Algorithm Analysis and Design

Pre-Requisites & Course Outcomes

PRE-REQUISITES:
1. Programming Skills
2. Discrete Structures
3. Data Structures

COURSE OUTCOMES (COs):
After completion of this course, the learners will be able to:-

CO #	Detailed Statement of the CO	BT Level	Mapping to PO #
CO1	Demonstrate P and NP complexity classes of the problem.	BTL2	PO1, PO2, PO3
CO2	Apply the concepts of asymptotic notations to analyze the complexities of various algorithms.	BTL4	PO1, PO2, PO3, PO4
CO3	Analyze and evaluate the searching, sorting and tree-based algorithms.	BTL5	PO1, PO2, PO3, PO4, PO5
CO4	Design efficient solutions using various algorithms for given problems.	BTL6	PO1, PO2, PO3, PO4, PO5, PO6, PO10
CO5	Develop innovative solutions for real-world problems using different paradigms.	BTL6	PO1, PO2, PO3, PO4, PO5, PO6, PO7, PO9, PO10,

Syllabus (Unit-I)

- **Performance Analysis of Algorithms:** Algorithm Specification, Performance Analysis: Space and Time Complexity, Correctness of Algorithms, Growth of Functions, Asymptotic Notations and Types, Concept of Randomized Algorithms.
- **Recurrences:** Substitution, Iteration, Master and Recurrence Tree method.
- **No. of Hours:** 09
- **Books:**
  - T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, "Introduction to Algorithms", PHI, 2nd Edition, 2006. **Chapters[1-5]**
  - S. Dasgupta, C. Papadimitriou and U.Vazirani, "Algorithms", McGraw Hill Higher Education, 1st Edition, 2017. **Chapters[0-2]**
  - J. Kleinberg and E. Tardos, "Algorithm Design", Pearson Education, 2nd Edition, 2009. **Chapters[2,5,13]**

Introduction

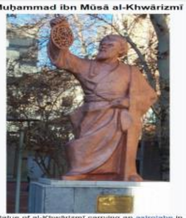
Why do we study this course?

- Why do we study Design and Analysis of Algorithm?
  - Benefit of Algorithm
    - ✓ Easy to understand.
    - ✓ Logic is developed before actual coding.
  - Benefit of Analysis of Algorithm
    - ✓ To find best version of solution from various solutions of same problem.
  - Benefit of Design of Algorithm
    - ✓ To create an efficient algorithm to solve a problem in an efficient way.

History of Algorithm

- The word 'algorithm' has its roots in Latinizing the *nisba*, indicating his geographic origin, of the name of Persian mathematician Muhammad ibn Musa al-Khwarizmi to *algorismus*
- *In late medieval Latin, algorismus, English 'algorism', the corruption of his name, simply meant the "decimal number system"*

Muḥammad ibn Mūsā al-Khwārizmī



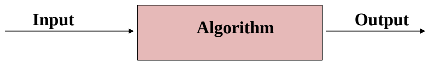
Statue of al-Khwārizmī carrying an astrolabe in Amir Kabir University, Tehran, Iran

Born	c. 780 Khwarezm
Died	c. 850 (aged c. 70) Baghdad, Iraq

Source: https://en.wikipedia.org/wiki/Algorithm

Algorithm: Definition

- An algorithm is any well-defined procedure that takes some values as input and produces some values as output.



- An algorithm is thus a finite sequence of computational steps that transforms the input into desired output in finite amount of time.

Algorithm for Problem Solving

- Problem definition
  - What task has to be done.
    - ✓ Calculation of mean, square root, shortest path etc.
- Algorithm design
  - Writing pseudo code, drawing flow chart etc. (*Decision of algorithm design model like Divide & Conquer, Dynamic Programming, Greedy etc.)*
- Algorithm analysis
  - Analysis of time and space required to run the algorithm.

Algorithm for Problem Solving (contd.)

- Implementation
  - Writing a program
- Testing
  - Testing of the output

Characteristics of Algorithm

- Input: An algorithm has zero or more input
- Output: An algorithm has one or more output.
- Finiteness: An algorithm must terminate after a finite number of steps.
- Definiteness: Each instruction must be clear and unambiguous.
- Effectiveness: An algorithm must be effective in such a way that its operations are sufficiently basic and feasible.

*Note: A procedure that has all the characteristics of an algorithm except finiteness is called **computational methods.***

 U1.10

Characteristics of Algorithm

- Input: An algorithm has zero or more input
- Output: An algorithm has one or more output.
- Finiteness: An algorithm must terminate after a finite number of steps.
- Definiteness: Each instruction must be clear and unambiguous.
- Effectiveness: An algorithm must be effective in such a way that its operations are sufficiently basic and feasible.

*Note: A procedure that has all the characteristics of an algorithm except finiteness is called **computational methods.***

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63 by Dr. Saumya Bansal U1.11

Example of an Algorithm

- Problem: To find the max element of an array

Algorithm arrayMax(A, n)

Input array A of n integers

Output maximum element of A

```
Max ←A[0]
for i ←1 to n-1 do
        if A[i] > Max then
        Max ← A[i]
End For
return Max
```

 U1.12

Example of an Algorithm

- Problem: To find the max element of an array

Algorithm arrayMax(A, n)

Input array A of n integers

Output maximum element of A

```
Max ← A[0]
for i ← 1 to n-1 do
    if A[i] > Max then
    Max ← A[i]
End For
return Max
```


Algorithm analysis

- Why do we analyse the algorithm?
  - To decide the better algorithm among various solutions of a given problem.
    - ✓ For example, better algorithm among all sorting algorithm
    - ✓ Suppose you have written an algorithm for a given problem and the solution of the problem is already exist. How do you prove your algorithm is better?
  - To check feasibility
    - ✓ Even if the solution is given first time of any given problem, the analysis of an algorithm can decide whether the algorithm will run with feasible recourse (Time and Space)

Algorithm analysis: Factors to analyze

- Time
- Space
- Correctness of an algorithm
- Communication time [For network solution]
- Power consumption [For Mobile App]

Algorithm Analysis

Priori Analysis	Posteriori analysis
Analysis is done before the real implementation of algorithm	Analysis is done after the real implementation of algorithm i.e. program
Priori analysis is an absolute analysis.	Posteriori analysis is a relative analysis.
Independent on the hardware and compiler	Dependent on the hardware and compiler
It gives approximate answer	It gives exact answer
The complexity remains same for every system	The complexity differs from system to system
Asymptotic notations are used to represent the complexity in terms of time and space functions	Complexity is represented in terms of watch time (milli second, nano second etc.) and bits/bytes (for space complexity)

 U1.16

Algorithm Analysis

- Algorithm Analysis involves mainly two types of analysis
  - Time complexity: The amount of time required to run an algorithm.
  - Space complexity: The amount of memory space required to run an algorithm.

