UNIT 1

1. Introduction and Overview of the Course

2. Pseudo code for expressing algorithms

3. Space complexity, Time complexity

4. Time complexity Examples

5. Introduction to Asymptotic Notation

6. Asymptotic Notation Types with examples

7. Probabilistic analysis, Amortized analysis

**Algorithm:**

An algorithm is a finite set of instructions that is followed to accomplish a particular task.

**Properties of algorithm:**

1. Input

2. Output

3. Definiteness

4. Finiteness

5. Efficiency

Input:

The input of zero or more number of quantities should be given to the algorithm.

Output:

Atleast one quantity is produced as an output.

Definiteness:

Each instruction in algorithm should be specific and unambiguous.

Finiteness:

The algorithm should be finite means after finite number of steps it should be terminated.

Efficiency:

Every step of algorithm should be feasible. In other words every step of algorithm should be carried out by pen and paper.

**Pseudo code for expressing an algorithm:**

An algorithm is basically a sequence of instruction written in English language and broadly divided into 2 sections.

1. Algorithm heading:

It consists of

1. Name of algorithm
2. Problem description
3. Input
4. Output
5. Algorithm body: It consists of logical body of algorithm.

**Rules for writing an algorithm:**

Step.1:

Algorithm is a procedure consisting of heading and body. The heading section consists of keyword algorithm, name of the algorithm and parameters list.

Syntax: Algorithm max(A[] , n )

Step.2:

In the heading section we should write the following things

Problem description

Input

Output

Step.3:

Comments begin with forward slash and continue until the end of the line.

Step.4:

Compound statement should be enclosed with in {and } .

Step.5:

An identifier begin with a letter and an identifier can be a combination of alpha numeric string.

Step.6:

Assigning values to variables is done by using assignment statement.

< Variable >:= < expression >

Step.7:

There are 2 boolean values “True” and “False”. In order to produce these values we use logical operators such as and, or, not and relational operators such as <, >, <=, >=, ==,!= .

Step.8:

Elements of multi-dimensional arrays are accessed using [and] .

Example:

If ‘a’ is a 2-dimensional array ( i , j )th element of an array is denoted by

a [ i , j ] . Array indexes start at Zero.

Step.9:

Looping statements are also employed.

Example:

While loop

while< condition > loop

{

< statement 1 >

< statement 2 >

.

.

.

.

< statement n >

}

General form of for loop

for variable := value 1 to n do

{

< statement 1>

( step i := i + 1 )

< statement 2 >

.

.

.

.

< statement n >

}

Here, value 1 is initialisation condition. Value n is terminating condition. Sometimes, a keyword ‘step’ used to denote increment or decrement the value.

A repeat until statement is constructed as

< statement 1 >

< statement 2 >

.

.

.

.

< statement n >

until < condition >

These statements are executed as long as the condition is true.

Step.10:

Input and output are done using the instruction read and write. No format is used to represent the size of input and output quantities.

Examples:-

1. Write an algorithm to find sum of n numbers.

Sol:-

Algorithm sum (n)

//problem description: Sum of given ‘n’ numbers.

//Input: 1 to n numbers.

//Output: Sum of ‘n’ numbers.

{

sum := 0;

for i := 1 to n do

sum := sum +i;

write (sum);

}

1. Write an algorithm to check whether given number is odd or even.

Sol:-

Algorithm check (n)

//problem description: To find given no is even or odd.

//Input: One integer value.

//Output: Given no is even or odd.

{

read (n);

if (n%2 == 0)

write (“n is even”);

else

write(“n is odd”);

}

1. Algorithm for finding maximum of given n numbers.

Sol:-

Algorithm Max (a [ ], n)

//problem description: To find maximum numbers in an array.

//Input: Array elements.

//Output: Maximum value.

{

read (n);

read (a [n]);

max := a[0];

for i:= 1 to n-1 do

{

if (a[i] > max) then

max := a[i];

}

write (max);

}

**Performance Analysis:**

The efficiency of an algorithm can be decided by measuring the performance of an algorithm.

