DESIGN AND ANALYSIS OF ALGORITHMS (CS-22203)
NERIST

INTRODUCTION

Definition of an algorithm:

By an algorithm, we mean a precise method usable by a computer for solving a problem. An algorithm is composed of a finite set of steps, each of which may require one or more operations. The following are the properties that an algorithm should possess:

- **Definiteness**. Each operation must be definite, i.e., it must be perfectly clear what must be done. E.g., 'Add 4 or 5 to x' is not a definite operation.

- **Effectiveness**. Each operation must be such that it can at least in principle be executed by a person using pencil and paper in a finite amount of time.

- **Input**. An algorithm may have zero or more inputs which are externally supplied.

- **Output**. An algorithm should produce one or more output.

- **Finiteness**. An algorithm should terminate after a finite amount of operations.

An algorithm that obeys all the above properties except the termination property is called a computational procedure.

A **program** is the expression of an algorithm in a programming language.

Aspects of the study of algorithms:

1) How to devise an algorithm – This concerns choosing a particular design for the algorithm to be created.

2) How to express an algorithm – This concerns converting an algorithm into a well-structured program.

3) How to validate an algorithm – This concerns validating that the algorithm computes the correct answer for all possible legal inputs.

4) How to analyze an algorithm - The process of determining how much computing time and storage an algorithm will require.

5) How to test a program – Debugging and Profiling. Debugging is the process of finding out errors in the program and correcting them whereas profiling is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

Throughout this course, we will be dealing with a lot of different strategies. And as we go along you will notice that a particular problem can be solved by one or more strategies. This helps us to develop the ability to compare the algorithms, which could be used to solve a particular problem, and finally decide which one is the best.

Throughout our study, we assume that we use a 'conventional' computer. By this we mean the instructions are carried out one at a time (not concurrently) and the major cost of an algorithm depends on the number of operations it requires.

Two phases are involved in the complete analysis of the computing time of an algorithm:

- **A priori analysis**. In this, we obtain a function (of some relevant parameters), which bound the algorithm's computing time. This is done before the actual execution of the algorithm. Hence the name, a priori (prior to execution).

- **A posteriori testing**. Here, we collect the actual statistics about the algorithm's consumption of time and space while it is executing.

Order of magnitude

Suppose there is the statement,
S: $x = y + z$
somewhere in the middle of a program.

If we wish to determine the total time this statement will spend executing, given some initial state of input data, we need two items of information which are the right hand factors in the following equation.

Total time S will spend executing = The no. of times S will be executed (frequency count) X the time taken for executing S once.

Note that the time per execution depends on:
i) the machine being used
ii) the programming language together with its compiler.

This would mean that we cannot find out the total time a statement spends executing (hence the total time an algorithm takes since an algorithm is nothing but a collection of statements) until we actually execute the algorithm on some machine.

But we would like to know how long an algorithm would take before we actually execute it for obvious reasons.

E.g., Suppose, I am given a new algorithm for a particular problem, I would like to know how long would it take because if it is going to take very long I am not interested. And I might as well search for another.

So we come to a compromise and say that if we would want an estimate of the time taken before the actual execution. Then, we use this estimate to valuate the algorithm.

Fortunately, such an estimate can be determined which is nothing but a function of the frequency counts all the statements in the algorithm. Notice that the *frequency count* is the left hand factor in the preceding equation. 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.

The order of magnitude of a statement refers to its frequency of execution (i.e., frequency count). The order of magnitude of an algorithm refers to the sum of the frequency counts of all its statements.

E.g., consider the three program segments a, b, c:

```
                      …….                  for  k = 1 to n do
….                    for  k = 1 to n do        for  j = 1 to n do
x = x  + y                 x = x  + y                  x = x  + y
….                    repeat                           repeat
                      ……                   repeat
  (a)                      (b)                  (c)
```

For each segment we assume that the statement x = x + y is contained within no other loop than what is already visible. Thus, for segment (a) the frequency count (or the order of magnitude) of this statement is 1. For segment (b) the count is n and for segment (c) it is n^2 .

