**Mathematical Analysis of Nonrecursive Algorithms**

**Simple Formula:**





**General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms**

1. Decide on a parameter (or parameters) indicating an input’s size.

2. Identify the algorithm’s basic operation.

3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.

4. Set up a sum expressing the number of times the algorithm’s basic operation is executed.

5. Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.

**EXAMPLE 1** Consider the problem of finding the value of the largest element in a list of n numbers.

**ALGORITHM MaxElement(A[0..n − 1])**

//Determines the value of the largest element in a given array

//Input: An array A[0..n − 1] of real numbers

//Output: The value of the largest element in A

maxval ←A[0]

for i ←1 to n − 1 do

if A[i]>maxval

maxval←A[i]

return maxval

**Basic Operation:** comparison A[i]> maxval





O(n)

**EXAMPLE 2** Consider the element uniqueness problem: check whether all the elements in a given array of n elements are distinct.

**ALGORITHM UniqueElements(A[0..n − 1])**

//Determines whether all the elements in a given array are distinct

//Input: An array A[0..n − 1]

//Output: Returns “true” if all the elements in A are distinct and “false” otherwise

for i ←0 to n − 2 do

for j ←i + 1 to n − 1 do

if A[i]= A[j ] return false

return true

**Basic Operation:** comparison A[i]=A[j]

The number of times the basic operation being executed depends on the type of input

If the first two elements in the array are equal the time complexity is denoted as

**Cbest (n) =(n)**

The worst case time complexity is the largest number of times the comparison operation is performed and is denoted as,

C(n)

We could also have computed the sum faster as follows:

**EXAMPLE 3** Given two n × n matrices A and B, find the time efficiency for computing their product C = AB.

**ALGORITHM MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])**

//Multiplies two square matrices of order n by the definition-based algorithm

//Input: Two n × n matrices A and B

//Output: Matrix C = AB

for i ←0 to n − 1 do

for j ←0 to n − 1 do

C[i, j ]←0.0

for k←0 to n − 1 do

C[i, j ]←C[i, j ]+ A[i, k] \* B[k, j]

return C

**Basic Operation:** Multiplication

Let M(n) denotes the number of times the multiplication operation is executed and is given as

**EXAMPLE 4** The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

**ALGORITHM Binary(n)**

//Input: A positive decimal integer n

//Output: The number of binary digits in n’s binary representation

count ←1

while n > 1 do

count ←count + 1



return count

**Basic Operation:** comparison n > 1

How many times we can divide n by 2 without getting a result less than 1?. In other words, what is the largest d for which n/2d ≥1?.

We solve for d:

n/2d ≥1

2d ≤ n

Taking log2 on both sides

log2 2d = log2 n

d =log2 n

**c(n)= log2 n**

**Properties of Logarithms**

All logarithm bases are assumed to be greater than 1 in the formulas below, lg x denotes the logarithm base 2, ln x denotes the logarithm base e=2.71828…, x,y are arbitrary positive numbers.



**Important Summation Formulas**



**Sum Manipulation Rules**



**Floor and Ceil formulas**

