Unit - 3

Chapter

13

Factoring Methods

Chapter Outline

13.1 Introduction

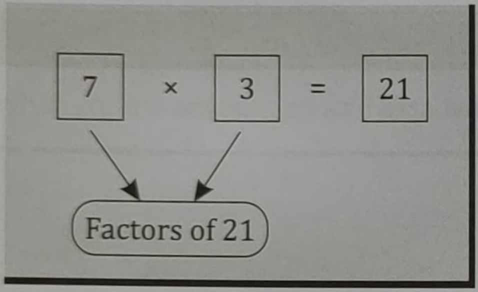
The **factor** is a number or algebraic expression that divides another number or expression evenly i.e its remainder is 0. (or) factors are small numbers when multiplied gives other numbers. For example, 3 and 6 are factors of 12 because 12 ÷ 3 = 4 exactly and 12 ÷ 6 = 2 exactly. The other factors of 12 are 1, 2, 4, and 12.

Factoring or **Factorization** is nothing but writing a number as the product of smaller numbers. It is the decomposition of a number (or) mathematical objects into smaller or simpler numbers/objects. In simple words, factoring is the process of finding the factors:

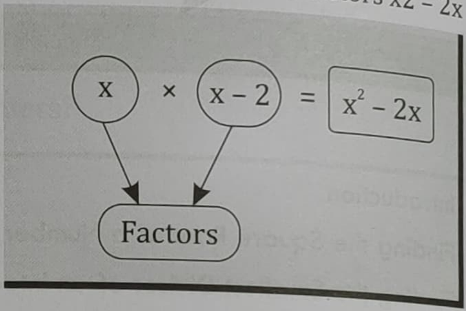
Note: Factoring is the reverse process of multiplication.

Example

For example, 7 and 3 are factors of 21.



For example, x and x – 2 are the factors x2 – 2x



Algebraic expressions also have factors.

Here, x × 2x is the factored form of $x^2 - 2x$.

So, factors can be numbers, algebraic expressions, or algebraic variables.

The Theory of Numbers, especially factoring methods, has been thoroughly studied in the field of mathematics. Factoring of integers into its prime numbers has many practical applications in encryption, hashing technique etc. A numerical simulation is a computer-based study that uses a software to implement a mathematical model of a physical system. In numerical simulations, random numbers are widely employed. Methods for factoring large whole numbers are of great importance in public-key cryptography.

13.2 Finding the Square Root of a Number

Problem

Given a number *m*, design an algorithm to calculate its square root.

There are in built functions in all programming languages to calculate square root of any given number but idea here is to find the square root of a number without using any predefined function. The algorithm should be designed to find the square root of a number using an iterative method.

A square root of a number is a value that, when multiplied by itself, gives the number.

We know that, 2 × 2 = 4, so square root of 4 is 2.

3 × 3 = 9, so square root of 9 is 3.

From the above examples it is clear that, In general, The square root 'n' of a number 'm. must satisfy the equation

$$n \times n = m \quad \text{..............(1)}$$

It's easy to find the square root of a perfect square. A perfect square is a positive number made by multiplying a number by itself. In other words, a perfect square is a number that is the result of an integer raised to the power of 2.

Perfect Square are: $1^2 = 1$, $2^2 = 4$, $3^2 = 9$, $4^2 = 16$, $5^2 = 25$, $6^2 = 36$, $7^2 = 49$, $8^2 = 64$ and so on.

We can use four methods to find the square root of numbers and those methods are as follows:

1. Repeated Subtraction Method of Square Root
2. Square Root by Prime Factorization Method
3. Square Root by Long Division Method
4. Square Root by Approximation Method (Newton's method).

Let us consider Newton's method of solving a square root of a number.

Example 1 **To Find the Square Root of 49**

1. To find the square root of 49, start with an initial guess of 9.
2. Multiply 9 by itself: 9 × 9 = 81, which is greater than 49, so 9 is too high.
3. Try a lower guess of 8.
4. Multiply 8 by itself: 8 × 8 = 64, which is still greater than 49 but closer.
5. Try a further lower guess of 7.
6. Multiply 7 by itself: 7 × 7 = 49, which exactly equals 49.
7. So, 7 is the square root of 49.
8. The number of tries needed depends on how close the initial guess is.

Example 2 **Find $\sqrt{6}$ to 4 Decimal Places**

Since $2^2 = 4$ and $3^2 = 9$, we know that $\sqrt{6}$ is between 2 and 3. Let's guess it to be 2.5. Squaring 2.5 gives us $2.5^2 = 6.25$, which is too high. So, we make a slightly lower guess of 2.4. To find an approximation to four decimal places, we need to keep refining our guess until we have five decimal places, and then we can round the result.

Guess	Square of guess	High/low
2.4	5.76	Too low
2.45	6.0025	Too high but real close
2.449	5.997601	Too low
2.4495	6.00005025	Too high, so between 2.449 and 2.4495
2.4493	5.99907049	Too low
2.4494	5.99956036	Too low, so between 2.4494 and 2.4495
2.44945	5.9998053025	Too low, so between 2.44945 and 2.4495.

This is sufficient because the result will round to 2.4495 (and not to 2.4494).

Newton's Method to Find the Square Root of a Number

Given an number 'm' and a tolerance level 'L', the task is to find the square root of that number using Newton Method.

1. The square root of 'm' can be given by the formula: root = 0.5 * (n + (m / n)) where 'n' is any guess which can be assumed to be m or m/2 or 1.
2. In this formula, 'n' is any assumed square root of 'm' and root is the calculated square root of m for assumed number 'n'.
3. The tolerance level 'L' is the maximum allowed difference between 'n' and 'root' to consider the estimate accurate enough.
4. We use this formula inside a loop and try to guess n, starting from number from m or m/2 and then we update it with each iteration with new guessed root. Continue this process until the difference between 'n' and 'root' is less than or equal to the tolerance level 'L'.
5. Once the difference meets the tolerance level, the loop stops, and the current value of 'root' is taken as the square root of 'm'.

