CA4CRT10 – Design and Analysis of Algorithm (Core)

Unit I

Introduction

The word algorithm comes from the name of a Persian author, Abu Ja'far Mohammed ibn Musa al Khowarizmi (c.825A.D.), who wrote a textbook on mathematics. This word has taken on a special significance in computer science, where "algorithm" has come to refer to a method that can be used by a computer for the solution of a problem. This is what makes algorithm different from words such as process, technique, or method.

Definition of Algorithm

Definition1.1 [Algorithm]: An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. 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 instructions of an algorithm, then for All cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness.** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in ccriterion3; it also must be feasible.

An algorithm is composed of a finite set of steps, each of which may require one or more operations. The possibility of a computer carrying out these operations necessitates that certain constraints be placed on the type of operations an algorithm can include.

Criteria1and 2 require that an algorithm produce one or more *outputs* and have zero or more *inputs* that are externally supplied. According to criterion 3, each operation must be *definite*, meaning that it must be perfectly clear what should be done. Directions such as "add 6 or 7 to *x*" or "compute 5/0"are not permitted because it is not clear which of the two possibilities should be done or what the result is.

The fourth criterion for algorithms we assume that they terminate after a finite number of operations. A related consideration is that the time for termination should be reasonably short. For example, an algorithm could be devised that decides whether any given position in the game of chess is a winning position. The algorithm works by examining all possible moves and counter moves that could be made from the starting position. The difficulty with this algorithm is that even using the most modern computers, it may take billions of years to make the decision. We must be very concerned with analyzing the efficiency of each of our algorithms.

Criterion 5 requires that each operation be effective; each step must be Such that it can, at least in principle, be done by a person using pencil and paper in a finite amount of time. Performing arithmetic on integers is an example of an effective operation, but arithmetic with real numbers is

not, since some values may be expressible only by infinitely long decimal expansion. Adding two such numbers would violate the effectiveness property.

Algorithms that are definite and effective are also called computational *procedures.* One important example of computational procedure is the operating system of a digital computer. This procedure is designed to control the execution of jobs, n such a way that when no jobs are available, it does not terminate but continues in a waiting state until a new job is entered. Though computational procedures include important examples such as this one, we restrict our study to computational procedures that always terminate.

To help us achieve the criterion of definiteness, algorithms are written in a programming language. Such languages are designed so that each legitimate sentence has a unique meaning. A program is the expression of an algorithm in a programming language. Sometimes words such as procedure, function, and subroutine are used synonymously for program.

Algorithm Design Techniques

- **Divide and Conquer Method**

In the divide and conquer approach, the problem is divided into several small sub-problems. Then the sub-problems are solved recursively and combined to get the solution of the original problem. The divide and conquer approach involves the following steps at each level −

- **Divide** − The original problem is divided into sub-problems.
- **Conquer** − The sub-problems are solved recursively.
- **Combine** − The solutions of the sub-problems are combined together to get the solution of the original problem.

The divide and conquer approach is applied in the following algorithms −

- Binary search
- Quick sort
- Merge sort
- Integer multiplication
- Matrix inversion
- Matrix multiplication

- **Greedy Method**

In greedy algorithm of optimizing solution, the best solution is chosen at any moment. A greedy algorithm is very easy to apply to complex problems. It decides which step will provide the most accurate solution in the next step. This algorithm is a called greedy because when the optimal solution to the smaller instance is provided, the algorithm does not consider the total program as a whole. Once a solution is considered, the greedy algorithm never considers the same solution again.

A greedy algorithm works recursively creating a group of objects from the smallest possible component parts. Recursion is a procedure to solve a problem in which the solution to a specific problem is dependent on the solution of the smaller instance of that problem.

- **Dynamic Programming**

Dynamic programming is an optimization technique, which divides the problem into smaller sub-problems and after solving each sub-problem, dynamic programming combines all the solutions to get ultimate solution. Unlike divide and conquer method, dynamic programming reuses the solution to the sub-problems many times.

Recursive algorithm for Fibonacci Series is an example of dynamic programming.

- **Backtracking Algorithm**

Backtracking is an optimization technique to solve combinational problems. It is applied to both programmatic and real-life problems. Eight queen problem, Sudoku puzzle and going through a maze are popular examples where backtracking algorithm is used.

