CSC13: Algorithmics and Program Design

Dr. S. Zeeshan Hussain
Dept. of Computer Science
Jamia Millia Islamia
New Delhi

Unit-Wise Syllabus

- **Algorithmic Problem Solving**: Algorithms; Problem Solving Aspect: Algorithm Devising, Design and Top-down Design; Algorithm Implementation: Essential and Desirable Features of an Algorithm; Efficiency of an Algorithm, Analysis of Algorithms, Pseudo codes; Algorithm Efficiency, Analysis and Order; Importance of Developing Efficient Algorithms; Complexity Analysis of Algorithms: Every-Case Time Complexity, Worst-Case Time Complexity, Average-Case Time Complexity, Best-Case Time Complexity.
- Basic Algorithms – Exchanging the Values of Two Variables, Counting, Summation of a Set of Numbers, Factorial Computation, Sine Function Computation, Generation of the Fibonacci Sequence, Reversing the Digits of an Integer, Base Conversion, etc. Flowchart. Flowchart – Symbols and Conventions, Recursive Algorithms.
- **Factoring: Finding the square root of number, Smallest Divisor of an integer, Greatest common divisor of two integers, generating prime numbers, computing prime factors of an integer, Generation of pseudo random numbers, Raising a number to a large power, Computing the *n*th Fibonacci number.**
- **Arrays, Searching and Sorting:** Single and Multidimensional Arrays, Array Order Reversal**,** Array counting, Finding the maximum number in a set, partitioning an array, Monotones Subsequence; Searching: Linear and Binary Array Search; Sorting: Sorting by selection, Exchange and Insertion. Sorting by diminishing increment, Sorting by partitioning.
- **Programming:** Introduction, Game of Life, Programming Style: Names, Documentation and Format, Refinement and Modularity; Coding, Testing and Further Refinement: Stubs and Drivers; Program Tracing, Testing, Evaluation; Program Maintenance: Program Evaluation, Review, Revision and Redevelopment; and Problem Analysis, Requirements Specification, Coding and Programming Principles.
- REFERENCES

- **Dromy: How to Solve by Computer, PE (Unit 1-4)**
- **Kruse: Data Structures and Program Design, PHI (Unit-5)**
- Robertson: Simple Program Design, A Step-by-Step Approach, Thomson

Factoring Methods

Topics to be covered:

- Finding the square root of number
- Smallest Divisor of an integer
- Greatest common divisor of two integers
- Generating prime numbers
- Computing prime factors of an integer
- Generation of pseudo random numbers
- Raising a number to a large power
- Computing the *n*th Fibonacci number

Finding the square root of number

Problem:

Given a number m, devise an algorithm to compute its square root.

Finding the square root of number

Algorithmic Description

Let us understand what is meant by “square root of number”

Taking specific examples, we know that square roo of 4 is 2, square root of 9 is 3 and the
square root of 16 is 4 and so on.

That is

Finding the square root of number

2 x 2 =4

3 x 3 = 9

4x4 =16

For these examples we can conclude that in general cases the square root n of an another number m must satisfy the equation,

$$n \times n = m \quad [1]$$

Finding the square root of number

- Suppose, for example, we do not know the square root 0f 36.
- We might given that 9 could be its square root
- Using equation [1] to check our guess we find that 9 x 9 =81 which is greater than 36.
- Our guess of 9 too high so we might next try 8.
- For example 8 x 8 =64 which is still greater

than 36 but closer than our original guess.

Finding the square root of number

The Investigation we have made suggests we could adopt the following systematic approach to solve the problem:

- Choose a number n less than number m we want the square of.
- Square n and if it is greater than m decrease n by 1 and
- repeat step 2 else go to step3.
- When the square of our guess at the square root is less than m we can start increasing n by 0.1 until we again compute a guess greater than m. At this point we start decreasing our guess by 0.01 and so on until we have completed the square root we require to the desired accuracy.

Finding the square root of number

In above algorithm the number of iterations required depends critically on how good our initial guess is e.g; if m is 10,0000 and our initial guess is 500 we will need 400 iterations.

To try to make a better algorithm let us again try the problem by finding the square root of 36 we found that

- $\mathbf{9^2 = 81}$ which is greater than 36.
- \- if 9 divides into 36 to give 4
- \- if we choose 4 as our square root candidate,we would have found $\mathbf{4^2 = 16}$ which is less than 36.

