Problem Solving Approaches

Discipline Courses-I
Semester-I
Paper: Programming Fundamentals
Unit-I
Lesson: Problem Solving Approaches
Lesson Developer: Rakhi Saxena
College/Department: Acharya Narendra Dev College, University of Delhi

Table of Contents

1.1 Computer Problem Solving

The design phase of the program development cycle is the most important phase of program development especially in the case of industry level code. In this chapter you will learn that dividing a complex problem into smaller tasks during the design phase makes it easier to solve and leads to development of modular programs.

You will also learn some program design tools that aid in the design of a program – pseudocode, flow charts and hierarchy charts.

Given a problem, finding a solution that can be built into a program for a computer is a creative and challenging task. Even though there is no general formula or recipe for writing a program, there are some strategies that can be used for coming up with a solution to a given problem. In this chapter, you will learn some general problem solving strategies for solving problems using computers.

Learning these skills will help you design top down modular programs using program design tools.

Learning Objectives

After reading this chapter you should be able to:

1. Use various techniques and tools to design programs.

1.1. Use the principles of top down modular program design.
1.2. Apply stepwise refinement to split a problem into simpler sub problems.
1.3. Describe the benefits of modular programming.
1.4. Use pseudocode to design a program.
1.5. Use hierarchy charts to depict a modular design.
1.6. Use flowchart symbols to design a program segment.
1.7. Describe general problem solving strategies.

Modular Programming

Imagine the complex task of building a house. Investing in first getting an architect to design the house and create a blueprint will obviously result in a good quality house that suits users' needs.

Similarly, a good detailed design is just like a blueprint that makes it easier to write good and correct program code. Spending effort and time on building a good design rather than rushing into coding cannot be more over-emphasized.

Also, the task of constructing the house cannot be done all together. The design has to be broken down into smaller manageable tasks such as – laying the foundation, building the walls, laying the roof, putting in doors and windows, laying electrical cables, and so on.

This division of tasks tells us that certain tasks need to be done before others – such as walls cannot be put up before the foundation has been laid. Also, we come to know that certain tasks can be performed at the same time – such as windows can be put up at the same time as the electrical wiring is being done.

It's the same with a program. It is easier to write large programs by breaking up the complex problem into smaller and simpler manageable units. A good way to begin designing a large program is to identify the major tasks that the program must accomplish. Each of these tasks then becomes a **module** in the program. Identifying the tasks and various subtasks involved in the program design is called **modular programming**.

A modular program, in contrast to one single chunk of source code is composed of smaller, separated chunks of code (modules) that are well isolated but have well defined interfaces for communication with other modules. These modules can be developed by separate teams with their own life cycles and their own schedules. The results can later be assembled together.

In C++ modules are implemented as **functions**. You will learn to write functions in the next unit.

Value addition: Did You Know?
Heading text – Guidelines for Module Division
Body text: The number and type of modules in a program are a matter of programmer's style. However, some guidelines for designing modules are: • A module should perform a single logical task/ function. • It should be self contained and independent of other modules. • It should be relatively short. • There should be a well defined interface for invocation by other modules.
Source: Self made

Stepwise Refinement

Once the fundamental tasks that are needed to solve a problem are identified, the higher level tasks can be further split into sub tasks if possible. The subtasks become **sub-modules** of the original parent task. The subdivision of tasks can continue as long as necessary or until the subtask cannot be further refined.

This process of breaking down a problem into simpler and smaller sub problems repeatedly until no further break down is possible is called **stepwise refinement** or **top-down design**. A top-down model is often specified with the assistance of "black boxes", these make it easier to understand.

To illustrate the modular programming technique, let us consider an example.

A bank needs to develop a program, which when given an amount/balance in a person's fixed deposit account, the rate of interest and the time period for the deposit, will compute the simple interest that the amount in the account has earned. This interest is then added to the account's balance to obtain the new balance amount.

The data variables needed to solve the problem are

- *amount* - the amount/balance in the account
- *roi* – the rate of interest
- *period* – the period for which the deposit has been held
- *interest* – the interest earned on the deposit amount

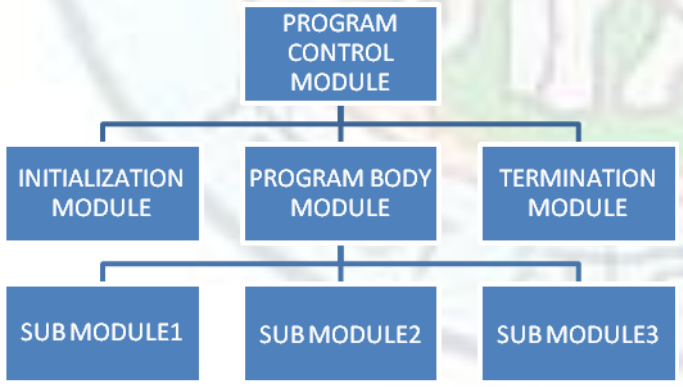
The fundamental tasks that we need to perform to solve the problem are as follows:

1. **Input Data:** Input variables – *amount, roi* and the *period.*
2. **Perform Calculations**:

   Compute the *interest*.

   Increment the *amount*.
3. **Display Result**: Display the *interest* earned on the deposit and the new *amount*.

The fundamental tasks can be further divided into smaller sub modules if possible.

Value addition: Did You Know?
Heading text – Hierarchy of Modules
Body text: Modular programs are generally hierarchical. The top-level Control Module makes all major decisions regarding flow of control through the program. A module can only be invoked (called) from a higher level module. On exiting a module control is returned to the statement immediately following the invocation.   
Source: Self made

Benefits of Modular Programming

Following are the reasons why you should follow a modular approach to program design:

- **Improved Readability** – The program is easier to read and understand. This reduces the time needed to locate errors in the program code.
- **Increased Productivity** – It is easier to design, code and test large programs one module at a time rather than all at once. This increases the productivity of the programmers.
- **Higher Maintainability** – It is easier to identify sections in code (in a specific module) where faults occur and make necessary modifications to it without affecting the rest of the modules.
- **Reduced Development Time** – Different program modules can be designed and coded by different programmers thus reducing the time to develop large complex software.
- **Smaller Programs** – The same module can be used/ invoked more than once in different parts of the code of the same program thus reducing size of the code.
- **Reusability** – Modules performing common tasks (such as searching or sorting) can be used in more than one program thus reducing the time and effort in program development.

Value addition: Did You Know?
Heading text – Structured Programming
Body text: Modular Programming is a part of structured programming. Structured Programming is the method used to design and code programs in a systematic and organized way. Structured programming concepts include – following the program development cycle, using comments and indentation, and designing programs in a top down modular method.
Source: Self made

Program Design Tools

There are a number of tools/ devices that aid in program design and are useful in the process of identifying exactly what steps to take to solve a programming problem. Of the program design tools available, three popular tools are - hierarchy charts, pseudocode and flowcharts.

Hierarchy Charts

In a complex program there will be numerous program modules and sub modules. A **hierarchy chart** helps us keep track of the relationships between modules in a visual way. Just like an organization chart determines who is responsible to whom in an organization, a hierarchy chart describes the relationships among modules in a program. A hierarchy chart is also called a **structure chart**.

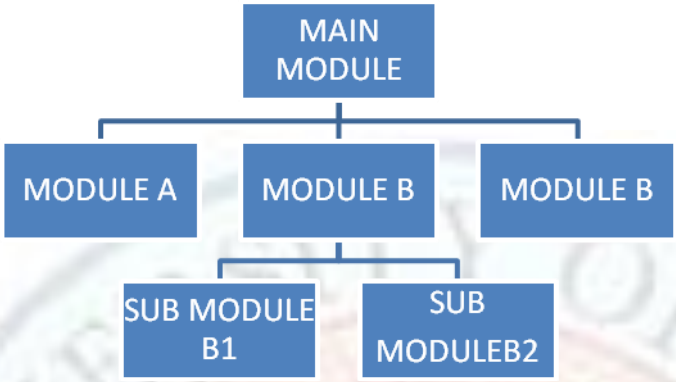
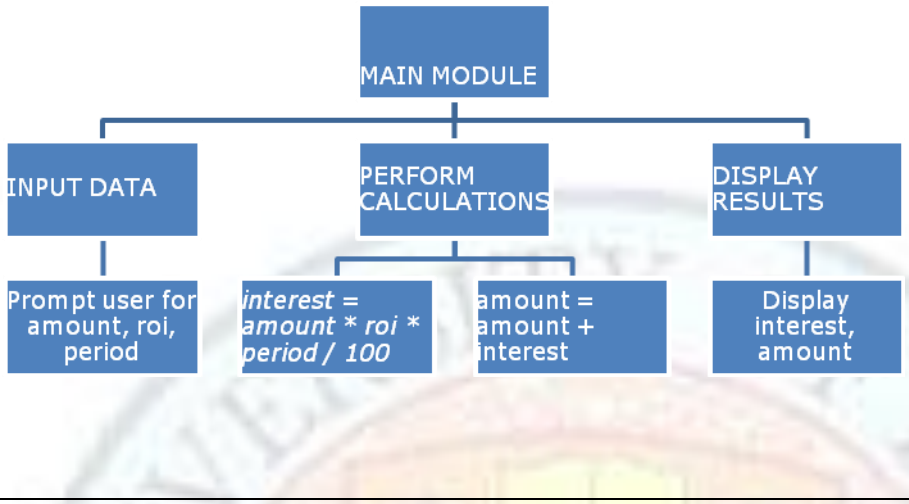


Figure 1.1 – A Hierarchy Chart

The main module where program execution begins sits at the top of a hierarchy chart. Think of the main module as the CEO of the organization. Below the main module are the highest level subordinates (labeled A, B, and C) that perform the fundamental tasks. The sub modules B1 and B2 are subordinates of module B. A line connecting a higher module to a lower one indicates that the former is the parent that calls the latter into action. The chart is read from top to bottom and from left to right.

The main benefit of hierarchy charts is in the initial planning of a program. The major tasks of a program are broken down so that we can see what must be done in general. From this point, we can then refine each module into more detailed plans using pseudocode or flowcharts.

Value addition: Example
Heading text – Hierarchy Chart for Interest Calculation Problem
The following is the hierarchy chart for the interest calculation problem:



Source: Self made

Pseudocode

Once the modules in a software system are identified, we must provide specific instructions to accomplish the task of the module. We can supply this detail using pseudocode. Pseudocode uses short English like phrases to describe the outline of a program. It is not actual code from any programming language but resembles the actual code. Pseudocode allows the programmer to focus on the steps required to solve a problem rather than on how to use the computer language.

Pseudocode has several advantages. It is compact and probably will not extend for many pages. Also, the plan looks like the code to be written and so is preferred by many programmers. One advantage of pseudocode over flowcharts is that pseudocode has no provision for unconditional branching and thus forces the programmer to write structured programs.

