LECTURE 2

DESIGN AND ANALYSIS OF ALGORITHMS

Performance Analysis of an Algorithm

How to Design an Algorithm?

In order to write an algorithm, following things are needed as a pre-requisite:

1. The problem that is to be solved by this algorithm.
2. The constraints of the problem that must be considered while solving the problem.
3. The input to be taken to solve the problem.
4. The output to be expected when the problem is solved.
5. The solution to this problem, in the given constraints.

Then the algorithm is written with the help of above parameters such that it solves the problem.

Example:

Consider the example to add three numbers and print the sum.

- **Step 1: Fulfilling the pre-requisites**

  As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled.

  1. The problem that is to be solved by this algorithm: Add 3 numbers and prints their sum.
  2. The constraints of the problem that must be considered while solving the problem: The numbers must contain only digits and no other characters.
  3. The input to be taken to solve the problem: The three numbers to be added.
  4. The output to be expected when the problem is solved: The sum of the three numbers taken as the input.
  5. The solution to this problem, in the given constraints: The solution consists of adding the 3 numbers. It can be done with the help of '+' operator, or bit-wise, or any other method.

- **Step 2: Designing the algorithm**

  Now, design the algorithm with the help of above pre-requisites:

  Algorithm to add 3 numbers and print their sum:

  1. START
  2. Declare 3 integer variables num1, num2 and num3.
  3. Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
  4. Declare an integer variable sum to store the resultant sum of the 3 numbers.
  5. Add the 3 numbers and store the result in the variable sum.

6. Print the value of variable sum
7. END

- **Step 3: Testing the algorithm by implementing it.**

  In order to test the algorithm, implement it in any programming language like C language.

  Program:

```
// C program to add three numbers
#include <stdio.h>
int main()
{
   // Variables to take the input of the 3 numbers
   int num1, num2, num3;
   // Variable to store the resultant sum
   int sum;
   // Take the 3 numbers as input
   printf("Enter the 1st number: ");
   scanf("%d", &num1);
   printf("%d\n", num1);
   printf("Enter the 2nd number: ");
   scanf("%d", &num2);
   printf("%d\n", num2);
   printf("Enter the 3rd number: ");
   scanf("%d", &num3);
   printf("%d\n", num3);
   // Calculate the sum using + operator
   // and store it in variable sum
   sum = num1 + num2 + num3;
   // Print the sum
   printf("\nSum of the 3 numbers is: %d", sum);
   return 0;
}
```

  Output:

```
Enter the 1st number: 2
Enter the 2nd number: 3
Enter the 3rd number: 5

Sum of the 3 numbers is: 10
```

One problem, many solutions: The solution to an algorithm can be or cannot be more than one. It means that while implementing the algorithm, there can be more than one method to do implement it. For example, in the above problem to add 3 numbers, the sum can be calculated with many ways like:

- + operator
- Bit-wise operators
- . . etc

How to Analyse and Algorithms?

For a standard algorithm to be good, it must be efficient. Hence the efficiency of an algorithm must be checked and maintained. It can be in two stages:

Priori Analysis: "Priori" means "before". Hence Priori analysis means checking the algorithm before its implementation. In this, the algorithm is checked when it is written in the form of theoretical steps. This Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation. This is done usually by the algorithm designer.

Posterior Analysis: "Posterior" means "after". Hence Posterior analysis means checking the algorithm after its implementation. In this, the algorithm is checked by implementing it in any programming language and executing it. This analysis helps to get the actual and real analysis report about correctness, space required, time consumed etc.

What is Algorithm Complexity and How to find it?

An algorithm is defined as complex based on the amount of Space and Time it consumes. Hence the Complexity of an algorithm refers to the measure of the Time that it will need to execute and get the expected output, and the Space it will need to store all the data (input, temporary data and output). Hence these two factors define the efficiency of an algorithm.

The two factors of Algorithm Complexity are:

- **Time Factor:** Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor:** Space is measured by counting the maximum memory space required by the algorithm.

Therefore, the complexity of an algorithm can be divided into two types:

Space Complexity: Space complexity of an algorithm refers to the amount of memory that this algorithm requires to execute and get the result. This can be for inputs, temporary operations, or outputs.

How to calculate Space Complexity?

The space complexity of an algorithm is calculated by determining following 2 components:

- **Fixed Part:** This refers to the space that is definitely required by the algorithm. For example, input variables, output variables, program size, etc.

- **Variable Part:** This refers to the space that can be different based on the implementation of the algorithm. For example, temporary variables, dynamic memory allocation, recursion stack space, etc.