Finding the square root of number

- In other words, the 9 and the 4 tend to cancel out each other by deviating from the sequence m in opposite direction. Thus,

Square	**Square root**
81	9
36	6
16	4

Thus the square root of 36 must lie somewhere between 9 which is t00 big and 4 which is too small.

Taking the average of 9 and 4 :

(9+4)/2 = 6.5

Finding the square root of number

- We find that $6.5^2 = 42.25$ which is greater than 36.
- Dividing this into 36:

  36/6.5 = 5.53

We see that it again has complimentary value (5.53) that is less than 36.

Square		Square root
81	Greater than 36	9
42.5	Do	6.5

36		??
30.5809	Less than 36	5.53
16	Do	4

Finding the square root of number

If we are still unsure as to what to do next, we can take a guess at the square root and then use the equation[1] to check whether or not we have guessed correctly.

For example:

- Suppose we do not know the square of 36.
- We might guess that 9 could be the square root.
- Using equation[1] we find that 9 x 9 = 81 which is greater than 36.
- Another guess like 8 will result to 8 x 8 = 64 which is again greater than 36 but closer than the previous result

Finding the square root of number

- We now have two estimates of the square root, one on either side, that are closer than our first estimates.
- We can proceed to get an even better estimate of the square root by averaging these two recent guesses:

  - **(6.5 + 5.53)/ 2 = 6.015**
  - Where 6.015^2 **=** = 36.6025 which is only slightly greater than the square we are seeking.
  - However we will assume that it is a good strategy for the development of algorithm.

Finding the square root of number

- To perform the average of g1 and g2 is :
- g2:= (g1 + (m/g1))/2

- To achieve the repitative interchanging of roles by setting up the following loop

    g1:=g2

  g2:= (g1 + (m/g1))/2

We can therefore terminate the algorithm when the difference between g1 and g2 becomes less than some error (0.0001)

Finding the square root of number

Algorithm Description

1. Choose m the number whose square root is required and the termination condition error e.

2. Set the initial guess g2 to m/2.

3. Repeatedly

(a) Let g1 assume the role of g2.

(b) Generate a better estimate g2 of the square root using the averaging formula until the absolute difference between g1 and g2 is less than error e

ALGORITHM : Ch-3 (Page-90)

Finding the square root of number

```
function sqroot(m,error: real): real;
var g1 {previous estimate of square root},
  g2 {current estimate of square root}: real;

begin {estimates square root of number m}
  {assert: m>0∧g1=m/2}
  g2 := m/2;
  {invariant: |g2 * g2−m|=<|g1 * g1−m|∧g1>0∧g2>0}
  repeat
    g1 := g2;
    g2 := (g1+m/g1)/2
  until abs(g1−g2)<error;
  {assert: |g2 * g2−m|=<|g1 * g1−m|∧|g1−g2|<error}
  sqroot := g2
end
```

The smallest divisor of an integer

Number = 36

Divisors =(2,3,4,6,9,12,18}

Number = 49

Divisors= {7}

The smallest divisor of an integer

Algorithm

func(n)

begin

s :=sqrt(n);

//Check for the smallestdivisor between 2 and less than equal to the square root of the number

for(i:=2;i<=sqrt(n);i++)