Algorithm analysis

Time Complexity **Space Complexity**

 U1.17

Algorithm Analysis: Time Analysis

- Time complexity of an algorithm can be analysed by following way.
  - Consider each basic step of algorithm takes 1 unit of time.
  - Count the frequency of the each step

Then, the time complexity of the algorithm, T(n), will be

$$T(n)= 1^* f_{S1} + 1^* f_{S2} +1^* f_{S3} ....+1^* f_{Sn}$$

Where,

f_{Sn} = frequency of step n

 U1.18

Algorithm Analysis: Time Analysis

Statement	Cost of execution	Frequency	Total (Cost * Frequency)
1. Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. sum←0;	1	1	1
4. for i←1 to n do	1	n+1	n+1
5. s←s+a[i];	1	n	n
6. end for	0	-	0
7. return s;	1	1	1
8. }	0	-	0
Total			**T(n)=2n+3**

 U1.19

Algorithm Analysis: Order of growth

- Order of growth of an algorithm predicts that how execution time or space of an algorithm changes with the input size.
  - Let's understand with an example,

Input size (n)	Algorithm A T(n)=100n+1	Algorithm B T(n)=n^2+n+1
10	1001	111
100	10001	10101
1000	10001	1001001
10000	1000001	>10^{10}

 U1.20

Algorithm Analysis: Order of growth

Input size (n)	Algorithm A T(n)=100n+1	Algorithm B T(n)=n^2+n+1
10	1001	111
100	10001	10101
1000	10001	1001001
10000	1000001	>10^{10}

- Observations:
  - At *n*=10, Algorithm A looks bad.
  - As n increases, the Algorithm A looks better. (Why?)
  - Regardless of the coefficients, there will always be some value of n where $an^2 > bn$.
  - Even if the run time of Algorithm A were n + 10000, it would still be better than Algorithm B for sufficiently large n.
- Conclusion: The coefficient and non-leading term do not affect the order of growth for some sufficient large value of n.

  The leading term is the term with the highest exponent.

 U1.21

Algorithm Analysis: Asymptotic notation

- The Asymptotic notations are used to describe the rate of growth of functions.
  - Following are the types of asymptotic notations, we generally use.

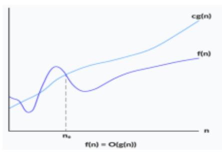
1. Big–OH (O)
2. Big–OMEGA (Ω)
3. Big–THETA (Θ)
4. Little–OH (o)
5. Little–OMEGA (ω)

Asymptotic notation: Big–OH (O)

- Let *f(n)* and *g(n)* be two functions from set of integers to real numbers , *f:Z→R*, then f(n) is O(g(n)) or f(n) = O(g(n))iff

$$0 \le f(n) \le C * g(n) \qquad \forall n \ge n_0$$

Where, C and n_0 are any positive real constants



Big Oh notation always gives **Upper Bound**

Asymptotic notation: Big–OH (O)

- Let f(n)= n^3+3n+5 then
  - f(n)=O(n^3)   True/ False
  - f(n)=O(n)   True/ False
  - f(n)=O(n^4)   True/ False
  - f(n)=O(nlogn)   True/ False
  - f(n)=O(n^{3logn})   True/ False
  - f(n)=O(n^3logn)   True/ False

Asymptotic notation: Big–OH (O)

- Let $f(n)= n^3+3n+5$ and $g(n)= n^3$. If $f(n)=O(g(n))$ then find C and n_0.
- Let $f(n)=n^2+3n+5$. if $f(n)=O(g(n))$ then what could be the possible values for $g(n)$?

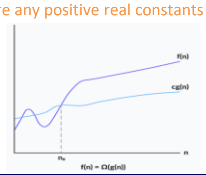
Asymptotic notation: Big–Omega (Ω)

- Let *f(n)* and *g(n)* be two functions from set of integers to real numbers , *f:Z→R*, then f(n) is Ω(g(n)) or f(n) = Ω(g(n)) iff

$$0 \leq C * g(n) \leq f(n) \quad \forall n \geq n_0$$

Where, C and n_0 are any positive real constants

Big Omega notation always gives Lower Bound



Asymptotic notation: Big–Omega (Ω)

- Let $f(n)= n^3+3n+5$ then
  - $f(n)=\Omega(n^3)$ True/ False
  - $f(n)= \Omega(n)$ True/ False
  - $f(n)= \Omega(n^4)$ True/ False
  - $f(n)= \Omega(n\log n)$ True/ False
  - $f(n)= \Omega(\log n)$ True/ False
  - $f(n)= \Omega(n^2\log n)$ True/ False

Asymptotic notation: Big–Omega (Ω)

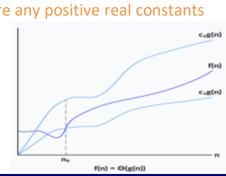
- Let f(n)= n+5 and f(n) ∈Ω (n). Calculate C and n_0.
- Let f(n)=n^2+3n+5. if f(n)=Ω (g(n)) then what could be the possible values for g(n)?

Asymptotic notation: Big–Theta (Θ)

- Let *f(n)* and *g(n)* be two functions from set of integers to real numbers , *f:Z→R*, then f(n) is Θ (g(n)) or f(n) = Θ (g(n)) iff

$$0 \le C_1 * g(n) \le f(n) \le C_2 * g(n) \quad \forall n \ge n_0$$

Where, C and n_0 are any positive real constants



Big Theta notation always gives **tight bound**

Asymptotic notation: Big–Theta (Θ)

- Let f(n)= n^3+3n+5 then
  - f(n)=Θ (n^3) True/ False
  - f(n)= Θ (n) True/ False
  - f(n)= Θ (n^4) True/ False
  - f(n)= Θ (nlogn) True/ False
  - f(n)= Θ (logn) True/ False
  - f(n)= Θ (n^2logn) True/ False

Asymptotic notation: Big–Theta (Θ)

- Let f(n)= n+5 and f(n) ∈Θ (n). Calculate C_1, C_2 and n_0.
- Let f(n)=n^2+3n+5. if f(n)=Θ (g(n)) then what could be the possible value(s) for g(n)?

 U1.31

Asymptotic notation: Big–Theta (Θ)

- Sometimes, we can not express the function in tight bound
  - For example, Let f(n)=n!

