Lecture -01

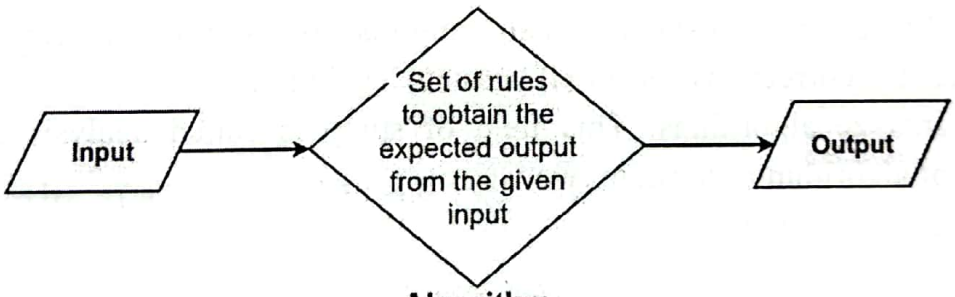
CSE 2201
Algorithm Analysis and Design

Reference Books

1. Fundamentals of Computer Algorithms
   By: Ellis Horowitz, Sartaj Sahni, and Sanguthrvar Rajasekaran
2. Introduction to Algorithms

   By: Thomas H cormen, Charles E. Leiserson, Ronal L. Rivest, and Cliford Stein
3. Algorithms in C++, parts 1-4: Fundamentals, Data Structure, Sorting, Searching (3rd edition)
   By: Rober Sedgewick
4. Beginning of Algorithms (Wrox beginning Guides)
   By: Simon Harris, and Janes Ross

Algorithms: An algorithm is a finite set of instructions that is followed, accomplishes a particular task.



Algorithm

Algorithms can be expressed as natural languages, programming languages, pseudocode, flowcharts and control tables. Natural language expressions are rare, as they are more ambiguous. Programming languages are normally used for expressing algorithms executed by a computer.

In addition, all algorithms must satisfy the following criteria:

1. Input: Zero or more quantities are externally supplied.
2. Output: At least one quantity is produced.

3. Definiteness: Each instruction is clear and unambiguous.
4. Finiteness: If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates often a finite number of steps.
5. Effectiveness: Every instruction must be very basic so that it carried out in principle.

The following are some examples of algorithms used in real life:

Traffic signals: Traffic signals use smart algorithms to manage traffic flow.

Sorting documents and papers: This is a great example of how algorithms can be used for various tasks and purposes, such as sorting files alphabetically, by word count, by date, or by any other specifications.

Searching for a book in the library: Finding a library book is like following an algorithm or a step-by-step plan.

The study of algorithm includes many important and active areas of research. There are four district areas of study one can identify.

1. How to devise algorithms: Creating an algorithm is an art which may never be fully automated study various design technique.
2. How to validate algorithms: Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs.
3. How to analyze algorithms: This field of study is called analysis of algorithms. Analysis of algorithm perform in two ways:
    - Time Complexity
    - Space Complexity
4. How to a test a program: Testing a program consists of two phases: Debugging and profiling (performance measurement)

Example:

```
Algorithm Max (A, n) //A is an array of size n
{
  Result= A [i]
  for   i = 2 to n do
  if (A [i]> Result) then Result = A[i]
  return Result ;
}
```

Performance Analysis (Priori estimates): Develop skill for making evaluative judgements about algorithms. Performance analysis is the priori estimates that is apply after developing the algorithm.

1. Does it do what we want it to do?
2. Does it work correctly according to the original specifications of the task?
3. Is there documentation that describes how to use it and how it works.
4. Are procedures created in such a way that they perform logical sub-function?
5. Is the code readable.

Performance measurement (Posteriori estimates/testing): There are other criteria for judging algorithms that have a more direct relationship to performance. These have to do with their computing time and storage requirements.

- Space complexity: The space complexity of an algorithm is the amount of memory it needs to run to completion.
- Time complexity: The time complexity of an algorithm is the amount of computing time it needs to run to completion. Performance evaluation can be loosely divided into two major phases.

