Algorithm Analysis

Algorithm Performance

- Performance Analysis
  - Mathematical Analysis of memory usage and time of execution
- Performance Measurement
  - Measurement of actual memory usage and absolute execution time

Algorithm complexity

- Space Complexity is the amount of main memory needed to run the program

- Time complexity is the amount of computer time needed to run to completion

Space Complexity

- Fixed space requirement
  - Code space, fixed size variables, constants and structures
- Variable Space requirement
  - Variable size structured variables and dynamic data structures
- Generally easy to estimate and measure

Time Complexity

- Compile time does not depend on the program instance
- Execution time depends on the kind of algorithm and the size of the input data
- A program may be compiled only once but may be run millions of time

Time Complexity Analysis Objective

- To find out execution time of algorithms as function of input size in a machine- and programmer-independent manner, so that several algorithms for the same problem can be compared

- Machine model : A normal single processor sequential computer where all instructions take equal amount of time, and has infinite memory.

- There is no resource constraint.

Time Complexity – unit of measurement

- Execution Time of a program instance may be measured in seconds with finite accuracy
- But this time is in the context of a particular processor and compilation technique
- A more general method is to count the number of program steps – syntactically or semantically meaningful program segments whose execution time is independent of the program instance characteristics

Time complexity – Step Count

- We can find out the step count for any program
- For different instances, the step count will differ and an idea of execution time will be available
- We can also find a mathematical expression for the step count
- In that case we can compare the step count expressions of different algorithms

Estimation

- Generally an estimate of the growth rate of execution time with input size is sufficient for comparison.

- An worst case estimate is normally computed because it provides an upper bound for all inputs including particularly the bad ones.

- Average case analysis does not provide the upper bound and is difficult to compute. Even the definition of average can affect the result.

Rules for computing running time

- 1. Sequence : Add the time of the individual statements. The maximum is the one that counts.

- 2. Alternative structures : Time for testing the condition plus the maximum time taken by any of the alternative paths.

- 3. Loops : Execution time of a loop is at most the execution time of the statements of the body (including the condition tests) multiplied by the no. of iterations.

Rules for computing running time …

4. Nested loops : Analyse them inside out.

5. Subprograms : Analyse them as separate algorithms and substitute the time wherever necessary.

6. Recursive Subprograms : Generally the running time can be expressed as a recurrence relation. The solution of the recurrence relation yields the expression for the growth rate of execution time.

Order functions

- T(n) = O(f(n)) if there are positive constants c and n_o such that $T(n) \leq c\ f\ (n)$ when $n \geq n_o$
- c f (n) is at least as large as T(n).
- f(n) is an upper bound of T(n), ignoring c
- growth rate of T(n) ≤ that of f(n)

Order functions …

- $T(n) = \Omega(g(n))$ if there are positive constants c & n_0 such that $T(n) \geq cg(n)$ for $n \geq n_o$
- g(n) is a lower bound
- growth rate of T(n) ≥ that of g(n)

Order functions …

- $T(n) = \theta(h(n))$ iff $T(n) = O(h(n))$ & $T(n) = \Omega(h(n))$
- growth rates are equal.

- $T(n)=o(\rho(n))$ if $T(n) = O(\rho(n))$ and $T(n) \neq \theta(\rho(n))$
- growth rate of $T(n) <$ that of $\rho(n)$

- $T(n) = \omega(q(n))$ if $T(n) = \Omega(q(n))$ and $T(n) \neq \theta(q(n))$
- growth rate of $T(n) >$ that of $q(n)$

- $T(n) = O(f(n))$ guarantees that $T(n)$ grows no faster than $f(n)$; $f(n)$ is an upper bound. We are interested in the lowest upper bound.

Properties of order functions

- 1. If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, then

- (a) $T_1(n) + T_2(n) = O(\max (f(n), g(n)))$
- (b) $T_1(n) * T_2(n) = O(f(n) * g(n))$

- 2. If T(x) is a polynomial of degree n
- then $T(x) = \theta(x^n)$ (prove it)

Growth Functions

Function	Name
C	constant
log n	logarithmic
$\log^2 n$	Log-squared
n	linear
n logn	
n^2	quadratic
n^3	cubic
2^n	exponential

Computation of Growth Rates

- The relative growth rates of two functions f(n) and g(n) can be determined by computing

  Lt $(f(n)/g(n))$ using L'Hospital's rule.
  $n \to \alpha$

- Usually the relation between f(n) and g(n) can be derived by simple algebra and from known facts.

