+n2 = θ(2n)**

There is only one possibility to represent any complexity with Theta notation. For example, the complexity **2n+3** can be represented using Theta notation as 2n+3=θ(n) only. It is not possible to represent 2n+3 as θ(n2) or θ(n3).

Suppose, if we want to represent 2n+3 as θ(n2) then we need to identify c1, c2 and n0 which collectively satisfies the condition

c1\*n2 ≤ 2n+3 ≤ c2\*n2

It is not possible to identify c1, c2 and n0 which collectively satisfie the above condition. So, it is not possible to represent 2n+3 as θ(n2).

Theta notation is used to represent the complexity accurately.

**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

Ex: Represent the complexity **2n+3** using small oh notation

f(n)=2n+3

g(n)=n2

So, f(n)=o(g(n))

2n+3=o(n2)

**Small omega notation (ω)**

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

Ex: Represent the complexity **2n+3** using small omega notation

f(n)=2n+3

g(n)=1

So, f(n)= ω(g(n))

2n+3=ω(1)

Ex: Represent the complexity 10n2+4n+6using small omega notation

f(n)=10n2+4n+6

g(n)=n

So, f(n)= ω(g(n))

10n2+4n+6=ω(n)

Ex: Show that **2n3+4n2+10=θ(n3)**

f(n)=2n3+4n2+10

g(n)=n3

c1\*g(n) ≤ f(n) ≤ c2\*g(n)

c1\*n3 ≤ 2n3+4n2+10 ≤ c2\*n3

c1=2, c2=3

2n3 ≤ 2n3+4n2+10 ≤ 3n3

n0=5

f(n)=θ(g(n))

2n3+4n2+10=θ(n3)

Ex: Show that **3n ≠ O(2n)**

f(n)=3n

g(n)=2n

f(n) ≤ c\*g(n)

3n ≤ c2n

it is not possible to identify c and n0 which satisfies the condition

3n ≤ c2n

So, 3n ≠ O(2n)

**DIVIDE AND CONQUER METHOD**

It is one of the methods for generating or developing algorithms. The following procedure is used to get solution or to develop an algorithm for any problem. If p is the problem to be solved then check whether p can be directly solvable. If the problem p can be solved directly then solve the problem and get the solution. Otherwise, divide the problem p into number of sub problems (p1, p2, p3,…). If the sub problems can be directly solvable then solve the sub problems and combine the solutions to get solution for entire problem. Otherwise, continue the division process until the directly solvable sub problems are obtained. Combining the solutions of sub problems is necessary for only some of the problems.

p

p1 p2 ………. pn

p11 p12 …….. p1m

**Control Abstraction of Divide and Conquer Method**

Control abstraction of Divide and Conquer method indicates the general procedure used to solve any problem or to develop an algorithm for any problem using Divide and Conquer method. Control abstraction of divide and conquer method is

Algorithm D&C(p)

// p is the problem to be solved

{

if **small**(p) then

**solve**(p)

else

{

Divide p into number of sub problems p1, p2,…..pn

D&C(p1);

D&C(p2);

.

D&C(pn);

Combine solutions of sub problems

}

}

**Small(**) is a user defined function which returns true if the problem is small enough to solve directly, otherwise it returns false.

**Solve()** is a user defined function which gets solution of the problem.

**Applications of Divide and Conquer Method**

1. Binary Search
2. Quick Sort
3. Finding Minimum and Maximum

For each problem, we discuss

1. Problem Description
2. Example
3. Algorithm
4. Calculation of Space and Time complexity

**Binary Search**

It is one of the techniques used to check the presence of a value in a list of values. The values in the list should be in sorted order. The procedure used in this method is as follows: the value to be searched is compared with the middle value in the list. If they match then the process is terminated and announced that the value is identified. Otherwise, the list is divided into two equal size parts based on middle position and then the value is searched in first part if the value to be searched is less than the middle value of list or in second part if the value to be searched is greater than the middle value of list. This process is repeated until the value is found or the list shrinks to a single value.

