Exponentiation:

We consider exponentiation which involves raising an integer base to an integer power. The basic strategy would be to use successive multiplication by the base. However, as the result becomes quite large even for small powers, it would work only if there is a machine to hold such large integers. For larger powers, one could use a stack.

Evaluation of x^n requires n-1 multiplications. Here is a simple recursive function to carry out the exponentiation.

```
power( x,  n)
{
      if ( n==0)
          return 1;
      if(n==1)
          return x;

       else
           return ( x * power( x , n – 1 )  ;
}
```

Complexity Analysis :

The problem size reduces from n to n – 1 at every stage, and at every stage one multiplication operation is involved. Thus total number of operations T(n) can be expressed as sum of T(n-1) and one operation as the following ***recurrence relation***:

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

In turn T(n-1) operations can expressed as a sum of T(n-2) and one operation

$T(n-1) = T(n-2) + 1$ …(2)

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

$T(n) = T(n-2) + 2$ …(3)

Also we note that

$T(n-2) = T(n-3) + 1$ …(4)

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

$T(n) = T(n-3) + 3$ … (5)

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

$T(n) = T(n - k) + k$ …(6)

To solve the generalized relation (6), we have to find the value of k. We note that T(1) , that is the number of operations to raise a number x to power 1 needs just one operation. In other words

$T(1) = 1$ ...(7)

Thus we set the first term on right hand side of (6) to 1 to get

$n - k = 1$

that is $k = n - 1$

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

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

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

$T(n) = n$ ..(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. So what is the time complexity of the exponentiation algorithm? The right hand side of (8) indicates that it is simply O(n) .

An efficient recursive algorithm for exponentiation:

Let us now develop an efficient version of the exponentiation algorithm. Note that x can be raised to power 16 by raising x^2 to the power 8

$x^{16} = (x^2)^8$

which can be viewed as two different operations
$y = x^2$

$x^{16} = (y)^8$
Thus instead of multiplying 15 times, we can get the result by multiplying x^2 seven times.

Further we note that y^8 can be obtained by a similar process.

$y^8 = (y^2)^4$

Thus at every stage the number of multiplications is reduced by half.

Note the similarity with binary search. A higher power can be obtained from its lower power (i.e. power/2).

We present below a recursive algorithm. Raising a number to power n can be reduced to the problem of raising x^2 to power n/2. The problem size can be reduced by recursively, till n equals 1. However note that when the power is odd, it needs one more multiplication.

$x^{17} = (x^2)^8 \cdot x$

Here is the algorithm which takes care of both even and odd powers:

```
power( x,  n)
{
      if ( n==0)
          return 1;
      if(n==1)
          return x;

      if (n is even)
          return power ( x * x,  n / 2);

      else
           return power( x * x, n / 2 ) * x  ;
}
```

Complexity of the efficient recursive algorithm for exponentiation:

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 an additional multiplication operation. We are now in a position to write the ***recurrence relation*** for this algorithm as

$T(n) = T(n/2) + 1$ ..(1)
$T(1) = 1$ ..(1)

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

$T(n/2) = T(n/4) + 1$ ..(2)

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

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

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

$$
\begin{aligned}
T(n) &= T(n/2) + 1 \\
&= [\ T(n/4) + 1\ ] + 1 \\
&= T(n/4) + 2 \\
&= [\ T(n/8) + 1\ ] + 2 \\
&= T(n/8) + 3
\end{aligned}
$$

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

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

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

$T(n) = T(n/ 2^4) + 4$

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

$T(n) = T(n/ 2^k) + k$

Since we know that $T(1) = 1$, we find a substitution so that the first term on the right hand side reduces to 1. The following choice will make this possible

$2^k= n$

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

$T(n) = 1 + \log n$

Thus the time complexity of this recursive algorithm in terms of Big-O is

$O(\log n)$

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

Decimal to Binary Conversion:

We count in base 10, possibly because we were born with 10 fingers. Computers count in base 2 because of the binary nature of electronics. One of the tasks a compute performs is to convert from decimal to binary. Here is a simplified version of the solution to this problem.

