

Text:

1. Analysis of Algorithms

Why do we bother to analyze an algorithm?

- ✓ For some of us analyzing algorithms is an intellectual activity that is fun.
- ✓ Another reason is the challenge of being able to predict the future and even though we are narrowing our predictions to algorithms, it is gratifying when we succeed.
- ✓ A third reason is because computer science attracts many people who enjoy being efficiency experts. Analyzing algorithms gives these people a chance to exhibit their skills by devising new ways of doing the same task even faster. This tendency has a large payoff in computing where time means money and efficiency saves dollars.

As an algorithm is executed, it makes use of the computer's central processing unit (*CPU*) to perform operations and it uses the memory (both immediate and auxiliary) to hold the program and its data. Analysis of algorithms refers to the process of determining how much computing time and storage an algorithm will require. This area is a challenging one which sometimes requires great mathematical skill. One important result of this study is that it allows one to make quantitative judgments about the value of one algorithm over another. Another result is that it allows us to predict if our software will meet any efficiency constraints which may exist. Questions such as how well does an algorithm perform in the best case, in the worst case, or on the average are typical. For each algorithm which is presented here, an analysis will also be given.

Before we can talk about how to analyze an algorithm we need to make explicit our assumptions about the kind of computer we expect the algorithm to be executed on. The assumptions we make can have important consequences with respect to how fast a problem can be solved. Though formal models of machines do exist (e.g. Turing machines or Random Access Machines), for most of this course it will be sufficient to consider our computer as a "conventional" one. By this we mean that the instructions of a program are assumed to be carried out one at a time and the major cost of an algorithm depends upon the number of operations it requires. We assume that a random access memory is available which permits one to either access or store any element in a fixed amount of time.

We admit that there are reasons to believe that these assumptions may become outmoded with future generations of machines. Already computers such as ILLIAC IV or the CDC STAR exist and offer a high degree of parallelism in the manner in which a sequence of operations can be executed. This invalidates to some extent the measurement of an algorithm's cost by the summing of its logical operations. A second though somewhat more remote factor is the dramatic decrease in the cost of logic circuits (microprocessors) to the point where configurations of these processors cause the movement of data to be more expensive than the arithmetic and logical operations.

If these trends continue, a new theory of computation will be required. But until such machines become more pervasive the model of counting and summing logical operations on a sequential processor remains the most accurate predictor of performance and the one we will use.

Given an algorithm to be analyzed, we have two tasks at hand:

- i. The first task is to determine which operations are employed and what their relative costs are. These operations may include
 - ✓ the four basic arithmetic operations on integers: addition, subtraction, multiplication and division
 - ✓ arithmetic on floating point numbers
 - ✓ comparisons
 - ✓ assigning values to variables and
 - ✓ executing procedure calls.

These operations typically take no more than a fixed amount of time and so we say that their time is bounded by a constant. This is not true of all operations of a computer.

Some may be composed of an arbitrarily long sequence of more basic operations. For example, a comparison of two character strings may use a character compare instruction which may, in turn, use a shift and bit-compare instruction. The total time for the comparison of two strings will depend upon their lengths, while the time for each character compare is bounded by a constant.

- ii. The second task is to determine a sufficient number of data sets which cause the algorithm to exhibit all possible patterns of behavior. This is one of the important and creative tasks of algorithm analysis. It requires us to understand the workings of the algorithm well enough to concoct the data configurations which produce the best or worst or typical behavior. We will say more about this when we discuss particular algorithms.

In producing a complete analysis of the computing time of an algorithm, we distinguish between two phases: a priori analysis and a posteriori testing.

1.1. Priori analysis:

In a priori analysis we obtain a function (of some relevant parameters) which bounds the algorithm's computing time.

1.2. Posteriori testing:

In a posteriori testing we collect actual statistics about the algorithm's consumption of time and space, while it is executing (program).

Priori analysis	Posteriori testing
Algorithm	Program
Independent of Language	Language Dependent
Hardware independent	Hardware Dependent
Time & Space Functions	Watch time and bytes

Suppose there is the statement " $x \leftarrow x + y$ " somewhere in the middle of a program. We wish to determine the total time this statement will spend executing, given some initial state of input data.

This requires essentially two items of information,

1. the statement's frequency count (i.e. the number of times the statement will be executed) and
2. the time for one execution.

The product of these two numbers is the total time.

Since the time per execution depends on both the machine being used and the programming language together with its compiler, a priori analysis limits itself to determining the frequency count of each statement. This number can be determined directly from the algorithm, independent of the machine it will be executed on and the programming language the algorithm is written in.