    We know that n!=1*2*3*......*(n-1)*n

Hence,

$$1 \le 1*2*3 \dots\dots * (n-1) * n \le n * n * n * \cdots * n$$

$$1 \le n! \le n^n$$

Here, f(n)=O(n^n) and f(n)=Ω(1). But no theta bound.

Same you can find for f(n)=logn!

 U1.32

Asymptotic notations: O, Θ and Ω

$$1 < logn < \sqrt{n} < n < nlogn < n^2 < n^3 \dots\dots . < 2^n < 3^n \dots\dots . < n^n$$

Lower Bound **Upper Bound**

Tight Bound
or
Average bound

Let f(n)=n^2+5 then f(n)=O(n^2) (How?) and f(n)=Ω (n^2) (How?) that implies f(n)= Θ (n^2).

 U1.33

Asymptotic notations: O, Θ and Ω

- Let f(n) and g(n) be two functions, from set of integers to real numbers , *f:Z→R*, such that

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

then,

if A=0 then f(n)=O(g(n)) but f(n) ≠Θ (g(n))

if A=∞ then f(n)=Ω (g(n)) but f(n) ≠Θ (g(n))

if A≠0 and A is finite f(n) =Θ (g(n))

Ponder: Can we compare order of growth from above statements?

Asymptotic notations: Little-OH (o)

- Let f(n) and g(n) be two functions, from set of integers to real numbers , *f:Z→R*, then f(n) is o(n) or f(n) ∈ o(n) iff

$$0 \le f(n) < c * g(n) \qquad \forall n \ge n_0$$

Where, C_1 and n_0 is any positive constant.

Mathematically, if

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

Then, we can say that f(n)=o(g(n)).

If f(n)=o(g(n)) then f(n)=O(g(n))? And if f(n)=O(g(n)) then f(n)=o(g(n))?

Asymptotic notations: Little-Omega(o)

- Let f(n) = n+3 and g(n)= n^2 be two functions then f(n) = o(n) or f(n) ∈ o(n) ?

We have to calculate the value of following limit

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

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

$$= \lim_{n\to\infty}\left(\frac{n}{n^2}+\frac{3}{n^2}\right) = \lim_{n\to\infty}\left(\frac{1}{n}+\frac{3}{n^2}\right) = 0$$

Hence, we can say that f(n)=o(n).

Asymptotic notations: Little-Omega(ω)

- Let f(n) and g(n) be two functions, , from set of integers to real numbers , *f:Z→R*, then f(n) = ω(n) or f(n) ∈ ω (n) iff

$$f(n) > c * g(n) \geq 0 \qquad \forall n \geq n_0$$

Where, C_1 and n_0 is any positive constant.

Mathematically, if

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

Then, we can say that f(n)=ω (g(n)).

If f(n)=ω (g(n)) then f(n)= Ω(g(n))? And if f(n)= Ω(g(n)) then f(n)=ω (g(n))?

Asymptotic notations: Little-Omega(ω)

- Let f(n) = n^2+3 and g(n)= n be two functions then f(n) = ω(n) or f(n) ∈ ω(n) ?
  We have to calculate the value of following limit

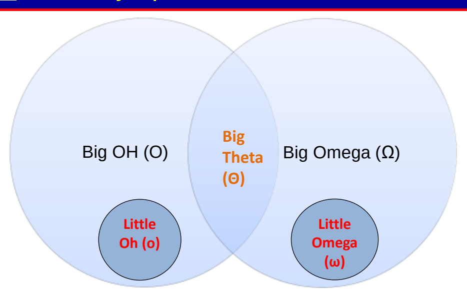
$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

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

$$= \lim_{n\to\infty} \left(\frac{n^2}{n} + \frac{3}{n}\right) = \lim_{n\to\infty} \left(n + \frac{3}{n}\right) = \lim_{n\to\infty} (n+0) = \lim_{n\to\infty} (n) = \infty$$

Hence, we can say that f(n)=ω (n).

Asymptotic notations: Relationship



Asymptotic notations: Properties

- Reflexive Property:
  - f(n)=O(f(n))
  - f(n)= Ω(f(n))
  - f(n)=Θ(f(n))
- Symmetric Property:
  - f(n)= Θ(g(n)) iff g(n)= Θ(f(n))
  - f(n)= O(g(n)) iff g(n)= Ω(f(n))
  - f(n)= o(g(n)) iff g(n)= ω(f(n))

 U1.40

Asymptotic notations: Properties

- Reflexive Property
  - f(n)= Θ(g(n)) and g(n)= Θ(h(n)) that implies f(n)= Θ(h(n))
  - f(n)=O(g(n)) and g(n)= O(h(n)) that implies f(n)= O(h(n))
  - f(n)=Ω(g(n)) and g(n)= Ω(h(n)) that implies f(n)= Ω(h(n))
  - f(n)=o(g(n)) and g(n)= o(h(n)) that implies f(n)= o(h(n))
  - f(n)=ω(g(n)) and g(n)= ω(h(n)) that implies f(n)= ω(h(n))
- O(f(n)+g(n))= O{max(f(n),g(n)}

 U1.41

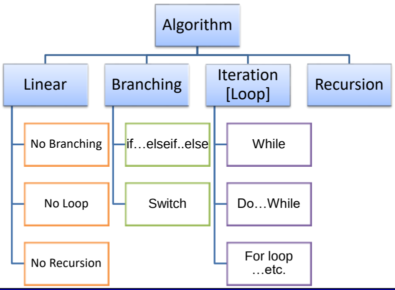
Asymptotic notations: Efficiency Classes

- Basic Asymptotic Efficiency Classes

Complexity	Efficiency Class
1	Constant
logn	Logarithmic
n	Linear
nlogn	n-log-n or Linearithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

 U1.42

Time Analysis of Algorithm



 U1.43

Time Analysis of Algorithm: Linear

1. Algorithm: FindSum(a,b)
2. {
3. input: integer a and integer b
4. Output: sum of a and b
5. sum←0;
6. sum=a+b;
7. return sum;
8. }

Analysis:

Statement 5: 1
Statement 6: 1
Statement 7: 1
Total:1+1+1=3

For any constant value of f(n) we always write O(1).

f(n)=3 i.e. f(n)=O(1).

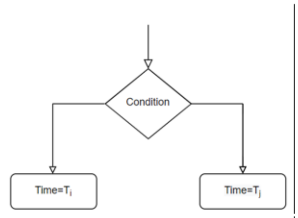
Can we say f(n)=Θ(1)?