Example: Space Complexity

Algorithm abc(a,b,c)

{

return a+b+b+c+(a+b+c)/(a+b) + 4.0

}

Sum ⟶ Computes $\sum_{i=1}^{n} a[i]$

Rsum ⟶ $\sum_{i=1}^{n} a[i]$

The space needed by each of these algorithms is seen to be the sum of the following components:

1. A fixed part that is independent of the characteristics (number, size) of the inputs and outputs.
    - Instruction space (space for the code) byte code

- Space for simple variables.
- Fixed size component variables.
- Space for constants

2. A variable part that consists of the space needed by components variables whose size is dependent on the particular problem instance being solved.
   - The referenced variable
   - The recursion stack

The Space Complexity, S(p)= C + Sp (instance)

Algorithm Sum(a,n) n= number of elements to be sum

{ S=0.0 ⟶ (1)

for i= 1 to n do ⟶ (n+1)

S = S + a [i] ⟶ (n)

n words

returns S

}

Time Complexity: T(P)= (2n+3)

Space complexity depends: 1.The number of elements to be summed.

per number need 1 word

integer

Sum (n) ≥(n+3)

1 word for n

1 word for i

1 word for S

Algorithm Rsum (a,n)

{

If ($n \leq 0$) then return 0.0

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

Space need: stack, the local variable, and return address.

Depth of recursion: (n+1).....S(p)>=3(n+1)

Time complexity: Time= Compile time (C) + Run (executing time) tp

↳ Not depend on instance characteristics (Actually it is the total number of comparisons)

C= Independent of the problem instance

tp= depends on the problem instance

T(p)= C + tp (instance)

How to measure T(p)?

- Measure Experimentally → Using a stop watch → T(P) obtained in see/msecs
- Count program steps, T(p) obtained as a step count.
- Arithmetic operations, (+,-,×), data movement control, decision making (if, while), comparisons.
- Fixed part is usually ignored, only the variable part tp = is measure →

Step count: what is step count?

- a+b+b+c + (a+b)/(a-b), 1 step
- comments, zero step
- while (<expr>) do, step count is equal to the number of times is <expr> executed
- for i=<expr> to <expr> do, step count equal to number of times <expr> is checked.

Algorithm Sum (a[],n,m)

{ for i=1 to n do	n+1
For j =1 to m do	n(m+1)
S= S +a [i][j]	nm
return S	1

$T(p) = (2nm+2n+2)$

Performance measurement: which is better → $T(p1)= n+1$ on $T(p2)=n^2+5$

$T(p1)= \log (n^2+1)/n$, $T(p2)=n^n(n\log n)/n^2$

Complex step count functions are difficult to compare

For complexity value of growth of time and space complexity functions is easy and sufficient.

Example Algorithm: MIN (a[1],……. a[n])

```
m ⟵ a[1];
for i ⟵ 2 to n
if a [i]<m
then m ⟵ a [i]
```

Running time: The No of primitive operations executed before terminations.
T(n) = 1[first step] + (n) for loop + (n-1) (>7 quantities) + (n-1) (the assignment in then) = (3n-1)

Typical running time functions:

- **1** (constant running time): Instructions are executed once or few times.
- **Log(n)** (logarithmic): A big problem is solved by dividing the original problem is smaller sizes by a constant fraction at each step.
- **N (linear):** A small amount of processing is done on each input element.
- **N log N:** A problem is solved by dividing it into smaller problems solving them independently and combining the solution.
- **N^2 (quadratic):** Typical for algorithms that process all pairs of data items (double nested loops)
- **N^3 (cubic):** Processing of triples of data (triple nested loops).
- **N^K(polynomial)**
- **2^K(Exponential:** Few exponential algorithms are appropriate for practical use.