Example 3 **Find the Square Root of 256 Using Newton's Method.**

Let m=256, L=0.1 and let us assume $n = \frac{m}{2} = \frac{256}{2} = 128$

Iteration 1: root $= 0.5 * \left(n + \left(\frac{m}{n}\right)\right)$

$= 0.5 * \left(128 + \left(\frac{256}{128}\right)\right)$

$= 0.5 * (128 + 2)$

$= 0.5 * 130$

$= 65$

(n=128 and root =65, the difference (n-root) is greater than tolerance limit 0.1 and hence lets continue with next iteration)

Assign root to n

Therefore n=65

teration 2: root $= 0.5 * \left(n + \left(\frac{m}{n}\right)\right)$

$= 0.5 * \left(65 + \left(\frac{256}{65}\right)\right)$

$= 0.5 * (65 + 3.93)$

$= 0.5 * 68.93$

$= 34.46$

(n=65 and root =34.46, the difference (n - root) is greater than tolerance limit 0.1 and hence lets continue with next iteration)

Assign root to n

Therefore n=34.46

Iteration 3: root $=0.5*\left(n+\left(\frac{m}{n}\right)\right)=0.5*\left(34.46+\left(\frac{256}{34.46}\right)\right)$

$= 0.5 * (34.46 + 7.42)$

$= 0.5 * 41.88$

$= 20.94$

(n=34.46 and root =20.94, the difference (n - root) is greater than tolerance limit 0.1 and hence lets continue with next iteration)

Assign root to n

Therefore n=20.94

Iteration 4: root $=0.5*\left(n+\left(\frac{m}{n}\right)\right)=0.5*\left(20.94+\frac{256}{20.94}\right)$

$= 0.5 * (20.94 + 12.22)$

$= 0.5 * 33.16$

$= 16.58$

(n=20.94 and root =16.58, the difference (n - root) is greater than tolerance limit 0.1 and hence lets continue with next iteration)

Assign root to n

Therefore n=16.58

Iteration 5: root $=0.5*\left(n+\left(\frac{m}{n}\right)\right)=0.5*\left(16.58+\left(\frac{256}{16.58}\right)\right)$

$= 0.5 * (16.58 + (256/16.58)$

$= 0.5 * (16.58 + 15.44)$

$= 0.5 * 32.02$

$= 16.01$

(n=16.58 and root =16.01, the difference (n - root) is greater than tolerance limit 0.1 and hence lets continue with next iteration)

Assign root to n

Therefore n=16.01

Iteration 6: root $=0.5*\left(n+\left(\frac{m}{n}\right)\right)=0.5*\left(16.01+\left(\frac{256}{16.01}\right)\right)$

$= 0.5 * (16.01 + (256/16.01)$

$= 0.5 * (16.01 + 15.99)$

$= 0.5 * 32$

$= 16$

(n=16.01 and root =16, the difference (n - root) is less than tolerance limit 0.1 and hence stop the iteration)

Therefore Square root of 256 is 16. We have calculated the square root of 256 in 6 iterations with initial guess value as m/2. The number iterations will vary based on the initial guess value.

Algorithm **To Find the Square Root of a Number using Newton's Method**

Let *m* be the number whose square root to be found and tolerance limit is L.

```
Step 1:   Start.
Step 2:   Declare variables m, n, L and root.
Step 3:   Read a number m and Tolerance Limit L
Step 4:   Initialize root to 0 (root=0)
Step 5:   set initial guess n=m/2
Step 6:   Repeat the following steps until absolute difference between n and root is
          less than tolerance limit L.
            (a)  root = 0.5 * (n + (m / n))
            (b)  n = root
Step 7:   Print root.
Step 8:   End
```

Program 1 **C Program to Find the Square Root of a Number using Newton's Method.**

```c
#include <stdio.h>
#include <stdlib.h>

double squareRoot(double m, double L)
{
        double root, n;
        root=0.0;
        n=m/2;
        while(1)
        {
            root = 0.5 * (n + (m / n));
            if(abs(root-n) < L)
                break;
            n = root;
        }
        return root;
}

void main()
{
        double m, L;
        printf("\n Enter a Number to Find the Square Root (m) : ");
        scanf("%lf",&m);
        printf("\n Enter Tolerance Limit (L)) : ");
        scanf("%lf",&L);
        printf("Approximate Square Root of %lf is : %lf",m,squareRoot(m,L));
}
```

Output :

```
Enter a Number to Find the Square Root (m) : 299
Enter Tolerance Limit (L)) : 0.00001
Approximate Square Root of 299.000000 is : 17.312026
```

Program 2 C Program to Find the Square Root of a Number using Predefined Function sqrt().

```
#include <stdio.h>
#include <math.h>

void main()
{
    double m;
    printf("\n Enter a Number to Find the Square Root : ");
    scanf("%lf",&m);
    printf("\n Square Root of %lf is : %lf", m, sqrt(m));
}
```

Output :

```
Enter a Number to Find the Square Root : 256
Square Root of 256.000000 is : 16.000000
```

13.3 Finding the Smallest Divisor of an Integer

Problem

Design an algorithm to find smallest exact divisor of an integer *n* other than one.

A factor of a number is an exact divisor of that number. An exact divisor of a number divides that number with no remainder. For any number *n*, there are at least two divisors. All divisors of *m* are between 1 and *n* and can be totally ordered.

Example 1

Consider the even number 64. The complete set of divisors is {1, 2, 4, 8, 16, 32}.

Smallest exact divisor of 64 other than 1 is 2.

Consider the odd number 81. The complete set of divisors is {1, 3, 9, 27, 81}

Smallest exact divisor of 81 other than 1 is 3.

Consider the prime number 31. The complete set of divisors is {1, 31}

Smallest exact divisor of 31 other than 1 is 31.

Example 2 How to find the smallest exact divisor of an integer

Suppose we have an integer n and need to find its smallest divisor other than 1. A common approach is to start from 2 and keep dividing n by successive numbers until we find one that divides n evenly.

Now, imagine n is a large prime number, like 1013. Would we really check every number from 2 up to 1012 to see if any of them divides 1013? This would require a lot of time and effort, making it an inefficient solution.

Do we have a better way?

Consider the number 64. The complete set of divisors other than one is {2, 4, 8, 16, 32}.

Let us take the first divisor 2. We can easily observe that:

$2 * 32 = 64$

Similarly we can take other numbers and observe the following:

$4 * 16 = 64$ $8 * 8 = 64$ $16 * 4 = 64$

So from this we can observe that the smallest divisor 2 is linked with the largest divisor 32. Similarly the second smallest divisor 4 is linked with second largest divisor 16. This goes on with 8 linked with 8 and finally 16 linked with 4.

Considering the complete set of divisors of 64, we can see that smallest divisor 2 is linked with largest divisor 32 , second smallest divisor is linked with second biggest divisor and so on.

Smaller factor is linked with	Bigger factor
2	32
4	16
8	8
16	4

So if we want to generalize this, we can observe that if any number n has a divisor then that divisor is always linked with another divisor. The general pattern is that a smaller divisor is linked with a larger one and vice versa.

smaller divisor * larger divisor = n

Another interesting thing that can be observed is that if we go on increasing the value of smaller divisor and consider the next larger divisor, at the same time if we go on decreasing the value of the larger divisor and consider the next smaller divisor then we reach a cross over point. At the cross over point we observe that smaller divisor = larger divisor.