if (n mod i ==0) return i or print i

Check if the number is prime i.e; divisors are 1 and the number itself

end

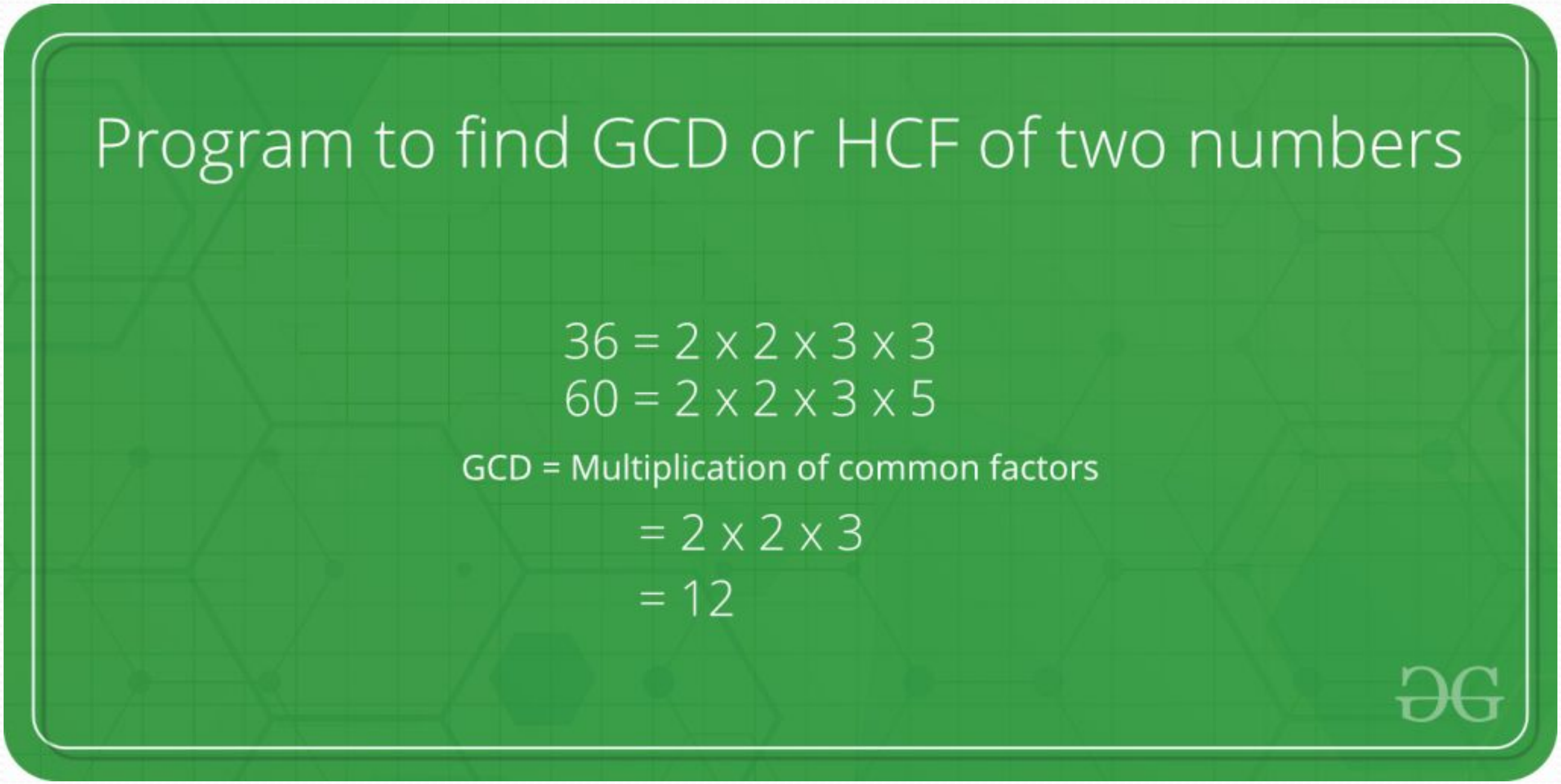
The smallest divisor of an integer

```
void smallest_divisor(int n) {

     int i=1;
     for (i = 2; i <= sqrt(n); ++i)
 {
          if (n % i == 0) { printf("\n Smallest divisor of %d is =
   %d",n,i);
               break;
          }
     }

int main() {
     int n;
     printf("Enter a Number: ");
```

Greatest common divisor of two integers



GCD of two numbers

Program to find the GCD or HCF of two numbers

Here we will discuss how to find the GCD or HCF of two numbers entered by the user using C++ programming language.

GCD i.e. Greatest Common Divisible or HCF i.e. Highest Common Factor of two numbers is the largest positive integer that can divide both the numbers

There are many methods to calculate GCD:

- **Using Prime Factorization,**
- **Euclid’s Algorithm,**
- **Lehmer’s GCD algorithm, etc**

Here we will use Euclid’s Algorithm to find the GCD, which is based on the idea that the GCD doesn’t change when smaller number is subtracted from the greater number. This keeps on going until only the GCD left.

Algorithm

Two inputs are taken from the user.

Inputs are stored in two **int** type variables say **first** and **second.**

A recursive program **findGCD** is called with parameters **first** and **second**.