We can measure the performance by computing two factors:

1. Space complexity.

2. Time complexity.

Performance evaluation can be divided into two major phases:

1. Prior estimates.

2. Posterior testing.

We refer to this as performance analysis and performance measurement.

1. Space complexity:

The space complexity can be defined as amount of memory required by an algorithm to run.

To compute this space complexity we use 2 factors:

1. Constant characteristics (fixed part)

2. Instance characteristics (fixed variable part)

Space requirement s(p) can be given as

s(p)= c + Sp

where c is a constant i.e., fixed part and it denotes the space of inputs and outputs

## This space is an amount of space taken by instruction, variables and identifiers.

Sp is a space depend upon instance characteristics .This is a variable part whose space requirements depends on particular problem instance.

Example: 1

Algorithm abc (a,b,c)

{

return a+b+b\*c+(a+b-c)/(a+b)+4.0;

}

Space complexity s(p)=3

Example: 2

Algorithm sum(a,n)

{

s := 0;

for i := 1 to n do

s := s+a[i];

return s;

}

Example: 3

Algorithm Rsum(a,n)

{

if (n<=0) then return 0;

else

return Rsum(a,n-1)+a[n];

}

Recursion stack space includes space for formal parameters, local variables and return address. Assume the return address requires one word of memory. Each call of recursion requires 3 words (space for value of n, return address and a pointer to an array) .

The depth of recursion is n+1. So, recursion stack space needed is greater than or equal to 3(n+1).

**Time Complexity:**

Time complexity of an algorithm is amount of computer time required by an algorithm to run to completion.

Time T(P) taken by a program P is the sum of the compile time and the run time. The compilation time does not depends upon characteristics instance. The runtime is denoted by

tp(n) = Ca ADD(n)+Cs SUB(n)+Cm MUL(n)+Cd DIV(n)+…………………….

# n denotes instance characteristic.

# Ca, Cs, Cm, Cd,…………………………. denotes time needed for an addition, subtraction, multiplication, division,………………..

# ADD, SUB, MUL, DIV,…………..are function whose values are no of additions, subtractions, multiplication, divisions,……………….. that are performed when code for P is used on an instance with characteristics 'n'.

# For calculating time complexity we build a table in which we list total no of steps contributed by each statement.

# In this we first determine this steps for execution (s/e) and total no of times each step executes (frequency).

# By combining these two we obtain a total contribution of each statement.

# By adding all contributions of all statements this step for entire algorithm is obtained

i.e., Total steps = s/e \* frequency

**Time complexity Examples :-**

1. Sum of an array elements

Algorithm sum(a, n)

{

s := 0;

for i := 1 to n do

s = s + a[i];

return s;

}

|  |  |  |
| --- | --- | --- |
| s/e | frequency | Total steps |
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 1 | n+1 | n+1 |
| 1 | n | N |
| 1 | 1 | 1 |
| 0 | 0 | 0 |

= 2n+3

1. Matrix addition

Algorithm add(a, b, c, m, n)

{

for i := 1 to m do

for j := 1 to n do

C[i, j] = a[i, j] + b[i, j];

}

|  |  |  |
| --- | --- | --- |
| s/e | frequency | Total steps |
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 1 | m+1 | m+1 |
| 1 | m(n+1) | m(n+1) |
| 1 | mn | Mn |
| 0 | 0 | 0 |

= m+1+mn+m+mn

= 2mn+2m+1

1. Recursive sum of array elements

Algorithm Rsum(a, n)

{

if (n <= 0) then

return 0;

else

return Rsum(a, n-1) + a[ ];

}

|  |  |  |
| --- | --- | --- |
| s/e | frequency  n=0 n>0 | Total steps  n=0 n>0 |
| 0 | 0 0 | 0 0 |
| 0 | 0 0 | 0 0 |
| 1 | 1 1 | 1 1 |
| 1 | 1 0 | 1 0 |
| 0 | 0 0 | 0 0 |
| 1+x | 0 1 | 0 1+x |
| 0 | 0 0 | 0 0 |

= 2 = 2+x

where x = trsum(n-1)

**NOTE:-**

The function is calling itself by using (n-1) but we don’t know how many times it calls itself. So we consider it as x.