$2 \rightarrow 4 \rightarrow 8$

$32 \rightarrow 16 \rightarrow 8$

In this example, cross over point is 8.

So what is 8 in our case? 8 is the square root of 64.

So once we find the square root of any number n we don't need to look beyond the value of the square root for the smallest divisor. The other side of the square root will always have the larger divisors.

Steps to find the the smallest exact divisor of an integer:

Step 1. Check n is divisible by 2, If yes then 2 will be the smallest divisor

Step 2. Iterate from i = 3 to sqrt(n) and making a jump of 2. If any value divides the n in that particular range. Then that will be the the smallest divisor.

Step 3. If nothing divides, then n is a prime number and it is a smallest divisor.

Further to improve the design of algorithm few important considerations are as follows:

1. If a number **n** is divisible by 2 (even) then smallest divisor is 2
2. if the number n is odd, test all the odd divisors (i.e., 3, 5, 7,.....) which are less than or equal to $\sqrt{n}$.
3. The divisors to test can be generated with equation div = div + 2 ,with initial value of div =3
4. To check a number is an exact divisor or number **mod** function is used. If ***n mod div = 0*** (i.e., no remainder after division) then **div** is an exact divisor.
5. If ***n*** is the smallest divisor of ***n*** then it is a prime number and the smallest divisor is 1.

Algorithm **To Find the Smallest Divisor of an Integer n**

```
        Start
step 1: Declare variables n, r, div
step 2: Read n
step 3: If n mod 2 is equal to 0 then print 2 is smallest divisor and go to Step 9
step 4: Compute r = square root of n
step 5: Initialize divisor div to 3
step 6: while div is less than or equalt to r do the following steps
step 7: (a) if (n mod div is equal to 0) then print div is the smallest divisor and
            go to step9
        (b) div = div + 2
step 8: print n is the smallest divisor
step 9: End
```

Program 3 **C Program to Find the Smallest Divisor of an Integer**

```c
#include <stdio.h>
#include <math.h>

int smallestDivisor(int number)
{
    if(number % 2 == 0)           //if given number is even then smallest divisor will be 2
      return 2;
    else
    {
      int r = sqrt(number);
      int div = 3;
      while(div <= r)             //looping from 3 to square root of given number
      {
         if(number % div == 0)  //checking for smallest divisor
               return div;
         div  = div + 2;
      }
    }
    return number;                //in case of prime number it will return number
}
void main()
{
    int n;
    printf("Enter the Number:");
    scanf("%d",&n);
    int result=smallestDivisor(n);
    printf("Smallest divisor: %d",result);
}
```

Output :

```
Enter the Number:100
Smallest divisor: 2
Enter the Number:725
Smallest divisor: 5
Enter the Number:1013
Smallest divisor: 1013
```

13.4 The Greatest Common Divisor of Two Integers

Problem

Design an algorithm to find greatest common divisor (GCD) of two positive non zero integers *a* and *b*.

The greatest common divisor (GCD) refers to the greatest positive integer that is a common divisor for a given set of positive integers. It is also termed as the highest common factor (HCF) or the greatest common factor (GCF).

For a set of positive integers (a, b), the greatest common divisor is defined as the greatest positive number which is a common factor of both the positive integers (a, b). (or) The GCD of two or more integers is the largest integer that divides each of the integers such that their remainder is zero.

GCD of 20, 30 = 10 (10 is the largest number which divides 20 and 30 with remainder as 0)

GCD of 42, 120, 285 = 3 (3 is the largest number which divides 42, 120 and 285 with remainder as 0)

GCD of any two numbers is never negative or 0 as the least positive integer common to any two numbers is always 1.

Example 1 **Find the Greatest Common Divisor of 13 and 48.**

We will use the below steps to determine the greatest common divisor of (13, 48).

Divisors of 13 are 1, and 13.

Divisors of 48 are 1, 2, 3, 4, 6, 8, 12, 16, 24 and 48.

The common divisor of 13 and 48 is 1.

The greatest common divisor of 13 and 48 is 1.

Thus, GCD(13, 48) = 1.

Example 2 **Find the Greatest Common Divisor of 10 and 20.**

Divisors of 10 are 1, 2, 5, and 10.

Divisors of 20 are 1, 2, 4, 5, 10, and 20.

The common divisor of 10 and 20 is 1, 2, 5, and 10

The greatest common divisor of 10 and 20 is 10.

Thus, GCD(10, 20) = 10.

Methods to Find GCD:

There are several methods to find the greatest common divisor of given two numbers.

1. Prime Factorisation Method
2. Long Division Method
3. Euclid's Division Algorithm

Euclid's Algorithm (or Euclidean Algorithm) is a method for efficiently finding the greatest common divisor (GCD) of two numbers. Let us discuss Euclid's algorithm to find the GCD of two numbers in this section.

"modulo" Operation:

The modulo operation gives the remainder when two positive integers are divided. In C Programming, The modulo operator is denoted by %. The modulo division operator produces the remainder of an integer division.

We write it as follows : A % B = R

This means, dividing A by B gives the remainder R, this is different from division operation (÷) which gives the quotient.

Example 3

7 % 2 = 1 (Dividing 7 by 2 gives the remainder 1)

42 % 7 = 0 (Dividing 42 by 7 gives the remainder 0)

The **modulo** operation computes the remainder after integer division and can be used to find an exact divisor of a number.

Steps to Find the GCD of Two Numbers a and b using Euclid's Algorithm

Step 1: If a=0 then GCD(a, b)=b since the Greatest Common Divisor of 0 and b is b. Return the value of 'b' and stop. Otherwise go to step 2.

Step 2: If b=0 then GCD(a,b)=a since the Greates Common Divisor of 0 and a is a. Return the value of 'a' and stop. Otherwise go to step 3.

Step 3: Let r be the remainder of dividing a by b assuming a > b

r = a % b

If r=0 then GCD(a,b)=b. Return the value of 'b' and stop. Otherwise go to step 4.

Step 4: The smaller integer ***b*** is taken as larger integer and remainder ***r*** is taken as the divisor.

Assign the value of 'b' to 'a' and 'r' to 'b'.

Go to Step 3 to Find GCD(b, r)

The process continues until remainder becomes zero in Step 3.

Example 4 **Find GCD of 285 and 741 using Euclid's algorithm.**

To calculate the GCD of 285 and 741:

1. Since 285 is less than 741, start by calculating GCD(741, 285).
2. Divide 741 by 285, which gives a remainder of 171. Now, calculate GCD(285, 171).
3. Divide 285 by 171, which gives a remainder of 114. Now, calculate GCD(171, 114).
4. Divide 171 by 114, which gives a remainder of 57. Now, calculate GCD(114, 57).
5. Divide 114 by 57, which gives a remainder of 0.