Input 1: List of values in sorted order

Input 2: Value to be searched

Two cases:

1. Successful search (value is present in the list)
2. Unsuccessful search (value is not present in the list)

Ex:

List of values: 10 20 30 40 50 60 70 80

a : 1 2 3 4 5 6 7 8

K is the value to be searched

1) K=60 (Successful search)

s=1, e=8

m=(s+e)/2=9/2=4

a[m]=a[4]=40

K>a[m]

search in the second part of list

s=m+1=4+1=5, e=8

m=(s+e)/2=(5+8)/2=6

a[m]=a[6]=60

K=a[m]

Value is identified. Searching process is terminated.

2) K=15 (Unsuccessful search)

s=1, e=8

m=(s+e)/2=9/2=4

a[m]=a[4]=40

K<a[m]

search in the first part of list

s=1, e=m-1=4-1=3

m=(s+e)/2=(1+3)/2=2

a[m]=a[2]=20

K<a[m]

search in the first part of list

s=1, e=m-1=2-1=1

m=(s+e)/2=(1+1)/2=1

a[m]=a[1]=10

K>a[m]

search in the second part of list

It is not possible to divide the list into two parts as the list contains only one value. So, the searching process is terminated.

**Algorithm**

Algorithm BinarySearch(a, n, k)

//’a’ is an array of size ‘n’ and ‘k’ is the value to be searched

{

s := 1;

e := n;

while s ≤ e do

{

m := (s+e)/2;

if k = a[m] then

{

write “value is found”;

return;

}

else

{

if k < a[m] then

e := m-1;

else

s := m+1;

}

}

write “value is not found”;

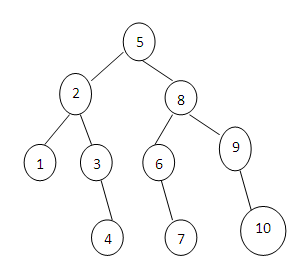
}

**Time complexity**

To calculate time complexity, decision tree is constructed. The decision tree is constructed based on mid position of list as well as sub lists that will be generated in the searching process.

**Ex**:-

If the number of elements in the list is n=10 then the decision tree is



The decision tree indicates the number of comparisons required to identify values at different positions of the list. For example, one comparison is required to identify the value at 5th position of the list. Two comparisons are required to identify the value at 2nd position. Three comparisons are required to identify the value at6th position.

Maximum number of comparisons required to identify a value is 4 (height of decision tree +1)

Height of decision tree is log2n. Where ‘n’ is number of positions in the list.

Time complexity of binary search is calculated by considering 2 cases

**Case 1:** Successful Search (value is present in the list)

We consider 3 cases here

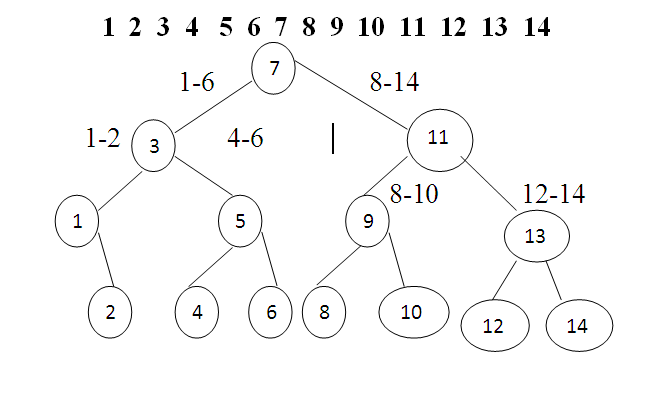
1. **Best Case**: Best case is encountered when the value is present at middle position of the list. One comparison is enough to identify the value at middle position. So, time complexity is **O(1).**
2. **Worst Case**: Worst case is encountered when the value is present at any position of the list. Maximum number of comparisons required to search the value at any position of list=height of decision tree+1=log2n+1. Time complexity is **O(log2n).**
3. **Average Case:** Average case is encountered when the value is present at any position of the list. Maximum number of comparisons required to search the value at any position of list=height of decision tree+1=log2n+1. Time complexity is **O(log2n).**

**Case 2:** Unsuccessful Search (value is not present in the list)

Maximum number of comparisons required to search a value that is not present in the list =height of decision tree+1=log2n+1

Time complexity is **O(log2n)**.