Value addition: Example

Heading text – Pseudocode for Interest Calculation Problem

The following is the pseudocode for the interest calculation problem:

Main Module
Declare amount, roi, period as Float
Write "Interest Calculation Program"
Call Input Data Module
Call Perform Calculations Module
Call Display Results Module
End Module

Input Data Module
Write "Enter the amount, rate of interest and the term for the deposit: "

```
        Prompt for amount, roi, period
End Module

Perform Calculations Module
        Set  interest = amount * roi * period / 100
        Set amount = amount + interest
End Module

Display Results Module
        Write  "Interest Earned: "
        Write interest
        Write "Updated Amount: "
        Write amount
End Module
```

Source: Self made

Flow Charts

Another common program design tool is the flowchart. A flowchart is a diagram that uses special symbols to display pictorially the flow of execution within a program or a program module. Flowcharts provide an easy, clear way to visualize how program execution will proceed.

A flowchart is drawn using certain special symbols such as rectangles, diamonds, ovals, and small circles. These symbols are connected by arrows called flow lines.
Flowcharts can clearly show how control structures (the sequence, selection and repetition structures) operate.

The following table shows the basic symbols that can be used in a flowchart:

Symbol	Name	Description
	Terminator	Represents the start or end of a program or a module
	Process	Represents any kind of processing functions, for example, a computation
	Input / Output	Represents an input or an output operation

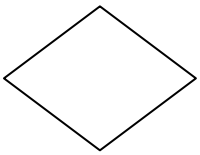
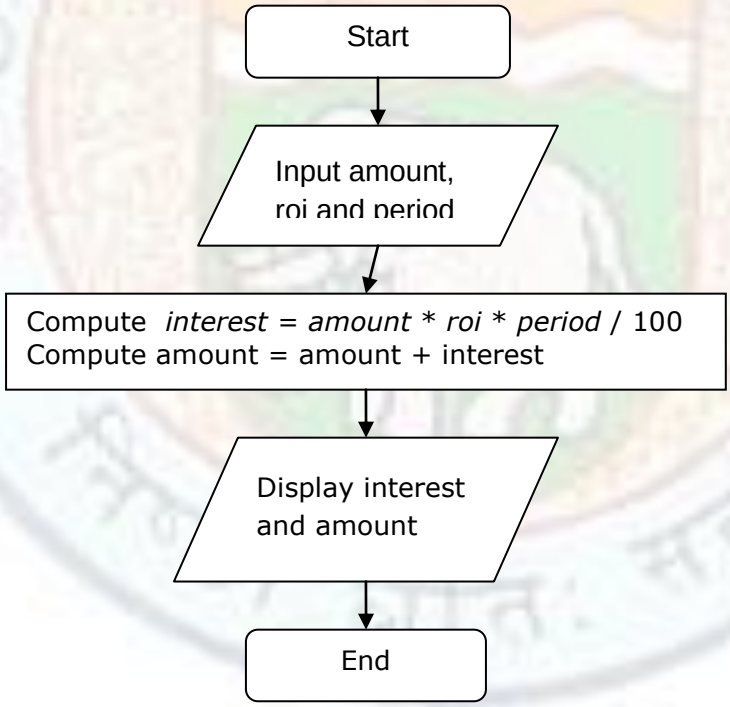
	Decisions	Represents a program branch point
	Connector	Represents an entry to or an exit from a program segment
	Flow Line	Represents the order in which actions are to be performed

Figure 1.2 Basic Flow Chart Symbols

Value addition: Example

Heading text – Flow Chart for Interest Calculation Problem

The following is the flowchart for the interest calculation problem:



Source: Self made

The main advantage of using a flowchart to design a program is that it provides a pictorial representation, which makes the logic of the program easier to follow. We can clearly see every step and how each is connected to the next.

The major disadvantage with flowcharts is that when a program is very large, the flowcharts may continue for many pages, making them difficult to follow and modify.

Flowcharts are time consuming to write and difficult to update. For this reason, professional programmers are more likely to favor pseudocode and hierarchy charts. Because flowcharts so clearly illustrate the logical flow of programming techniques, however, they are a valuable tool in the education of programmers.

Problem Solving Strategies

Even though you have learned how to design a program and have also seen some design tools, you are probably wondering how to develop the logic for the program. Most students when given a problem they have never seen before get stuck on where and how to begin writing a program for the same - that is, how to solve the given problem?

A great deal of the art of problem solving is to understand the kind of problem that is posed and the kind of solution that is demanded. There is no general problem solving strategy. Different strategies work for different people.

However, there are some general problem-solving techniques that you can use to derive a solution to a problem. These include:

- **Abstraction -** solve the problem in a specific example of the system to reach the solution to the general problem.
- **Analogy -** modify a solution that solved a similar problem.
- **Hypothesis Testing** - assume a possible solution to the problem and work backwards from the solution to the initial conditions.
- **Reduction** – reduce a problem into a smaller problem iteratively to arrive at the solution.
- **Divide and Conquer** - break down a large, complex problem into smaller, solvable problems and combine the results.

You will learn to apply these techniques in subsequent chapters in this unit.

Value addition: Did you Know?
Heading text – Problem Solving Tips
Successful computer problem solving requires the right state of mind. Here are some points to remember when faced with difficult problems: • Don't panic — in most cases you can turn off your computer and get back to the problem later. Of course, many problems occur when you do have a deadline looming (or passed). If you can wait, even for a few minutes, to take a few deep breaths, do so.

- The problem may not be as big as it seems — After calmly looking over the situation you may be back up and running in a few minutes.
- Don't jump to conclusions before you understand the problem and look at every angle.
- Think through all possibilities before you begin.
- Work on the problem when you have time.
- Know when to take a break — if you start feeling worn out, step away from the computer. Get some rest, relax, refuel then jump back in with a fresh outlook.
- Don't be shy — Ask for help. If you exhaust all your possibilities or get in over your head, ask a teacher or a friend who can help.

The right frame of mind is an important ingredient in successful computer problem solving. Keep a cool head and you're more likely to solve the problem.

Source: Self made

1.2 Fundamental Algorithms

The process by which new problems are solved begins with an understanding of all aspects of the problem and ends when a solution has been found.

In the previous unit, you learned that there are various problem solving strategies to solve a given problem. In this unit you will learn how to apply some problem solving approaches given unfamiliar problems and design algorithms to solve them.

One problem solving strategy for problems that need iteration is to first decide the initialization steps, then decide the iterative steps and lastly decide the termination steps. We will apply iteration to generate the Fibonacci series and to sum the digits of a number.

Another strategy is analogy - a solution from a previously solved problem can be modified to solve a similar problem. We will apply this strategy to reverse the digits of a number.

1.2.1 Learning Objectives

After reading this chapter you should be able to:

2. Develop algorithms to solve simple problems.
Define what an algorithm is and describe its properties.
Design an algorithm to generate the Fibonacci series.
Design an algorithm to sum the digits of a number.
Design an algorithm to reverse the digits of a number.

1.2.2 What is an Algorithm?

Before we start learning about different problem solving approaches to designing algorithms, we should first understand what an algorithm really is. If you were given a new problem and asked to write a program to solve it, you would need to develop a precise step-by-step sequence of instructions that usually begins with an input value and produces an output value in a finite number of steps. This sequence of steps specifying how to solve some problem is called an **algorithm**.

In other words, an algorithm is a well-defined step-by-step method used to achieve a desired result.

An algorithm should have certain properties

- It should terminate/halt in a finite number of steps
- The instructions should be precise and computable.
- It should produce a result.

An algorithm is independent of the programming language used. Once you have designed the algorithm it should be possible to code the corresponding program in any programming language.

In the previous unit, you have already seen some algorithms – to exchange the values of two variables, to perform summation of a series of numbers, to compute the factorial of a number and so on.

In this unit we will develop iterative algorithms that generally have three parts:

- Initialization
- Iteration / Looping
- Termination

One problem solving strategy is thus to first decide the initialization steps, then decide the iterative steps and lastly decide the termination steps.

1.2.3 Generating the Fibonacci Sequence

Our first example of an algorithm will be one that computes the Fibonacci series. The Fibonacci numbers are the series of numbers 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

Notice that except the first two numbers in the series, each number in the series is the sum of the previous terms. Thus, the series is defined as F_0, F_1, F_2, F_3, F_4, … such that

$F_0 = 0$
$F_1 = 1$
$F_2 = F_0 + F_1 = 0 + 1 = 1$
$F_3 = F_1 + F_2 = 1 + 1 = 2$
$F_4 = F_2 + F_3 = 1 + 2 = 3$
$F_5 = F_3 + F_4 = 2 + 3 = 5$

Let's say, the given problem is to design an algorithm to generate the Fibonacci series up to the n^{th} term. Let us define

fib3 – new term
fib2 – preceding term
fib1 – term before the preceding term

From the problem definition we know that to generate a new term in the series, we can use the formula

New term = preceding term + term before the preceding term (1)

Initialization: We can initialize the first two terms of the Fibonacci series as:

```
fib1 = 0          //first term
fib2 = 1          //second term
```

Iteration: We can use formula (1) for generating successive terms of the series.

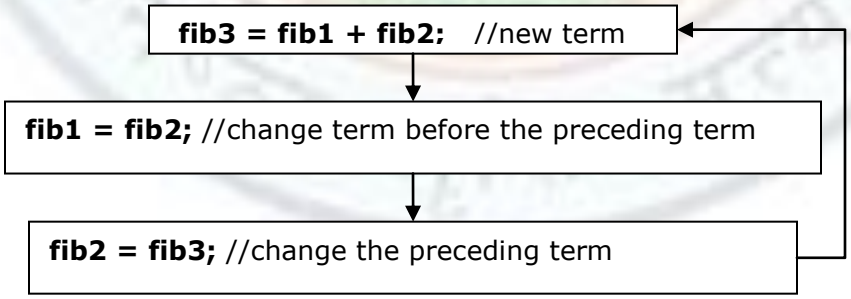
```
fib3 = fib1 + fib2;          //third term
```

However, to generate the fourth term, we need to make some adjustments. We need to sum the second and the third term (and not the first and second terms). This means we need to change the two values – fib2 (preceding term) and fib1 (term before the preceding term). What should these values be?

First, the term before the preceding term (fib1) should be replaced by the second term – this value is presently in fib2 with fib1 = fib2

Then, the preceding term (fib2) should be replaced by the third term – this value is presently in fib3 with fib2 = fib3

Thus, to generate the new term and then to change the values, the assignments needed are:



To successively generate the Fibonacci series up to the n^{th} we can put the three assignment statements in a loop.

Termination: When should we stop generating the series? We stop when we have generated the n^{th} term. This means we need to keep track of the number of terms

generated. This as you have learned earlier can be done using a counter variable initialized to 1. The counter is incremented as soon as a new term is generated. The loop is repeated until the counter reaches n.