The process stops here because the remainder is 0. Therefore, the Greatest Common Divisor (GCD) of 285 and 741 is 57. This result was obtained by repeatedly dividing and taking the remainder until we reached a remainder of 0, confirming that 57 is the largest number that divides both 285 and 741 without leaving a remainder.

Example 5 Find GCD of 40 and 64 using Euclid's algorithm

64 % 40 = 24

40 % 24 = 16

24 % 16 = 8

16 % 8 = 0

We stop here since we've already got a remainder of 0. The last number we used to divide is 8 so the GCD of 40 and 64 is 8.

Example 6 Find GCD of 16 and 28 Euclid's algorithm

28 % 16 =12

16 % 12 = 4

12 % 4 = 0 (remainder is zero)

Hence GCD (16,28) is 4.

Algorithm To Find Greatest Common Divisor of Two Integers a and b

```
Step 1: Start
Step 2: Declare variables a, b, r
Step 3: Read two numbers a and b
Step 4: if a is 0, GCD of a and b is b. return b and stop
Step 5: if b is 0, GCD of a and b is a. return a and stop
Step 6: Repeat below steps until remainder 'r' becomes zero
            (a) r = a % b
            (b) a = b
            (d) b = r
Step 7: Return b
Step 8: End
```

Program 4 C Program to Find Greatest Common Divisor of Two Integers

```c
#include <stdio.h>

int GCD(int a,int b)
{
        if(a==0)
            return b;
        if(b==0)
            return a;
        int r;
        while(a % b > 0)
        {
            r = a % b;
            a = b;
            b = r;
        }
        return b;
}
```

Output :

```
Enter Two Numbers:0 20
GCD of 0 and 20 are: 20
Enter Two Numbers:25 0
GCD of 25 and 0 are: 25
Enter Two Numbers:285 741
GCD of 285 and 741 are: 57
```

```
void main()
{
    int a,b;
    printf("\n Enter Two Numbers:");
    scanf("%d %d",&a,&b);
    printf("\n GCD of %d and %d are: %d",a,b, GCD(a,b));
}
```

13.5 Generation of Prime Numbers

A number is prime if it has exactly two positive divisors, 1 and itself. Otherwise it is composite. The number 1 is usually regarded as being neither prime nor composite.

2, 3, 5, 7, 11, 13, 17, 19, 23 are first few prime numbers. All prime numbers other than 2 are odd.

A method to generate prime numbers, should check whether a number is prime or not. From the definition of prime number we know that, if a number is prime it will have no exact divisors other than 1 and itself .

A straightforward method used for the generation of prime numbers is the **Sieve of Eratosthenes.**

Eratosthenes, a famous Greek mathematician devised a 'sieve' to identify prime numbers. The Sieve of Eratosthenes is a simple and ancient algorithm for finding all prime numbers up to a specified integer. The Sieve of Eratosthenes drains out composite numbers and leaves prime numbers behind. This generates all of the prime numbers less than or equal to n. Start by writing the numbers from 2, 3, 4 ... n in a line. Then keep repeating the following process until all the elements or numbers less than or equal to $\sqrt{n}$ have been crossed out or circled.

Circle the next number which is neither circled nor crossed out, and cross out all other multiples of that number which are in the list (some of these are probably already crossed out).

The numbers which are not crossed out when the process terminates are all of the primes between 2 to n.

Example

To list out the prime numbers less than or equal to 36.

i.e., when n = 36

2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28
29	30	31	32	33	34	35	36	

The square root of 36 is 6 and therefore we have to repeat the process mentioned earlier until all the elements are crossed out or circled less than or equal to 6.

Consider a first number 2, which is not yet circled or crossed out. Circle the number 2 and cross-out all the multiples of 2 except 2 i.e., 4, 6, 8, 10 ...... 36.

②	3	$\cancel{4}$	5	$\cancel{6}$	7	$\cancel{8}$	9	$\cancel{10}$
11	$\cancel{12}$	13	$\cancel{14}$	15	$\cancel{16}$	17	$\cancel{18}$	19
$\cancel{20}$	21	$\cancel{22}$	23	$\cancel{24}$	25	$\cancel{26}$	27	$\cancel{28}$
29	$\cancel{30}$	31	$\cancel{32}$	33	$\cancel{34}$	35	$\cancel{36}$	

Consider the next number which is neither crossed out nor circled, i.e., number 3. Now, circle the number 3 and cross out all the multiples of 3 except 3.

i.e., (6, 9, 12, 15, ... 33, 36).

If some of them are already crossed out, then no need to cross them again.

Consider the next number which is neither crossed out nor circled, i.e., number 5. Now, circle the number 5 and cross out the multiple of 5 except 5. i.e., 10, 15, ... 35.

②	③	$\cancel{4}$	⑤	$\cancel{6}$	7	$\cancel{8}$	$\cancel{9}$	$\cancel{10}$
11	$\cancel{12}$	13	$\cancel{14}$	$\cancel{15}$	$\cancel{16}$	17	$\cancel{18}$	19
$\cancel{20}$	$\cancel{21}$	$\cancel{22}$	23	$\cancel{24}$	$\cancel{25}$	$\cancel{26}$	$\cancel{27}$	$\cancel{28}$
29	$\cancel{30}$	31	$\cancel{32}$	$\cancel{33}$	$\cancel{34}$	$\cancel{35}$	$\cancel{36}$	

Now, we cannot consider any number since all the numbers from 2 to 6 [Sqrt (36)] are crossed out or circled. So we terminate the process.

The numbers that are not crossed out are the prime numbers between 2 and 36.

2	3	5	7	11	13	17	19	23	29	31

13.6 Computing the Prime Factors of an Integer

Problem

Every integer can be expressed as a product of prime numbers. Design an algorithm to compute all the prime factors of an integer n.

? What are Factors and Prime Factors?

Factors of a number are the numbers that can be multiplied together to get the original number. For example, 4 and 5 are factors of 20 because $4 \times 5 = 20$.

Prime factors of a number are the prime numbers that, when multiplied together, give the original number. For instance, the prime factors of 20 are 2, 2, and 5 because $2 \times 2 \times 5 = 20$. A prime factor is simply a factor of the number that is also a prime number.

For example, consider the number 30. We can factorize 30 as $30 = 5 \times 6$, but 6 is not a prime number. We can further break down 6 into 2×3, where 2 and 3 are prime numbers. So, the prime factors of 30 are $2 \times 3 \times 5$, where all the factors are prime numbers.

? What is the Meaning of Prime Factorization?

