Design & Analysis of Algorithms

For

Computer Science

&

Information Technology

By

THE GATE© ACADEMY

A Forum of IIT / IISc Graduates

www.thegateacademy.com

✆080-40611000

Syllabus for Design & Analysis of Algorithms

Searching, Sorting, Hashing, Asymptotic Worst Case Time and Space Complexity, Algorithm Design Techniques, Greedy, Dynamic Programming and Divide-and-Conquer, Graph Search, Minimum Spanning Trees, Shortest Paths.

Previous Year GATE Papers and Analysis

GATE Papers with answer key

thegateacademy.com/gate-papers

Subject wise Weightage Analysis

thegateacademy.com/gate-syllabus

Contents

"Dream no small dreams for they have
no power to move the hearts of men."
...Goethe

CHAPTER 1 Algorithm Analysis

Learning Objectives

After reading this chapter, you will know:

1. Definition of Algorithm
2. Need for Analysis
3. Algorithm Analysis
4. Asymptotic Notation
5. Recurrence

Introduction

Once an algorithm is given for a problem and decided to be correct, then an important step is to determine how much in the way of resources, such as time or space, the algorithm will be required.

The analysis required to estimate use of these resources of an algorithm is generally a theoretical issue and therefore a formal framework is required. In this framework, we shall consider a normal computer as a model of computation that will have the standard repertoire of simple instructions like addition, multiplication, comparison and assignment, but unlike the case with real computer, it takes exactly one unit time to do anything (simple) and there are no fancy operations such as matrix inversion or sorting, that clearly cannot be done in one unit time. We always assume infinite memory also.

Definition of Algorithm

An Algorithm is a finite sequence 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 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 criteria 3; it also must be feasible.

Algorithm Development Stages

There are five phases of Algorithm Development Stages:

(i) **Requirements:** Make sure you that we understand the information that is given (the input) and what results that are produce (the output)

(ii) **Design:** For each object there will be some basic operations to perform on it. These operations already exist in the form of procedures and write an algorithm which solves the problem according to the requirements.

(iii) **Analysis:** Can we think of another algorithm? If so, write it down. Next, try to compare these two methods. It may already be possible to tell if one will be more desirable than the other. If you can't distinguish between the two, choose one to work on for now and we will return to the second version later.

(iv) **Refinement and Coding:** Modern approach suggests that all processing which is independent of the data representation be written out first.

(v) **Verification:** Verification consists of three distinct aspects;
(1) Program Proving, (2) Testing and (3) Debugging

Need for Analysis

Analysis is the study we perform in order to figure out what to do. It uses a high-level description of the algorithm instead of an implementation and it characterizes running time as function of the input size, n. it takes into account all possible inputs and allows us to evaluate the speed of an algorithm independent of the hardware/software environment.
The "Analysis" deals with performance evaluation (Complexity Analysis). One of the goals of analysis is to compare algorithm mainly in terms of running time but also in terms of other factors (e.g., memory requirements, programmer's effort etc.)

Algorithm Analysis

Types of Analysis/Behavior of Algorithm

Worst Case
- Provides an upper bound on running time
- An absolute guarantee that the algorithm would not run longer, no matter what the inputs are

Best Case
- Provides a lower bound on running time
- Input is the one for which the algorithm runs the fastest

Average Case
- Provides a prediction about running time
- Assumes that the input is random
- Lower Bound $\leq$ Running Time $\leq$ Upper Bound

The following two components need to be analyzed for determining algorithm efficiency. If we have more than one algorithms for solving a problem then we really need to consider these two before utilizing one of them.

Time Complexity

The time complexity of an algorithm is to find the time taken by an Algorithm to complete its execution. There two methods

1. **A Priori Analysis:** It is based on determining the order of the magnitude of the statement, construct or data structure. This method is independent of machine, programming language

and operating system. If algorithm has to be analyzed further in detail then each operation (Arithmetic, Relational and Logical) is considered to take 1 unit of time.

2. **Posteriori Testing:** In this method an algorithm is converted into a program using any programming language, executed on a particular machine. Algorithm's execution time is taken with the systems watch time. Result of this analysis will be dependent on the machine and language used. It will be real time.
   Running Time Complexity: The time required for running an algorithm.

Space Complexity

The amount of space required at run-time by an algorithm for solving a given problem.
In general these measurements are expressed in terms of asymptotic notations, like Big-Oh, Theta, Omega etc.

General Rules for Space Complexity Calculation

While computing space complexity we need to analyze whether the memory requirement is dependent on input size. Three things mainly need to be considered;

1) Maximum width of system stack which holds the activation records during the run time of the function.
2) Whether the size of activation record is dependent on input size.
3) And in each invocation how much memory is being used from heap.
   Then space complexity is as follows:
   Max-system-stack-width*((Size of the memory allocated on activation record which is dependent on n)+ (Size of the memory allocated from heap which is dependent on n))

Example: Consider the simple program fragment for analyzing space complexity.

```
int sum(int n)
{
int partialSum = 0;
for(int i = 0; i<n; i++)
partialSum =  partialSum + i * i * i;
return partialSum;
 }
```

Notice that the memory used by this program is absolutely independent of input size because this is non-recursive program and has only one activation record to be pushed on the system stack whose size is not going to change with n and also each invocation doesn't have any heap space requirement which is dependent on n**.** So whether n = 10, 20, 100, etc, the number of records to be pushed is O (1). Therefore Space Complexity is O (1).