In backtracking, we start with a possible solution, which satisfies all the required conditions. Then we move to the next level and if that level does not produce a satisfactory solution, we return one level back and start with a new option.

- **Branch and Bound**

A branch and bound algorithm is an optimization technique to get an optimal solution to the problem. It looks for the best solution for a given problem in the entire space of the solution. The bounds in the function to be optimized are merged with the value of the latest best solution. It allows the algorithm to find parts of the solution space completely.

The purpose of a branch and bound search is to maintain the lowest-cost path to a target. Once a solution is found, it can keep improving the solution. Branch and bound search is implemented in depth-bounded search and depth–first search.

- **Linear Programming**

Linear programming describes a wide class of optimization job where both the optimization criterion and the constraints are linear functions. It is a technique to get the best outcome like maximum profit, shortest path, or lowest cost.

In this programming, we have a set of variables and we have to assign absolute values to them to satisfy a set of linear equations and to maximize or minimize a given linear objective function.

Algorithm Analysis

The study of algorithms includes many important and active areas of research. There are four distinct areas of study and they are specified below:

1. *How to devise algorithms* - Creating an algorithm is an art which may never be fully automated. A major goal is to study various design techniques that have proven to be useful in that they have often yielded good algorithm. By mastering these design strategies, it will become easier to devise new and useful algorithms. Dynamic programming is one such technique. Some of the techniques are especially useful in fields other than computer science such as operations research and electrical engineering. All of the approaches we consider have applications in a variety of areas including computer science. Other some important design techniques are linear, nonlinear, and integer programming.

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. We refer to this process as *algorithm validation.* The purpose of the validation is to assure us that this algorithm will work correctly independently of the issues concerning the programming language it will eventually be written in. Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is referred to as program proving or sometimes as *program verification*.

A proof of correctness requires that the solution be stated in two forms. One form is usually as a program which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in the predicate calculus. The second form is called a *specification*, and this may also be expressed in the predicate calculus. A proof consists of showing that these two forms are equivalent in that for every given legal input, they describe the same output. A complete proof of program correctness requires that each statement of the programming language be precisely defined and all basic operations be proved correct. All these details may cause a proof to be very much longer than the program.

3. *How to analyze algorithms* - This field of study is called analysis of algorithms. As an algorithm is executed, it uses the computer's central processing unit (CPU) to perform operations and its memory (both immediate and auxiliary) to hold the program and data. Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage an algorithm requires. This is a challenging area which sometimes requires great mathematical skill. An important result of this study is that it allows to make quantitative judgments about the value of one algorithm over another. Another result is that it allows to predict whether the software will meet any efficiency constraints that exist. Questions such as how well does an algorithm performs in the best case, in the worst case, or on the average are typical.

4. *How to test a program* – Testing a program consists of two phases: Debugging and profiling (or performance measurement).Debugging is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them. However, as E. Dijkstra has pointed out, "debugging can only point to the presence f errors, but not to their absence". In cases in which it cannot verify the correctness of output on sample data, the following strategy can be employed:

Let more than one programmer develop programs for the same problem, and compare the outputs produced by these programs. If the outputs match, then there is a good chance that they are correct. A proof of correctness is much more valuable than a thousand tests (if that proof is correct), since it guarantees that the program will work correctly for all possible inputs. *Profiling* or *performance measurement* is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results. These timing figures are useful in that they may confirm a previously done analysis and point out logical places to perform useful optimization.

Performance Analysis

There are many criteria upon which we can judge an algorithm. For instance:

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

These criteria are all vitally important when it comes to writing software, most especially for large systems. 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.

Definition 1.2 [Space/Time complexity]: The space complexity of an algorithm is the amount of memory it needs to run to completion. The time complexity of an algorithm is the amount of computer time it needs to run to completion.

Performance evaluation can be loosely divided into two major phases: (1) a priori estimates and (2) a posterior testing. We refer to these as performance analysis and performance measurement respectively.