 U1.44

Time Analysis of Algorithm: Linear

 U1.45

Time Analysis of Algorithm: Branch



If (Condition)
    Statement 1
    Statement 2
    .
    .
    Statement n
else
    Statement 1
    Statement 2
    .
    .
    Statement n
end if

Time=T_i

Time=T_j

Total time T(n)= max { T_i, T_j}

Time Analysis of Algorithm: Branch

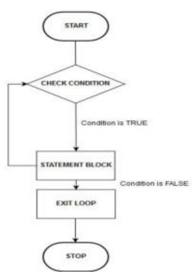
1. *if (a >b)*
2. z← a*a
3. print z
4. *else*
5. z← b*b
6. k← a+z
7. print k
8. *end if*

Total time complexity T(n)= max (2,3)

i.e. T(n)=2

Therefore, T(n) =O(1)

Time Analysis of Algorithm: Loop



while (Condition) do
    Statement 1
    Statement 2
    .
    .
    Statement n
end while

for i ← 0 to n do
    Statement 1
    Statement 2
    .
    Statement n
end for

Time Analysis of Algorithm: Loop

```
for (i=0;i<n;i++)
    Statement 1          →  O(n)

for (i=0;i<n;i=i+2)
    Statement 1          →  O(n/2)=O(n)

for (i=0;i<n;i=i*2)
    Statement 1          →  O(logn)

for (i=n;i>0;i=i/2)
    Statement 1          →  O(logn)
```


Time Analysis of Algorithm: Loop

```
for (i=0;i<5;i++)
    Statement 1          →  O(1)

i←1; s ←1;
 while(s<=n) do
    i++;
    s=s+i;               →  O(√n)
    print("*")
end while
```


Time Analysis of Algorithm: Loop

```
if(a>b)
   for (i=0;i<5;i++)
     Statement 1         →  O(n) Why?
else
   for (i=0;i<n;i++)
     Statement 2
end if

for (i=0; i<n;i++)
  for(j=0; j<5; j++)     →  O(n)
    Statement 1
```


Time Analysis of Algorithm: Loop

```
for (i=0; i<n;i++)
  for(j=0; j<n-1; j++)
    Statement 1
```
→ $O(n^2)$

```
for (i=0; i<n;i++)
  for(j=i; j<n-1; j++)
    Statement 1
```
→ $O(n^2)$

```
for (i=0; i<n;i++)
  for(j=0; j<n; j=j*2)
    Statement 1
```
→ O(nlogn)

 U1.52

Time Analysis of Algorithm: Loop

```
for (i=0; i*i<n;i++)
    Statement 1
```
→ $O(\sqrt{n})$

```
for (i=n/2; i≤n;i++)
  for(j=1; j≤ n; j=2*j)
     for(k=1; k≤ n-1; k=k*2)
    Statement 1
```
→ $O(n\log^2 n)$

```
for (i=0; i<n;i++)
  for(j=0; j< n; j=j++)
     for(k=0; k< n; k=k++)
    Statement 1
```
→ $O(n^3)$

 U1.53

Time Analysis of Algorithm: Loop

```
for (i=0; i<n;i++) do
  for(j=0; j<5; j++) do
    if  a>b Then
       for(j=0; j<n; j++)
          Statement
    else
       for(j=0; j<n; j=j*2)
          Statement
    end if
  end for
end for
```
→ $O(n^2)$ **Why?**

 U1.54

Time Analysis of Algorithm: Loop

```
int j = 1;
  for (int i = 0; i < n; i++) {
    for (int k = j; k > 0; k--) {
        print("*");
    }
    j *= 2;
  }
```

$O(2^n)$ **Why?**

for i=0 : inner loop j=1 i.e 2^0
for i=1 : inner loop j=2 i.e 2^1
for i=2 : inner loop j=4 i.e 2^2
:
:
for i=n : inner loop j=1 i.e 2^n

Sum= $2^0+2^1+2^2+.......+2^n= 2^n - 1$ (Geometric Series)= $O(2^n)$

 U1.55

Loop and Recursion

```
for (i=n;i>0;i--)
print(i);
```

```
void fun(int n)
{
    if(n==0)
    return;
    else
    {
        print(n);
        fun(n-1);
    }
}
```

Base Case:
Loop Exit condition
Recursion Body:
Body of the loop
Recursion Call
increment/Decrement of loop variable

 U1.56

Loop and Recursion

```
for (i=1;i<n;i++)
print(i);
```

```
void fun(int n)
{
    if(n==0)
    return;
    else
    {
        fun(n-1);
        print(n);
    }
}
```

Base Case:
Loop Exit condition
Recursion Body:
Body of the loop
Recursion Call
increment/Decrement of loop variable

 U1.57

Loop and Recursion

```
for (int i = 1; i <= n; i++)
      fact = fact * i;

int fun(int n)
{
    if(n==1)
    return 1;
    else
    {
        return n*fun(n-1);
    }
}
```

Base Case:
Loop Exit condition
Recursion Body:
Body of the loop
Recursion Call
increment/Decrement of loop variable

Recursion and Recurrence Relation

```
void fun(int n)
 {
   if (n==0)
   return;
   else
    {
    fun(n-1);
    print(n);
    }
 }
```

Recurrence Relation:

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

Can you guess the output?

Recursion and Recurrence Relation

```
void fun(int n)
 {
   if (n==0)
   return;
   else
    {
    for i←1 to n
    print("*");
    fun(n-1);
    }
 }
```

Recurrence Relation:

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + n & n > 0 \end{cases}$$

Can you guess the output?

Recursion and Recurrence Relation

```
void fun(int n)
 {
  if (n==0)
  return;
  else
   {
   fun(n-1);
   fun(n-2);
   print("*");
   }
 }
```

Recurrence Relation:

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

Can you guess the output?

Recursion and Recurrence Relation

BINARY SEARCH

```
int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l) {
      int mid = l + (r - l) / 2;
         if (arr[mid] == x)
         return mid;
      if (arr[mid] > x)
         return binarySearch(arr, l, mid - 1, x);
        return binarySearch(arr, mid + 1, r, x);
   }
    return -1;
}
```

Recurrence Relation:

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

Recursion and Recurrence Relation

MERGE SORT

```
void mergeSort(int arr[],int l,int r){
   if(l>=r){
      return;
   }
   int m =l+ (r-l)/2;
   mergeSort(arr,l,m);
   mergeSort(arr,m+1,r);
   merge(arr,l,m,r);
}
```

Recurrence Relation:

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

Recursion and Recurrence Relation

Recurrence	Algorithm	Solution
T(n) = T(n/2) + O(1)	binary search	O(log n)
T(n) = T(n-1) + O(1)	sequential search	O(n)
T(n) = 2T(n/2) + O(1)	tree traversal	O(n)
T(n) = T(n/2) + O(n)	quicksort partition	O(n)
T(n) = 2T(n/2) + O(n)	mergesort, quicksort	O(n log n)
T(n) = T(n-1) + O(n)	selection or bubble sort	O(n^2)