Example: Consider the recursive C program which prints the null terminated string in the reverse order.

```
void printRev(char *str)
{
     if( *str == '\0') return;
```

```
        printRev(str+1);
        printf("%c",*str);
    }
```

Maximum width of the system stack is O(L) where L is string length. And the size of activation record is constant w.r.t length L. Therefore, space complexity is O(L).

General Rules for Running Time Calculations

Rule 1-for/While Loops

The running time of a for/while loop is = (The number of iteration perform)∗ (the running time of statements inside the loop).

Example: Consider the simple program fragment whose running time cost is O(n) where n is a positive integer.

```
int sum(int n)
{
int partialSum = 0;
for (int i = 1; i<= n; i++)
partialSum = partialSum + i * i * i;
return partialSum;
}
```

Lines 1 and 4 count one unit each. Line 3 counts for four units per time executed (two multiplications, one addition, and one assignment) and is executed n times, for a total of 4n units. Line 2 has hidden costs of initializing i, testing i <= n and incrementing i. The total cost of all these is 1 to initialize, n+1 for all the tests, and n for all the increments, which 2n+2.Thus, total cost of 6n+4, which is O(n).

Rule 2 –Nested loops

The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all loops.

As an example following program fragment is $O(n^2)$.

```
for( i = 0; i< n ; i++)
for( j = 0; j<n; j++)
k++;
```

Rule 3 - Consecutive Statements

Just add running time of all these statements.

As an example, the following program fragment, which has O(n) work followed by $O(n^2)$ work, is also $O(n^2)$.

```
for( i = 0; i< n ; i++)
    k++;
for( i = 0; i< n ; i++)
  for( j = 0; j<n; j++)
    k++;
```

Rule 4 – if/else

For the fragment

```
if (condition )
S1
else
S2
```

Running time of an if/else statement is never more than the running time of test plus the larger of the running times of S1 and S2.

Rule 5 – Recursive function

Deriving the recurrence relation and then solving it for getting the running time is always the better way than any other solution.

Example: Consider the code given below for printing null terminated string in reverse order.
Let T (n) be the running time of printRev ( ) for which string length is n. Then, $T(n-1)$ would represent running time of the same function which is given string of length n – 1.
Thus

$$\boxed{T(n) = T(n-1) + O(1)}$$

Logarithm Time Complexity

There are several algorithms, which require logn cost in worst case. Binary Search, Heap ADT operations, etc are examples of that.
Typically, an algorithm is O (logn) if it takes constant time to cut the problem size by a half. That's what exactly happens in binary search algorithm.

```
int binarySearch(int a[ ], int len, int x)
{
    int low = 0 , high = len – 1;
    while(low < = high)
    {
        int mid = (low + high)/2;
        if( a[mid] < x)
            low = mid + 1;
        else if(a[mid] > x)
            high = mid – 1;
        else
        return mid;
    }
    return NOT_FOUND;
}
```

At first look this algorithm gives the illusion of O (n) cost as we might think that since it has a while loop and that will always run for the entire length of the given array. A careful examination would expose that the loop will not run more than O (logn) times since in each iteration high or low getting adjusted such that the problem size decreases by half of the current size. The following recurrence relation can best represent the running time of binary search algorithm. T (n) = T (n/2) + 1, where T (n) be the running time for n input elements.

Look at another interesting code given below.

```
sum = 0;
for( i = 1; i< n ; 2 * i)
sum++;
```

The i variable values getting incremented in each iteration such that its becoming double of current hence certainly would not take much longer than logn to reach its value as n or more.

Amortized Analysis

An amortized analysis is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive. Even though we take averages, however, probability is not involved. An amortized analysis guarantees the average performance of each operation in the worst case.

Types of Amortize Analysis

There are three common amortization arguments:

- The Aggregate method.
- The Accounting method and
- The Potential method.

Online Algorithm

Definition: An algorithm that must process each input in turn, without detailed knowledge of future inputs. In computer science, an online algorithm is one that can process its input piece-by-piece in a serial fashion, i.e., in the order that the input is fed to the algorithm without having the entire input available from the start. In contrast, an offline algorithm is given the whole problem data from the beginning and is required to output an answer which solves the problem at hand (for example, selection sort requires that the entire list be given before if can sort it, while insertion sort doesn't). Since it does not know the whole input, an online algorithm is forced to make decisions that may later turn out not be optimal, and the study of online algorithms has focused on the quality of decision-making that is possible in this setting. Competitive analysis formalizes this idea by comparing the relative performance of an online and offline algorithm for the same problem instance.

A problem exemplifying the concepts of online algorithms is the Canadian Traveler Problem, The goal of this problem is to minimize the cost reaching a target in weighted graph where some of the edges are unreliable and may have been removed from the graph. However, that an edge has been removed (failed) is only revealed to the traveler when she/he reaches one of the edges endpoints.

The worst case for this problem is simply that all of the unreliable edges fail and the problem reduces to the usual shortest path problem. An alternative analysis of the problem can be made with the help of competitive analysis, for this method of analysis. The offline algorithm known in advance which edges will fail and the goal is to minimize the ratio between the online and offline algorithm performance. This problem is PSPACE-complete.