**ASYMPTOTIC NOTATIONS :**

**#** To choose best algorithm we use space complexity and time complexity.

# Asymptotic notation is another method of finding time complexity of an algorithm.

# Generally an algorithm that is asymptotically more efficient will be best choice.

# There are 3 types .They are

1. Big - Oh (O)

2. Big - Omega (Ω)

3. Big - Theta (Θ)

**Big ‘O’ notation:**

# It is a method of representing the upper bound of algorithm running time.

# let f(n) and g(n) are two non-negative functions used for representing big O notations.

**If f(n)=O(g(n)) if and only if f(n)<=c\*g(n) for all n, n>=n0**

where c, n0 are positive constants. g(n) represents the upper bound of f(n). c>0 and n0>=1.

Diagram :-

No 
C g(n) 
fl n) 
Input - N 

EXAMPLES:

1) 3n+2=O(n)

Here f(n)=3n+2

g(n)=n

f(n)<=c\*g(n)

3n+2<=c\*n

c=4

O(1):

g(n)=1

3n+2<=c\*g(n)

IF n=1

3\*1+2<=4\*1

5<=4

which is false

O(n):

g(n)=n

3n+2<=c\*g(n)

if n=2

3\*2+2<=4\*2

6+2<=8

8<=8

which is true

if n=3

3\*3+2<=4\*3

9+2<=12

11<=12

which is true

O(n^2):

g(n)=n^2

3n+2<=c\*g(n)

if n=2

3\*2+2<=4\*2\*2

6+2<=8\*2

8<=16

which is true

if n=3

3\*3+2<=4\*3\*3

9+2<=12\*3

11<=36

which is true

O(n^3) :

g(n)=n^3

3n+2<=c\*g(n)

if n=2

3\*2+2<=4\*2\*2\*2

6+2<=8\*4

8<=32

which is true

if n=3

3\*3+2<=4\*3\*3\*3

9+2<=12\*9

11<=108

which is true

HENCE O(1) IS NOT POSSIBLE.

O(n),O(n^2),O(n^3)... ARE POSSIBLE.

2) 10n^2+3n+3=O(n^2)

Here f(n)=10n^2+3n+3

g(n)=n^2

f(n)<=c\*g(n)

10n^2+3n+3<=c\*n^2

c=11

O(1):

g(n)=1

10n^2+3n+3<=c\*g(n)

IF n=1

10\*1^2+3\*1+3<=11\*1

10+3+3<=11

16<=11 WHICH IS FALSE

O(n):

g(n)=n

10n^2+3n+3<=c\*g(n)

IF n=2

10\*2^2+3\*2+3<=11\*2

40+6+3<=22

49<=22 WHICH IS FALSE

IF n=3

10\*3^2+3\*3+3<=11\*3

90+9+3<=33

102<=33 WHICH IS FALSE

O(n^2) :

g(n)=n^2

10n^2+3n+3<=c\*g(n)

IF n=3

10\*3^2+3\*3+3<=11\*3\*3

90+9+3<=99

102<=99 WHICH IS FALSE

IF n=4

10\*4^2+3\*4+3<=11\*4\*4

160+12+3<=176

175<=176 WHICH IS TRUE

IF n=5

10\*5^2+3\*5+3<=11\*5\*5

250+15+3<=275

268<=275 WHICH IS TRUE

O(n^3) :

g(n)=n^3

10n^2+3n+3<=c\*g(n)

IF n=4

10\*4^2+3\*4+3<=11\*4\*4\*4

160+12+3<=11\*64

175<=704 WHICH IS TRUE

IF n=5

10\*5^2+3\*5+3<=11\*5\*5\*5

250+15+3<=55\*25

268<=1375 WHICH IS TRUE

HENCE O(1),O(n) IS NOT

POSSIBLE.

O(n^2),O(n^3)..... ARE POSSIBLE.

THE ORDER OF GROWTH IS AS FOLLOWS

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

**OMEGA 'Ω' NOTATION:**

# It is a method of representing the lower bound of algorithm running time.

# let f(n) and g(n) are two non-negative functions used for representing omega notations.

**If f(n)=Ω(g(n)) if and only if f(n)>=c\*g(n) for all n, n>=n0**

where c, n0 are positive constants. g(n) represents the lower bound of f(n). c>0 and n0>=1.