- **Space Complexity**

Algorithm *abc* computes a+ b+b*c+(a+ b-c)/(a+b)+4.0; Algorithm *Sum* computes $\sum_{i=1}^{n} a[i]$ iteratively, where the a[i]'s are real numbers; and *RSum* is a recursive algorithm that computes $\sum_{i=1}^{n} a[i]$.

```
float abc (float a, float b, float c)
{
        return a + b + b * c+ (a + b -c)/(a+ b) + 4.0;
}


float Sum(float a[], int n)
{
        float s = 0.0;
        for(int i=1; i<=n; i++)
                s += a[i];
        return s;
 }
```

Recursive function

```
float RSum(float a[], int n)
{
        if (n <0)  return (0.0);
        else return RSum (a, n - 1)+ a[n];
 }
```

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 (e.g., number, size) of the inputs and outputs. This part typically includes the instruction space (i.e., space for the code), space for simple variables and fixed-size component variables (also called aggregate),space for constants, and soon.

2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics and the recursion stack space (in so far as this space depends on the instance characteristics).

The space requirement S (P) of any algorithm *P* may therefore be written as $S(P)= c+S_p$(instance characteristics) where *c* is a constant.

When analyzing the space complexity of an algorithm, we concentrate solely one estimating S_p (instance characteristics). For any given problem we need first to determine which instance characteristics to use to measure the space requirements. This is very problem specific, and we resort to examples to illustrate the various possibilities. Generally speaking our choices are limited to quantities related to the number and magnitude of the inputs to and outputs from the algorithm. At times, more complex measures of the interrelationship among the data items are used.

- **Time Complexity**

The time T(P) taken by a program P is the sum of the compile time and the run (or execution)time. The compile time does not depend on the Instance characteristics. Also, we may assume that a compiled program will be run several times without recompilation. Consequently, we concern ourselves with just the run time of a program. This run time is denoted by t_P (instance characteristics).

Because many of the factors t_P depends on are not known at the time a program is conceived it, is reasonable to attempt only to estimate t_P. If we knew the characteristics of the compile to be used, we could proceed to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores, and soon, that would be made by the code for P. So, we could obtain an expression for $t_P(n)$ of the form

$t_P(n) = c_aADD(n) + c_sSUB(n) + c_mMUL(n)+ c_dDIV(n) + ...$

where *n* denotes the instance characteristics, and c_a, c_s, c_m, c_d, and so on, respectively, denote the time needed for an addition, subtraction, multiplication, division, and so on, and ADD, SUB, MUL, DIV, and so on, are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and so on, that are performed when the code for *P* is used on an instance with characteristic *n*.

So, the time complexity is the number of operations an algorithm performs to complete its task (considering that each operation takes the same amount of time). The algorithm that performs the task in the smallest number of operations is considered the most efficient one in terms of the time complexity. However, the space and time complexity are also affected by factors such as the operating system and hardware.

Example:

Compare two different algorithms, which are used to solve a particular problem, the problem is searching an element in an array (the array is sorted in ascending order). To solve this problem two algorithms are used:

1. Linear Search.
2. Binary Search.

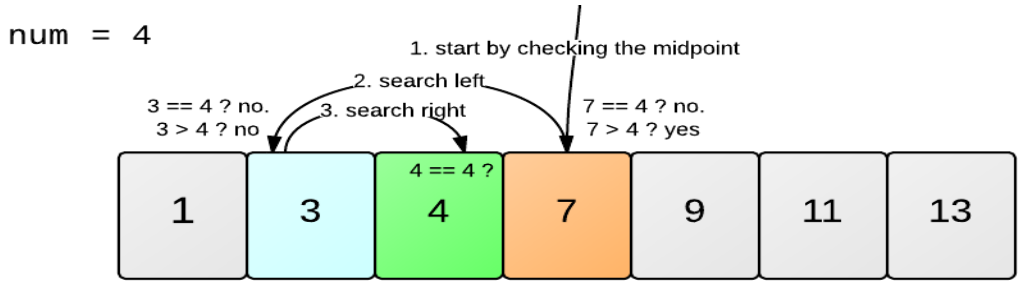
The array contains ten elements, and to find the number 10 in the array.

```
const int array [] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

const int search_digit = 10;
```

Linear search algorithm will compare each element of the array to the **search_digit.** When it finds the **search_digit** in the array, it will return **true**. Now let's count the number of operations it performs. Here, the answer is 10 (since it compares every element of the array). So, Linear search uses ten operations to find the given element. These are the maximum number of operations for this array; in the case of Linear search, this is also known as the *worst case* of an algorithm.