Our algorithm to generate the Fibonacci series is thus:

1. Prompt the user for the number of terms to be generated.
2. Initialize fib 1 = 0, fib2 = 1, counter = 1
3. If counter < number of terms to be generated
4. fib3 = fib1 + fib2
5. fib1 = fib2
6. fib2 = fib3
7. counter = counter + 1
8. Go to step 3

Value addition: Source Code

Heading text – The Fibonacci Program

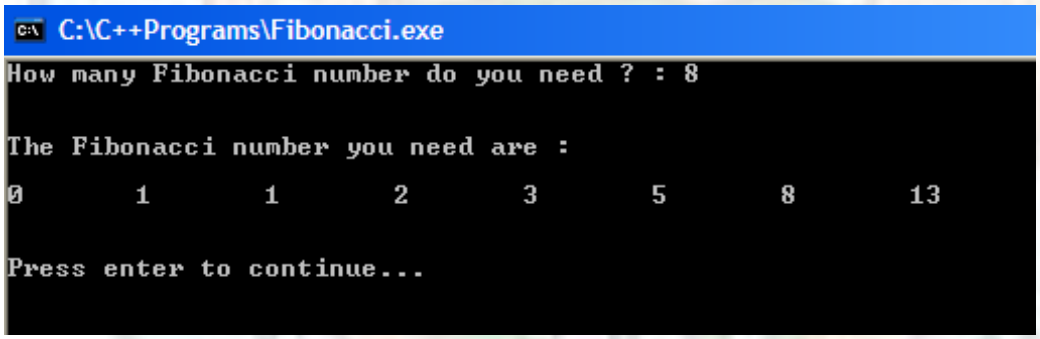
```
/* This program generates the Fibonacci Series up to the nth term*/

#include <iostream>
using namespace std;
#include <conio.h>

int main()
{
   int fib1 = 0;
   int fib2 = 1;   //define first two terms of the Fibonacci series

   int fib3;  //variable to store the next term of the Fibonacci series

    int numbers = 2;  //Variable to store how many numbers to be printed

    cout << "How many Fibonacci number do you need ? : " ;

    cin >> numbers;

    cout << "\n\nThe Fibonacci number you need are : \n\n" ;
    cout << fib1;   //Print the first term
    if (numbers == 1) return 0; //If number of terms to be printed is 1 work is done

    cout << "\t" << fib2;   //Print the second term
    if (numbers == 2) return 0; //If number of terms to be printed is 1 work is done
```

```
    //loop to calculate the new element of the series and printing the same.
     for (int i = 3; i <= numbers; i++)
     {
        fib3 = fib1 + fib2;
        cout << "\t" << fib3;
        fib1 = fib2;
        fib2 = fib3;
     }

    cout << endl;
    cout << "\n\nPress enter to continue...";
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – The Fibonacci Program



Source: Self made

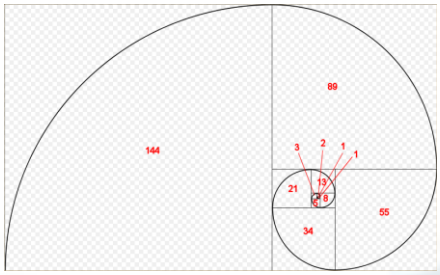
Value addition: Did you Know?

Heading text – The Fibonacci Series

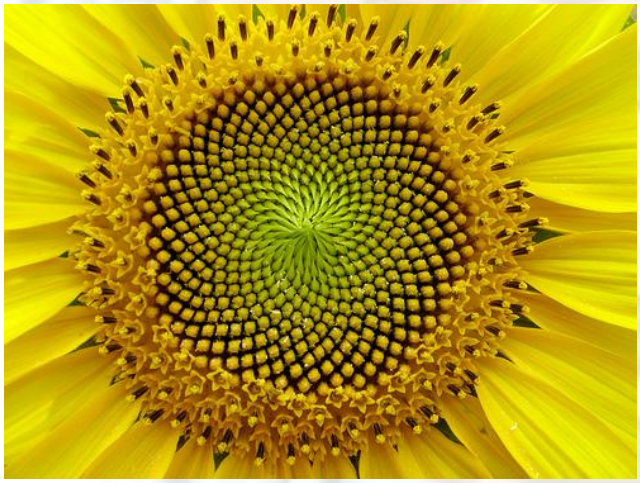
Body text:

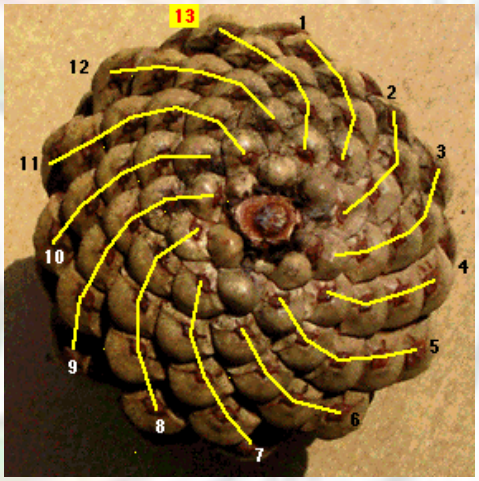
The 12th monk Leonardo Fibonacci (1175) is commonly cited as having discovered the Fibonacci Series that appears throughout nature.

A Fibonacci Spiral is created by drawing arcs connecting the opposite corners of squares, whose relative sizes follow the Fibonacci Sequence. Many shells follow the shape of the Fibonacci Spiral.



The individual florets of the sunflower grow in two spirals extending out from the center in opposite directions. The first spiral has 21 arms, while the other has 34. These are Fibonacci numbers.
Similarly, pinecones have 5 and 8 arms, or 8 and 13 arms depending on their size. This arrangement has been identified as the most efficient way of filling the space on the pinecone with seeds.





Most daisies have 21, 34, 55, or 89 petals - all Fibonacci numbers.

Source: http://hynesva.com/blogs/character_and_excellence

1.2.4 Summing the Digits of a Number

The next algorithm that we will design is to sum the digits of a number taken as input. As an example,

If the input number is: 45632

The output should be: 20

From the statement of the problem, it is obvious that we somehow need to successively extract each digit from the input number and then accumulate it in a sum. We can either start from the most significant digit or the least significant digit (that is either from the right or from the left of the number). However, since we do not how many digits there are in the number before taking user input, we cannot start from the left (most significant position) – we have to start from the right (least significant position).

Initialization: Prompt the user for a number. Also initialize the sum to 0.

Iteration: To extract the units place in a number, we can divide the number by 10. The integral division will give a remainder – the digit in the units place!

digit in units place = Number % 10

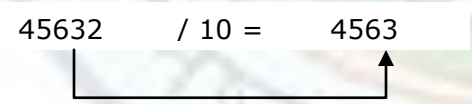
For example, 45632 % 10 will give us the digit in the units place, 2.

Now we can add this digit to the sum:

sum = sum + 2

Since we have already extracted the original digit that was in the units place we do not need it any longer. Now to extract the digit in the tens place, we can move the tens place digit to the units place. To do this, we can divide the original number by 10.

Number = Number / 10



This newly extracted digit can again be added to the sum in the iterative step.

Termination: We stop when we have extracted all the digits, that is, when the original number after repeated division becomes zero.

Our algorithm to sum the digits of a number is thus:

1. Prompt the user for a number.
2. Initialize sum to 0.
3. If the number > 0 then

4. digit = number % 10
5. sum = sum + digit
6. number = number / 10
7. Go back to step 2.

Value addition: Source Code

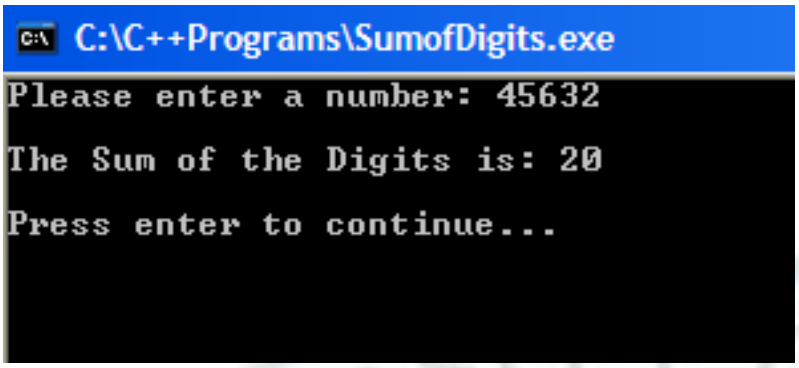
Heading text – The SumofDigits Program

```
/* This program sums the digits of an input number*/

#include <iostream>
using namespace std;
#include <conio.h>

int main()
{
   int number;  // variable to store the input number
   int sum = 0; //variable to accumulate
   int digit;   // variable to successively hold the digit
                // extracted from the units place

    cout << "Please enter a number: " ;
    cin >> number;

    while (number > 0)
    {
         digit = number % 10;   //extract digit in units place
         sum = sum + digit; // insert digit into the reversed number
         number = number / 10;  // move the next successive digit to the units place
    }

   cout << endl;
   cout << "The Sum of the Digits is: " << sum;
   cout << endl << endl<< "Press enter to continue...";
   getchar();
   return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – The SumofDigits Program



Source: Self made

1.2.5 Reversing the Digits of a Number

In this problem the problem solving strategy that we can us is **analogy**. In this strategy, we take the solution to a previously solved similar problem and modify it to solve a new problem. Since we have already developed a similar problem to sum the digits of an integer, we can use our learning there to develop an algorithm for reversing the digits of a number taken as input.

As an example,

If the input number is: 45632

The output should be: 23654

From the statement of the problem, it is obvious that we somehow need to successively extract each digit from the input number and then make them part of the output number. From the sum of digits algorithm, we already know how to extract the digits of the given number one by one. The only difference is how to place these digits in the reversed number.

Initialization: First initialize the reversed number to 0.

Computation: Then extract the digit in the units place from the input number by performing number % 10.

Now we can add this digit to the reversed number:

Reversed number = 2

Then we extract the digit in the tens place, by first dividing the number by 10 and then performing the modulus 10 operation.

But what about making this digit part of the reversed digit? If we just add it to the reversed number, we would be summing the digits. We don't want that. We should first vacate the place in the units place in the reversed number before adding this new extracted digit.

Notice in our example, the reversed number has 2 in the units place. We can move this number to the next higher place, in this case, the tens place (by multiplying it by 10) and then add the newly extracted digit.

Reversed number = 2 * 10 + 3 = 23

Reversed number = reversed number * 10 + digit of number in units place

This step is performed repeatedly to extract all the digits in the input number.

Termination: We stop when we have extracted all the digits, that is, when the original number after repeated division becomes zero.

Our **algorithm** to reverse the digits of a number is thus:

1. Prompt the user for a number.
2. If the number > 0 then
3. digit = number % 10
4. reversed digit = reversed digit * 10 + digit
5. number = number / 10
6. Go back to step 2.