1. First it is checked whether any of the input is 0

if (first == 0)

return second;

if (second==0)

return first;

2. If both input numbers are equal return any of the two numbers

if (first == second)

return second;

3. If **first** is greater than the **second**

Recursively call **findGCD** function with parameters ‘**first-second**’, **second**.

findGCD(first – second, second);

4. Otherwise recursively call **findGCD** function with parameters ‘**first**’, ‘**second-first**’.

findGCD(first, second – first);

Program to find GCD of two numbers

//C++ Program

```
//GCD of Two Numbers
#include<iostream>
using namespace std;
// Recursive function declaration
int findGCD(int, int);
// main program
int main()
{
        int first, second;
        cout<<“Enter First Number: “;
        cin>>first;
        cout<<“Enter second Number: “;
        cin>>second;
        cout<<“GCD of “<<first<<” and “<<second<<” is “<<findGCD(first,second);
        return 0;
}
//body of the function
int findGCD(int first, int second)
{
        // 0 is divisible by every number
```

To Check whether a given number is prime or not

```
procedure primenumber : number
FOR i = 2 to number – 1
 check if number is divisible by i
IF divisible
RETURN "NOT PRIME" END IF
END FOR
RETURN "PRIME"
end procedure
```

Prime Number

```
#include <stdio.h>
int main() {
int loop, number; int prime = 1;
 number = 11;
for(loop = 2; loop < number; loop++) {
 if((number % loop) == 0) { prime = 0; }
}
if (prime == 1) printf("%d is prime number.", number); else
   printf("%d is not a prime number.", number);
return 0;
}
```