Algorithm Analysis Example

Ex-1 To find out the value of an expression

$x^3 + 9x^2 - 25x + 20$

Algo :

```
read x ;
y = x*x*x + 9*x*x - 25*x + 20 ;
write y ;
```

Algorithm Analysis Example ...

Ex-2 Iterative computation of factorial of n

```
Algo : read n ; factorial = 1 ;

       if (n < 0)
       then error
       else     if (n == 0)
                then factorial = 1
                else     for i = 1 to n do
                                factorial = factorial * i;
       write factorial ;
```

Algorithm Analysis Example ...

Ex-3 Finding the average of a sequence of positive integers terminated by a 0.

```
Algo :   read x ; sum = 0 ; i = 0 ;
         while (x != 0)  do
                  { i = i + 1 ;
                  sum = sum + x ;
                  read x ;}

                  avg = sum/ i ;
                  write  avg ;
```

Is this algorithm correct ?

Algorithm Analysis Example …

Ex- 4 Read an array and bubble sort in ascending order

Algo :

```
for i = 1 to n  do
                read  A[i ] ;
for i = 1 to n-1       do
        for j = n down to i +1      do
                if (A[j-1] > A[j])
                then swap (A[j-1], A[j]);
 for i = 1 to n              do
                write A[i ] ;
```

- Swap (x,y) ≡

```
temp = x ;
x   = y ;
y   = temp ;
```

Logarithmic time Algorithms

- An algo is logarithmic time O(log n) if it takes const time to cut the problem size by a fraction (usually ½).

- **Ex-1** Binary Search

- **Ex-2** Euclid's GCD Algorithm

```
while n > 0 do
        { rem = m %  n ;
        m = n    ;
        n  = rem ; }
 gcd = m ;
```

- After two iterations, the remainder is at most half of its original value. ∴ the no. of iterations is at most 2log n

  ∴ Execution time is O(log n)

Logarithmic time Algorithms …

Ex-3 Exponentiation: find x^n , n is an integer

```
Algo : if n = 0
       then pow = 1
       else    if even (n)
               then  pow = pow (x*x, n / 2)
               else pow = x*pow(x*x, n / 2) ;
```

Fibonacci Series

Algo : if (n==0) or (n==1)
then return 1;
else return (fib(n-1) + fib(n-2));

$T(n) = T(n-1) + T(n-2) + 2$

By analogy, $T(n) \geq fib(n)$ ..... (1)

since $fib(n) = fib(n-1)+fib(n-2)$

(1) can be proved by induction.

Fibonacci Series …

Now,

$$\text{Fib}(n) \geq (3/2)^{n-1}$$

can be proved by induction.

$\therefore\ T(n) \geq (3/2)^{n-1}$

Thus $T(n) = \Omega\ ((3/2)^{n-1})$

Even the lower bound is exponential !

To Summarize

- 1. Factorial time algorithms are worst.
- 2. Factorial and exponential time algorithms are impractical.
- 3. Even polynomial time algorithms of > $O(n^3)$ are impractical.
- 4. It may be possible to reduce the time complexity drastically by changing the algorithm design strategy.
- 5. For logarithmic time algorithms, if the input is to be read, the total solution may become linear time.

Performance Measurement

- For space measurement, we can use the summation of the memory space used by the compile time variables and runtime memory allocation
- For time measurement, use the time functions in the C library
- Be careful not to include the data reading and writing time
- Only the CPU time used in the processing of data is to be measured

Time measurement

- This may be tricky in multi-programming OS like Linux, Windows XP, Vista, etc.
- Consult Horowitz & Sahni
- Use the program segments in your programs
- For embedded and real-time programs, time is calculated by observing the machine code and the number of clock cycles required for running the program. Then the no. of clock cycles is multiplied by the clock cycle time to arrive at the actual time required in micro- or millisecond