The process of expressing a number as the product of prime numbers is called prime factorization. Prime numbers are those that have only two factors: 1 and the number itself. Examples of prime numbers include 2, 3, 5, 7, 11, 13, 17, 19, and so on.

Prime factorization involves breaking down a number into prime numbers that, when multiplied together, give the original number. In other words, prime factorization is the method of decomposing a number into its prime factors, which, when multiplied, produce the original number.

Example 1

The Prime factors of 108 = 2, 2, 3, 3, 3

$$\begin{array}{r|r} 2 & 108 \\ \hline 2 & 54 \\ \hline 3 & 27 \\ \hline 3 & 9 \\ \hline & 3 \end{array}$$

Product of prime factors = 2 × 2 × 3 × 3 × 3 = 108

Therefore product of prime factors represents the number 108

Example 2

The prime factors of 288 = 2, 2, 2, 2, 2, 3, 3

$$\begin{array}{r|r} 2 & 288 \\ \hline 2 & 144 \\ \hline 2 & 72 \\ \hline 2 & 36 \\ \hline 2 & 18 \\ \hline 3 & 9 \\ \hline & 3 \end{array}$$

Product of prime factors = 2 × 2 × 2 × 2 × 2 × 3 × 3 = 288

Therefore product of prime factors represents the number 288.

Example 3

Prime factorization of 12 is $2 \times 2 \times 3 = 2^2 \times 3 = 12$

Prime factorization of 18 is $2 \times 3 \times 3 = 2 \times 3^2 = 18$

Prime factorization of 24 is $2 \times 2 \times 2 \times 3 = 2^3 \times 3 = 24$

Prime factorization of 20 is $2 \times 2 \times 5 = 2^2 \times 5 = 20$

Prime factorization of 36 is $2 \times 2 \times 3 \times 3 = 2^2 \times 3^2 = 36$

Note

The input number need not be a prime number. But the factors of the number must be prime number.

For example, 24 and 35 are not prime numbers. But the prime factors of 24 are 2, 2, 2, and 3. Here both 2 and 3 are prime numbers. Similarly, prime factors of 35 are 5 and 7. Both 5 and 7 are prime numbers.

Algorithm To Compute the Prime Factors of an Integer

```
Step 1: Start
Step 2: Declare variables n and i
Step 3: Read number n
Step 4: While n is divisible by 2, repeat the following steps
            (a) Print 2
            (b) Divide n/2 (n = n/2)
Step 5: After step 4, n must be odd.
            i = 3 to the square root of n, repeat the following steps
            (a) while i divides n, print i and divide n by i (n = n/i)
            (b) After i fails to divide n, increment i by 2 and continue.
Step 6: if n is greater than 2, print n
Step 7: End
```

Steps to Compute the Prime Factors of an Integer

Let us consider n = 36.

Step 1: Since 36 % 2 = 0 $\boxed{\text{Print 2}}$ Divide 36/2 So, n becomes 18 (n=18)

Since 18 % 2 = 0 $\boxed{\text{Print 2}}$ Divide 18/2 So, n becomes 9 (n=9)

Since 9 % 2 = 1, stop iterating and go to next step

At the end of this step n becomes odd.

n = 9 at the end of step 1.

Step 2: Square root of n = Square root of 9 = 3

Iterate this step from i = 3 to sqrt(9) i.e., i = 3 to 3 (It means this step will be executed only once)

Check n % i is divisible, where i = 3

Since 9 % 3 = 0 $\boxed{\text{Print 3}}$ Divide 9/3, So, n becomes 3 (n=3)

Increment i by 2, So i becomes 5

Loop will terminate as i value is 5.

n = 3 at the end of step 2.

Step 3: Check if n is greater than 2

Since 3 > 2 is true, print the value of n $\boxed{\text{Print 3}}$

We have printed 2, 2, 3 and 3 in above steps.

The prime factors of 36 are 2, 2, 3, 3

Product of prime factors = 2 × 2 × 3 × 3 = 36

Program 5 C Program to Compute the Prime Factors of an Integer.

```
#include <stdio.h>
#include <math.h>

void primeFactors(int n)

{   // Print the number of 2's that divide n
    printf("\n Prime Factors of %d are : ",n);
    int i;
    while (n % 2 == 0)
    {
        printf(" %d ", 2);
        n = n/2;

    }
    // n must be odd at this point. So we can skip one element (Note i = i +2)
    for (i = 3; i <= sqrt(n); i = i + 2)
    {
        // While i divides n, print i and divide n
        while (n % i == 0)
        {
            printf(" %d ", i);
            n = n/i;
        }
    }
    // This condition is to handle the case when n is a prime number greater than 2
    if (n > 2)
        printf(" %d ", n);
}

void main()
{
    int n;
    printf("\n Enter a number to find the prime factors : ");
    scanf("%d",&n);
    primeFactors(n);
}
```

Output 1 :

```
Enter a number to find the prime factors : 108
Prime Factors of 108 are :  2  2  3  3  3
```

Output 2 :

```
Enter a number to find the prime factors : 36
Prime Factors of 108 are :  2  2  3  3
```

Output 3 :

```
Enter a number to find the prime factors : 20
Prime Factors of 20 are :  2  2  5
```

Output 4 :

```
Enter a number to find the prime factors : 315
Prime Factors of 315 are :  3  3  5  7
```

13.7 Generation of Pseudo-Random Numbers

Problem

Use the Linear Congruential Method to generate a uniform set of pseudorandom numbers.

A random number generator is a hardware device or software algorithm that generates a number that is taken from a limited or unlimited distribution and outputs it. These numbers are generated using some kind of mathematical algorithms which uniformly distributes all numbers preset in the sequence.

In simple words, a random number is a value produced by a hardware device or software algorithm, and we don't know what it will be until it is generated.

Generating truly random numbers is very challenging. The main difficulty lies in the fact that a computer is a machine that follows precise instructions, making it impossible for it to create something genuinely unpredictable. Computers are deterministic, meaning their behavior is entirely predictable, and nothing they do is truly random. To create numbers that appear random, computers use mathematical algorithms designed to simulate randomness. These random number generators are an attempt to produce results that seem unpredictable, even though they are based on predefined algorithms.

What are the applications of Random Numbers?

Random numbers have many applications, but one of the most important is in cryptography, where they play a key role in creating encryption keys. The quality of random numbers used in cryptography directly affects the security of the system. If the random number generator is strong and produces high-quality random numbers, it becomes very difficult for anyone to break into the system.

Random number generators are also used across various industries to produce random values. These industries include gaming, financial institutions, secure communications, cybersecurity, and many others. The reliability of random numbers is crucial in ensuring the effectiveness and security of these applications.

Types of Random Numbers

Random numbers are classified in two categories true random numbers and pseudo random numbers.