Recursive Algorithm Analysis

- Recursive Algo Analysis
  - Tree Method
  - Iteration Method
  - Master Method
  - Substitution Method

Recursive Algorithm Analysis : Useful Formulae

$$\log x^y = y \log x \qquad \log n = \log_{10}^{n}$$

$$\log xy = \log x + \log y \qquad \log^k n = (\log n)^k$$

$$\log \log n = \log(\log n) \qquad \log \frac{x}{y} = \log x - \log y$$

$$a^{\log_b^x} = x^{\log_b^a} \qquad \log_b^x = \frac{\log_a^x}{\log_a^b}$$

Recursive Algorithm Analysis : Useful Formulae

Arithmetic series

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Geometric series

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

Harmonic series

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \log n$$

Other important formulae

$$\sum_{k=1}^{n} \log k \approx n \log n$$

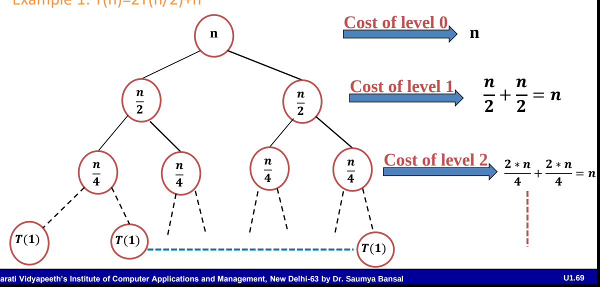
Recursive Algorithm Analysis : Tree Method

- Steps
  - Draw the recursion tree
  - Find cost of each level
  - Count the height of the tree [Maximum number of levels]
  - Count total number of leaf node [last level]
  - Find out cost of last level
  - Calculate total cost

    = Sum of the cost of each level + cost of last level

Recursive Algorithm Analysis : Tree Method

- Example 1: T(n)=2T(n/2)+n



Recursive Algorithm Analysis : Tree Method

Height of the tree

- Let after k level, we reach at T(1).
- We can observe, in the tree, that the tree grows as $\frac{n}{2^l}$ where l is the no of levels.
- Hence, After k level, $\frac{n}{2^k}$= T(1)

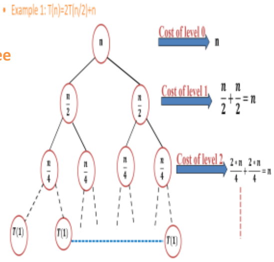
$\frac{n}{2^k}$= 1 (A/C recurrence relation)

=>$n = 2^k$

taking log both the side

$$\log_2 n = \log_2 2^k$$
$$\log_2 n = k$$

Hence, Height of the tree k = $\log_2 n$



Recursive Algorithm Analysis : Tree Method

No of Leaf node:

If the root is level 0, then the K-th level of the N-ary tree will have N^K nodes.

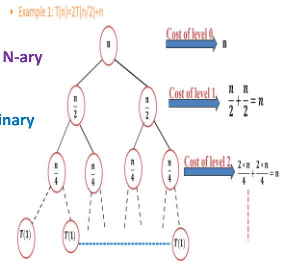
From the tree, it is observed that the tree is binary tree (two children of each node).

Hence, the total no of leaf node=2^k = $2^{\log_2 n}$

= $n^{\log_2 2}$

= n

Total cost of leaf node= n*T(1)=n*1=n



Recursive Algorithm Analysis : Tree Method

Total Cost=

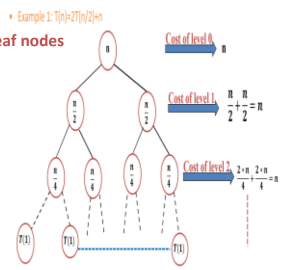
Sum of cost of each internal level + cost of leaf nodes

= n+n+n+......... k terms + Θ(n)

= k*n + Θ(n)

= nlogn + Θ(n)

=O(nlogn)



Recursive Algorithm Analysis : Tree Method

- Example 2: T(n)=T(n/2)+n

Recursive Algorithm Analysis : Tree Method

- Example 3: T(n)=T(n/3)+T(2n/3)+n

Recursive Algorithm Analysis : Iteration Method

- In the iteration method we iteratively "unfold" the recurrence until we "see the pattern".
- After getting pattern, we do summation and get the total cost of recurrence relation
- Example: T(n)=T(n-1)+1

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

After k terms,
=T(n-k) +(1+1+1……k times)

T(0)=1
Let after k term we get T(0)
Hence, n-k=0=>n=k
Therefore,
T(n-n) +(1+1+1……n times)
T(0) +(1+1+1……n times)
1+n= O(n)

Recursive Algorithm Analysis : Iteration Method

- Example 2: $T(n)=2T(n/2)+n$

Recursive Algorithm Analysis : Iteration Method

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

Recursive Algorithm Analysis : Substitution Method

- Idea: Make a guess for the form of the solution and prove by induction.
  - How do we make a good guess?
    - ✓ Use Tree/ Iteration method

Example: $T(n) = 2T(n/2) + n$

Guess: $T(n) = O(n\log n)$ i.e $T(n) \le cn\log n$

Proof:

Base Case: We need to show that our guess holds for some base case (not necessarily n = 1, some small n is ok).

Let n=2,

$2T(n/2)+n=2T(2/2)+2=2*1+2=4$

$n\log n=2\log 2=2$

That means, $T(n) \le C*n\log n$

Recursive Algorithm Analysis : Substitution Method

Guess: T(n) = O(nlogn) i.e T(n) ≤ cnlogn
Proof:

Induction Step: Assume holds for n/2: $T(\frac{n}{2}) \le c\frac{n}{2}\log\frac{n}{2}$

Now we prove that holds for n: T(n) ≤ cnlogn

T(n) = 2T(n/2) + n
≤ 2(cn/2 logn/2) + n
= cnlogn/2 + n
= cnlogn − cnlog2 + n
= cnlogn − cn + n
≤ cnlogn +n(1-c)

What if we guess T(n)=O(n^2)?

Thus, T(n)=O(nlogn)

Similarly, it can be shown that T(n) = Ω(nlogn)

 U1.79

Recursive Algorithm Analysis : Master Theorem

Let a ≥ 1 and b > 1 be constants, let *f(n)* be a function, and let *T(n)*, monotonically Increasing function, be defined on the nonnegative integers by the recurrence

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

where, we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then *f(n)* has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$ ($\epsilon \in \mathbb{R}^+$) , then $T(n)=\Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n)=\Theta(n^{\log_b a} \lg n)$
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$ ($\epsilon \in \mathbb{R}^+$) , and if $af(n/b) \le cf(n)$ for some constant then c<1 and all sufficiently polynomialy large n, then $T(n)=\Theta(f(n))$

source: Introduction to Algorithms, MIT Press by T Cormen, C Leiserson, et al.

