COURSE HANDOUT

Course Code	ACSC13
Course Name	Design and Analysis of Algorithms
Class / Semester	IV SEM
Section	A-SECTION
Name of the Department	CSE-CYBER SECURITY
Employee ID	IARE11023
Employee Name	Dr K RAJENDRA PRASAD
Topic Covered	Complexity of an Algorithm
Course Outcome/s	Analyse the complexity of algorithm with time and space requirement values.
Handout Number	6
Date	27 March, 2023

Content about topic covered: Complexity of an Algorithm

Complexity of an Algorithm:

Explanation about time and space complexity with examples.

Space complexity: The space complexity of an algorithm is the amount of memory it needs to run to completion.

The space needed by each of these algorithms is the sum of the following components.

1. A ***fixed component*** that is unaffected by the features (such as quantity and size) of the inputs and outputs. This section normally consists of the instruction space (also known as the space for the code), the space for simple variables and constants, and so on

2. A ***variable part*** that consists of the recursion stack space (to the extent that this space depends on the instance characteristics), the space needed by referenced variables (to the extent that this depends on instance characteristics), and the space needed by component variables, the size of which depends on the specific problem instance being solved.

So, The space requirement S(P)of any algorithm P may therefore be written as

$$S(P)= C + Sp(\text{instance characteristics}),\text{where C is a constant}$$

Eg 1.

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

}

The particular values of a, b, and c define the problem instance. We can see that the space required by abc is independent of the instance characteristics by making the assumption that one word is sufficient to hold the values of each of a, b, and c as well as the result.

Sp(instance characteristics) = 0.

Eg 2.

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

The amount of elements that must be summed, or n, defines the issue instances. Given that it is an integer, n requires one word of space. The amount of space required by variables of the type array of floating point numbers is the same as the amount required by a. Since a needs to be substantial enough to accommodate the n elements to be added, this is at least n words.

Eg 3.

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

The formal parameters, local variables, and return address are all included in the recursion stack area. Suppose that all that is needed to store the return address is a single word. A minimum of three words are needed for each call to RSum (containing storage for the n-values, the return address, and a pointer to a[]). Recursion stack space is 3(n+1) because the depth of recursion is n + 1.

Time complexity:

The time complexity of an algorithm is the amount of computer time it needs to run to completion.

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.

Therefore, the time complexity of an algorithm is determined by the number of steps it requires to compute the function for which it was constructed.

There are two methods we can figure out how many steps a program needs to take in order to solve a specific problem.

1. Introduction of global variable called count.
2. To build a table in which we list the total number of steps contributed by each statement.

Examples

1. Count method:

Eg 1.

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

The time complexity is O(1).

Eg 2.

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

So, the total steps required = 2 n + 3;

Time complexity = O(n)

Eg 3. Algorithm RSum(a,n)

```
{
    count = count +1;        // for the if condition
    if (n ≤ 0) then
    {
        count = count +1;        //for the return statement
        return 0.0;
    }
    else
    {
        count = count +1;        //function invocation
        return RSum (a,n-1)+ a[n];
    }
}
```

When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count, for example

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

$$\begin{aligned} t_{Rsum}(n) &= 2 + t_{Rsum}(n-1) \\ &= 2 + 2 + t_{Rsum}(n-2) \\ &= 2(2) + t_{Rsum}(n-2) \\ &= 2(2) + 2 + t_{Rsum}(n-3) \\ &= 2(3) + t_{Rsum}(n-3) \\ &\vdots \\ &= 2(n) + t_{Rsum}(n-n) \\ &= 2(n) + t_{Rsum}(0) \\ &= 2n + 2 \end{aligned}$$

So, the time complexity = O(n).

2. Table Method

An algorithm's step count is computed by first calculating the total number of times (frequency) that each statement is executed overall, as well as the number of steps required for its execution (s/e). The s/e of a statement is the amount by which the count differs from the original value as a result of the statement's execution. These two numbers are multiplied to determine the contribution of each statement as a whole. The total number of steps for the entire algorithm can be found by totalling the from each statement.

Eg 1. Sum of elements of an array.

Statement	s/e	Frequency	Total steps
1. Algorithm Sum(a, n)	0	-	0
2. {	0	-	0
3. s:=0.0;	1	1	1
4. for i :=1 to n do	1	n + 1	n +1
5. s:=s+ a[i];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2 n + 3

Time complexity = O (n).

Eg 2. Matrix multiplication.

Statement	s/e	Frequency	Total steps
1. Algorithm Add (a, b, c, m, n)	0	-	0
2. {	0	-	0
3. for i :=1 to m do	1	m +1	m +1
4. for j :=1 to n do	1	m(n+1)	mn + m
5. C[i,j] := a[i,j] +b[i,j];	1	mn	mn
6.	0	-	0
Total			2mn +2m +1

Time complexity = O (mn).