1. True Random Numbers

To generate a "true" random number, a computer measures a physical phenomenon that occurs outside the computer. For example, it might measure the radioactive decay of an atom. According to quantum theory, the exact moment when radioactive decay occurs is unpredictable, making this a source of "pure randomness" from the universe. Because this process is inherently unpredictable, an attacker wouldn't be able to guess the random value.

2. Pseudorandom Numbers

Pseudorandom numbers are generated by computers. They aren't truly random because a computer's behavior is entirely predictable—computers are deterministic devices, meaning their actions follow a set sequence based on initial conditions. To simulate unpredictability, computers use mathematical algorithms to generate numbers that appear random, but they are actually "random enough" for most purposes.

A Pseudorandom Number Generator (PRNG) is a program or function that uses a mathematical process to produce numbers that simulate randomness. PRNGs may also be called Digital Random Number Generators (DRNG) or Deterministic Random Bit Generators (DRBG).

PRNG requires only two steps:

1. Provide the PRNG with an arbitrary seed.
2. Ask for the next random number.

The seed value acts as the "starting point" for generating random numbers. This value influences the sequence of numbers that the PRNG produces. If the seed value changes, the generated numbers change as well. However, the same seed value will always produce the same sequence of numbers, which means the numbers aren't truly random—true randomness can't be recreated.

A common practice is to use the current time as a unique seed value. For example, the precise time of May 5, 1977, at 5:03 A.M. and 7.01324 seconds UTC can be converted into an integer and used as a seed. Since that exact moment will never occur again, a PRNG seeded with that time should generate a unique sequence of random numbers.

Pseudorandom numbers offer an alternative to "true" random numbers. A random number generator that doesn't rely on real-world phenomena to produce its sequence is referred to as a **Pseudorandom Number Generator (PRNG).**

Linear Congruential Generator to Generate Pseudorandom Numbers

Linear Congruential Generator is one of the oldest and best known pseudorandom number generator algorithms. Liner congruential generator is a simple example of pseudorandom number generator. It was developed by D. H. Lehmer in 1949.

Sequence of pseudorandom numbers $x_1, x_2, \ldots$ between 0 and m-1 are generated using the below expression.

$$x_{n+1} = (a\,x_n + b) \bmod m$$

Each of these members have to satisfy the following conditions:

$m > 0$	(the modulus is positive),
$0 < a < m$	(the multiplier is positive but less than the modulus),
$0 \le b < m$	(the increment is non negative but less than the modulus), and
$0 \le x_0 < m$	(the seed is non negative but less than the modulus).

When appropriate values are chosen for these parameters, the algorithm can produce a long sequence that appears random. However, it's important to note that pseudorandom generators are deterministic. This means that the sequence is generated by following a predictable pattern based on the initial conditions.

Theprocessworksbygeneratingthenextrandomintegerusingthepreviousone,alongwiththeconstantsa, b, and m. To start the process, an initial seed x_0 must be provided. The appearance of randomness is achieved by using modulo arithmetic, which keeps the numbers within the desired range.

However, if an attacker knows a small number of values in the sequence and has information about a, b, and m, they could potentially reconstruct the entire sequence. To reduce this risk, the value of m should be very large, allowing the generation of a long series of distinct random numbers. A common choice for m is a value close to or equal to 2^{31}, which is near the maximum representable non-negative integer for many computers.

A computer executes code that is based on a set of rules to be followed. For PRNGs in general, those rules revolve around the following:

1. Accept some initial input number x_0, that is a seed or key.
2. Apply that seed in a sequence of mathematical operations to generate the result. That result is the random number.
3. Use that resulting random number as the seed for the next iteration.
4. Repebat the process to generate pseudorandom numbers.

Example

Using Linear Congruential method, generate a sequence of Random numbers with $x_0 = 27$, $a = 17$, $b = 43$ and $m = 100$ using expression $x_{n+1} = (ax_n + b) \bmod m$ for $n \ge 0$

$x_1 = x_{0+1} = (ax_0 + b) \bmod m$ $= [(17 \times 27) + 43] \bmod 100$ $= 502 \bmod 100$ $= 2$	$x_2 = (ax_1 + b) \bmod m$ $= (17 \times 2) + 43] \bmod 100$ $= 77 \bmod 100$ $= 77$

$x_3 = (ax_2 + b) \bmod m$ $= [(17 \times 77) + 43] \bmod 100$ $= 1352 \bmod 100$ $= 52$	$x_4 = (ax_3 + b) \bmod m$ $= [(17 \times 52) + 43] \bmod 100$ $= 927 \bmod 100$ $= 27$
$x_5 = (ax_4 + b) \bmod m$ $= [(17 \times 27) + 43] \bmod 100$ $= 502 \bmod 100$ $= 2$	$x_6 = (ax_5 + b) \bmod m$ $= [(17 \times 2) + 43] \bmod 100$ $= 77 \bmod 100$ $= 77$

By observing the above random numbers, it is understood that the sequence repeats. (This indicates a weakness of our example generator: If the random numbers are between 0 and 99 then one would like every number between 0 and 99 to be a possible member of the sequence. The parameters a, b and m determine the characteristics of the random number generator, and the choice of x_0 (the seed) determines the particular sequence of random numbers that is generated. If the generator is run with the same values of the parameters, and the same seed, it will generate a sequence that's identical to the previous one. In that sense the numbers generated certainly are not random. They are therefore sometimes referred to as pseudo random numbers.

For example, the sequence obtained when $x_0 = a = b = 7$, $m = 10$, is

7, 6, 9, 0, 7, 6, 9, 0, ...

As this example shows, the sequence is not always "random" for all choices of x_0, a, b, and m; the way of choosing these values appropriately is the most important part of this method.

Because X_{n+1} is determined by X_n, so once some number in the sequence get repeated, the sequence will get into a cycle. And because there are only m possible different values for X_n, so the sequence will get into a cycle in at most m steps and the period is at most of length m.

It's very reasonable that we want the sequence to have long period so it might look random. So m is chosen to be very big, e.g. 2^{31}.