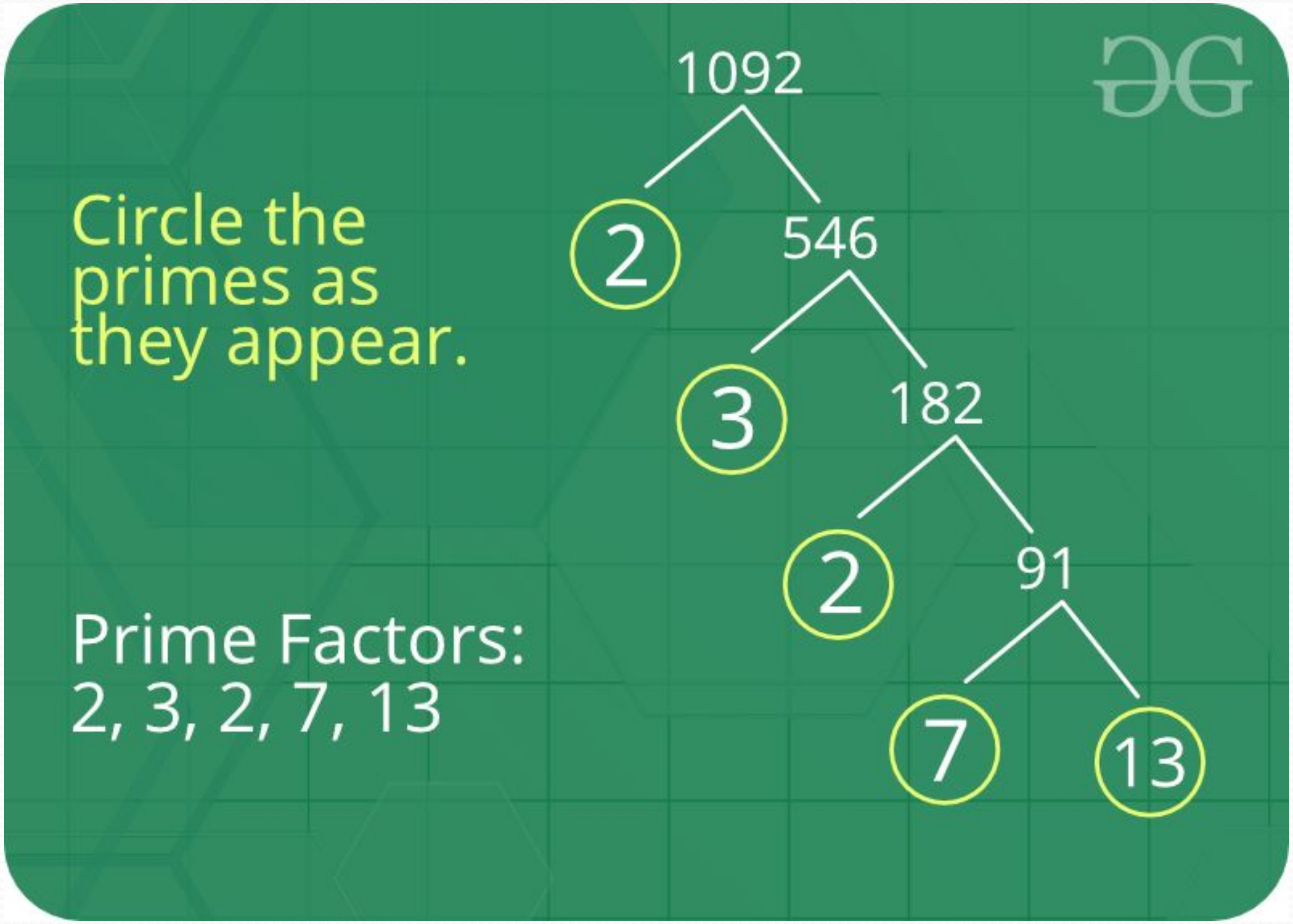
Output

11 is prime number.

To Compute Prime factors of a given number

- Prime factor is the factor of the given number which is a prime number. Factors are the numbers you multiply together to get another number. In simple words, prime factor is finding which prime numbers multiply together to make the original number.

- **Example:** The prime factors of 15 are 3 and 5 (because 3×5=15, and 3 and 5 are prime numbers).


1092
2
546
3
182
2
91
7
13
Circle the primes as they appear.
Prime Factors:
2, 3, 2, 7, 13

Prime Factors

- We will see how we can get all the prime factors of a number in an efficient way. There is a number say n = 1092, we have to get all prime factors of this. The prime factors of 1092 are 2, 2, 3, 7, 13. To solve this problem, we have to follow this rule −
- When the number is divisible by 2, then print 2, and divide the number by 2 repeatedly.
- Now the number must be odd. Now starting from 3 to square root of the number, if the number is divisible by current value, then print, and change the number by dividing it with the current number then continue.
- Let us see the algorithm to get a better idea.

Algorithm for Prime factors

```
printPrimeFactors(n)
begin    while n is divisible by 2, do
  print 2
  n := n / 2
done
for i := 3 to √n, increase i by 2, do
    while n is divisible by i, do
 print i
    n := n / i
done
 done
 if n > 2, then
    print n
end if
end
```

Program to compute Prime factors

```
#include<stdio.h>
#include<math.h>
void primeFactors(int n) {
   int i;    while(n % 2 == 0)
{     printf("%d, ", 2);        n = n/2; //reduce n by dividing this by
   2    }
   for(i = 3; i <= sqrt(n); i=i+2)
{ //i will increase by 2, to get only odd numbers
while(n % i == 0) {
 printf("%d, ", i);
      n = n/i;  }
}
 if (n > 2) { printf("%d, ", n);   }
}
```

Generation of pseudo random numbers

- **Pseudo Random Number Generator(PRNG)** refers to an algorithm that uses mathematical formulas to produce sequences of random numbers. PRNGs generate a sequence of numbers approximating the properties of random numbers.
- A PRNG starts from an arbitrary starting state using a **seed state**. Many numbers are generated in a short time and can also be reproduced later, if the starting point in the sequence is known. Hence, the numbers are **deterministic and efficient**.

Why do we need PRNG?

- With the advent of computers, programmers recognized the need for a means of introducing randomness into a computer program. However, surprising as it may seem, it is difficult to get a computer to do something by chance as computer follows the given instructions blindly and is therefore completely predictable. It is not possible to generate truly random numbers from deterministic thing like computers so PRNG is a technique developed to generate random numbers using a computer.

How PRNG works?

- **<u>Linear Congruential Generator</u>** is most common and oldest algorithm for generating pseudo-randomized numbers. The generator is defined by the recurrence relation:
- $\mathbf{X_{n+1} = (aX_n + c) \bmod m}$ where X is the sequence of pseudo-random values m, 0 < m - modulus a, 0 < a < m - multiplier c, $0 \le c < m$ - increment x_0, $0 \le x_0 < m$ - the seed or start valueWe generate the next random integer using the previous random integer, the integer constants, and the integer modulus. To get started, the algorithm requires an initial Seed, which must be provided by some means. The appearance of randomness is provided by performing **modulo arithmetic.**.

Characteristics of PRNG

- **Efficient:** PRNG can produce many numbers in a short time and is advantageous for applications that need many numbers
- **Deterministic:** A given sequence of numbers can be reproduced at a later date if the starting point in the sequence is known.Determinism is handy if you need to replay the same sequence of numbers again at a later stage.
- **Periodic:** PRNGs are periodic, which means that the sequence will eventually repeat itself. While periodicity is hardly ever a desirable characteristic, modern PRNGs have a period that is so long that it can be ignored for most practical purposes.