Why faster Algorithms

$f(n) = n$

$f(n) = \log n$

$f(n) = n \log n$

$f(n) = n^2$

$f(n) = n^3$

$f(n) = 2^n$

Algorithm Efficiency VS Speed: Example:

Sorting n numbers=sort 10^6 numbers.

Friend's compute = 10^9 instruction/second

Friend's algorithms= $2n^2$ instructions

Your compute = 10^7 instruction/second

Your algorithm =50nlog n instructions,

$$\text{Your friend} = \frac{2\times(10^6)^2 \text{ instruction}}{10^9 \text{ instruction/second}} = 2000 \text{ second}$$

$$\text{You} = \frac{2\times(10^6)\log 10^6 \text{ instructions}}{10^7 \text{ instruction/second}} = 1000 \text{ second}$$

Discussion: Performance Analysis

Problems and algorithms to solve this problem: Sorting data

Example: Sorting data in ascending order

Algorithms: Which is the best algorithm for the problem? How do we judge?

Lecture -02

CSE 2201

Algorithm Analysis and Design

Why study Algorithms

Necessary in any computer programming problem

- Improve algorithm efficiency run taste process more data, do something that would otherwise be impossible.
- Solve problems of significantly large data size.
- Technology can only improve things by a constant factor.
- Compare algorithms
- Algorithms as a field of study: learn about a standard set of algorithms, new discoveries arise, numerous application areas, learn techniques of algorithm design and analysis.

Application

- Multimedia: CD player, DVD, MP3, JPG, Diva, HDTV
- Internet: Packet routing, data retrieval (Google)
- Communication: Cell phones, e- commerce.
- Computers: Circuit layout, file systems
- Science: Human genome
- Transportation: Airline crow scheduling, UPS deliveries

Roadmap

Different problems

- Sorting
- Searching
- String processing
- Graph problems
- Geometric problems
- Numerical problems

Different Design problems

1. **Greedy Method:** In the greedy method, at each step, a decision is made to choose the *local optimum*, without thinking about the future consequences. **Example:** Fractional Knapsack, Activity Selection.
2. **Divide and Conquer:** The Divide and Conquer strategy involves dividing the problem into sub-problem, recursively solving them, and then recombining them for the final answer. **Example:** Merge sort, Quicksort.
3. **Dynamic Programming:** The approach of Dynamic programming is similar to divide and conquer. The difference is that whenever we have recursive function calls with the same result, instead of calling them again we try to store the result in a data structure in the form of a table and retrieve the results from the table. Thus, the overall time complexity is reduced. "Dynamic" means we dynamically decide, whether to call a function or retrieve values from the table. **Example:** 0-1 Knapsack, subset-sum problem.
4. **Linear Programming:** In Linear Programming, there are inequalities in terms of inputs and maximizing or minimizing some linear functions of inputs. **Example:** Maximum flow of Directed Graph
5. **Reduction (Transform and Conquer):** In this method, we solve a difficult problem by transforming it into a known problem for which we have an optimal solution. Basically, the goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms. **Example:** Selection algorithm for finding the median in a list involves first sorting the list and then finding out the middle element in the sorted list. These techniques are also called transform and conquer.
6. **Backtracking:** This technique is very useful in solving combinatorial problems that have *a single unique solution*. Where we have to find the correct combination of steps that lead to fulfillment of the task. Such problems have multiple stages and there are multiple options at each stage. This approach is based on exploring each available option at every stage one-by-one. While exploring an option if a point is reached that doesn't seem to lead to the solution, the program control backtracks one step, and starts exploring the next option. In this way, the program explores all possible course of actions and finds the route that leads to the solution. **Example:** N-queen problem, maize problem.
7. **Branch and Bound:** This technique is very useful in solving combinatorial optimization problem that have *multiple solutions* and we are interested in find the most optimum solution. In this approach, the entire solution space is represented in the form of a state space tree. As the program progresses each state combination is explored, and the previous solution is replaced by new one if it is not the optimal than the current solution. **Example:** Job sequencing, Travelling salesman problem.
8. Incremental
9. Randomized/probabilistic.