Our real examples will have large and safe values, for example

a=21751, x_0=3553, b=10653, and m=2147483648

Algorithm **To Generate Pseudorandom Numbers using Linear Congruential Generator**

```
Step 1: Set parameters values for multiplier a, Increment b ,modulus m and Initial seed value $x_0$
Step 2: Generate successive members of Linear congruential sequence using
        $x_{n+1} = (ax_n + b) \bmod m$ for $n \geq 0$ ,where n is number of Psuedo numbers to be generated.
Step 3: Repeat step 2 for n numbers.
```

Program 6 C Program to Generate Pseudo-Random Numbers using Linear Congruential Generator

```c
#include <stdio.h>
#include <math.h>

void generatePseudoRandomNumbers(int n)
{
    int a=109, b=853, m=40960, i;
    long long int x= 3553;
    printf("\n Pseudorandom Numbers are : ");
    for(i = 1; i <= n; i++)
    {
        x = (a*x + b) % m;
        printf("\t %lli \t",x);
    }
}
void main()
{
    int n;
    printf("\n Enter Number of Pseudorandom Numbers to be Generated :");
    scanf("%d",&n);
    generatePseudoRandomNumbers(n);
}
```

Output :

```
Enter Number of Pseudorandom Numbers to be Generated :5
Pseudorandom Numbers are :      19490     36303     25720     19053     29630
```

How to Generate Pseudorandom Numbers using Predefined Functions in C :

Let us understand how we can generate the random number in the C programming language using predefined or inbuilt functions. As we know, the random function is used to find the random number between any two defined numbers. In the C programming language, the random function has two inbuilt functions: **rand()** and **srand()** function. In order to use these functions, we must include the <stdlib.h> library in the program. These two functions are closely related to each other. Without the srand() function, the rand() function would always generate the same number each time the program is run. Let's understand these functions in the C language.

1. rand() Function:

In the C programming language, the rand() function is a library function that generates the random number in the range [0, RAND_MAX]. RAND_MAX is a constant which is platform dependent and equals the maximum value returned by rand function. the rand() function does not contain any seed number. Therefore, when we execute the same program again and again, it returns the same values.

Syntax

```
int rand(void);
```

The function does not need any parameters. The rand() function will return a pseudo-random number in a certain range (up to RAND_MAX).

Program 7 **C Program to Generate Pseudo-Random Numbers using rand() Function**

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    // This program will create same sequence of random numbers on every program run
    int n,i;
    printf("\n Enter Number of Pseudorandom Numbers to be Generated : ");
    scanf("%d",&n);
    printf("\n Pseudorandom Numbers are : ");
    for( i = 0; i<n; i++)
     printf(" %d ", rand());
}
```

Output 1:

```
Enter Number of Pseudorandom Numbers to be Generated : 5
Pseudorandom Numbers are :  41  18467  6334  26500  19169
```

Output 2:

```
Enter Number of Pseudorandom Numbers to be Generated : 5
Pseudorandom Numbers are :  41  18467  6334  26500  19169
```

Output 3:

```
Enter Number of Pseudorandom Numbers to be Generated : 5
Pseudorandom Numbers are :  41  18467  6334  26500  19169
```

2. srand() Function:

The srand() function is a C library function that determines the initial point to generate different series of pseudo-random numbers. A srand() function cannot be used without using a rand() function. The srand() function is required to set the value of the seed only once in a program to generate the different results of random integers before calling the rand() function. If srand() is not called, the rand() seed is set as 1 by default.

Syntax `void srand(unsigned int);`

The function needs an unsigned int as a mandatory value that will set the seed.

The pseudo-random number generator should only be seeded once, before any calls to rand(), and the start of the program. It should not be repeatedly seeded, or reseeded every time we wish to generate a new batch of pseudo-random numbers.

Standard practice is to use the result of a call to srand(time(0)) as the seed. However, time() returns a time_t value which vary every time and hence the pseudo-random number vary for every program call.

Program 8 **C Program to Generate Pseudo-Random Numbers using srand() and rand() functions**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void main()
{
        int n,i;
        printf("\n Enter Number of Pseudorandom Numbers to be Generated : ");
        scanf("%d",&n);
        srand(time(0));
        printf("\n Pseudorandom Numbers are : ");
        for( i = 0; i<n; i++)
          printf(" %d ", rand());
}
```

Output 1:

```
Enter Number of Pseudorandom Numbers to be Generated : 5
Pseudorandom Numbers are :  22622  28802  3179  18146  31579
```

Output 2:

```
Enter Number of Pseudorandom Numbers to be Generated : 5
Pseudorandom Numbers are :  22720  23574  14816  19149  13336
```

Output 3:

```
Enter Number of Pseudorandom Numbers to be Generated : 5
Pseudorandom Numbers are :  22825  7075  29414  2743  15723
```

13.8 Raising a Number to a Large Power

Problem

Given some integer x, compute the value of x^n where n is a positive integer considerably greater than n.

The power (or exponent) of an Integer is how many times number is a multiplied by itself. Power is represented with **a base number** and **an exponent.**

The **base number** is **what number is being multiplied.** The power or **exponent is how many** times the base number is being multiplied. Power written above and to the right of the base number.

Example

1. $2^4 = 2 \times 2 \times 2 \times 2 = 16$

   Here 2 is the base number and 4 is the power or exponent.

   The rule for multiplying two integers of same base are $(x^a)*(x^b) = x^{a+b}$

2. $x^3 * x^4 = x^{3+4} = x^7$.

   where $x^3 = x*x*x$ and $x^4 = x*x*x*x$

Although most programming languages have a built-in or predefined library functions like pow() that computes powers of a number, we can write a similar function without using predefined functions, and it can be very efficient. There are so many methods to find the powers of a number. Let us understand some of the methods.

1. Naive Iterative Method (Simple Method)

The simplest approach to find the value of x^n is to repetitively multiply x exactly n times and print the product. Consider the expression $p = x^n$ where p is the accumulating product p, integer x and n is the power.

The result is derived in a linear fashion multiplying x by n times. So Product is computed as

```
Product (p) = x * x * x * ... ... n times.
```

The approach is shown below:

```
p = 1;
for i = 1 to n do
    p = p * x
```

Program 9 **C Program to Find the Power of a Number using Naive Iterative Method**

```c
#include <stdio.h>
// Naive iterative solution to calculate "pow(x, n)"
long power(int x, unsigned n)
{
    // initialize result by 1
    long pow = 1;
    int i;
    // multiply 'x' exactly 'n' times
    for (i = 0; i < n; i++)
    {
        pow = pow * x;
    }
    return pow;
}
```

```
void main()
{
        int x,n;
        printf("\n Enter Base x : ");
        scanf("%d",&x);
        printf("\n Enter Power n : ");
        scanf("%d",&n);
        printf("pow(%d, %d) = %d", x, n, power(x, n));
}
```

Output :

```
Enter Base x : 3
Enter Power n : 9
pow(3, 9) = 19683
```

The time complexity of the above method is **O(n)** as it requires **n** multiplications. Is the above method efficient? Not really, as we require 'n' multiplications, and practically the value of 'n' can be very large. Although this works with small exponents, for larger exponents, it's quite a bit of work. Surely there must be a better way. In fact, there's a much more efficient method.

2. Exponentiation by Squaring

The efficient exponentiation algorithm is based on the concept of squaring. Let us understand the basics of this concept.