**Ex:** Construct decision tree for 14 positions.



**Recursive algorithm for Binary Search**

Algorithm RBinarySearch(a, s, e, k)

// ‘a’ is an array containing the list of values

// ‘s’ is starting position and ‘e’ is ending position of the array ‘a’

// ’k’ is the value to be searched

{

if s ≤ e then

{

m:=(s+e)/2;

if k=a[m] then

{

write ‘value is found’;

return;

}

else

{

if k < a[m] then

e := m-1;

else

s := m+1;

}

RBinarySearch(a, s, e, k);

}

write ‘value is not present’;

}

**Time Complexity of Recursive Binary Search Algorithm**

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

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

T(n)=T(n/8)+c+2c=T(n/8)+3c

T(n)=T(n/16)+c+3c=T(n/16)+4c

.

.

.

After k times

T(n)=T(n/2k)+kc

Assume 2k=n

Taking log2 on both sides

log2 2k=log2 n

k\*log22= log2 n

k= log2 n

T(n)=T(n/n)+c\* log2 n=T(1)+c\* log2 n

T(n)=c\* log2 n

T(n)=O(log2 n)

**Quick Sort**

Quick sort is a technique to sort a list of values. The procedure for sorting a list of values is as follows:

Check whether the list is containing more than one value or not. If the list contains more than one value then select a value (generally the first value) in the list as pivot value. Set a pointer (i) to the starting position of list. Set a pointer (j) to the ending position of list. Move the pointer ‘i’ in forward direction until the value at ith position is less than or equal to the pivot value. Move the pointer ‘j’ in backward direction until the value at jth position is greater than the pivot value. If the position of ‘i’ is less than the position of ‘j’ then swap the values at ‘i’ and ‘j’ positions and then move the ‘i’ and ‘j’ pointers again. Otherwise, swap the pivot value with the value at jth position and divide the list into two parts based on position ‘j’. The first part includes the values from the starting of list to (j-1)th position and the second part includes the values from (j+1)th position to ending of list. Now, sort the first and second parts separately using same procedure.

Ex: sort the following list of values using quick sort technique

p

40 60 10 80 50 70 30 20

a: 1 2 3 4 5 6 7 8

i j

p

40 60 10 80 50 70 30 20

a: 1 2 3 4 5 6 7 8

i j

p

40 60 10 80 50 70 30 20

a: 1 2 3 4 5 6 7 8

i j

p

40 20 10 80 50 70 30 60

a: 1 2 3 4 5 6 7 8

i j

p

40 20 10 80 50 70 30 60

a: 1 2 3 4 5 6 7 8

i j

p

40 20 10 80 50 70 30 60

a: 1 2 3 4 5 6 7 8

i j

p

40 20 10 80 50 70 30 60

a: 1 2 3 4 5 6 7 8

i j

p

40 20 10 30 50 70 80 60

a: 1 2 3 4 5 6 7 8

i j

p

40 20 10 30 50 70 80 60

a: 1 2 3 4 5 6 7 8

i j

p

40 20 10 30 50 70 80 60

a: 1 2 3 4 5 6 7 8

i j

p

40 20 10 30 50 70 80 60

a: 1 2 3 4 5 6 7 8

ij

p

40 20 10 30 50 70 80 60

a: 1 2 3 4 5 6 7 8

j i

p

30 20 10 40 50 70 80 60

a: 1 2 3 4 5 6 7 8

j

1st part:

30 20 10

a: 1 2 3

2nd part:

50 70 80 60

a: 5 6 7 8

**Algorithm**

Algorithm Quicksort(a, s, e)