Diagram :

Time 
C g(n) 
Input - N 

**EXAMPLES:**

1) 3n+2=**Ω**(n)

Here f(n)=3n+2

g(n)=n

f(n)>=c\*g(n)

3n+2>=c\*n

c=2

**Ω**(1):

g(n)=1

3n+2>=c\*g(n)

IF n=1

3\*1+2<=2\*1

5>=2 WHICH IS TRUE

**Ω**(n):

g(n)=n

3n+2>=c\*g(n)

IF n=2

3\*2+2>=2\*2

6+2>=4

8>=4 WHICH IS TRUE

IF n=3

3\*3+2>=2\*3

9+2>=6

11>=6 WHICH IS TRUE

**Ω**(n^2):

g(n)=n^2

3n+2>=c\*g(n)

IF n=2

3\*2+2>=2\*2\*2

6+2>=8

8>=8 WHICH IS TRUE

IF n=3

3\*3+2>=2\*3\*3

9+2>=6\*3

11>=18 WHICH IS FALSE

**Ω**(n^3) :

g(n)=n^3

3n+2>=c\*g(n)

IF n=2

3\*2+2>=2\*2\*2\*2

6+2>=4\*4

8>=16 WHICH IS FALSE

IF n=3

3\*3+2>=2\*3\*3\*3

9+2>=6\*9

11>=54 WHICH IS FALSE.

HENCE **Ω**(1)**,Ω**(n) IS POSSIBLE.

**Ω**(n^2),**Ω**(n^3)... ARE NOT POSSIBLE.

2) 10n^2+3n+3=**Ω**(n^2)

Here f(n)=10n^2+3n+3

g(n)=n^2

f(n)>=c\*g(n)

10n^2+3n+3>=c\*n^2

c=9

**Ω**(1):

g(n)=1

10n^2+3n+3>=c\*g(n)

IF n=1

10\*1^2+3\*1+3>=9\*1

10+3+3>=9

16>=9 WHICH IS TRUE

**Ω**(n):

g(n)=n

10n^2+3n+3>=c\*g(n)

IF n=2

10\*2^2+3\*2+3>=9\*2

40+6+3>=18

49>=18 WHICH IS TRUE

IF n=3

10\*3^2+3\*3+3>=9\*3

90+9+3>=27

102>=27 WHICH IS TRUE

**Ω**(n^2) :

g(n)=n^2

10n^2+3n+3>=c\*g(n)

IF n=3

10\*3^2+3\*3+3<=9\*3\*3

90+9+3>=9\*9

102>=81 WHICH IS TRUE

IF n=4

10\*4^2+3\*4+3>=9\*4\*4

160+12+3>=9\*16

175>=144 WHICH IS TRUE

**Ω**(n^3) :

g(n)=n^3

10n^2+3n+3>=c\*g(n)

IF n=2

10\*2^2+3\*2+3>=9\*2\*2\*2

40+6+3>=18\*4

49>=72 WHICH IS FALSE

IF n=3

10\*3^2+3\*3+3>=9\*3\*3\*3

90+9+3>=27\*9

102>=243 WHICH IS FALSE

HENCE **Ω**(1),**Ω**(n),**Ω**(n^2) ARE POSSIBLE.

**Ω**(n^3)..... ARE NOT POSSIBLE.

THE ORDER OF GROWTH IS AS FOLLOWS

**Ω**(1)<**Ω**(log n)<**Ω**(n)<**Ω**(n\* log n)<**Ω**(n^2)<**Ω**(n^3)<**Ω**(2^n)

**THETA 'Θ' NOTATION :**

# It is a method of representing both lower and upper bound of algorithm running time.

# let f(n) and g(n) are two non-negative functions used for representing theta notations.

**If f(n)=Θ(g(n)) if and only if c1\*g(n)<=f(n)<=c2\*g(n) for all n, n>=n0**

where c1,c2, n0 are positive constants. g(n) represents the average boundary level of f(n).

c1>0,c2>0 and n0>=1.