 U1.80

Recursive Algorithm Analysis : Master Theorem

- Example 1: T(n)=9T(n/3)+n
  - Compare the equation with *T(n)=aT(n/b)+f(n)*
  - a=9 b=3 f(n)=n
  - Calculate $n^{\log_b a} = n^{\log_3 9} = n^2$

  Now check each case of Master Theorem one by one.

  Let's check first case, $f(n) = O(n^{\log_b a-\epsilon})$ i.e. n ≤ cn^2 ?

  Since, , $f(n) = O(n^{\log_3 9-\epsilon})$ where ϵ=1 we can apply case1.

  Hence, according to Master Theorem, T(n)=Θ(n^2)

 U1.81

Recursive Algorithm Analysis : Master Theorem

- Example 2: T(n)=T(2n/3)+1
  - Compare the equation with *T(n)=aT(n/b)+f(n)*
  - a=1 b=3/2 f(n)=1
  - Calculate $n^{\log_b a} = n^{\log_{3/2} 1} = 1$

Now check each case of Master Theorem one by one.

Let's check first case, *f(n)* $= O(n^{\log_b a-\epsilon})$

Since, , *f(n)* $\neq O(n^{\log_{3/2} 1-\epsilon})$ where $\epsilon > 0$, we can't apply case1. Why?

Let's Check for second case i.e. *f(n)* $= \Theta(n^{\log_b a})$ *i.e.* $c_2 n^{\log_b a} \le f(n) \le c_1 n^{\log_b a}$

Since *f(n)* $= \Theta(n^{\log_b a})$, *we can apply case 2*

Hence, according to Master Theorem, $T(n)=\Theta(n^{\log_b a}\lg n) = \Theta(\log n)$

Recursive Algorithm Analysis : Master Theorem

- Example 3: T(n)=3T(n/4)+nlogn
  - Compare the equation with *T(n)=aT(n/b)+f(n)*
  - a=3 b=2 f(n)=nlogn
  - Calculate $n^{\log_b a} = n^{\log_4 3} \approx n^{0.79}$

Now check each case of Master Theorem one by one.

Since *f(n)* $= \Omega(n^{\log_b a+\epsilon})$, *we can apply case 3, but before that we must check two conditions.*

1. *f(n) should be polynomially larger than* $n^{\log_b a}$
2. *Check Regularity condition (af(n/b) ≤ cf(n)* for some constant then c<1)

Recursive Algorithm Analysis : Master Theorem

- Example 3: T(n)=3T(n/4)+nlogn
  - Compare the equation with *T(n)=aT(n/b)+f(n)*
  - a=3 b=2 f(n)=nlogn
  - Calculate $n^{\log_b a} = n^{\log_4 3} \approx n^{0.79}$

Condition 1:f(n) should be polynomially larger than $n^{\log_b a}$

"Polynomially larger" means that the ratio of the functions falls between two polynomials, asymptotically. Specifically, f(n) is polynomially greater than g(n) if and only if there exist generalized polynomials (fractional exponents are allowed) p(n),q(n) such that the following inequality holds asymptotically: $p(n) \le f(n)/g(n) \le q(n)$

https://math.stackexchange.com/questions/1614848/meaning-of-polynomially-larger

$p(n) \le n\log n / n^{0.79} \le q(n)$ => $p(n) \le n^{0.21} \log n \le q(n)$ => $n^{.01} \le n^{0.21} \log n \le n^2$

*Hence, **f(n) should be polynomially larger than*** $n^{\log_b a}$

Recursive Algorithm Analysis : Master Theorem

- Example 3: T(n)=3T(n/4)+nlogn
  - Compare the equation with *T(n)=aT(n/b)+f(n)*
  - a=3 b=2 f(n)=nlogn
  - Calculate $n^{\log_b a}$ =$n^{\log_4 3}$ ≈ $n^{0.79}$

Condition 2: Check Regularity condition *(af(n/b) ≤ cf(n) for some constant then c<1)*

af(n/b)=3(n/4)log(n/4)≤(3/4) nlogn, here c=3/4 and c<1

Now, Both the conditions have been satisfied, therefore, we can apply case 3.

Hence, T(n)=ϴ(nlogn)

Recursive Algorithm Analysis : Master Theorem

- Example 4: T(n)=2T(n/2)+nlogn
  - Compare the equation with *T(n)=aT(n/b)+f(n)*
  - a=2 b=2 f(n)=nlogn
  - Calculate $n^{\log_b a}$ =$n^{\log_2 2}$ = n

Since *f(n) = Ω($n^{\log_b a+\epsilon}$), we can apply case 3*

- Condition 1: f(n) should be polynomially larger than $n^{\log_b a}$

p(n)≤ *nlogn/*n ≤q(n) =>p(n)≤ *logn*≤q(n)

We cannot find any polynomial for p(n).

Hence, f(n) is not polynomially larger than $n^{\log_b a}$

Here, Master theorem can't be applied

Recursive Algorithm Analysis : Master Theorem

- Some more recurrence relation where Master theorem can't be applied
  - **T (n) = 2^nT (n/2) + n^n ⇒ Does not apply** (a is not constant)
  - **T (n) = 2T (n/2) + n/log n ⇒ Does not apply** (f(n) is not polynomially larger than $n^{\log_b a}$)
  - **T (n) = 0.5T (n/2) + 1/n ⇒ Does not apply** (a < 1)
  - **T (n) = 64T (n/8) − n^2logn ⇒ Does not apply** (f(n) is not positive)
  - **T(n)=sinn ⇒ Does not apply** (T(n) is not monotone)

Recursive Algorithm Analysis : Master Theorem

- Extended Master Theorem: *(Source: Data Structure and Algorithm Made Easy by Narasimha Karumanchi)*

  If the recurrence is of the form ,

  $$T(n)=aT(n/b)+\Theta(n^k \log^p n)$$

  where $a \geq 1, b > 1, k \geq 0$ and p is a real number, then:

  - 1) If $a > b^k$, then $T(n)=\Theta(n^{\log_b a})$
  - 2) If $a = b^k$
    - ✓ a. If $p > -1$, then $T(n)= \Theta(n^{\log_b a} \log^{p+1} n)$
    - ✓ b. If $p = -1$, then $T(n)= \Theta(n^{\log_b a} \log\log n)$
    - ✓ c. If $p < -1$, then $T(n)= \Theta(n^{\log_b a})$
  - 3) If $a < b^k$
    - ✓ a. If $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
    - ✓ b. If $p < 0$, then $T(n) = O(n^k)$

Recursive Algorithm Analysis : Master Theorem

- Solve following recurrence relation using extended Master Theorem
  - $T(n)=3T(n/4)+n\log n$ Solution: $\Theta(n\log n)$
  - $T(n)=2T(n/2)+n\log n$ Solution: $\Theta(n\log^2 n)$
  - $T(n)=3T(n/2)+n^2$ Solution: $\Theta(n^2)$
  - $T(n)=4T(n/2)+n^2$ Solution: $\Theta(n^2 \log n)$
  - $T(n)=16T(n/4)+n$ Solution: $\Theta(n\log n)$
  - $T(n)=2T(n/2)+n/\log n$ Solution: $\Theta(n\log\log n)$
  - $T(n)=2T(n/4)+n^{0.5}$ Solution: $O(n^{0.5})$

Recursive Algorithm Analysis : Master Theorem

- Solve following recurrence relation using extended Master Theorem
  - $T(n)=T(\sqrt{n})+1$

  Let $n=2^m$

  $T(2^m)= T(2^{m/2}) + 1$ …….. Eqn(1)

  Let $S(m) = 2^m$, Now, Eqn(1) can be rewritten as

  $S(m)=S(m/2) +1$

  Now, use the master theorem,

  $S(m)=\Theta(\log m)$.

  We have $n=2^m$, take log both the side, $m=\log n$

  Now, put the value of n, hence, $T(n)= \Theta(\log\log n)$

Recursive Algorithm Analysis : Master Theorem

- Master Theorem for Subtract and Conquer. *(Source: Data Structure and Algorithm Made Easy by Narasimha Karumanchi)*
  Let $T(n)$ be a function defined on positive n, and having the property

$$T(n) = \begin{cases} c, & if\ n \le 1 \\ aT(n-b) + f(n), & if\ n > 1 \end{cases}$$

for some constants c, $a > 0$, $b \ge 0$, $k \ge 0$, and function $f(n)$. If $f(n)$ is in $O(n^k)$, then

$$T(n) = \begin{cases} O(n^k) & if\ a < 1 \\ O(n^{k+1}) & if\ a = 1 \\ O\left(n^k a^{\frac{n}{b}}\right) & if\ a > 1 \end{cases}$$

- Variant of Subtraction and Conquer Master Theorem
  The solution to the equation $T(n) = T(\alpha n) + T((1-\alpha)n) + \beta n$, where $0 < \alpha < 1$ and $\beta > 0$ are constants, is $O(n\log n)$.

Space Complexity

Space Complexity

Definition: The amount of space required to solve an instance of a problem against the input size "n".

Auxiliary space: Space other than that consumed by the input.

We often speak of **Auxiliary space (extra memory)** needed, not counting the memory needed to store the input itself.

Space Complexity

Example:

```
int FindMax(int A[], int n)
 {
   int max=A[0];
   for (int i=0;i<n;i++){
     if(A[i]>max)
       max=A[i];
   }
 return max;
 }
```

Auxiliary space= 4 integer
= O(1)

We can use bytes, but it's easier to use, say, the number of integers used, the number of fixed-sized structures, etc.

Space Complexity

Example:

```
int Fn( int n)
 {
   if(n==0)
    return;
   else
    Fn(n-1);
 }
```

Auxiliary space= Stack used for each recursion= O(logn)

In recursive algo, the function call stack is also considered.

Space Complexity

Example:

```
int Fn( int n)
 {
   if(n==0)
    return;
   else
    Fn(n/2);
 }
```

Auxiliary space= Stack used for each recursion= O(logn)

In recursive algo, the function call stack is also considered.

Space Complexity

Example:

```
int Fn( int n)
{
  if(n==0)
   return;
  else
   Fn(n/2);
   Fn(n/2);
}
```

Auxiliary space= Stack used for each recursion= O(logn)

In recursive algo, the function call stack is also considered.

Note that the stack space is reused here. Once a function call terminates, it removes from the stack.

Recursive Algorithm Analysis : Correctness of Algorithm

Definition: An algorithm Is called totally correct for the given specification if and only if for any correct input data it:

- Terminates and
- Returns correct output

- Correct input data is the data which satisfies the initial condition of the specification
- Correct output data is the data which satisfies the final condition of the specification

Recursive Algorithm Analysis : Correctness of Algorithm

Problem: "Given the array and its length compute the sum of numbers in the array"

The corresponding Specification could be:

Name: Sum (Arr, len)

input: (initial condition)

Algorithm gets 2 following arguments (input data):

1. Arr - array of integer numbers
2. len - length of Arr (natural number)

output:(Final condition)

Algorithm must return:

sum - sum of the numbers in the array Arr (integer number)

Recursive Algorithm Analysis : Correctness of Algorithm

The Proof of total correctness of the algorithm involves following steps.

1. Prove that the algorithm always terminates for any correct input data.
2. Prove that the algorithm produces correct output for any correct input data. (Partial Correctness)

Partial Correct Algorithm: An algorithm is said to be partial correct if it guaranties the correct output for any correct input data.

Partial correct algorithm does not make the algorithm stop.

The proof of termination can never be fully automated, since the halting problem is undecidable problem.

Recursive Algorithm Analysis : Correctness of Algorithm

There are two main methods to prove correctness of an algorithm.

Empirical Method

- Run the program and check its correctness

Formal Reasoning

- Loop Invariant Method

Correctness of Algorithm: Empirical Method

Empirical Method is based on the actual implementation (Program) of the algorithm and observation of output.

Problem: To find the maximum in the a given array

Algorithm	Implementation

FindMax(Arr, len)
{
max ← -1
for i←0 to len{
if (Arr[i] > max) {
max ← Arr[i]
}
}
return max
}
	int FindMax(int a[], int n)
	{
	int i=0, max=-1;
	for(i=0;i<n;i++){
	if(a[i]>max)
	max=a[i];
	}
	return max;
	}

Correctness of Algorithm: Empirical Method

Now, Check if the algorithm totally correct i.e.

1. whether the algorithm stops?
2. whether algorithm gives correct output for every valid input?

