# Time Complexity Analysis of Recursive functions:

## Using Recurrence relations

A **recurrence relation**, also known as a **difference equation**, is an equation which defines a sequence recursively: each term of the sequence is defined as a function of the preceding terms.

Let us look at the powerA function (exponentiation).

**int powerA( int x, int n)**

**{**

**if (n==0)**

**return 1;**

**if (n==1)**

**return x;**

**else**

**return x \* powerA( x , n–1 );**

**}**

The problem size reduces from n to n – 1 at every stage, and at every stage two arithmetic operations are involved (one multiplication and one subtraction).

Thus total number of operations needed to execute the function for any given n, can be expressed as sum of 2 operations and the total number of operations needed to execute the function for n-1.

When n=1, it just needs one operation to execute the function

T(1) = 1

We need to solve this to express T(n) in terms of n.

In other words T(n) can be expressed as sum of T(n-1) and two operations using the following ***recurrence relation***:

T(n) = T(n – 1 ) + 2

The solution to the recurrence relation proceeds as follows.

Given the relation

T(n) = T(n – 1 ) + 2 …………………………………………………….(1)

We try to reduce the right hand side till we get to T(1) , whose solution is known to us.

We do this in steps.

First of all we note (1) will hold for any value of n.

Let us rewrite (1) by replacing n by n-1 on both sides to yield

T(n – 1 ) = T(n – 2 ) + 2 …………………………………………………….…(2)

Substituting for T(n – 1 ) from relation (2) in relation (1) yields

T(n ) = T(n – 2 ) + 2 (2) …………………………………………………….…(3)

Also we note that

T(n – 2 ) = T(n – 3 ) + 2 …………………………………………………….…(4)

Substituting for T(n – 2 ) from relation(4) in relation (3) yields

T(n ) = T(n – 3 ) + 2 (3) …………………………………………………….… (5)

Following the pattern in relations (1), (3) and (5) now we can write a generalized relation

T(n ) = T(n – k ) + 2(k) …………………………………………………….…(6)

To solve the generalized relation (6), we have to find the value of k.

We note that T(1) needs just one operation from (1)

T(1) = 1 ……………………………………………………....(7)

We can reduce T(n-k) to T(1) by setting

n – k = 1

and solve for k to yield

k = n – 1

Substituting this value of k in relation (6), we get

T(n) = T(1) + 2(n – 1 )

Now substitute the value of T(1) from relation (7) to yield the solution

T(n) = 2n– 1 ……………………………………………………...(8)

When the right hand side of the relation does not have any terms involving T(..), we say that the recurrence relation has been solved. The algorithm is linear in n.

<http://www.cs.ucf.edu/~dmarino/ucf/cop3502/lec_biswas/recursion12.pdf>

### Complexity of the efficient recursive algorithm for exponentiation:

We now consider the powerB function.

**int powerB( int x, int n)**

**{**

**if ( n==0)**

**return 1;**

**if(n==1)**

**return x;**

**if (n is even)**

**return powerB( x \* x, n/2);**

**else**

**return powerB( x \* x, n/2 ) \* x ;**

**}**

At every step the problem size reduces to half the size.

When the power is an odd number, an additional multiplication is involved.

To work out time complexity, let us consider the ***worst case,*** that is we assume that at every step an additional multiplication is needed.

Thus total number of operations T(n) will reduce to number of operations for n/2, that is T(n/2) with three additional arithmetic operations (the odd power case).

We are now in a position to write the ***recurrence relation*** for this algorithm as

T(n) = T(n/2) + 3 ……………………………………………………...(1)

T(1) = 1 ……………………………………………………...(1)

To solve this recurrence relation, we note that T(n/2) can be expressed as

T(n/2) = T(n/4) + 3 ……………………………………………………...(2)

Substituting for T(n/2) from (2) in relation (1) , we get

T(n) = T(n/4) + 3(2)

By repeated substitution process explained above, we can solve for T(n) as follows

T(n) = T(n/2) + 3

= [ T(n/4) + 3 ] + 3

= T(n/4) + 3(2)

= [ T(n/8) + 3 ] + 3(2)

= T(n/8) + 3(3)

Now there is a relationship between 3 and 8, and we can rewrite the recurrence relation as

T(n) = T( n/ 23 ) + 3(3)

We can continue the process one step further, and rewrite the relation as

T(n) = T( n/ 24 ) + 3(4)

Now we can see a pattern running through the various relations and we can write the generalized relation as

T(n) = T( n/ 2k) + 3(k)

Since we know that T(1) = 1, we find a substitution such that the first term on the right hand side reduces to 1.

The following choice will make this possible

2k= n

We can get the value of k by taking log of both sides to base 2, which yields

k = log n

Now substituting this value in the above relation, we get

T(n) = 1 + 3 log n

Thus we can say that this algorithm runs in LOGARITHMIC TIME, and obviously is much more efficient than the previous algorithm.

Consider 2 Power 6 and run through the steps above.

1 2 3 4 5 6

2 X 2 X 2 X 2 X 2 X 2

### Complexity of the recursive Binary search function Rbinary

The time complexity of the algorithm can be obtained by noting that every time the function is called, the number of elements to be handled reduces to half the previous value.

Assuming it takes one operation for each of the 3 IF statements, and 2 operations for computing mid, the total number of operations can be expressed through the recurrence relation

T(n) = T( n/2) + 5

with

T(1) = 1

Note this relation is very similar to the one that we had for the efficient version of the exponentiation algorithm, and it can be shown that the complexity of this algorithm is also logarithmic.

### Recurrence relation for the recursive Array handling function arrayOdd

**int arrayOdd(int A[ ], int n)**

**{**

**if(n < 1)**

**return 0;**

**else**

**return A[n-1] % 2 + arrayOdd( A, n-1);**

**}**

In this function, the total number of operations reduce from size n to n-1 and every time additional 4 arithmetic operations are carried out.

Thus the recurrence relation for this function is

T(n) = T(n – 1) + 4

T(1) = 1

**Recurrence relation for a hypothetical recursive function**

**int hypo(int a, int n)**

**{**

**if(n == 1)**

**return 0;**

**else**

**return a + hypo(a,n-1) \* hypo(a,n-1);**

**}**

T(n) = 2 T(n – 1) + 3

T(1) = 1