For example consider the three program segments (a), (b), & (c):

$x \leftarrow x + y$	For $i \leftarrow 1$ to n do $x \leftarrow x + y$ repeat	For $i \leftarrow 1$ to n do For $i \leftarrow 1$ to n do $x \leftarrow x + y$ repeat repeat
(a)	(b)	(c)

For each segment we assume the statement " $x \leftarrow x + y$ " is contained within no other loop than what is already visible. Thus for segment (a) the frequency count of this statement is 1. For segment (b) the count is n and for segment (c) it is n^2 . These frequencies 1, n , & n^2 are said to be different, increasing orders of magnitude. An order of magnitude is a common notion with which we are all familiar; for example, walking,

bicycling, riding in a car and flying in an airplane represent increasing orders of magnitude with respect to the distance we can travel per hour.

In connection with algorithm analysis, the order of magnitude of a statement refers to its frequency of execution, while the order of magnitude of an algorithm refers to the sum of the frequencies of all of its statements. Given three algorithms for solving the same problem whose orders of magnitude are n , n^2 , and n^3 , naturally we will prefer the first since the second and third are progressively slower.

For example, if $n = 10$ then these algorithms will require 10, 100, and 1000 units of time to execute respectively (assuming all basic operations are of equal duration). Determining the order of magnitude of an algorithm is very important and producing an algorithm which is faster by an order of magnitude is a significant accomplishment. The a priori analysis of algorithms is concerned chiefly with order of magnitude determination. Fortunately, there is a convenient mathematical notation for dealing with this concept.

2. Time and Space Complexity

Sometimes, there are more than one way to solve a problem. We need to learn how to compare the performance of different algorithms and choose the best one to solve a particular problem. While analyzing an algorithm, we mostly consider time complexity and space complexity. Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Similarly, Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

Time and space complexity depends on lots of things like hardware, operating system, processors, etc. However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm.

Let's start with a simple example. Suppose you are given an *array A* and an integer x and you have to find if x exists in *array A*.

Simple solution to this problem is traverse the whole *array A* and check if the any element is equal to x .

```
for i : 1 to length of A
  if A[i] is equal to x      -----  $N*c$ 
    return TRUE
  else
    return FALSE          -----  $c$ 
```

Each of the operation in computer take approximately constant time. Let each operation takes c time. The number of lines of code executed is actually depends on the value of x . During analyses of algorithm, mostly we will consider worst case scenario, i.e., when x is not present in the array *A*. In the worst case, the *if* condition will run N times where N is the length of the array *A*. So in the worst case, total

execution time will be $(N*c + c)$. $N*c$ for the *if* condition and c for the return statement (ignoring some operations like assignment of i).

As we can see that the total time depends on the length of the array A . If the length of the array will increase the time of execution will also increase.

2.1. Order of growth

Order of growth is how the time of execution depends on the length of the input. In the above example, we can clearly see that the time of execution is linearly depends on the length of the array. Order of growth will help us to compute the running time with ease. We will ignore the lower order terms, since the lower order terms are relatively insignificant for large input. We use different notation to describe limiting behavior of a function.

2.1.1. Frequency Count Method:

One of the ways to calculate the running time complexity of an algorithm is frequency count method. In this method, we simply count the number of times an operation is carried out in our algorithm and add up the individual counts to calculate the overall time complexity of the algorithm. Let us see some examples to understand the concept.

Example 4.1:

	Algorithm Sum (A, n)	Time	Space
Step 1	Begin		A = n S = 1 n = 1 i = 1
Step 2	S <- 0	1	
Step 3	for i = 0 to n do	n+1	
Step 4	S = S + A[i]	N	
Step 5	Repeat		
Step 6	Return S	1	
Step 7	End		
		T(n)=2n+3	S(n)=n+3
		Degree(n) or O(n)	

Example 4.2:

	Algorithm Add (A, B, n)	Time	Space
Step 1	Begin		$A = n^2$ $B = n^2$ $C = n^2$ $n = 1$ $i = 1$ $j = 1$
Step 2	for i = 0 to n do	$n + 1$	
Step 3	for j = 0 to n do	$n(n + 1)$	
Step 4	$C[i,j] = A[i,j] + B [i,j]$	$n \times n$	
Step 5	Repeat		
Step 6	Repeat		
Step 7	End		
		$T(n)=2n^2+2n+1$	$S(n)= 3 n^2+1$
		Degree(n^2) or $o(n^2)$	

Example 4.3:

	Algorithm Multiply (A, B, n)	Time	Space
Step 1	Begin		$A = n^2$ $B = n^2$ $C = n^2$ $n = 1$ $i = 1$ $j = 1$ $k = 1$
Step 2	for i = 0 to n do	$n + 1$	
Step 3	for j = 0 to n do	$n \times (n + 1)$	
Step 4	$C[i,j]=0$	$n \times n$	
Step 5	for k = 0 to n do	$n \times n \times (n+1)$	
Step 6	$C[i,j]=C[i,j]+ A[i,k] * B [k,j]$	$n \times n \times n$	
Step 7	Repeat		
Step 8	Repeat		
Step 9	Repeat		
Step 10	End		
		$T(n)=2n^3+3n^2+2n+1$	$S(n)= 3 n^2+4$
		$o(n^3)$	$o(n^2)$

2.1.1.1. Analysis of For Loop:

Example 4.4:

```

for (i = n; i > 0 ; i --)
{
    Stmt;
}

```

n
n
O(n)

Example 4.5:

```

for (i = n; i > 0 ; i --)
{
    Stmt;
}

```

n
n
O(n)

Example 4.6:

```

for (i = 0; i < n ; i+2 )
{
    Stmt;
}

```

n/2
O(n)

Example 4.7:

```

for (i = 0; i < n ; i ++ )
{
    for (j = 0; j < i ; j ++ )
    {
        Stmt;
    }
}

```

i	j	Stmt
0	0	0
1	0	0
	1	1
2	0	0
	1	1
	2	1
3	0	0

	1	1	
	2	1	
	3	1	3
...	
n	0	0	
	1	1	
	..	1	
	n	1	n

$$\text{Total} = 1 + 2 + 3 + \dots + n = n(n+1)/2 = O(n^2)$$

Example 4.9:

```

for (i = 1; i < n ; i * 2 )
{
    Stmt;
}
i = 1, 2, 2^2, 2^3, 2^4, .... 2^k

```

Assume, $i \geq n$

$$2^k \geq n$$

$$\text{If, } 2^k = n$$

$$k = \log_2 n$$

$$O(\log_2 n)$$

2.1.1.2. Floor & Ceiling

For any real number x , we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read “the floor of x ”) and the least integer greater than or equal to x by $\lceil x \rceil$ (read “the ceiling of x ”). For all real x ,

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

For any integer n ,

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$$

and for any real number $x \geq 0$ and integers $a, b > 0$,

$$\left\lceil \frac{\lfloor x/a \rfloor}{b} \right\rceil = \left\lfloor \frac{x}{ab} \right\rfloor$$

$$\left\lfloor \frac{\lceil x/a \rceil}{b} \right\rfloor = \left\lceil \frac{x}{ab} \right\rceil$$

$$\left\lfloor \frac{a}{b} \right\rfloor \leq \frac{a + (b - 1)}{b}$$

$$\left\lceil \frac{a}{b} \right\rceil \geq \frac{a - (b - 1)}{b}$$

The floor function $f(x) = \lfloor x \rfloor$ is monotonically increasing, as is the ceiling function $f(x) = \lceil x \rceil$.

```

for (i = 1; i < n ; i * 2 )
{
    Stmt;
}
O(log2 n)

```

If $n = 8$; $i = 1, 2, 4$ (3 times) and $\log_2 8 = 3$

If $n = 10$; $i = 1, 2, 4, 8$ (4 times) but $\log_2 10 = 3.2$

Therefore, we should take ceil value $\lceil \log_2 n \rceil$

Example 4.11:

```

P=0
for (i = 1; i < n ; i = i * 2 )
{
    P ++;                               // log n
}
for (j = 1; j < P ; j = j * 2 )
{
    Stmt;                               // log P
}
P = log n
log P = log log n
=> O (log log n)

```

2.1.1.3. If and While Loops:

Example 4.13:

```

i = 0;           ----- 1
while (i < n)   ----- n + 1
{
    Stmt;       ----- n
    i++;        ----- n
}
                                     3n + 2
                                     O(n)

```

Example 4.14:

```

a = 1;
while (a < b)
{
    Stmt;
    a = a*b;
}

```

$$a = 1, 2, 2^2, 2^3, 2^4, \dots, 2^k$$

When $a = 2^k$, loop will terminate

$$\Rightarrow a \geq b$$

$$\Rightarrow 2^k \geq b$$

$$\text{If } 2^k = b, k = \log b$$

3. Designing Algorithms

The act of creating an algorithm is an art which may never be fully automated. Various design techniques exist which have proven to be useful in that they have often yielded good algorithms. By mastering these design strategies, it will become easier for us to devise new and useful algorithms. Some of these techniques may already be familiar, and some have been found to be so useful that books have been written about them. Some of the techniques are especially useful in fields other than computer science such as operations research and electrical engineering. Here, we intend to give an introduction to these many approaches to algorithm formulation. All of the approaches we consider have applications in a variety of areas including computer science. Some of the prominent techniques are:

- Divide and conquer
- Greedy Method
- Dynamic Programming
- Backtracking
- Branch & Bound
- Linear Programming