Analysis of Algorithm

Muhammad Athar email id: athar@northern.edu.pk WhatsApp# 0333-5077664

(Week 03) Lectures 05 & 06

Objectives: Learning objectives of these lectures are

- How to Compute Running Time of the Algorithms?
- What are the Basic Operations?
- How to Compute Basic Operations in an algorithm?
- Rules to compute the operations for different statements

Text Book & Resources:

1. Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, The MIT Press; 3rdEdition (2009). ISBN-10: 0262033844
2. Introduction to the Design and Analysis of Algorithms by Anany Levitin, Addison Wesley; 2ndEdition (2006). ISBN-10: 0321358287
3. Algorithms in C++ by Robert Sedgewick (1999). ASIN: B006UR4BJS
4. Algorithms in Java by Robert Sedgewick, Addison-Wesley Professional; 3rd Edition(2002). ISBN-10: 0201361205

Analysis of Algorithm

Muhammad Athar email id: athar@northern.edu.pk WhatsApp# 0333-5077664

How to Compute Running Time for Algorithm?

By inspecting the pseudo code, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

Example 1:

Algorithm *arrayMax*(*A*, *n*)

	# operations
currentMax ← *A*[0]	1
for (*i* =1; i<n; i++)	2*n*
(i=1 once, i<n n times, i++ (n-1) times)	
if *A*[*i*] > *currentMax* then	2(*n* – 1)
currentMax ← *A*[*i*]	2(*n* – 1)
return *currentMax*	1
Total	6*n* –2

Example 2 (Initializing array with zero):

Algorithm 1

	Cost
arr[0] ← 0;	c_1
arr[1] ← 0;	c_1
arr[2] ← 0;	c_1
...	...
arr[N-1] ← 0;	c_1

$= c_1+c_1+...+c_1$

$= c_1 \times N$

Algorithm 2

	Cost
for(i←0; i<N; i++)	c_2
arr[i] ← 0;	c_1

$= (N+1) \times c_2 + N \times c_1$

$= (c_2 + c_1) \times N + c_2$

Example 3:

Algorithm	***Cost***
sum = 0;	c_1
for(i=0; i<N; i++)	c_2
for(j=0; j<N; j++)	c_2
sum += arr[i][j];	c_3

$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2$

Basic Operations in Algorithms

Analysis of Algorithm

Muhammad Athar email id: athar@northern.edu.pk WhatsApp# 0333-5077664

An algorithm to solve a particular task employs some set of basic operations. When we estimate the amount of work done by an algorithm we usually do not consider all the steps such as e.g. initializing certain variables. Generally, the total number of steps is roughly proportional to the number of the basic operations. Thus, we are concerned mainly with the basic operations - how many times the basic operations have to be performed depending on the size of input.

Typical basic operations for some problems are the following:

Problem	Operation
Find x in an array	Comparison of x with an entry in the array
Multiplying two matrices with real entries	Multiplication of two real numbers
Sort an array of numbers	Comparison of two array entries plus moving elements in the array
Traverse a tree	Traverse an edge

The work done by an algorithm, i.e. its complexity, is determined by the number of the basic operations necessary to solve the problem.

Some algorithms are not dependent on the size of the input - the number of the operations they perform is fixed. Other algorithms depend on the size of the input, and these are the algorithms that might cause problems. Before implementing such an algorithm, we have to be sure that the algorithm will finish the job in reasonable time.

What is size of input? We need to choose some reasonable measure of size. Here are some examples:

Problem	Size of input
Find x in an array	The number of the elements in the array
Multiply two matrices	The dimensions of the matrices
Sort an array	The number of elements in the array
Traverse a binary tree	The number of nodes
Solve a system of linear equations	The number of equations, or the number of the unknowns, or both

Counting the number of operations

The core of the algorithm analysis: to find out how the number of the basic operations depends on the size of the input.

Rules to compute the operations for different statements

Sequential Statements: Just add the running time of the statements

Example: ***Algorithm 1***

Cost

arr[0] ← 0; c_1

arr[1] ← 0; c_1

arr[2] ← 0; c_1

... ...

arr[N-1] ← 0; c_1

$c_1+c_1+...+c_1 = c_1 \times N$

Iterative Statements

Iteration is at most the running time of the statements inside the loop (including tests) times the number of iterations.

Examples

The running time of ***for*** loop is at most the running time of the statements inside the loop times the number of iterations.

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

```
for( i = 0; i < n; i++)      // i = 0;    executed only once: 1
                             // i < n;    n + 1 times          n+1
                             // i++       n times              n
  // total time of the loop heading:
     // 1+ n+1+ n = 2n+2
        sum = sum + i;       // 2 operations ,       2n
        Total = 2n+2n+2=4n+2
```

Sometimes a loop may cause the if-else rule not to be applicable. Consider the following loop:

```
while (n > 0)
    if (n % 2 = = 0)
        print n
         n = n / 2
    else
        print n
        print n
         n = n – 1
```

The else-branch has more basic operations; therefore one may conclude that the loop is n times. However the if-branch dominates. For example if n is 60, then the sequence of n is: 60, 30, 15, 14, 7, 6, 3, 2, 1, and 0. Hence the loop is logarithmic.

Nested Loops

Muhammad Athar email id: athar@northern.edu.pk WhatsApp# 0333-5077664

Analyze these inside out. 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 size of all the loops. The total running time is the running time of the inside statements times the product of the sizes of all the loops.

Selection Statements

If then else

- if (condition) S_1 else S_2

  Running time of the test plus the larger of the running times of S_1 and S_2.

Example 1: (Search in unordered array)

```
for (i = 0; i < n; i++)
        if (a[i] = = x)
                return 1; // 1 means succeed
        else if(i= =n-1)
                return -1; // -1 means failure, the element is not found
```

The basic operation in this problem is comparison, so we are interested in how the number of comparisons depends on **n**.
Here we have a loop that runs at most **n** times:
If the element is not there, the algorithm needs **n** comparisons. If the element is at the end, we need **n** comparisons. If the element is somewhere in between, we need less than **n** comparisons.
In the **worst case** (element not there, or located at the end), we have **n** comparisons to make.
Hence the number of operations is **N**.
Else portion will be execute only once in case when number not found.
Here we will select if portion having larger running time.

Example 2:

Analysis of Algorithm

Muhammad Athar email id: athar@northern.edu.pk WhatsApp# 0333-5077664

If Statement: Take the complexity of the most expensive case :

```
char key;
   int[][] A = new int[5][5];
   int[][] B = new int[5][5];
   int[][] C = new int[5][5];
   ........
   if(key == '+')  {
      for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
           C[i][j] = A[i][j] + B[i][j];
   } // End of if block

   else if(key == 'x')
      C = matrixMult(A, B);

   else
      System.out.println("Error! Enter '+' or 'x'!");
```

n^2

n^3

1

Overall complexity

n^3

Switch Statement

Switch: Take the complexity of the most expensive case

```
char key;
int[] X = new int[5];
int[][] Y = new int[10][10];
........
switch(key)  {
    case 'a':
       for(int i = 0; i < X.length; i++)
         sum += X[i];
    break;
    case 'b':
       for(int i = 0; i < Y.length; j++)
         for(int j = 0; j < Y[0].length; j++)
              sum += Y[i][j];
    break;
  }  // End of switch block
```

→ n

→ n^2

Overall Complexity: n^2

<u>Function Calls</u>:

Analyzing from inside to out. If there are function calls, these must be analyzed first.