Value addition: Source Code

Heading text The ReversedDigits Program

Body text:

```
/* This program reverses the digits of an input number*/

#include <iostream>
using namespace std;

int main()
{
   int number;     // variable to store the input number
   int reversedNumber = 0; //variable to compute the reversed number
   int digit;  // variable to successively hold the extracted digit from the units place

    cout << "Please enter a number: " ;
    cin >> number;

    while (number > 0)
    {
        digit = number % 10;   //extract digit in units place
        reversedNumber = reversedNumber * 10 + digit; // insert digit into the
                                                      //reversed number
```

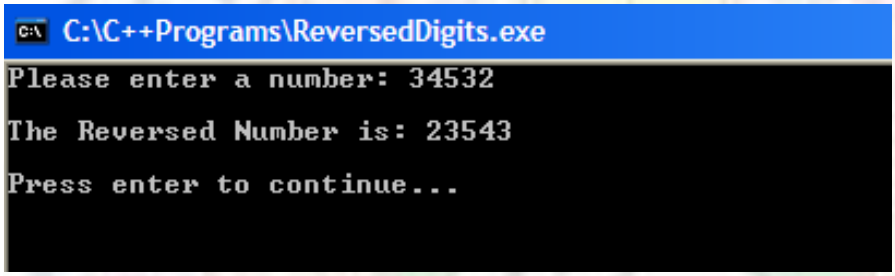
```
        number = number / 10;  // move the next successive digit to the units place
    }

    cout << endl;
    cout << "The Reversed Number is: " << reversedNumber;
    cout << endl << endl<< "Press enter to continue...";
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – The ReversedDigits Program

Body text:



Source: Self made

1.3 More Fundamental Algorithms

In this chapter you will learn to develop some more fundamental algorithms for base conversion, character to number conversion and for generating pseudorandom numbers using problem solving techniques such as abstraction and analogy.

Learning these problem solving strategies will empower you with skills to write programs for new problems that you are unfamiliar with and for ones that you have never seen before.

1.3.1 Learning Objectives

After reading this chapter you should be able to:

3. Develop algorithms to solve simple problems.
Design an algorithm to convert decimal number to octal/binary.
Design an algorithm for sine series computation.
Design an algorithm to generate pseudo random numbers.

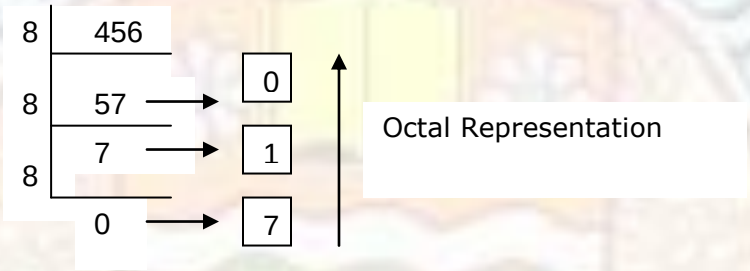
1.3.2 Base Conversion

We will next devise an algorithm to convert bases. Given a number in base X we would like to transform this number into a new one that uses Y base.

The specific problem we consider is – given a number in decimal convert it to a corresponding number in octal.

This time the problem solving strategy that we use is **abstraction**. We will first try to work out the algorithm by solving the problem for a specific example and then convert it to a general solution.

The algorithm that we will use is the method of continuous dividing. Consider converting the decimal number 456 into its equivalent octal number. We continuously divide the given number by the required base (8) keeping track of the remainder after each successive division.



The conversion is given by the remainders in reverse order of generation:

$(456)_{10} = (710)_8$

To determine the remainder, we use the modulus operation. Each successive remainder can be added to the converted number after first multiplying it with its correct place value. The place value for first remainder is 1, for the next 10, for the next 100, and so on.

For example, let us look at the iterations for conversion of $(456)_{10}$ to octal.

Iteration Number	**Decimal Number**	**Remainder**	**Place Value for remainder(i)**	**Octal Number**
Initialization	456	--	--	0
1	456	456 % 8 = **0**	**1**	0 + (**0** * **1**) = 0
2	456/8 = 57	57 % 8 = **1**	1*10 = **10**	0 + (**1*** **10**) = 10
3	57/8 = 7	7 % 8 = **7**	10*10= **100**	10 +(**7*** **100**)= 710
4	7/ 8 = 0	--As Number == 0 ; Iteration stops---		

For computing the place value (i) for the remainder, we can initialize i to 1 and in successive iterations multiply i by 10 each time.

Our algorithm for decimal to octal/binary conversion is thus:

1. Prompt user for base (either 8 or 2) and the decimalNumber.
2. Initalize convertedNumber = 0, i = 1.
3. If (decimalNumber > 0)
4. rem = decimalNumber % base
5. convertedNumber = convertedNumber + (rem *i)
6. i = i * 10;
7. decimalNumber = decimalNumber / base
8. Go back to step 3.

Value addition: Source Code

Heading text – The BaseConversion Program

Body text:

```
/* This program converts a decimal number to octal/binary */

#include <iostream>
using namespace std;

int main()
{
   int decimalNumber; // variable to store the input number
   int base; // variable for base for conversion
   long convertedNumber = 0; //variable to compute the reversed number
   int rem;  // variable to successively hold the remainder
   int i=1;  // variable for place value of remainder

   cout << "Decimal to Octal/ Binary Conversion"<<endl<<endl;
   do {
     cout << "Enter base for conversion: (8-Octal 2-Binary): ";
     cin >> base;
   }
   while ( base != 8 && base !=2);

   cout << endl<<"Enter a decimal number: ";
   cin >> decimalNumber;

   while (decimalNumber > 0)
   {
        rem = decimalNumber % base;    //extract remainder after division by base
        convertedNumber = convertedNumber + (rem *i);
                  // insert remainder into correct place value in the converted number
        i = i * 10;
        decimalNumber = decimalNumber / base;
```

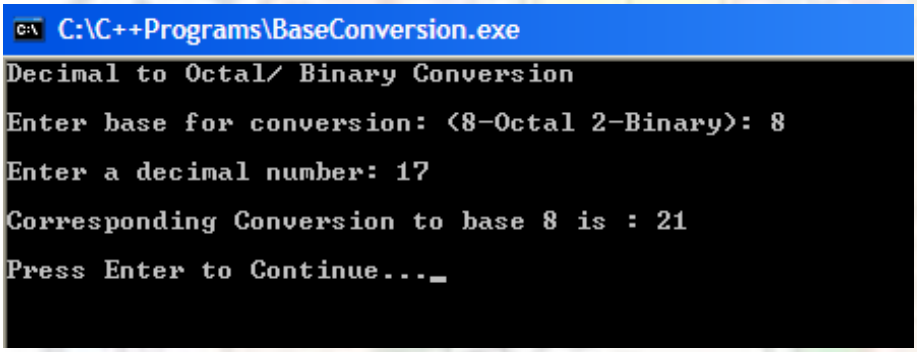
```
    }
    cout << endl;
    cout << "Corresponding Conversion to base " << base;
    cout << " is : "<<convertedNumber;

    cout << endl<<endl<<"Press Enter to Continue...";
    getchar();
    return 0;
}
```

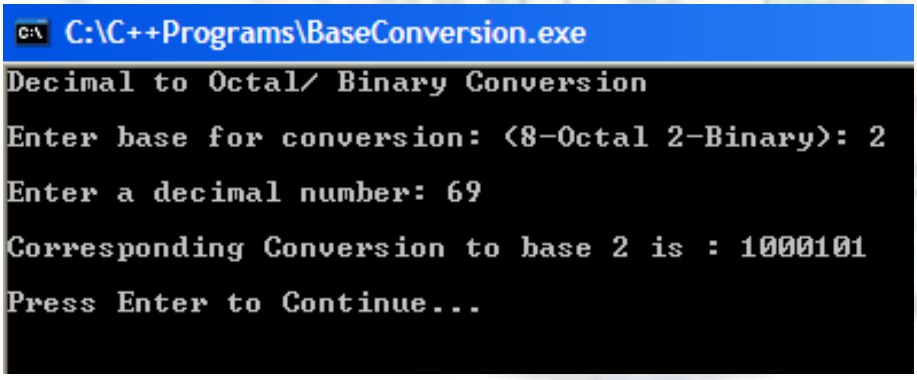
Source: Self made

Value addition: Program in Execution

Heading text – The BaseConversion Program

Body text:





Source: Self made

1.3.3 Sine Series Computation

Problem Solving Approaches

We will next devise an algorithm to compute the sine series given by the following infinite series:

Sin(x) =

The specific problem we consider is – given an angle x in radians, approximate the value of the sine of the angle using the sine series.

This time the problem solving strategy that we use is **analogy**. We will use some of the techniques we have seen earlier in summation of a series and in factorial computation.

Notice that the powers in the numerators and the factorials in the denominator follow the sequence:

1 3 5 7 9 11 …

This sequence can be easily generated by initializing a counter to 1 and incrementing it by 2 in each iteration.

Also, the general term of the series is given by the formula

i^{th} Term	Computation Required

The table illustrates that to compute the general term we need not compute i! and x^i each time. The computation of the $(i-1)^{th}$ term of the previous iteration holds the partial result for the i^{th} computation.

Thus

This means at every iteration we need to multiply the previous numerator by x^2 and the previous denominator by (i-1)* (i).

Current term = Previous term * (x*x) / ((i-1)* (i)).

For the summation the terms have to be added and subtracted alternatively. This can be achieved by multiplying term with a sign variable (initialized to 1).

In every iteration, the assignment

sign = -sign

causes sign to alternate between +1 and -1 for alternate terms. Multiplying term by sign will make term positive and negative alternatively.

Initialization: The following initializations are required for sine computation:

```
term = x;  // the variable term is used to compute a term of the series

sine = x; // the variable sine will hold partial sums of sine series computation

i = 1; // the variable i will keep track of iterations

sign = 1; // will alternates between +1 and -1 for alternate terms
```

Iteration:

```
i = i + 2;

sign = -sign;

term = term * (x*x)/ (i * (i-1));

sine = sine + (sign * term);
```

Termination: When should we stop generating the series? Since the value of x is between -1 and +1, raising it to large powers will cause the value to become very very small and these terms will progressively become less significant for adding to the series. We can therefore decide an acceptable error level, say - if the term becomes less than 10^{-6} we can stop the computation.

Algorithm:

Our algorithm to compute the Sine series is thus:

1. Prompt the user for the angle in degrees
2. Convert the angle in degrees to radian using the formula x = PI * (angle/180.0)
3. Initialize

   term = x

   sine = x

   i = 1

sign = 1

error = 1.0e-6

4. If (term > error)
5. i = i + 2
6. sign = -sign
7. term = term * (x*x)/ (i * (i-1))
8. sine = sine + (sign * term)
9. Go to step 4

Value addition: Source Code

Heading text – The SineSeries Program

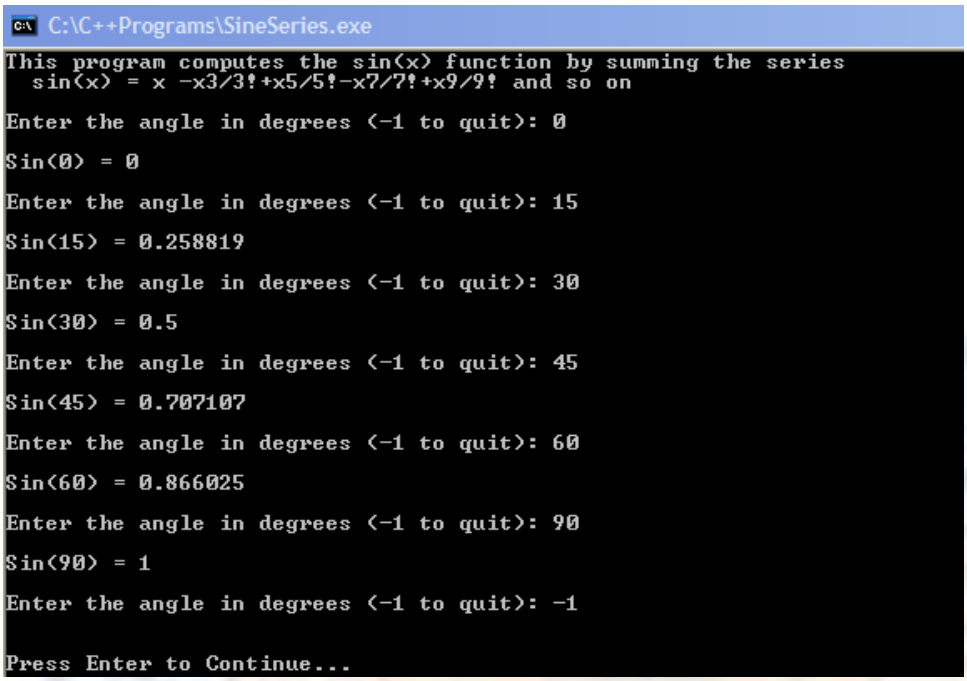
Body text:

```
/*
 This program computes the sin(x) function by summing the series
 sin(x) = x -x3/3!+x5/5!-x7/7!+x9/9!...
 The angle x is in radians. The user input is taken in degrees
 and converted to radians by multiplying angle by pi/ 180
*/

#include <iostream>
using namespace std;

#define PI 3.141592653589793

int main()
{
   int i; // to keep track of iterations
   int angle; //user input for angle
   float x;  // to hold value of angle converted to radians
   float term;  // to successively compute the terms of the series
   float sine;  // to hold partial sums of sine series computation
   float error = 1.0e-6;  // to decide when to terminate computation
   int sign; // alternates between +1 and -1 for alternate terms

   cout << "This program computes the sin(x) function by summing the series" << endl;
   cout << "  sin(x) = x -x3/3!+x5/5!-x7/7!+x9/9! and so on "<<endl<<endl;

   cout << "Enter the angle in degrees (-1 to quit): ";
   cin >> angle;

   while (angle != -1)
   {
      x = PI * (angle/180.0);  //convert angle to radians
```

```
        term = x;
        sine = x;
        i = 1;
        sign = 1;

        while (term > error)
        {
            i = i + 2;
            sign = -sign;
            term = term * (x*x)/ (i * (i-1));
            sine = sine + (sign * term);
        }
        cout << endl;
        cout << "Sin(" << angle << " ) = " << sine<<endl<<endl;

        cout << "Enter the angle in degrees (-1 to quit): ";
        cin >> angle;
    }
    cout << endl<<endl<<"Press Enter to Continue...";
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution
Heading text – The SineSeries Program



Source: Self made

1.3.4 Pseudo Random Number Generation

A random number is just that – a number generated randomly. A random number between 1 and 6 can be generated by throwing a dice – the number that is thrown is random – it cannot be predicted accurately. Computers are not very good at generating truly random numbers. It is difficult to get a computer to do something by chance. A computer follows its instructions blindly and is therefore completely predictable. To be truly random, the value must not be predictable. However, computers use complex mathematical formulas to generate values which *look* random, but really aren't. Even though the formulas are complex, they're still predictable. Therefore random numbers generated using algorithms are called pseudo random number.

One of the most popular methods for generating random numbers is the **linear congruential generator** that uses the formula

$$X_{i+1} = (a * X_i + b) \bmod m$$

The values of the variables used in the formula should satisfy the constraints given below:

Variable	Name	Constraint
m	Modulus	• m ≥ length of the random sequence required • m should be greater than x_0, a and b

		• m and b are *Relatively Prime* so the gcd(m, b) = 1.
a	Multiplier	• if m is a power of 2, then a must satisfy the condition a mod 8 = 5 • if m is a power of 10, then a must satisfy the condition a mod 200 = 21. • $\sqrt{m} \leq a \leq m - \sqrt{m}$ • (a-1) should be a multiple of every prime dividing into m: (a-1) is divisible by all prime factors of m • If m is a multiple of 4 then (a-1) should also be a multiple of 4 : (a-1) mod 4 = 0 if (m mod 4) = 0
b	Increment	Should be odd and not a multiple of 5
X_0	Seed Value	$0 \leq X_0 < m$

This linear congruential generator generates a sequence of numbers within a specified range. Because of the modulus arithmetic involved, this will generate every number in the range once before repeating.

Armed with knowledge lets make a sequence 256 elements long. So $m=2^8$. This makes choosing b simple as it can be any odd number less than m. Lets choose b = 27. Now choosing a value for a is also easy, choose a multiple of 4 and add 1. a = 65. The initial seed X_0 can be chosen as 0.

The algorithm for pseudo random number generation is:
1. Initialize $m=2^8$, b = 27, a = 65, x =0
2. x = (a * x + b) % m

Value addition: Source Code

Heading text – The PseudoRandom Program

```
/* This program generates pseudo random numbers using the
   Linear congruential method formula
   Xi+1 = (a * Xi + b) mod m */

#include <iostream>
using namespace std;

int main()
{
   int a = 65; //multiplier
   int b = 27; //increment
```

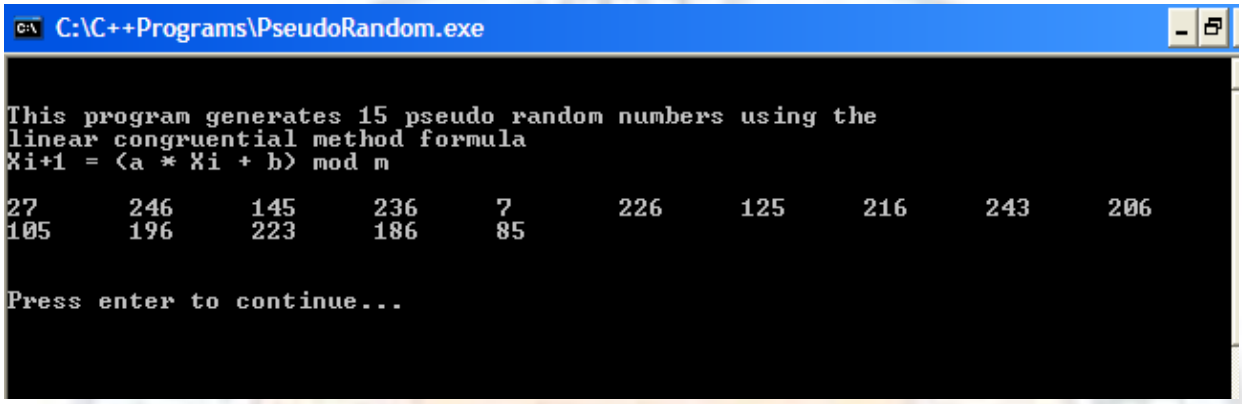
```
    int m = 256; //modulus
    int x = 0; //initial seed value
    int i = 1; //keeps track of number of values generated

    cout << endl<<endl;
    cout << "This program generates 15 pseudo random numbers using the"<<endl;
    cout << "linear congruential method formula" <<endl;
    cout << "Xi+1 = (a * Xi + b) mod m"<<endl<<endl;

    while (i <= 15)
     {
         x = (a * x + b) % m; //linear congruential method formula
         cout << x << "\t";
         i = i + 1;
     }

    cout << endl;

    cout << endl << endl<< "Press enter to continue...";
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – The PseudoRandom Program

Body text:



Source: Self made

Value addition: Did you Know?

Heading text – The rand() and srand() functions in C++

Body text:

In the C++ standard library <cstdlib> the two functions related to random number generation are rand() and srand().

Generate random number

int rand (void);

Returns a pseudo-random integral number in the range 0 to RAND_MAX. This number is generated by the linear congruential method algorithm that returns a sequence of apparently non-related numbers each time it is called. This algorithm uses a seed to generate the series, which should be initialized to some distinctive value using srand.

A typical way to generate pseudo-random numbers in a determined range using rand is to use the modulo of the returned value by the range span and add the initial value of the range:

(value % 100) is in the range 0 to 99
(value % 100 + 1) is in the range 1 to 100
(value % 30 + 1985) is in the range 1985 to 2014

Initialize random number generator

void srand (unsigned int seed);

The pseudo-random number generator is initialized using the argument passed as *seed*.

For every different *seed* value used in a call to srand, the pseudo-random number generator can be expected to generate a different succession of results in the subsequent calls to rand.

Example

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main ()
{
  int iSecret, iGuess;

  /* initialize random seed: */
  srand ( time(NULL) );

  /* generate secret number: */
  iSecret = rand() % 10 + 1;

  do {
    cout << "Guess the number (1 to 10): ";
    cin >> iGuess;
    if (iSecret<iGuess)
       cout << "The secret number is lower";
    else if (iSecret>iGuess)
       cout << "The secret number is higher";
  } while (iSecret!=iGuess);

  cout << "Congratulations!";

  getchar();
  return 0;
}
```

Source: http://www.cplusplus.com

1.4 Factoring Methods

A factor is a number that completely divides into another number. Factoring methods – finding factors - are used for finding whether a number is prime, finding the greatest common divisor, adding fractions, solving polynomial equations, and many other purposes**.**

In this chapter you will learn to solve some factoring problems using the problem solving strategies – hypothesis testing and reduction. We will apply hypothesis testing to find the square root of a number. We will apply reduction to find the greatest common divisor of a pair of integers.

We will also learn that a simple solution to a problem can be improved upon to get a more efficient algorithm. This will be demonstrated through the problem of finding the smallest divisor of an integer.

Learning these problem solving strategies will help you apply them to new problems that you encounter for programming.

1.4.1 Learning Objectives

After reading this chapter you should be able to:

4. Apply problem solving strategies to develop algorithms for factoring problems.

Apply hypothesis testing to design an algorithm to find square root of a number.

Apply reduction to design an algorithm to find the greatest common divisor of two integers.

Design an algorithm to find the smallest divisor of an integer.

