Lecture 1 What is an Algorithm

What is an algorithm?

Finite set of steps to solve a problem is called an algorithm.
Step => Instructions

Instructions are those which contains fundamental operators.

What are the characteristics of instructions?

Definitive

Every instruction must be defined without ambiguity. **Example:**

i = i Δ 1 is invalid, as it is unclear what Δ means. i = i + 1 is valid, as it is clear that i is incremented by 1.

Finiteness

Every instruction must pe terminated within a finite amount of time.

Example:
Invalid Instruction:
Define an instruction i (+) 1

```
i = 1;
while (1) {
  i = i + 1;
}
```

Has no terminating condition, and is, therefore infinite.

Every instruction must accept at least 0 input and provide at most 1 output.

Example: The plus operator '+' takes 2 inputs and gives 1 output.

Steps involved in solving a problem

1. Identify the problem

We have to identify the problem and what the problem wants us to do.

2. Identify the constraints

To solve the problem, we need to know what the constraints are.

3. Select the Design Logic

Depending on problem statement and constraints, following design strategies may be used:

1. Divide and Conquer
2. Greedy
3. Dynamic Programming
4. Branch and Bound
5. Backtracking

4. Validate

Validate whether our algorithm works on every test case. This can be done using following methods:

Principle of Mathematical Induction Proof by Contradiction

Analysis

This is used to compare algorithms w.r.t. time and space complexity and decide which is more suitable.

Types of Analysis

Priory Analysis

- Analysis is done before executing any algorithm.
- x = x + 1
- **Principle:** Frequency count of fundamental instructions We can see the number of times the fundamental instructions are getting executed. Example:

```
// # First Fundametal Instruction
int n, sum = 0;
scanf("%d", &n);
// Second Fundametal Instruction
for(int i = 0; i < n; i++) {
  sum += i;
}
```

- Provides the estimated value.
- Provides Uniform Value.
- Independent of System
- Can be used to compare two algorithms.

Posterior Analysis

- Analysis is done after execution.

```
x = x + 1

- sys1 : 0.1ns
- sys2 :0.2ns
```

```
- sys3 : 0.16ns
- sys4 : 0.3ns
```

- Provides the exact value.
- Dependent on systems input. (Non-uniform value) Lets say there are 2 algorithms: algo1 and algo2. algo1 --1K Inputs--> 0.1ns on System 1 algo2 --1M Inputs--> 1ns on System 2 We cannot tell which is faster, as the input size of different.
- Cannot be used to compare two algorithms.

Lecture 2 - Asymptotic Notation

Asymptotic means towards infinity. When using asymptotic notation, there is an inherent assumption that our input size is towards infinity, aka very large.

Complexities

There are two types of complexities:

1. Time Complexity
2. Space Complexity

Time Complexity

Time complexity is the amount of time required to execute the algorithm for a given number of inputs. Approximate number of instructions to execute a particular algorithm.

Space Complexity

Extra space required by the algorithm except input.

Notations Used

1. Big O Notation
2. Big Omega Notation
3. Theta Notation

Big O Notation

f(n) = O(g(n)) g(n) is "tight" upper bound of f(n), i.e. f(n) can never go beyond g(n). f(n) = O(g(n)) iff f(n) <= g(n) for all n >= n0, and for some constant c > 0.

f(n) will never perform worse than g(n).

Algo1 => $O(n^2 + 3n + 4)$
Algo2 => $O(2n^2 + 4)$

-> Constants are ignored if it comes as addition, subtraction, multiplication, division. Algo1 => $O(n^2 + n)$ Algo2 => $O(n^2)$

-> Lower order terms are ignored in addition and subtraction. Algo1 => $O(n^2)$
Algo2 => $O(n^2)$

> Shortcut: Take the highest order term.

Big Omega Notation

f(n) Ω(g(n)) g(n) is "tight" lower bound of f(n), i.e. f(n) can never go below g(n). f(n) = O(g(n)) iff f(n) >= g(n) for all n >= n0, and for some constant c > 0.

f(n) will never perform better than g(n).

Theta Notation

Theta gives both upper and lower bound.