- The base case is when n = 0, and $x^0 = 1$
- When n is positive and even. We know that when we multiply powers of x, we add the exponents: $x^a \cdot x^b = x^{a+b}$ for any base x and any exponents a and b. Therefore, if n is positive and even, then

  $x^n = x^{n/2} \cdot x^{n/2}$.
- If n is positive and odd, then, $x^n = x^{n-1} \cdot x$

This method can compute x^n with only log(n) multiplications instead of n multiplications, which is a significant improvement, especially when *n* is large. This technique, known as "exponentiation by squaring," is widely used in algorithms where efficiency is crucial, such as in cryptography.

Example 1 **Computing x^6**

We know that computing x^6 takes 6 multiplications in naive method. Let us see how we can reduce the number of multiplications in exponentiation by squaring approach.

$P_1 = x^1 = x$ $\quad$ $P_2 = x^2 = x * x$

$P_3 = x^3 = x^2 * x$ $\quad$ $P_4 = x^4 = x^3 * x$

$P_5 = x^5 = x^4 * x$ $\quad$ $P_6 = x^6 = x^5 * x$

It is clear from the above steps that the power of x is increasing by one at each step. And the multiplication at each step is addition of powers of x. This method will be efficient if the power is a small number. But we know that $x^6 = x^3 * x^3$

The number of mutilations to generate $x^6 = x^3 * x^3$ is only one here. (Adding the powers 3 + 3 = 6). Further to reduce the number of steps, identify smallest pair of numbers that add up to 6.

The smallest pair of numbers that add up to 6 is 3 + 3.

So

$x^2 = x^1 * x^1$

$x^3 = x^2 * x^1$

$x^6 = x^3 * x^3$

In this approach, it requires only 3 multiplications rather than 6 multiplications and hence it is efficient compared to naive iterative method.

Example 2 **Computing 3^{13}**

We know that computing 3^{13} takes 13 multiplications in naive method. Let us see how we can reduce the number of multiplications in squaring approach.

1. The first multiplication gives us . $3^2 = 3 * 3 = 9$
2. With the second multiplication we compute, $3^4 = 3^2 * 3^2 = 9 * 9 = 81$
3. With the third multiplication we compute, $3^8 = 3^4 * 3^4 = 81 * 81 = 6561$
4. The fourth and fifth multiplication yield the desired result
   $3^{13} = 3 * 3^4 * 3^8 = 3 * 81 * 6561 = 1594323$

In this approach, it requires only 5 multiplications rather than 13 multiplications and hence it is efficient compared to naive iterative method.

From the above examples. We can derive at two rules for power evaluation process.

Every stage of power generation one of these rules are applicable:

1. If the power to be generated is odd, it must be generated from power one less than it. (Example: $x^{11} = x^{10} * x^1$)
2. If the power to be generated is even, it must be generated from power that is half its size. (Example: $x^6 = x^3 * x^3$)

Algorithm **To Raise a Number to a Large Power**

Let x be the integer to be raised by a power n.

```
Step 1: Start
Step 2: Declare x, n and product
Step 3: initialize value of product = 1
Step 4: Read x and n
Step 5: While n > 0
        (a) if `n` is odd, multiply the result by 'x'
            product = product * x
        (b) Divide n by 2
        (c) multiply x by itself
            x = x * x
Step 6: Print product
Step 7: End
```

Example 3 Computing 3^9

Here, x=3 and n=9. The initial value of product is 1 (product =1)

Iteration	while (n > 0)	if n is odd	n = n/2	x = x * x

1	while (9 > 0) is True	n is odd, so
		product = product * x
		product = 1 * 3
		product = 3
			n=n/2
			n=9/2
			n=4
				x= x * x
				x= 3 * 3
				x=9

2	while (4 > 0) is True	n is not odd	n=n/2
			n=4/2
			n=2
				x= x * x
				x= 9 * 9
				x=81

3	while (2 > 0) is True	n is not odd	n=n/2
			n=2/2
			n=1
				x= x * x
				x= 81 * 81
				x=6561

4	while (1 > 0) is True	n is odd , so
		product = product * x
		product = 3 * 6561
		product = 19683
			n=n/2
			n=1/2
			n=0
				x= x * x
				x= 6561 * 6561
				x=43046721

5	while (0 > 0) is False			

The final value of product is : 19683 = 3^9

Program 10 **C Program to Find the Power of a number using Exponentiation by Squaring Method**

```
#include<conio.h>
#include<stdio.h>

int power(int x,int n)
{
    int product = 1;
    if(n == 0) // if power is zero then return 1
      return 1;
    while(n > 0)
    {
      if(n % 2 == 1)
        product = product * x;
      n = n/2;
      x = x * x;
    }
    return product;
}
```

Output :

```
Enter Base x : 3
Enter Power n : 9
pow(3, 9) = 19683
```

```
void main()
{
    int x,n;
    printf("\n Enter Base x : ");
    scanf("%d",&x);
    printf("\n Enter Power n : ");
    scanf("%d",&n);
    printf("pow(%d, %d) = %d", x, n, power(x, n));
}
```

13.9 Review Questions

1. What is Factoring or Factorization?
2. Write the Applications of Factoring in Computer Science.
3. Write an Algorithm to Find the Square Root of a Number using Newton's Method.
4. Write a C Program to Find the Square Root of a Number using Newton's Method.
5. Write a C Program to Find the Square Root of a Number using Predefined C Function.
6. Write an Algorithm to Find the Smallest Divisor of an Integer.
7. Write a C Program to Find the Smallest Divisor of an Integer.
8. Write an Euclid's Algorithm to Find the GCD of Two Numbers a and b.
9. Write a C Program to Find Greatest Common Divisor of Two Integers a and b.
10. What is the Meaning of Prime Factorization?
11. Write an Algorithm to Compute the Prime Factors of an Integer.
12. Write a C Program to Compute the Prime Factors of an Integer.
13. What is Random Number? What are the Applications of Random Numbers?
14. What is Pseudorandom Numbers?
15. Explain Linear Congruential Generator Method to Generate Pseudorandom Numbers.
16. Write an Algorithm to Generate Pseudorandom Numbers using Linear Congruential Generator.
17. Write a C Program to Generate Pseudorandom Numbers using Linear Congruential Generator.
18. Explain rand() and srand() functions in C with an example.
19. Write a C Program to Generate Pseudo-Random Numbers using rand() and srand() functions.
20. Write a C Program to Find the Power of a Number using Naive Iterative Method.
21. Write an Algorithm to Find the Power of a Number using Exponentiation by Squaring Method.
22. Write C Program to Find the Power of a Number using Exponentiation by Squaring Method.