For example if n is 19, the binary equivalent is

$10011 = 1. 2^4 + 0. 2^3 + 0. 2^2 + 1. 2^1 + 1. 2^0.$

Take a close look at the binary representation. It can be thought to be made up of two parts.

1001 and 1 where

$1001 = 1. 2^4 + 0. 2^3 + 0. 2^2 + 1. 2^1$

$= 9$ (which is 19/2)

and $1 = 19 \% 2.$

Again look at two parts of 1001

$100 = 1. 2^4 + 0. 2^3 + 0. 2^2$

$= 4$ (which is 9/2)

and $1 = 9 \% 2.$

This shows that the rightmost bit has the value of n%2; the other bits are the binary equivalent of n/2. But the bits are being collected from the right hand side. This suggests that we must perform all of the calculations before we return the result.

Speaking recursively, we need to calculate the binary equivalent of n/2 before we append the value of n%2. We will stop when n is 1 or 0. In both the cases the binary bit is same as the integer value.

Since the binary equivalent tends to have large number of bits, we will store the binary equivalent in a String object. Here is the code to convert a group of decimal numbers into their binary equivalents.

```
import java.io.*;
import java.util.*;
import java.lang.*;
```

```
class conversion
{

public static String getbinary( int n)
{
        if (n<= 1)
                return Integer.toString(n);
        return getbinary(n/2) + Integer.toString(n%2);
}
public static void main( String [ ] args )throws
IOException
{
     int b[] = { 0, 1, 4, 10, 19};
     String bin;
     for(int  i=0; i<5 ; i++)
     {
         bin =getbinary( b[i]);
         System.out.println(" "+b[i]+"   "+      bin);
     }
}

}
```

Here is the corresponding output:

```
 0    0
 1    1
 4    100
 10   1010
 19   10011
```

Binary search (recursive version)

Divide and conquer implementation

We have already looked at the iterative version of binary search algorithm. Given an array with elements in the ascending order, the problem is to search for a target element in the array. Here is the recursive version of the binary search. It returns the position of the element which matches the target. If the target is not found, it returns -1.
Use this code to implement binary search program, and test for arrays containing odd numbered elements as well as arrays containing even numbered elements. In order to understand the working of the program introduce a statement to print the middle element

every time. Modify the program so that it can handle an array containing elements in descending order.

```
#include <stdio.h>


 int targetsearch ( int target, int A[], int  n )
 {
        int temp;
        temp = binary(target, A, 0, n-1) ;
        return temp ;
 }


 int binary ( int target, int A [ ], int left, int right)
 {
    int mid;
    if ( left > right ) return -1 ;
    mid =  ( left + right )/ 2;
    if   ( target ==  A [mid] ){
            return (mid);}
    if  ( target < A [mid])
            return (binary (target, A, left, mid - 1 ));
    else
            return (binary (key, A, mid + 1, right )) ;
 }
```

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

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

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 works out to O(log n).

The Greatest common Divisor (GCD)

The GCD of Two non-negative integers is the largest integer that divides evenly into both.

For example, numbers 2, 3, 4, 6 and 12 divide evenly into both the numbers 24 and 36 . So GCD of 24 and 36 is 12, the largest integer. The GCD can be obtained by factorizing the numbers and picking up all the common elements. However for very large numbers it would involve large number of operations. For example consider obtaining GCD of 129618 and 576234 by factorizing.