Binary search algorithm first compares **search_digit** with the middle element of the array, that is 5. Now since 5 is less than 10, then we will start looking for the **search_digit** in the array elements greater than 5, in the same way until we get the desired element 10.



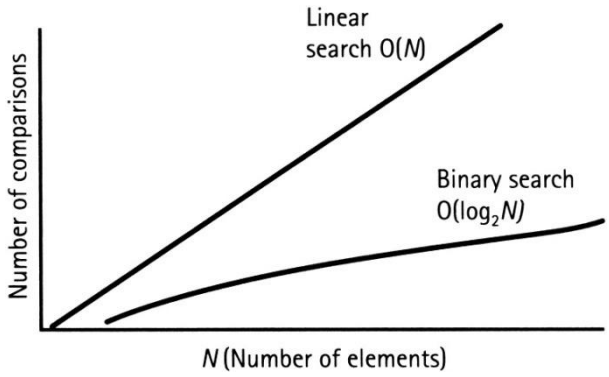
While apply this logic now try to count the number of operations binary search took to find the desired element. It took approximately four operations. Now, this was the worst case for binary search. This shows that there is a logarithmic relation between the number of operations performed and the total size of the array.

Number of operations = log (10) = 4 (approx) *for base 2.*

We can generalize this result for Binary search as, for an array of size **n**, the number of operations performed by the Binary Search is: **log(n)**

The Big O Notation

An array of size **n**, Linear search will perform **n** operations to complete the search. On the other hand, Binary search performed **log(n)** number of operations (both for their worst cases). We can represent this as a graph (**x-axis**: number of elements, **y-axis**: number of operations).



It is quite clear from the figure that the rate by which the complexity increases for Linear search is much faster than that for binary search. When we analyse an algorithm, we use a notation to represent its time complexity and that notation is Big O notation.

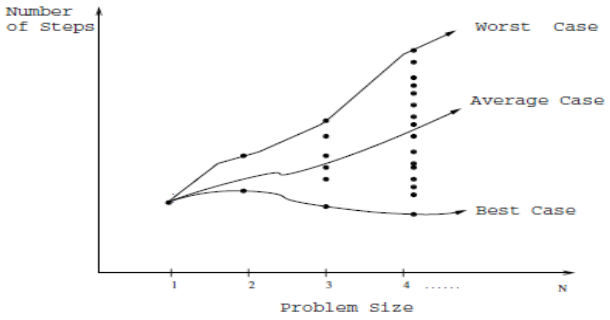
For Example: time complexity for Linear search can be represented as **O(n)** and **O(log n)** for Binary search (where, **n** and **log(n)** are the number of operations). The Time complexity or Big O notations for some popular algorithms are listed below:

1. Binary Search: O(log n)
2. Linear Search: O(n)
3. Quick Sort: O(n * log n)
4. Selection Sort: O(n * n)
5. Travelling Salesperson: O(n!)

Best, Worst, and Average-Case Complexity

Using the RAM model of computation, we can count how many steps our algorithm will take on any given input instance by simply executing it on the given input. However, to really understand how good or bad an algorithm is, we must know how it works over *all* instances.

To understand the notions of the *best*, *worst*, and *average-case* complexity, one must think about running an algorithm on all possible instances of data that can be fed to it. For the problem of sorting, the set of possible input instances consists of all the possible arrangements of all the possible numbers of keys. We can represent every input instance as a point on a graph, where the *x*-axis is the size of the problem (for sorting, the number of items to sort) and the *y*-axis is the number of steps taken by the algorithm on this instance. Here we assume, quite reasonably, that it doesn't matter what the values of the keys are, just how many of them there are and how they are ordered.



(Best, worst, and average-case complexity)

As shown in figure, these points naturally align themselves into columns, because only integers represent possible input sizes. Once we have these points, we can define three different functions over them:

- The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*. It represents the curve passing through the highest point of each column.
- The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*. It represents the curve passing through the lowest point of each column.

- Finally, the *average-case complexity* of the algorithm is the function defined by the average number of steps taken on any instance of size *n*.

∞ ∞ ∞