**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

**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: <, ≤, >, ≥, =, ≠ 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

{

}

else

{

}

Block of statements

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”;

# 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);

# 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 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

# 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)

//

{

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=2m2n+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+……+T(1)

=2+2+2+2+…… 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+……+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(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+4+0+0+0+4(2n-1) = c+4+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

# Frequency Count

T(n)=5n+1

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 used to represent space and time complexity of algorithms. Commonly used asymptotic notations are

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

# Big oh 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 exists two positive constants c and no such that f(n) ≤ c\*g(n), for all values of n where n ≥ n0.

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 n0=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 n0=6

So, 100n+6=O(n)

Ex3: if the complexity of an algorithm is 10n2+4n+6 then 10n2+4n+6=O(n2) f(n)= 10n2+4n+6

g(n)=n2

10n2+4n+6 ≤ 11n2, n≥6 c=11 and n0=6

So, 10n2+4n+6=O(n2)

Ex4: if the complexity of an algorithm is 6\*2n + n2 then 6\*2n + n2=O(2n) f(n)=6\*2n + n2

g(n)=2n

6\*2n + n2≤7\*2n, n≥1 c=7 and n0=1

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

Actually, 3n+2 can be represented as 3n+2=O(n) because 3n+2≤4n where c=4 and n0=2

or as 3n+2=O(n2)

because 3n+2≤4n2 where c=4 and n0=2 or as 3n+2=O(n3)

because 3n+2≤4n3 where c=4 and n0=2

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

# Omega 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 exists two positive constants c and no such that f(n) ≥ c\*g(n), for all values of n where n ≥ n0.

Ex1: if the complexity of an algorithm is 3n+2 then 3n+2=Ω(n) f(n)=3n+2

g(n)=n

3n +2 ≥ 3n, n≥1

c=3 and n0=1 So, 3n+2=Ω(n)

Ex2: if the complexity of an algorithm is 100n+6 then 100n+6=Ω(n) f(n)=100n+6

g(n)=n

100n+6 ≥ 100n, n≥1 c=100 and n0=1

So, 100n+6=Ω(n)

Ex3: if the complexity of an algorithm is 10n2+4n+6 then 10n2+4n+6=Ω(n2) f(n)= 10n2+4n+6

g(n)=n2

10n2+4n+6 ≥ 10n2, n≥1 c=10 and n0=1

So, 10n2+4n+6=Ω(n2)

Ex4: if the complexity of an algorithm is 6\*2n + n2 then 6\*2n + n2=Ω(2n) f(n)=6\*2n + n2

g(n)=2n

6\*2n + n2 ≥ 6\*2n, n≥1 c=6 and n0=1

So, 6\*2n + n2=Ω(2n)

Actually, 3n+2 can be represented as 3n+2=Ω(n) because 3n+2≥3n where c=3 and n0=1

or as 3n+2=Ω(1)

because 3n+2≥3 where c=3 and n0=1

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

# Theta 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 exists three positive constants c1, c2 and no such that c1\*g(n) ≤ f(n) ≤ c2\*g(n), for all values of n where n ≥ n0.

Ex1: if the complexity of an algorithm is 3n+2 then 3n+2=θ(n) f(n)=3n+2

g(n)=n

3n ≤ 3n +2 ≤ 4n, n≥2

c1=3, c2=4 and n0=2 So, 3n+2=θ(n)

Ex2: if the complexity of an algorithm is 100n+6 then 100n+6=θ(n) f(n)=100n+6

g(n)=n

100n ≤ 100n+6 ≤ 101n, n≥6

c1=100, c2=101 and n0=6 So, 100n+6=θ(n)

Ex3: if the complexity of an algorithm is 10n2+4n+6 then 10n2+4n+6=θ(n2) f(n)= 10n2+4n+6

g(n)=n2

10n2 ≤ 10n2+4n+6 ≤ 11n2, n≥6 c1=10, c2=11 and n0=6

So, 10n2+4n+6=θ(n2)

Ex4: if the complexity of an algorithm is 6\*2n + n2 then 6\*2n + n2=θ(2n) f(n)=6\*2n + n2

g(n)=2n

6\*2n ≤ 6\*2n + n2 ≤ 7\*2n, n≥1 c1=6, c2=7 and n0=1

So, 6\*2n + n2=θ(2n)

# Small Oh 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

lim 𝑓(𝑛) = 0

𝑛→∞

𝑔(𝑛)

Ex1: if the complexity of an algorithm is 3n+2 then 3n+2=o(n2) as

lim𝑛→∞ 3𝑛+2 = 0

𝑛2

Ex2: if the complexity of an algorithm is 10n2+4n+6 then 10n2+4n+6=o(n3) as

lim𝑛→∞ 10𝑛2+4𝑛+6 = 0

𝑛3

# Small Omega 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

lim 𝑔(𝑛) = 0

𝑛→∞

𝑓(𝑛)