Diagram:

Time 
C2g n) 
Input - N 

**EXAMPLES:**

1) 3n+2=**Θ**(n)

Here f(n)=3n+2

g(n)=n

c1\*g(n)<=f(n)<=c2\*g(n)

c1\*n<=3n+2<=c2\*n

c1=3 and c2=4

**Θ**(1):

g(n)=1

3\*n<=3n+2<=4\*n

IF n=1

3\*1<=3\*1+2<=4\*1

3<=5<=4 WHICH IS FALSE

**Θ**(n):

g(n)=n

3\*n<=3n+2<=4\*n

IF n=2

3\*2<=3\*2+2<=4\*2

6<=8<=8 WHICH IS TRUE

IF n=3

1. 3\*3<=3\*3+2<=4\*3

9<=11<=12 WHICH IS TRUE

**Θ**(n^2):

g(n)=n^2

3\*n\*n<=3n+2<=4\*n\*n

IF n=2

3\*2\*2<=3\*2+2<=4\*2\*2

12<=8<=16 WHICH IS FALSE

IF n=3

3\*3\*3<=3\*3+2<=4\*3\*3

27<=11<=36 WHICH IS FALSE

**Θ**(n^3) :

g(n)=n^3

3\*n\*n\*n<=3n+2<=4\*n\*n\*n

IF n=2

3\*2\*2\*2<=3\*2+2<=4\*2\*2\*2

24<=8<=32 WHICH IS FALSE

IF n=3

3\*3\*3\*3<=3\*3+2<=4\*3\*3\*3

81<=11<=108 WHICH IS FALSE.

HENCE **Θ**(n) IS POSSIBLE.

**Θ(1),Θ**(n^2),**Θ**(n^3)... ARE NOT POSSIBLE.

2) 10n^2+3n+3=**Θ**(n^2)

Here f(n)=10n^2+3n+3

g(n)=n^2

c1\*g(n)<=f(n)<=c2\*g(n)

c1\*n^2<=10n^2+3n+3<=c2\*n^2

c1=10 and c2=11

**Θ**(1):

g(n)=1

c1\*1<=10n^2+3n+3<=c2\*1

IF n=1

10\*1<=10\*1^2+3\*1+3<=11\*1

10<=16<=11 WHICH IS FALSE

**Θ**(n):

g(n)=n

c1\*n<=10n^2+3n+3<=c2\*n

IF n=2

10\*2<=10\*2^2+3\*2+3<=11\*2

20<=49<=22 WHICH IS FALSE

IF n=3

10\*3<=10\*3^2+3\*3+3<=11\*3

30<=102<=33 WHICH IS FALSE

**Θ**(n^2) :

g(n)=n^2

10\*n\*n<=10n^2+3n+3<=11\*n\*n

IF n=2

10\*2\*2<=10\*2^2+3\*2+3<=11\*2\*2

40<=40+6+3<=44

40<=49<=44 WHICH IS FALSE

IF n=3

10\*3\*3<=10\*3^2+3\*3+3<=11\*3\*3

90<=90+9+3<=99

90<=102<=99 WHICH IS FALSE

IF n=4

10\*4\*4<=10\*4^2+3\*4+3<=11\*4\*4

160<=175<=176 WHICH IS TRUE

**Θ**(n^3) :

g(n)=n^3

10\*n\*n\*n<=10n^2+3n+3<=11\*n\*n\*n

IF n=2

10\*2\*2\*2<=10\*2^2+3\*2+3<=11\*2\*2\*2

80<=49<=88 WHICH IS FALSE

IF n=3

10\*3\*3\*3<=10\*3^2+3\*3+3<=11\*3\*3\*3

270<=102<=297

WHICH IS FALSE

HENCE **Θ**(n^2) IS POSSIBLE.