1.4.2 Square Root of a Number

We will now devise an algorithm to determine the square root of a number. Given a number m we would like to determine the square root of the number, n, such that

$$n * n = m$$

The square root of a number is a factor which when squared gives back the number.

For example, square root of 81 is 9. The square root of 1448.64 is 38.061.

This time the problem solving strategy that we use is **hypothesis testing**. In hypothesis testing, a solution to the problem is assumed. This guess is then tested to determine if it is the required solution. If not, then we improve upon the guess in such a way that the new guess is closer to the desired solution. The process is repeated until the guess converges to the solution.

To find the square root of a number,

1. Guess a solution, n_0, to the problem (we will make an intelligent guess of course – not just a random one!).

2. Test whether n_0 is the square root of m (by testing whether $n_0 * n_0$ is equal to m or not).

3. If n_0 is not the solution, make another guess at the solution, say, n_1 (again we will estimate this intelligently so that n_1 converges towards the solution).

4. Keep updating the guesses until we reach the actual solution – the square root of m.

An intelligent guess for the initial solution n_0 is m/2 – since the square root of m cannot be less than half the number.

To determine subsequent guesses we will use **Heron's principle** of finding square root of a number:

> If you divide m by a number n_0 which is not the square root, you will get the quotient (q) different from the square root. However the average of n_0 and q is closer to the actual root than the starting number n_0.

Thus we can improve upon the original guess by estimating the new guess for the square root as the average of the original guess and the quotient.

For example, to find the square root of 500, let us guess that the square root is 250.
Since 250 * 250 = 62500, 250 is too large a guess for the square root of 500.

Next we divide 500 by 250 to get the quotient 2.

Since 2 * 2 = 4, 2 is too small a guess for the square root of 500.

Thus, when we eliminate 250 (our guess) as a solution for the square root – because it's too large, at the same time we can also eliminate 2 (number/guess) as a solution – because it's too small.

So a good intelligent new guess will be somewhere between 250 and 2.

We take the average of these two numbers as the new guess which is (250+2)/2 = 126. This 126 is closer to the actual root of 500 than the initial estimate of 250.

Repeating the above process to whatever level of accuracy is desired, we will arrive at the solution. The actual square root of 500 is 22.3607 as shown by the Figure 4.1.

Guess	**Quotient**	**Average (New Guess)**
250	500/250 = 2	(250+2)/2 = 126
126	500/126=3.96825	64.9841
64.9841	500/64.8641=7.69419	36.3392
36.3392	500/36.3392=13.7593	25.0492
25.0492	500/25.0492=19.9607	22.505

22.505	500/22.505=22.2173	22.3611
22.3611	500/22.3611=22.3602	22.3607
22.3607	500/22.3607=22.3607	22.3607

Figure 3.1 – Heron's Method to calculate square root of 500

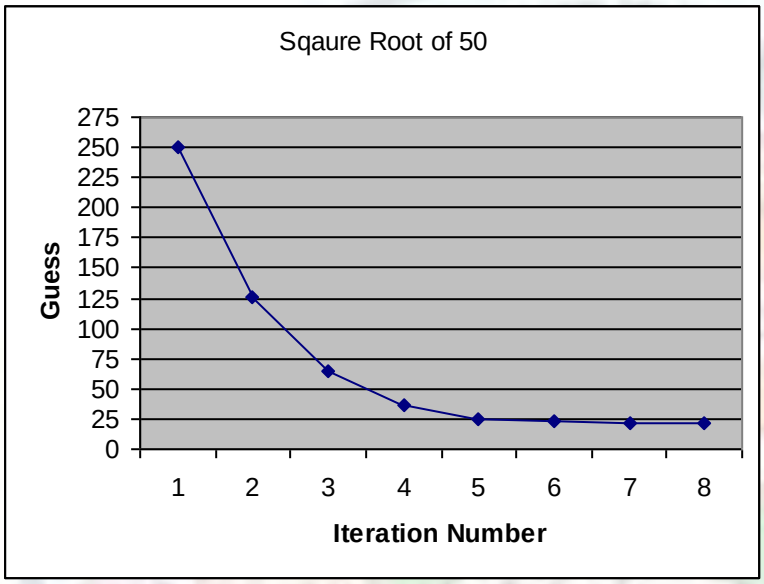


Figure 3.2 – Subsequent guesses converge to the solution

The **algorithm** for square root computation is as follows:

1. Prompt the user for a number.
2. Make a guess for the square root of the number as number/2.
3. Repeat
4. Divide the number by the guess to get a quotient.
5. Average the guess and the quotient.
6. Make this average value your new "guess"
7. Until difference between the original guess and the new guess is less than the precession level required.

Value addition: Source Code
Heading text – The SquareRoot Program

```
/* This program finds the square root of a number using Heron's method */

#include <iostream>
using namespace std;

double Abs(double nbr){     //function to return the absolute value of a number
      if( nbr >= 0 )
         return nbr;
      else
         return -nbr;
}

int main()
{

   double number;
   double guess;
   double newguess;
   const double precission = 1.0e-7;

   cout << "This program finds the square root of a number using Heron's method";
   cout << "\n\nEnter a number: ";
   cin >> number;

   newguess = number/2;

   cout << "\n\n\tGuess\t\tQuotient\tAverage(New Guess)"<<endl;
   cout << "\t    \t\t("<<number<<"/Guess)\t(Guess+Quotient)/2"<<endl<<endl;
   do
   {
     guess = newguess;
     newguess = (guess + number/guess) /2;
     cout << "\t" << guess << "\t\t" << number /guess<< "\t\t" <<newguess<<
endl;
   }
   while ( Abs(guess - newguess) > precission);

   cout << endl;
   cout << "The square root of " << number << " is " << newguess << "\n\n";
   cout << "Press Enter to Continue...";
   getchar();
   return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – The SquareRoot Program

Body text:

```
C:\C++Programs\SquareRoot.exe
This program finds the square root of a number using Heron's method

Enter a number: 500

        Guess           Quotient        Average(New Guess)
                        (500/Guess)     (Guess+Quotient)/2

        250             2               126
        126             3.96825         64.9841
        64.9841         7.69419         36.3392
        36.3392         13.7593         25.0492
        25.0492         19.9607         22.505
        22.505          22.2173         22.3611
        22.3611         22.3602         22.3607
        22.3607         22.3607         22.3607

The square root of 500 is 22.3607

Press Enter to Continue...
```

Source: Self made

Value addition: Did you Know?

Heading text – The Method of Heron of Alexandria

Heron of Alexandria (Circa 10 CE to 75 CE)

Heron of Alexandria lived in the first century BC and contributed especially to mechanics, optics, and geometry. Today he is mainly remembered for a remarkable way of computing the area of a triangle (Heron's Formula) -- namely as the square root of a four-dimensional volume!

Physical evidence exists that the Babylonians had a method of calculating the square root of some numbers as early as 2000 years before the birth of Christ. In the Yale collection there is an artifact that shows the calculation of

the square root of two to five decimal places accurately. the ancient Babylonian method seems to be the same as the Heron's method.
Source: Self made Image from The MacTutor History of Mathematics archive.

1.4.3 Greatest Common Divisor

Next we will write an algorithm to determine the greatest common divisor (gcd) of a pair of numbers. The problem solving strategy that we use here is **reduction**.

In reduction, we reduce a given problem to a smaller problem successively until we reach a solution.

The gcd of two integers is the largest integer that will divide exactly into the two integers leaving no remainder. For example, the gcd of 30 and 18 is 6. One way to find the gcd of two integers is to factor them and search for the largest common factor. This method is not very efficient.

There is a better, more efficient algorithm attributed to the famous Greek Mathematician Euclid known as the Euclid's gcd Algorithm.

The algorithm is based on the following two observations:

1. If a is completely divisible by b, then gcd(a, b) = b.

   This is because no integer (b, in particular) can have a divisor greater than the number itself.

2. If *r* is the remainder when *a* is divided by *b*, then gcd(a, b) = gcd(b, r).

   That is every common divisor of *a* and *b* also divides r. Thus gcd(a,b) divides r. But, of course, gcd(a, b) is completely divisible by b. Therefore, gcd(a, b) is a common divisor of b and r and hence gcd(a, b) ≤ gcd(b, r). The reverse is also true because every divisor of b and r also divides a.

Thus, by observation 2, we reduce the problem of computing gcd(a,b) to the problem of computing gcd(b,r). Note that the problem is reduced because (b,r) is a smaller pair than (a,b).

Starting with any two positive integers and performing repeated reductions will always eventually produce a pair where the second number is 0. Then the GCD is the other number in the pair.

For example,

gcd(206,40) = gcd(40,6) since 206%40 = 6
= gcd(6,4) since 40%6 = 4

= gcd(4,2) since 6%4 = 2
= gcd(2,0) since 4%2 = 2
= 2

The gcd algorithm is efficient, even for large numbers. The remainder is at most half the divisor, so it doesn't take long to converge to the gcd.

The **algorithm** for finding greatest common divisor of two numbers is as follows:

1. Prompt the user for two numbers a and b.
2. Initialize remainder r = -1
3. Repeat
4. r = a % b;
5. a = b;
6. b = r;
7. Until r becomes 0
8. gcd = a

Value addition: Source Code

Heading text – The GCD Program

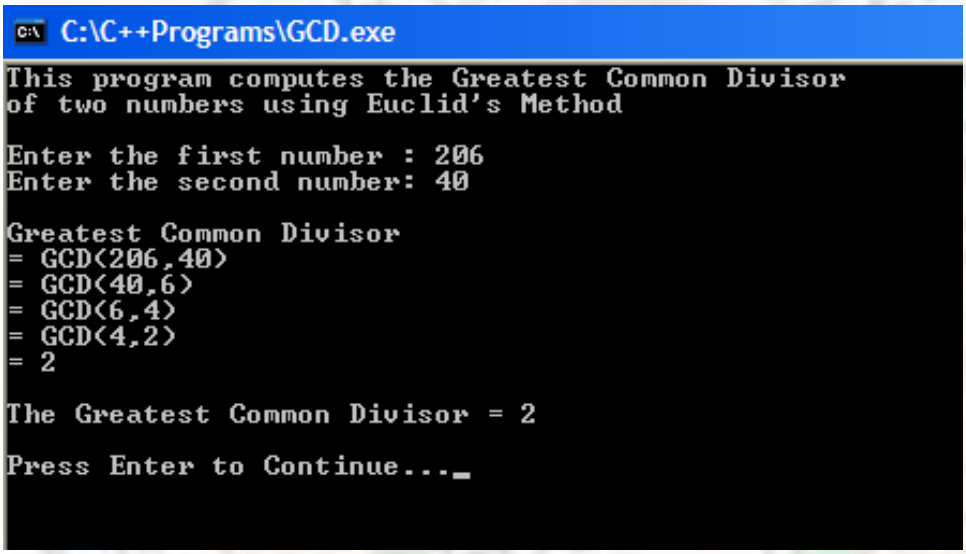
```
/* This program computes the Greatest Common Divisor of
   two numbers using Euclid's Method */

#include <iostream>
using namespace std;

int main()
{
   int a, b, r = -1, gcd;
   cout << "This program computes the Greatest Common Divisor\n";
   cout << "of two numbers using Euclid's Method \n\n";

   cout << "Enter the first number : ";
   cin >> a;
   cout << "Enter the second number: ";
   cin >> b;

   cout << "\nGreatest Common Divisor\n";
   while( r != 0 )
        {
                cout << "= GCD("<<a<<","<<b<<")\n";
                r = a % b;
                a = b;
                b = r;

        }
    gcd = a;
    cout << "= " << gcd<<endl;

    cout << "\nThe Greatest Common Divisor = " << gcd << endl;
```

```
    cout << "\nPress Enter to Continue...";
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – The GCD Program



Source: Self made

1.4.4 Smallest Divisor of an Integer

The next problem we consider is to determine the smallest divisor of a number. Given an integer n determine the smallest integer (other than 1) that completely divides n.