// a is an array containing list of values to be sorted

// s is starting position and e is ending position of list

{

if s < e then

{

j := Partition(a, s, e);

Quicksort(a, s, j-1);

Quicksort(a, j+1, e);

}

}

Algorithm Partition(a, s, e)

{

p := a[s];

i := s;

j := e;

while i < j do

{

while a[i] ≤ p do

i := i+1;

while a[j] > p do

j := j-1;

if i < j then

{

t := a[i];

a[i] := a[j];

a[j]: = t;

}

}

t := a[s];

a[s] := a[j];

a[j] := t;

return j;

}

**Time Complexity**

**Time Complexity of Partition Algorithm:**

Partition algorithm moves the pointer ‘i’ towards right for a maximum of ‘n’ positions and the pointer ‘j’ towards left for a maximum of ‘n’ positions. So, time complexity of partition algorithm is n+n+c=2n+c=cn.

Where ‘c’ is some constant.

**Time Complexity of Quick Sort Algorithm:**

If the list is containing ‘n’ number of values then the time complexity for sorting the ‘n’ values is

T(n)=Time for partition + Time for sorting first part + Time for sorting second part

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

Assuming that the number of values in first part is ‘i’.

Time complexity is measured by considering 3 cases

**Worst Case**

Worst case is encountered when we try to sort the values which are already in sorting order.

Ex: 10 20 30 40 50 60

p i j

10 20 30 40 50 60

p j i

When the list is divided into two parts, the first part is empty, second part includes (n-1) values.

T(n)=cn+T(0)+T(n-1-0)

T(n)=cn+0+T(n-1)

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

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

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

.

.

After n times

T(n)=T(n-n)+ncn-nc

T(n)=0+cn2-cn

T(n)=cn2-cn

T(n)=**O(n2)**

**Best Case**

Best case is encountered when the list is divided into 2 equal size parts each time. So, Time complexity is

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

T(n)=2T(n/2)+cn=21T(n/21)+1cn

T(n)=2[2T(n/4)+cn/2]+cn=4T(n/4)+cn+cn=4T(n/4)+2cn=22T(n/22)+2cn

T(n)=4[2T(n/8)+cn/4]+2cn=8T(n/8)+cn+2cn=8T(n/8)+3cn= 23T(n/23)+3cn

T(n)=8[2T(n/16)+cn/8]+3cn=16T(n/16)+cn+3cn=16T(n/16)+4cn= 24T(n/24)+4cn

.

.

.

After k times

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

Assume that 2k=n

Taking log2 on both sides

log2 2k= log2 n

k\*log22= log2 n

k= log2 n

T(n)=nT(n/n)+cn\*log2 n=nT(1)+ cn\*log2 n=n\*1+ cn\*log2 n=n+ cn\*log2 n

**T(n)=O(nlog2 n)**

**Average Case**

Average case is encountered when the list is divided into two unequal size parts. The average case time complexity is calculated by considering all possible sizes of first part and second part.

Ex: if the list is containing 8 values then the possible sizes of first and second parts are

**n=8**

**x** **n-1-x**

0 7

1 6

2 5

3 4

4 3

5 2

6 1

7 0

Time complexity is

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

T(n)=cn+2/n[T(0)+T(1)+……..+T(n-1)]

After solving the above equation, the time complexity is

T(n)=**O(nlog2n)**

**Randomized Quick Sort**

Randomized quick sort is a variant of quick sort. In Quick sort, generally first value of the list is selected as pivot element. In randomized quick sort, any value of the list can be selected as pivot element. While sorting different parts of list, values at different positions of the lists can be selected as pivot elements. For example, for sorting first part of the list the value at third position can be selected, for sorting second part of the list the value at second position can be selected.