In 3rd century B.C., the great mathematician Euclid had proposed an algorithm to find GCD. It was based on the facts that GCD(p, q) = GCD(q, p), and GCD(p, 0) = p

Euclid's algorithm:

1. Make p the larger of two numbers p and q.

2. Divide p by q. Let r be the remainder. If r is zero, then q is the gcd.

3. Else, gcd of p and q equals gcd of q and r.

Using this algorithm we can find the gcd of 1296 and 576 as follows;

gcd(1296,576)

= gcd (576, 1296 % 576)

= gcd (576, 144)

= 144 as 144 divides 576 without leaving a remainder.

```
public static long gcd( long a, long b)
{
     if(b == 0)
          return a;
     else
          return gcd( b, a%b);
}
```

Fibonacci Sequence

This is a very famous sequence which can be generated through a recursive solution. It has a small history. In 1202, Italian mathematician Leonardo Fibonacci posed a problem that has had a wide influence on many fields. The problem is related to growth in population of rabbits, generation to generation. It is assumed that the rabbits are reproduced according to the following rules:

1. Each pair of fertile rabbits produces a new pair of offspring each month.
2. Rabbits become fertile in their second month of life.
3. No rabbit dies.

Suppose we start with a pair introduced in January. Thus
on Jan 1 , there are no rabbits and

on Feb 1 , there is one pair. Since they are not yet fertile
on March 1, it is still single pair. In march they produce another pair, so
on April 1, the count is 2 pairs. In April the original pair produces another pair, so
on May 1 there are 3 pairs.

We continue further with the same logic and find that the number of rabbits in each month is given by the sequence 0,1,1,2,3,5,8… This sequence is known as the *Fibonacci sequence*:

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
0	1	1	2	3	5	8	13	21	34 …

A very interesting point about this sequence is that each element in this sequence is the sum of the two preceding elements. Further no two consecutive elements are even numbers. The sequence can be mathematically described through the relations:

$$t_n = \begin{cases} n & \text{if } n \text{ is } 0 \text{ or } 1 \text{ (i.e. n < 2)} \\ t_{n-1} + t_{n-2} & \text{otherwise} \end{cases}$$

How fast do the numbers grow? If 4th term is 2, 7th term is 8, 12th term is 89, 20th term is 4181, and so on. The numbers grow exponentially.

Generating the Fibonacci Sequence:

The sequence is generated by adding previous two generated sequence values. Thus we can write a method which recursively computes the previous values. The recursion stops when we reach the initial values of n = 1 and n = 0.

```
public static long fib (int n)
{
  if (n <= 0)
     throw new IllegalArgumentException();
  if (n <= 2 )
    return 1;
  else
    return fib (n - 1) + fib (n - 2);
}
```

This algorithm has an exponential time complexity. It is because each call generates two recursive calls to the same function.

An Efficient Algorithm for Fibonacci Sequence:

If we exploit the structure of the problem we can considerably reduce the time complexity, while still using a recursive solution.

```
int fib(int n)
{
    return fastfib (n, 0, 1);
}
```

•The body consists of a single line of code that does nothing but call another function, passing along the additional arguments.

•Functions of this sort, are called wrapper functions. Wrapper functions are very common in recursive programming

Here is an efficient algorithm:

```
public static long fastfib ( int n, long previous,long
                                                  current)
{
    if (n == 0)   return previous;
    if (n == 1)   return current;
    return  fastfib (n-1, current,  previous+current);
}
```

•Using this AdditiveSeq function, let’s determine the value of Fibonacci(6).

```
fib(6)
  = AdditiveSeq (6, 0, 1)
    = AdditiveSeq (5, 1, 1)
      = AdditiveSeq (4, 1, 2)
        = AdditiveSeq (3, 2, 3)
          = AdditiveSeq (2, 3, 5)
            = AdditiveSeq (1, 5, 8)
              = 8
```

•Notice how much more efficiently the recursion occurs in the AdditiveSeq function.

The time to find nth Fibonacci number reduces to that of finding the (n-1)th number, with extra time needed for one addition operation and one subtraction operation:

$T(n) = T(n – 1) + 2$

What is the time complexity of this recurrence relation?