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. Subsequently, a function of the order of magnitude may be found out.

Asymptotic Notation

An a priori analysis ignores all of the factors which are machine or programming language dependent and concentrates on determining the order of magnitude of the frequency of execution of statements.

Example algorithms:

1) Linear Search:

Searching for an element, x is an unsorted list, A of n elements

```
LSearch(A,x,n)
// Returns the position where x is found, otherwise 0.
Begin
for(i =1;i<=n;i++)                 min=1, max=n+1
{
        if x == A[i] then                  min=1, max =n
                return i                   min=0, max = 1
        endif
}
return 0                                   min=0, max =1
end                                        --------------------
                                           min=1, max=n+1
```

2) Finding the maximum of a list of n elements:

```
Max(A,n)   A={1, 2, 4, 7, 10} , n=5
begin
// Returns maximum element from the list, A
max = -999                         min=1,      max=1
for i = 1 to n                     min=n+1, max=n+1
{
        if  A[i] > max then        min=n,      max=n
                max = A[i]         min=1,   max=n
        endif
}
return max                              min=1,  max=1
                                           ---------------------
                                        min=n+1, max=n+1
```

The following mathematical notations are used for a priori analysis.

O-notation. $f(n) = O(g(n))$ iff there exist two positive constants c and n_o such that
$| f(n) | \leq c \, | g(n) |$ for all $n \geq n_o$

where, f(n) is the computing time of the algorithm. The variable n may be the number of inputs or outputs, their sum or the magnitude of one of them.

Since f(n) is machine dependent, an a priori analysis will not suffice to determine it. However, an a priori analysis can be used to determine a g(n) such that $f(n) = O(g(n))$.

When we say that an algorithm has computing time O(g(n)) we mean that if the algorithm is run on some computer on the same type of data but for increasing values of n, the resulting times will always be less than some constant times |g(n)|.

We shall always try to obtain the smallest g(n) such that $f(n) = O(g(n))$.

Theorem: If $A(n) = a_m n^m + . . . \; . . . . + a_1 n + a_o$ is a polynomial of degree m then $A(n) = O(n^m)$.
Proof:
$A(n) = a_m n^m + . . . \; . . . . + a_1 n + a_o$

$= n^m (a_m + . . . \; . . . . + a_{1/} n^{m-1} + a_{o\,/} n^m)$

$<= n^m (|a_m| + . . . \; . . . . + |a_1| + |a_o|)$

O-notation. $f(n) = O(g(n))$ iff there exist two positive constants c and n_o such that

$| f(n) | \leq c \, | g(n) |$ for all $n \geq n_o$

Let $c = |a_m| + \ldots \quad \ldots + |a_1| + |a_o|$,

and $n_o=0$.

Therefore according to the defn. of Big-O, $A(n) = O(n^m)$.

Ω-notation. $f(n) = \Omega(g(n))$ iff there exist two positive constants c and n_o such that
$| f(n) | \geq c \, | g(n) |$ for all $n \geq n_o$.

θ-notation. $f(n) = \theta(g(n))$ iff there exist positive constants c_1, c_2, and n_o such that
$c_1 | g(n) | \leq | f(n) | <= c_2 | g(n) |$ for all $n \geq n_o$.