**Θ**(1),**Θ**(n)**,Θ**(n^3)..... ARE NOT POSSIBLE.

THE ORDER OF GROWTH IS AS FOLLOWS

**Θ**(1)<**Θ**(log n)<**Θ**(n)<**Θ**(n\* log n)<**Θ**(n^2)<**Θ**(n^3)<**Θ**(2^n)

**Little ‘o’ notation:-**

The function f(n) = o g(n) if and only if

lim f(n)/g(n)=0  
 n→∞

**Example:-**

**3n+2 = o(1)**

**f(n) = 3n+2 g(n) = 1**

**o(1)**

**lim ((3n+2)/1)**

n→∞

= lim 3n + lim 2

n→∞ n→∞

∞+2 = 0 (which is not possible)

o(n)

lim ((3n+2)/n)

n→∞

= lim (3n/n) + lim (2/n)

n→∞ n→∞

= lim 3 + 0

n→∞

= 3 = 0 (which is not possible)

o(n2)

lim ((3n+2) n2 )

n→∞

= lim (3n/ n2) + lim (1/ n2)

n→∞ n→∞

= 0+0

Therefore 3n+2 = o(n2) is the solution.

**Probabilistic Analysis:**

Probability Analysis is the use of probability in the analysis of the problems.

Most commonly we use probability analysis to analysis the running time of an algorithm. For this they consider the hiring problem.

**Hiring Problem:-**

Hire Assistant (n):

1. best = 0//Candidate 0 is atleast qualified dummy candidate.

2. for i = 1 to n

3. Interview candidate i

4. If candidate i is better than candidate best

5. best = i

6. hire candidate i

In an interview initially there was no best candidate and so it is zero. Now first person is interviewed and if that person is better than best then he is considered to be the best and that candidate is hired. And now i value is incremented and second person is interviewed and if this person is better than the best person then he is considered to be best and is replaced with the present best. This process is continued……………………

Worst Case Analysis:

In the worst case we actually hire every candidate we interviewed. This situation occurs if the candidates comes in strictly increasing order of quality in this case we hire ‘n’ times for a total hiring cost of O(n\*c h. ).

Rank Based Hiring:

We can represent a particular input by listing in order,the rank of the candidates

(rank(1),rank(2),. . . . . . . . . . . . .,rank(n))

i.e., A = {1,2,3,. . . . . . . . . ,n}

In rank based hiring we need 10 people out of 100 then we will select first 10 numbers based on rank where as in hire assistant if we need 10 people out of 100 we should interview each and every candidate.

**Amortized Analysis:**

This analysis is used for finding average time of an algorithm. Amortized analysis initially used for specific algorithms particularly those including binary tree and union operations. There are three different ways we can platform the Amortized analysis.

\*Aggregate Method

\*Accounting Method

\*Potential Method

Here the only requirement is that the sum of the amortized complexities of all the operations in any sequence of operations must be greater than or equal to their sum of the actual complexities i.e.,

(∑(1 <= i <= n) amortized(i) ) >= ( ∑(1 <= i <= n) actual(i))

When amortized(i) and actual(i) respectively denote the amortized and actual complexities of ith operation in the sequence of operations.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Month | | 1 | 2 | | 3 | | 4 | 5 | | 6 | 7 | 8 | | 9 |
| Actual | | 50 | 50 | | 100 | | 50 | 50 | | 100 | 50 | 50 | | 100 |
| Amortized | | 75 | 75 | | 75 | | 75 | 75 | | 75 | 75 | 75 | | 75 |
| P() | | 25 | 50 | | 25 | | 50 | 75 | | 50 | 75 | 100 | | 75 |
| 10 | 11 | | | 12 | | 13 | | | 14 | | 15 | | 16 | |
| 50 | 50 | | | 200 | | 50 | | | 50 | | 100 | | 50 | |
| 75 | 75 | | | 75 | | 75 | | | 75 | | 75 | | 75 | |
| 100 | 125 | | | 0 | | 25 | | | 50 | | 25 | | 50 | |

Relative to the actual and amortized costs of each operation in a sequence of n operations, We define the potential function as

P(i) = amortized(i) – actual(i) + P(i-1)

Generalized form:

∑(1 <= i <= n) P(i) = ∑(1<=i<=n) (amortized(i) – actual(i) + P(i-1))

Aggregate Method:-

In this method we uniformly assign the amortized to the each operation.

Accounting Method:-

In this method we assign amortized costs to the operators by guessing what assignments will work.

Potential Method:-

Here we start with a potential function which is obtained by using good guess work.