Applications of PRNG

- PRNGs are suitable for applications where many random numbers are required and where it is useful that the same sequence can be replayed easily. Popular examples of such applications are **simulation and modeling applications**. PRNGs are not suitable for applications where it is important that the numbers are really unpredictable, such as **data encryption and gambling.**

Implementation

x1=(a*xo+c) mod m,
where, xo=seed,
x1=next random number that we will generate
a=constant multiplier,
c=increment,
m=modulus

After calculating x1, it is copied to xo(seed) to find new x1.
ie, after calculating x1, xo=x1

Program in C for PRNG

```
#include<stdio.h>
#include<conio.h>
int main() { int xo,x1; /*xo=seed,x1=next random number that we will
   generate */
int a,c,m; /*a=constant multiplier, c=increment, m=modulus */
int i,n; /*i for loopcontrol, n for how many random numbers */
 int array[20]; /*to store the random numbers generated */
printf(“("\nEnter the seed value xo: ");
scanf("%d",&xo);
printf("("\nEnter the constant multiplier a: ");
scanf("%d",&a);
printf("("\nEnter the increment c: "); scanf("%d",&c);
printf("("\nEnter the modulus m: "); scanf("%d",&m);
 printf(“\nHow many random numbers you want to generate: ");
   scanf("%d",&n);
 for(i=0;i<n;i++) /* loop to generate random numbers */ {
  x1=(a*xo+c) %m;
array[i]=x1; xo=x1; }
```

Output

C:\Users\skunwar\Desktop\lcm.exe

```
Enter the seed value xo: 118

Enter the constant multiplier a: 4

Enter the increment c: 22

Enter the modulus m: 1000

How many random numbers you want to generate: 4

The generated random numbers are: 494   998    14     78
```

Algorithm for power to a number

```
procedure power(base,exp) : integer
while (exp != 0) do
 BEGIN
 result *= base;
 --exp;
 END while
print  result
end procedure
```

Program to find Power of a number

```
#include <stdio.h>
int main() { int base, exp;  long result = 1;
printf(“\nEnter a base number: "); scanf("%d", &base);
printf(“\nEnter an exponent: "); scanf("%d", &exp);
while (exp != 0)
 { result *= base; --exp; }
printf(“\nAnswer = %ld", result);
return 0; }
```

Output

Enter a base number: 3

Enter an exponent: 4

Answer = 81

Raising a number to a large power

- We have given two numbers x and n which are base and exponent respectively. Write a function to compute x^n where 1 <= x, n <= 10000 and overflow may happen.
  Examples:

- Input : x = 5, n = 20
- Output : 95367431640625
- Input : x = 2, n = 100
- Output : 1267650600228229401496703205376

Raising a number to a large power

- In the above example, 2^100 has 31 digits and it is not possible to store these digits even if we use long long int which can store maximum 18 digits. The idea behind is that multiply x, n times and store result in res[] array.
  Here is the algorithm for finding power of a number.
  Power(n)
  1. Create an array res[] of MAX size and store x in res[] array and initialize res_size as the number of digits in x.
  2. Do following for all numbers from i=2 to n
  …..Multiply x with res[] and update res[] and res_size to store the multiplication result.
  Multiply(res[], x)
  1. Initialize carry as 0.
  2. Do following for i=0 to res_size-1
  ….a. Find prod = res[i]*x+carry.
  ….b. Store last digit of prod in res[i] and remaining digits in carry.
  3. Store all digits of carry in res[] and increase res_size by number of digits.

C program to compute large power of a number

C program to compute large power of a number

```
#include <stdio.h>
 // Maximum number of digits in  output
#define MAX 100000

// This function multiplies x with the number represented by res[].
// res_size is size of res[] or number of digits in the number represented by res[]. This function
// uses simple school mathematics for multiplication. This function may value of res_size
// and returns the new value of res_size

int multiply(int x, int res[], int res_size) {
 // Initialize carry
int carry = 0;
 // One by one multiply n with individual digits of res[]
for (int i = 0; i < res_size; i++) {
    int prod = res[i] * x + carry;

    // Store last digit of 'prod' in res[]
    res[i] = prod % 10;

    // Put rest in carry
    carry = prod / 10;
}
 // Put carry in res and increase result size
while (carry) {
    res[res_size] = carry % 10;
    carry = carry / 10;
    res_size++;
}
```

Q & A