To solve the problem, we can iteratively divide n by integers starting from 2, 3, 4, and so on up to n. As soon as we find an integer that completely divides n, we can stop – this integer is the smallest divisor.

However, when n is large, this approach is very inefficient.

Notice that if the number n is even – 2 will be its smallest divisor. If the number is odd then we need not check for divisibility by even numbers. We can iteratively divide n by integers starting from 3, 5, 7 and so on. Taking this approach, we have reduced the possible number of divisibility checks by half!

Also, it is not necessary to look for the smallest divisor of n beyond the $\sqrt{n}$. Let us look at an example to see why this is so.

Problem Solving Approaches

Consider the integer 225 and all its divisors {3, 5, 9, 15, 25, 45, and 75}.

Observe that if 3 is a factor, then 225/3 = 75 is also a factor – since 3 completely divides 225 leaving remainder 0 and quotient 75, 75 will also completely divide 225 leaving remainder 0 and quotient 3.

The smallest factor pairs up with the largest factor. The second smallest pairs with the second largest and so on.

3 pairs up with 75

5 pairs up with 45

9 pairs up with 25

15 pairs up with 15

We see that each factor from the left of the divisor list (l) pairs up with a number from the right of the list (r).

The factors l and r are such that l * r = n.

The crossover point in the list will occur when l = r → l * l = n → when l is the square root of n.

Thus we can run our test for divisibility for odd integers with divisor starting from 3 , incrementing divisor by 2 in every iteration and going at most up to √n.

The **algorithm** for finding smallest divisor of an integer is as follows:

1. Prompt the user for a number n.
2. Initialize smallest-divisor = number for the worst case
3. if number is divisible by 2 then smallest-divisor = 2
4. else
5. limit = √number
6. divisor = 3
7. if (number is not divisible by divisor and divisor <= limit)
8. increment divisor by 2
9. go back to step 7
10. if divisor is less than the limit then smallest-divisor = divisor

Value addition: Source Code

Heading text – The SmallestDivisor Program

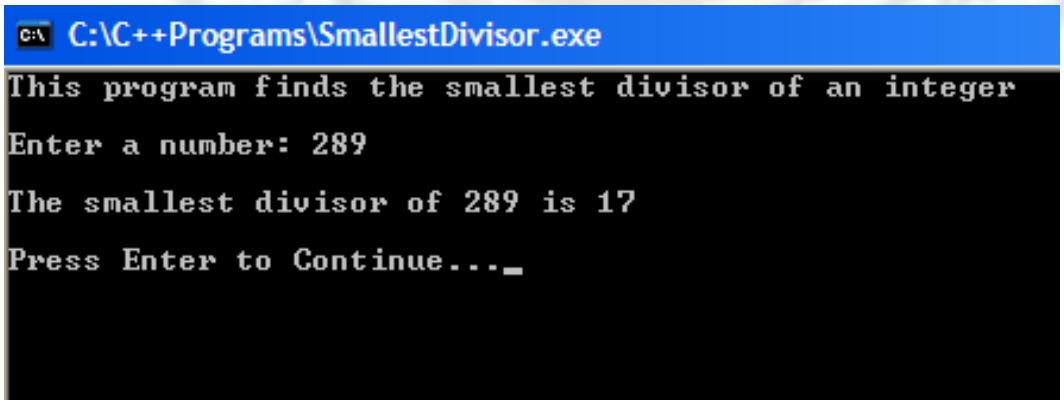
```
/* This program finds the smallest divisor of an integer */

#include <iostream>
#include <cmath>
using namespace std;

int main()
{
```

```
    int number;
    int limit;
    int divisor;
    int smallest;

    cout << "This program finds the smallest divisor of an integer";
    cout << "\n\nEnter a number: ";
    cin >> number;

    smallest = number; //worst case - smallest divisor is the number itself

    if ( (number % 2) == 0)
          smallest = 2;  //if number is even, smallest divisor is 2
    else
    {
       limit = (int) sqrt(number);   //limit upto which to test for divisibility
       divisor = 3;  //begin checking whether 3 is a divisor
       while (number % divisor != 0 && divisor <= limit)
          divisor = divisor + 2;  //increment divisor to next odd number
    }
    if (divisor <= limit)
      smallest = divisor;  // divisor did not cross limit
                        //this means smallest divisor was found

    cout << "\nThe smallest divisor of " << number << " is ";
    cout << smallest << "\n\n";
    cout << "Press Enter to Continue...";
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution

Heading text – The SmallestDivisor Program



Source: Self made

1.5 More Factoring Methods

In this chapter you will learn to solve some more factoring problems using the problem solving strategy – divide and conquer.

We will apply this strategy to raise a number to a large power and to compute the nth Fibonacci number. We will also learn how to improve algorithms by developing an efficient algorithm for generating prime numbers.

Learning these problem solving strategies will help you apply them to new problems that you encounter for programming.

1.5.1 Learning Objectives

After reading this chapter you should be able to:

5. Apply problem solving strategies to develop algorithms for factoring problems.
   Design an algorithm to generate prime numbers.
   Apply divide and conquer to design an algorithm to raise a number to a large power.

1.5.2 Generating Prime Numbers

The next problem that we consider is generation of prime numbers. The specific problem is – generate all prime numbers in the first N integers.

Recall that a number is prime if it is exactly divisible by only 1 and the number itself – it has no other factors. The prime numbers up to between 1 and 40 are:

2 3 5 7 11 13 17 19 23 29 31 37

The simplest method for prime number generation was developed by Eratosthenes in the 3rd century B.C. Here's how it works:

To find all prime numbers below a given number N:

1. Write all integers from 1 through N in order.

2. One is a not a prime number and is crossed out right away.

3. On every step, find the first number not yet crossed, circle it and cross out all of its remaining multiples. For example, in the first step, circle 2 and cross out all multiples of 2. (2, 4, 6, 8, 10, ...)

4. Repeat this step while the least available number does not exceed the square root of N.

When the algorithm stops, the prime numbers are either marked with circles or not crossed.

We need to check only up to the square root of N because for a composite number A= a*b less or equal to N, the factors a and b can't both exceed $\sqrt{N}$. Thus any prime factor of A

can't exceed √A, let alone √N.

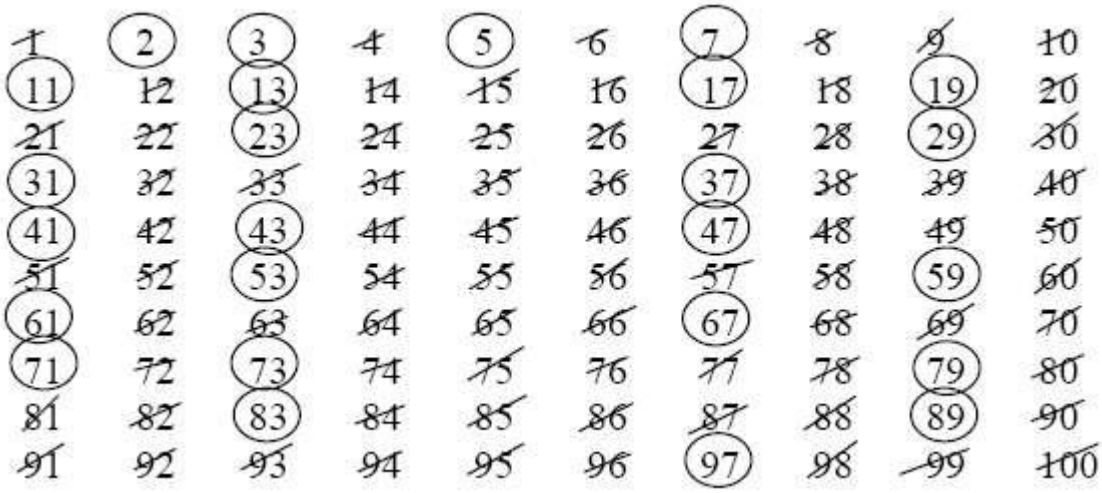


Figure 5.1 – Sieve of Eratosthenes

Unfortunately, the Eratosthenes method is rather time-consuming when the numbers you are looking for are much larger. Also we well need storage space proportional to N for storing the numbers.

How can we improve upon this method and make it more time and space efficient?

First of all, instead of storing the numbers, we generate the numbers one by one up to N. We do not need to generate even numbers since they cannot be prime (except for 2).

Also notice that after striking out multiples of first 2 and then 3 we are left with the sequence:

5 7 11 13 17 19 23 25 29 …

Difference between successive pairs in this sequence alternates between 2 and 4.

(7-5 = **2;** 11 – 7 = **4**; 13 -11 = **2**; 17 -13 = **4**; 19 – 17 = **2**; 23 – 19 = **4**; …)

This means, beyond 5, we need to only generate numbers in this alternating sequence. This can be done with the following two constructs:

```
dx = abs (dx – 6)          // dx is initialized to 2; abs → absolute value
X = X + dx
```

This reduces the number we check for primality by 2/3rd and we end up with an efficient algorithm (though not the most efficient!!).
For each newly generated number, X, determine if it is prime by checking for its divisibility by only **odd** integers from 3 up to the √X.

The **algorithm** for generating prime numbers is thus as follows:

1. Prompt the user for a number N.
2. Print the first three primes (2, 3, and 5) if required.
3. Initialize X = 7 and dx = 2

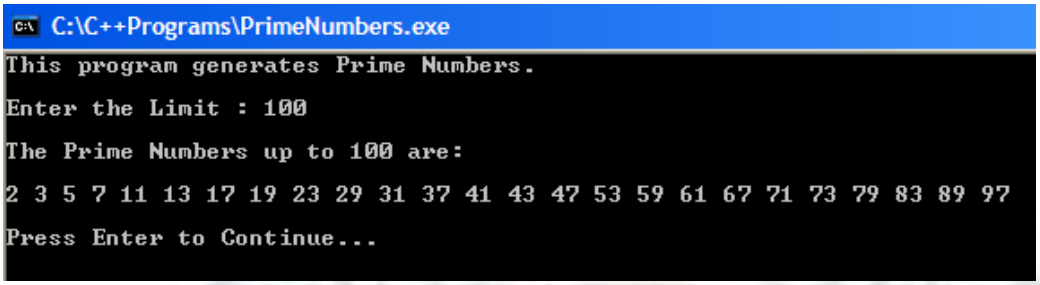
4. While X <= N.
5. Test whether X is prime dividing by all odd integers from 3 up to √X
6. Generate next X using X = X + dx.
7. Update dx = abs (dx – 6) for next number generation.

Value addition: Source Code

Heading text – The PrimeNumbers Program