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

lim 1 = 0

𝑛→∞

3𝑛+2

Ex2: if the complexity of an algorithm is 10n2+4n+6 then 10n2+4n+6= ω(n) as

lim 𝑛 = 0

𝑛→∞

10𝑛2+4𝑛+6

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 3n3+2n2=O(n3) f(n)=3n3+2n2

g(n)=n3

3n3+2n2 ≤ 4n3, n≥2

c=4, n0=2 3n3+2n2=O(n3)

Ex2: Show that 3n ≠ O(2n)

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

It is not possible to identify c and n0 such that 3n ≤ c2n is satisfied. So, 3n ≠ O(2n) Ex3: Show that 3n3+2n2=Ω(n3)

f(n)=3n3+2n2 g(n)=n3

3n3+2n2 ≥ 3n3, n≥1 c=3, n0=1

3n3+2n2=Ω(n3)

Ex4: Show that 3n3+2n2=θ(n3)

f(n)=3n3+2n2 g(n)=n3

3n3 ≤ 3n3+2n2 ≤ 4n3, n≥2 c1=3, c2=4, n0=2

3n3+2n2=θ(n3)

# Amortized analysis (or) amortized complexity

Charging the execution time of an operation over other operations (which appears before or after the operation) in the sequence of operations is called amortization.

The execution time of an operation can be overcharged on other operations in the sequence of operations in forward or reverse direction. One or more units of execution time of an operation can be overcharged on other operations. The execution time of an operation can be overcharged on either one or more operations in the sequence of operations.

After performing amortization, the execution time of an operation is called as amortized time. The amortized time of an operation is equal to or less than or more than its actual execution time.

Ex: Consider the following binary search tree

60

30

90

20

50

80

100

10

40

70

Consider the sequence of operations for searching the values 20, 70, 90, 60, 10, 30, 40, 100. The actual execution times for these operations are

Actual execution times: 3, 4, 2, 1, 4, 2, 4, 3

If one unit of execution time of second operation is overcharged on third operation, one unit of execution time of second operation is overcharged on fourth operation and one unit of execution time of seventh operation is overcharged on sixth operation then the execution times of operations becomes

Amortized execution times: 3, 2, 3, 2, 4, 3, 3, 3

Amortized analysis is used to calculate average execution time of an operation in a sequence of operations using any of the following three methods

1. Aggregate Method
2. Accounting Method
3. Potential Method

## Aggregate method

The average execution time of an operation in a sequence of operations is calculated using the following formula

Average execution time of an operation=Sum of execution times of all operations in the sequence/Number of operations in the sequence

In above example, the average execution time of an operation is Average execution time = (3+4+2+1+4+2+4+3)/8=2.87

## Accounting method

The average execution times of operations are calculated by guessing their values where the guessed values satisfies the following condition

p(n)≥0 where,

p(i) = [(averageexecutiontime(i) – actualexecutiontime(i)] + p(i-1) and p(0)=0

„i‟ is ith operation in the sequence of operations, averageexecutiontime(i) is average execution time of ith operation, actualexectiontime(i) is actual execution time of ith operation and p is potential function.

Ex: In above example, if we guess the average execution time of all operations in the sequence as 2 then

p(1) = 2-3+0= -1

p(2) = 2-4-1= -3

p(3) = 2-2-3= -3

p(4) = 2-1-3= -2

p(5) = 2-4-2= -4

p(6) = 2-2-4= -4

p(7) = 2-4-4= -6

p(8) = 2-3-6= -7

p(n)=p(8)<0

So, 2 is not good guess as average execution time.

If we guess the average execution time of all operations in the sequence as 3 then p(1) = 3-3+0= 0

p(2) = 3-4+0= -1

p(3) = 3-2-1= 0

p(4) = 3-1+0= 2

p(5) = 3-4+2= 1

p(6) = 3-2+1= 2

p(7) = 3-4+2= 1

p(8) = 3-3+1= 1 p(n)=p(8)>0

So, 3 is a correct guess for average execution time of all operations in the sequence.

## Potential method

The average execution times of operations are calculated by guessing their values where the guessed values satisfies the condition

*n*

p≥0 where p= [*averageexecutiontime*(*i*)  *actualexecutiontime*(*i*)]

*i*1

Ex: In above example, if we guess the average execution time of all operations in the sequence as 2 then

p=(2-3)+(2-4)+(2-2)+(2-1)+(2-4)+(2-2)+(2-4)+(2-3)=-7

So, 2 is not good guess as average execution time as p<0.

If we guess the average execution time of all operations in the sequence as 3 then p=(3-3)+(3-4)+(3-2)+(3-1)+(3-4)+(3-2)+(3-4)+(3-3)=1

So, 3 is a correct guess for average execution time of all operations in the sequence.