  The space requirement S(P) of any algorithm P may therefore be written as S(P)= c + Sp ,where c is a constant and Sp is variable part instance characteristics.

  These are some different examples for calculation of space complexity:

1. Algorithm abc(a,b,c, output)
   {
   Output= a + b + b * c+ (a+ b-c) / (a + b) + 4.0;
   return Output;
   }

   Here, the space needed by a, b, c and Output is independent of the variable part instance characteristics and one word is adequate to store the values of each of a, b, c, Output each. So, S_{abc} (variable part instance characteristics)=0
   .
   Only constant part space required for a,b,c and Ouput. So, complexity is 4 words (constant).

2. Algorithm Sum(a,n)
   {
   s:=0.0;
   for i :=1 to n do
   s:=s+ a[i];

return;
}

This algorithm is characterized by value of n, the number of elements to be summed. The space needed by n is one word, since it is of type integer. The space needed by a is the space needed by variables of type array of floating point numbers. This is at least n words, since a must be large enough to hold the n elements to be summed.

So, $S_{sum}(n)>=(n+3)$ (n for a[ ],one each for n, i and s).

3. Algorithm RSum(a,n)
{
if (n <0) then return 0.0;
else
return RSum (a,n-1)+ a[n];
}

As in the case of Sum, the instances are characterized by n. The recursion stack space includes space for the formal parameters, the local variables, and the return address. Assume that the return address requires only one word of memory. Each call to RSum requires at least three words (including space for the values of n, the return address, and a pointer to a[ ]). Since the depth of recursion is n + 1, the recursion stack space needed is >3(n+ 1)

Time Complexity: Time complexity of an algorithm refers to the amount of time that this algorithm requires to execute and get the result. This can be for normal operations, conditional if-else statements, loop statements, etc.

How to calculate Time Complexity?

The time complexity of an algorithm is also calculated by determining following 2 components:

- **Constant time part:** Any instruction that is executed just once comes in this part. For example, input, output, if-else, switch, etc.

- **Variable Time Part:** Any instruction that is executed more than once, say n times comes in this part. For example, loops, recursion, etc.

There are two different methods for calculation:

1. Count Method
2. Tabular Method

1. Count Method:

In this method, Count only the number of program steps. A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics.

Let us understand with examples:

```
Algorithm abc (a,b,c, output)
{
Output= a + b + b * c+ (a+ b-c)/(a + b) + 4.0;
count=count+1; // For expression
count=count+1; // For return
return Output;
}
```

Number of step count=2

```
Algorithm Sum(a,n)
{
s:=0.0;
count=count+1; //For initalization
for i :=1 to n do
{
count=count+1; //For for loop
s:=s+ a[i];
count=count+1; //For expression
}
count=count+1; //For last for statement
count=count+1; //For return
return;
}
```

Number of step count=2n+3

```
Algorithm RSum(a,n)
{
count=count+1; // For if statement

if (n <0) then
{
count=count+1; // For return statement
return 0.0;
}
else
{
count=count+1; // For return
return RSum (a,n-1)+ a[n];
}
}
```

Number of step counts=

$$\begin{cases} 2 & \text{if } n = 0 \\ 2+t_{RSum}(n-1) & \text{if } n>0 \end{cases}$$

2. Tabular Method

This method is to determine the step count of an algorithm is to build a table in which list the total number of steps contributed by each statement.

Statement	Total s/e (Steps per execution)	Frequency	Total steps
Algorithm Sum(a,n)	0		0
{	0		0
s:=0.0;	1	1	1
for i :=1 to n do	1	n+1	n+1
s:=s+a[i];	1	n	n
return;	1	1	1
}	0		0

Total number of steps count = 2n+3

Statement	Total s/e (Steps per execution)	Frequency		Total steps	
		n=0	n>0	n=0	n>0
Algorithm RSum(a,n)	0			0	0
{	0			0	0
if (n <0) then	1	1	1	1	1
return 0.0;	1	1		1	0
else	0			0	0
return RSum (a,n-1)+ a[n];	1+x	0	1	0	1+x
}	0			0	0
Total number of steps count=				**2**	**2+x**

References:

1. Computer algorithms, Ellis Horowitz, Rajasekaran S. and Sartaj Sahni, 1997, Computer Science Press.
2. **www.wikipedia.org**
3. www.tutorialspoint.com
4. www.geeksforgeeks.org
5. Introduction to Algorithms, Thomas H Cormen, Charles E Leiserson and Ronald L Rivest: 1990, TMH