**Algorithm**

Algorithm RandomQuickSort(a, s, e)

// a is an array containing the list of values to be sorted

// s is starting position of array and e is ending position of array

{

if s < e then

{

j := Partition(a, s, e);

RandomQuickSort(a, s, j-1);

RandomQuickSort(a, j+1, e);

}

}

Algorithm Partition(a, s, e)

{

k := Random(s, e);

p := a[k];

i := s;

j := e;

while i < j do

{

while a[i] ≤ p do

i := i+1;

while a[j] > p do

j := j-1;

if i < j then

{

t := a[i];

a[i] := a[j];

a[j]: = t;

}

}

t := a[s];

a[s] := a[j];

a[j] := t;

return j;

}

Random(s, e) is a user defined function which returns a value between s and e.

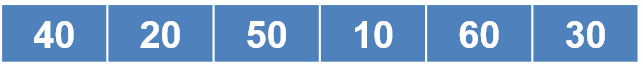
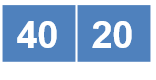
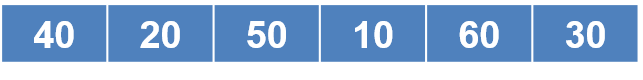
**Finding Minimum and Maximum**

Input: List of values

Output: minimum value and maximum value in the list

The procedure for identifying minimum and maximum value in a list is as follows:

Divide the list into two equal parts based on middle position. First part includes the values from starting position to middle position. Second part includes the values from middle+1 position to ending position. Divide each part into two equal parts again. Continue this division process until we get parts with either one value or two values. Now, combine the parts in reverse direction. When the sub list contains one value, then the minimum and maximum is that value only. When the sub list contains two values, then the minimum and maximum are identified by comparing the two values. While combining the two sub lists, identify the minimum and maximum of the combined list by comparing the minimum and maximum values of the two sub lists.



1 2 3 4 5 6

1 2 3

4 5 6

1 2

6

4 5

3

min=30 max=30

min=20 max=40

min=50 max=50

min=10 max=60

min=10 max=60

min=20 max=50

min=10 max=60

**Algorithm**

Algorithm MinMax(a, s, e, min, max)

// ‘a’ is an array containing the list of values

// ‘s’ and ‘e’ are starting and ending positions of the array

//’min’ is used to store minimum value in the list

//’max’ is used to store maximum value in the list

{

if s=e then //list contains one value

{

min := a[s];

max := a[s];

}

else

{

if s=e-1 then //list contains two values

{

if a[s] < a[e] then

{

min := a[s];

max := a[e];

}

else

{

min := a[e];

max := a[s];

}

}

else //list contains more than two values

{

m := (s+e)/2;

MinMax(a, s, m, min1, max1);

MinMax(a, m+1, e, min2, max2);

if min1 < min2 then

min := min1;

else

min := min2;

if max1 > max2 then

max := max1;

else

max := max2;

}

}

}

**Time Complexity**

When n=1

T(n)=T(1)=1+1+1=3

When n=2

T(n)=T(2)=1+1+1+1+1=5

When n>2

T(n)=1+1+1+T(n/2)+T(n/2)+1+1+1+1=7+2T(n/2)

**T(n)=7+2T(n/2)**=20\*7+21T(n/21)

T(n)=20\*7+2[7+2T(n/4)]=20\*7+21\*7+22 T(n/22)

T(n)= 20\*7+21\*7+22 [7+2T(n/8)]=20\*7+21\*7+22 \*7+23T(n/23)

.

.

After k times

T(n)= 20\*7+21\*7+22\*7+23\*7+…….+2k-1\*7+2kT(n/2k)

Assume 2k=n

Taking log2 on both sides

log2 2k= log2 n

k\*log22= log2 n

k=log2 n

T(n)= 20\*7+21\*7+22\*7+23\*7+…….+2k-1\*7+2kT(n/2k)

T(n)= (20+21+22+23+…….+2k-1)\*7+2kT(n/2k)

T(n)= (20+21+22+23+…….+2k-1)\*7+nT(n/n)

T(n)=(2k-1)\*7+nT(1)

T(n)=(n-1)\*7+n\*3

T(n)=10n-7

**T(n)=O(n)**