Example algorithms:
1) Linear Search:
Searching for an element, x is an unsorted list, A of n elements

```
LSearch(A,x,n)
// Returns the position where x is found, otherwise 0.
Begin                                              lower, up
for(i =1;i<=n;i++)              min=1, max=n+1;    Ω(1), O(n)
{
        if x == A[i] then       min=1, max =n;     Ω(1), O(n)
                return i        min=0, max = 1     Ω(0), O(1)
        endif
}
return 0                        min=0, max =1      Ω(0), O(1)
end                             ---------------------------------
                                min=1, max=n+1     Ω(1), O(n)
```

How much time will be taken by this algorithm?
$f(n) = \Omega(1)$
$f(n) = O(n)$

2) Finding the maximum element in a list of n elements:

```
Max(A,n)
begin
// Returns maximum element from the list, A
max = -999  // Assign a very small number
for i = 1 to n
do
        if  A[i] > max then
                max = A[i]
        endif
done
return max
```

min=n+1, max=n+1
$f(n) = \Omega(n)$ and $f(n)= O(n)$

---

then it means $f(n)=\Theta(n)$

{5, 4, 2, 7, 3,1}

How much time will be taken by this algorithm?

Comparison of Algorithms

If we have two algorithms which perform the same task on n inputs, and the first has a computing time of 2n and the second n^2. Notice that the computing time (a.k.a. time complexity) of the first algorithm in O(n) and that of the second is $O(n^2)$. Which algorithm is better?

We can see that the second algorithm will take more time that the first for sufficiently large values of n. Thus we say that the first algorithm is better. Further explanation follows.

E.g., if the actual computing times for these algorithms are 2n and n^2 respectively, then we can see below their values for increasing values of n.

	Algo.1	Algo.2
n	2n	n^2
0	0	0
1	2	1
2	4	4
3	6	9
4	8	16
...	...	...
...	...	...

From the above figures, one can easily make out that for initial values of n, algorithm 2 is better. But notice that for all values of n > 2, algorithm 1 is better. In practice, the constants associated with the computing time depends on many factors such as the language and the machine one is using. For a priori analysis, we can find out only the order of magnitude.

You will notice that however large the value of the constant be, as n tends to infinity, the value of the computing time of the algorithm with the higher order of magnitude will surpass the other at some value of n. Thus we say that algorithm with the lower order of magnitude is always better, even though it may do worse than the other for some small values of n.

Thus we may have the following relations:
$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

O(1) means that the number of executions is fixed and hence the total time is bounded by a constant. The first six orders of magnitude are all bounded by a polynomial. But, there is no integer m such that n^m bounds 2^n, i.e., $2^n \neq O(n^m)$. An algorithm whose computing time is bounded below by $\Omega(2^n)$ is said to require exponential time. As n gets large, there becomes a tremendous difference between exponential and polynomial time algorithms.

Exercises:

1. Is $4n+2 = O(n)$? c=6 , no = 1
2. Is $4n+1 = \Omega(n)$? (or prove $4n+1 = \Omega(n)$) c = 4, no = 0
3. Prove $\frac{1}{2} n^2 - 3n = \Theta(n^2)$ c1=0.1 , c2=0.5, no=7

Little oh (o-notation). f(n) = o(g(n)) iff there exist two positive constants c and n_o such that
$| f(n) | < c| g(n) |$ for all $n \geq n_o$

Little Omega (ω-notation). f(n) = **ω** (g(n)) iff there exist two positive constants c and n_o such that
$| f(n) | > c| g(n) |$ for all $n \geq n_o$.

Summation Formulas

1. Sum of consecutive integers.

$$\sum_{1}^{n} i = \frac{n(n+1)}{2}$$

Write out the integers from 1 to n.

1 2 3 4 ……………………… n-3 n-2 n-1 n

Add up the 1st integer and the last, the second and the second last, and so on. Each of these sums in n+1.

Case i) n is even.

There are $\frac{n}{2}$ such pairs. Thus the sum of the series is $\frac{n(n+1)}{2}$.

Case ii) n is odd.

There are $\frac{n-1}{2}$ such pairs and the extra term $\frac{(n+1)}{2}$ Thus the sum of the series is $\frac{(n-1)(n+1)}{2} + \frac{(n+1)}{2} = \frac{n(n+1)}{2}$

2. Sum of squares.

$$\sum_{1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6}$$

3. Powers of 2

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

--**--