Algorithm	Implementation

FindMax(Arr, len)
{
max ← -1
for i←0 to len{
  if (Arr[i] > max) {
    max ← Arr[i]
  }
}
return max
}
	int FindMax(int a[], int n)
	{
	int i=0, max=-1;
	for(i=0;i<n;i++){
	  if(a[i]>max)
	    max=a[i];
	}
	return max;
	}

Input 1: Arr={10,15,6,4,9}

Input 2: Arr={-10,-4,-15,-3,-15}

Partial Correctness is difficult to prove using empirical method

Correctness of Algorithm: Loop Invariant Method

There are three steps involved in loop invariant method.

- Initialization: It is true prior to the first iteration of the loop.
- Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
- Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Note the similarity to mathematical induction, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration corresponds to the base case, and showing that the invariant holds from iteration to iteration corresponds to the inductive step.

Correctness of Algorithm: Loop Invariant Method

Problem: "To find factorial of any positive integer."

Algorithm:

```
Fact(n)
{
  i←1
  fact ← 1
  while( i ≤ n)
   {
     fact ← fact*i
     i++;
   }
return fact
}
```

Initialisation:

Prior to the first loop, Lets take n=1 then fact=1 i.e. fact=1!

Maintenance:

Lets assume the algorithm gives the output k! for valid input k i.e. fact =k! for input k
In the next iteration, for k+1,
fact=k!*(k+1) which returns fact=(k+1)!
Hence, loop invariant holds.

Termination:

The condition i>n cause the while loop to terminate. The condition can be reached because i increments by one in each iteration.

Randomized Algorithm

Definition: Any algorithm that make use of randomness as part of its logic or procedure.

Example: Find a number "X" in the given array, in which first half are 'a's and the other half are 'b's. X={a,b}

Algorithm:

```
FindElement(A, n)
begin
  repeat
    Randomly select one element out of n elements.
  until 'X' is found
end
```

(Source : https://en.wikipedia.org/wiki/Randomized_algorithm)

 U1.106

Monte Carlo Algorithms

Monte Carlo Algorithms:

- The Algorithm terminates when either it gets successful or reach at most k steps
- The algorithm has the deterministic time complexity.
- Easier to analyze for worst case.

Example:

```
FindElement(A, n, k) //k =limit of finding steps. A is array and n is the length of the array
begin
  i=0
  repeat
    Randomly select one element out of n elements.
    i=i+1
  until i=k or 'X' is found
end
```

 U1.107

Analysis : Monte Carlo Algorithms

Analysis:

- **If an 'X' is found, the algorithm succeeds, else the algorithm fails.**
- **After k iterations, the probability of finding an 'X' is:**

$$Pr[find\ X] = 1 - \left(\frac{1}{2}\right)^k$$

```
FindElement(A, n, k) //k =limit of finding steps.
begin
  i=0
  repeat
    Randomly select one element out of n elements.
    i=i+1
  until i=k or 'X' is found
end
```

- **This algorithm does not guarantee success, but the run time is bounded.**
- **The number of iterations is always less than or equal to k. Taking k to be constant the run time (expected and absolute) is Θ (1).**

 U1.108

Las Vegas Algorithm

Las Vegas Algorithm:

- The algorithm keep running infinite times. It terminates only when it gets success.

Example:

```
FindElement(A, n) // A is array and n is the length of the array
begin
repeat
    Randomly select one element out of n elements.
until 'X' is found //X={a,b}
end
```


Analysis : Las Vegas Algorithm

Analysis:

- **If an 'X' is found, the algorithm succeeds, else the algorithm fails.**
- **Expected no of iteration to find 'X'**

  E[X]=1/(1/2)=2

  If probability of success is p in every trial, then expected number of trials until success is 1/p

```
FindElement(A, n) // A is array and n is the length of the array
begin
repeat
    Randomly select one element out of n elements.
until 'X' is found //X={a,b}
end
```

- **This algorithm does guarantee success, but the run time is determined as expected value (Not Deterministic).**

Cases to be Analysed in Algorithm

The resource of an algorithm is analyzed on the following criteria:

Best case: Best case performance measures the minimum resource utilization of algorithm with respect to input n.

Worst case: Best case performance measures the maximum resource utilization of algorithm with respect to input n.

Average case: Best case performance measures the average resource utilization of algorithm with respect to input n. It is calculated as the average of all possible inputs.

Cases to be Analysed in Algorithm

Example: "Find an element 'X' in a given integer array of length n."

Best Case: Element 'X' is found at the first attempt. (Not analysed Generally)

Worst case: Element 'X' is found at the last attempt or Element 'X' could not be found. (Mostly Done)

Average case: The average of finding 'X' over all possible inputs. (Sometimes Done)

Cases to be Analysed in Algorithm

Example: "Find an element 'X' in a given integer array of length n."

Best Case: Element 'X' is found at the first attempt.

Algorithm:

```
FindMax(Arr, len)
{
 max ← Arr[0]
 for i←0 to len{
    if (Arr[i] > max) {
       max ← Arr[i]
    }
 }
return max
}
```

Best Case: Element found at first place, Hence, only one comparison is needed. therefore time complexity for best case is O (1)

Worst Case: Element found at last index. It is assumed that elements are not repeated. The n comparison is required. Hence, Time complexity for worst case is O(n)

Cases to be Analysed in Algorithm

Example: "Find an element 'X' in a given integer array of length n."

Best Case: Element 'X' is found at the first attempt.

Algorithm:

```
FindMax(Arr, len)
{
 max ← Arr[0]
 for i←0 to len{
    if (Arr[i] > max) {
       max ← Arr[i]
    }
 }
return max
}
```

Average Case: In average case, we take all calculated computing time and divide it by the no of input.

$$\frac{\sum_{i=1}^{n+1} \theta(i)}{n+1} = \frac{\theta(\frac{(n+1)(n+2)}{2})}{n+1} = \frac{(n+1)(n+2)}{2(n+1)} = \frac{(n+2)}{2} = O(n)$$

Topics We Have Learned So Far

- Notion of Algorithm
- Importance of Analysis of Algorithm
- Time and Space Complexities
- Asymptotic Notations (O,Θ,Ω,o, and ω) and their properties
- Growth of Function
- Measuring Time complexity of Non-recursive Algorithm
- Measuring Time complexity of Recursive Algorithm
  - Recursion Tree Method
  - Iterative Method
  - Master Method
  - Substitution Method

 U1.115

Topics We Have Learned So Far

- Measuring Space Complexity
- Correctness of Algorithm
- Analysis of Randomized Algorithm
- Best, Worst and Average case analysis.

 U1.116

The test can be scheduled in any lecture next week. Be Prepared.

BHARATI VIDYAPEETH PUNE

Thank You

Bharati Vidyapeeth's Institute of Computer Applications and Management (GGS IP University) New Delhi, India by Dr. Saumya Bansal