```
/* This program generates Prime Numbers from 1 to N*/

#include <iostream>
#include <cmath>
using namespace std;

int main()
{
   int N, X, j, limit,dx=2;
   bool prime;

   cout << "This program generates Prime Numbers.\n\n";

   cout << "Enter the Limit : ";
   cin >> N;

   cout << "\nThe Prime Numbers up to " << N << " are: \n\n";

   if (N >= 2) cout <<"2 ";
   if (N >= 3) cout <<"3 ";
   if (N >= 5) cout <<"5 ";

   for (X= 7; X <= N; X = X+dx)     //generate alternating sequence upto N
   {
       j = 3;                    //Check for divisibility of X by all odd numbers
       limit = (int) sqrt(X);        //from 3 up to the square root of X
       prime = true;                 //assume X is prime
       while(prime && (j <= limit))
            {
                  prime = (X % j != 0); //If X is divisible by j
                                        //then prime will become false
         j = j + 2;           //Check for divisibility only by odd integers
       }
       if (prime)
         cout << X << " ";
       dx = abs(dx -6);
    }
    cout << "\n\nPress Enter to Continue...";
    getchar();
    return 0;
}
```

Source: Self made

Value addition: Program in Execution
Heading text – The PrimeNumbers Program
Body text: 
Source: Self made

1.5.3 Raising a Number to a Large Power

In this section, we will use the **divide and conquer** problem solving strategy. In this strategy, the given problem is reduced to smaller problems. The solution to the smaller problems is combined to get a solution to the original problem.

We will use divide and conquer to raise a number to a large power.

The specific problem is – Given an integer a compute the value of a^N where N is a positive integer considerably larger than 1.

We know that raising a number to a power is just a matter of accumulating products.

```
product = 1;
for (int i = 1; i <= N ; i++)
        product = product * a;
```

Lets say N = 8, then, 7 multiplications are performed by the above algorithm.

$b_1 = a * a$
$b_2 = b_1 * a$
$b_3 = b_2 * a$

$b_4 = b_3 * a$
$b_5 = b_4 * a$
$b_6 = b_5 * a$
$b_7 = b_6 * a$
$= a^8$

We can reduce the number of multiplications required to three using the following strategy:

$b_1 = a * a$ $(b_1 = a^2)$
$b_2 = b_1 * b_1$ $(b_2 = a^2 * a^2 = a^4)$
$b_3 = b_2 * b_2$ $(b_3 = a^4 * a^4 = a^8)$
$= a^8$

More generally, how do we implement $b = a^N$ with the smallest possible number of multiplications?

Put in other words, what is the smallest pair of numbers that will add up to N (we need a pair of numbers since we can only multiply two numbers at a time). Obviously it is N/2 and N/2 – that is, to compute a^N we have $a^N = a^{N/2} * a^{N/2}$. Similarly to compute $a^{N/2}$ we halve the powers and multiply $a^{N/4}$ with $a^{N/4}$. For example, $a^8 = a^4 * a^4$.

However, if the power is odd, say 11, then halving the power gives 6.5 which is not an integer – we are forced to evaluate a^{11} as $a^{11} = a^{10} * a = a^5 * a^5 * a$.

Thus we conclude –

Action **0**: If the power is even, it can be computed by squaring a power that is half its size. (For example, $a^4 = a^2 * a^2$)

Action **1**: If the power is odd, it can be computed using a power that is one less and hence by squaring a power that is half its size and the number. (For example, $a^{11} = a^5 * a^5 * a$)

This power halving (or repeated division of power by 2) approach can continue iteratively until the power is halved down to 1.

Computation Required	$a^{11} = a^5 * a^5 * a$	$a^5 = a^2 * a^2 * a$	$a^2 = a^1 * a^1$	$a = a^0 * a^0 * a$
Action Taken	**1**	**1**	**0**	**1**

If you have noticed it, the second row read from left to right is nothing but the binary representation of 11 - that is $(1011)_2$ in reverse. This is not a coincidence. Recall that when we compute the binary representation of N, we repeatedly divide N by 2 to get the binary representation in reverse.

This gives us the idea that we can use the binary representation of the number to decide how to accumulate the product.

We compute successive squared powers of a as: a, a^2, a^4, a^8,… These terms are accumulated in the product only when they appear in the binary representation.

For example, since binary representation of 11 is 1011, it indicates a^{11} can be computed as $a^{11} = (1.a^{8)} * (0.a^4) * (1.a^2) * (1.a^1)$. So, only the terms a, a^2, and a^8 are multiplied with product – that is only when the corresponding binary digit is 1.

We maintain two variables: the successive member of the power sequence, **psequence** and the accumulated product, **product.**

These values are initialized as **product = 1** and **psequence = a.**

In every iteration,

The psequence is updated with
psequence = psequence * psequence

The product is updated (only when the corresponding binary digit is 1) with

product = product * psequence.

For example, for computing a^{11}

- psequence is squared in successive iterations and takes the values: a, a^2, a^4, a^8, …
- product is multiplied with psequence only when psequence is a, a^2, and a^8 (or n%2 == 1)

Since $a^{11} = a^8 * a^2 * a$: a^4 is calculated but not multiplied with product

n		11	5	2	1
n%2		1	1	0	1
product	1	1*a	$a*a^2$	No change	$a*a^2*a^8$
psequence	a	$(a*a) = a^2$	$(a^2*a^2) = a^4$	$(a^4*a^4) = a^8$	$(a^8*a^8) = a^{16}$

The **algorithm** for raising a number to a large power is thus as follows:

1. Prompt the user for a and n.
2. Initialize product = 1, psequence = a.
3. while (n > 0)
4. if (n % 2 == 1)
5. product = product * psequence;
6. n = n / 2;
   psequence = psequence * psequence;

Value addition: Source Code

Heading text – The LargePowers Program

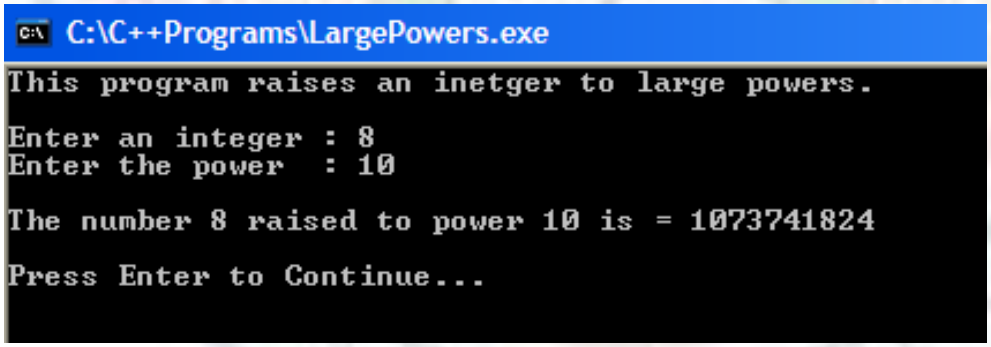
```
/* This program raises an inetger to large powers
   more efficiently than accumulating products*/

#include <iostream>
#include <cmath>
using namespace std;

int main()
{
   long a, n, n1;
   long product = 1;
   long psequence;

   cout << "This program raises an integer to large powers.\n\n";

   cout << "Enter an integer : ";
```

```
    cin >> a;
    cout << "Enter the power  : ";
    cin >> n;

    n1 = n;
    psequence = a;

    while (n > 0)
    {
        if ( n % 2 == 1)
            product = product * psequence;
        n = n / 2;
        psequence = psequence * psequence;
    }

    cout << "\nThe number " << a << " raised to power " << n1 << " is = " <<
product;
    cout << "\n\nPress Enter to Continue...";
    getchar();
    return 0;
}
```

Source:

Value addition: Program in Execution

Heading text – The LargePowers Program



Source: Self made

Summary

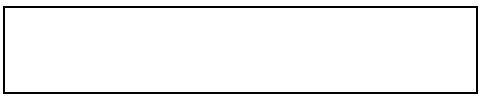
- The design phase of the program development cycle is the most important phase of program development especially in the case of industry level code.
- A good way to begin designing programs is to identify the major tasks that the program must accomplish. Each of these tasks then becomes a **module** in the program.

Problem Solving Approaches

- Identifying the tasks and various subtasks involved in the program design is called **modular programming**.
- This process of breaking down a problem into simpler and smaller sub problems repeatedly until no further break down is possible is called **stepwise refinement** or **top-down design**.
- A modular approach to program design has many benefits including - improved readability, increased productivity, higher maintainability, reduced time for development, smaller programs and reusability.
- A **hierarchy chart** helps us keep track of the relationships between modules in a visual way.
- **Pseudocode** uses short English like phrases to describe the outline of a program giving specific instructions to accomplish each task.
- A **flowchart** is a diagram that uses special symbols to display pictorially the flow of execution within a program or a program module.
- There are some general problem-solving techniques that can be used to derive a solution to a problem. These include – Abstraction, Analogy, Hypothesis Testing and Divide and Conquer.

- An algorithm is a precise step-by-step sequence of instructions that usually begins with an input value and produces an output value in a finite number of steps.
- The properties that an algorithm should have include - It should terminate/halt in a finite number of steps; The instructions should be precise and computable; It should produce a result.
- An algorithm is independent of the programming language used. Once you have designed the algorithm it should be possible to code the corresponding program in any programming language.
- A simple problem solving strategy is to first decide the initialization steps, then decide the iterative steps and lastly decide the termination steps.

- The method of continuous dividing is used for base number conversion.
- The algorithm to compute the sine series given by the following infinite series:

  Sin(x) = 

  can reduce calculations by noticing that

Current term = Previous term * (x*x) / ((i-1)* (i)).

- One of the most popular methods for generating random numbers is the **linear congruential generator** that uses the formula

$$X_{i+1} = (a * X_i + b) \bmod m$$

The values of the variables used in the formula should satisfy certain constraints.

- Hypothesis testing is a problem solving strategy where first a solution to the problem is assumed, then tested to determine if it is the required solution. A new guess is made iteratively until the guess converges to the solution.
- Reduction is a problem solving strategy where we reduce a given problem to a smaller problem successively until we reach a solution.
- To determine the square root of a number, we can use the Heron's method.
- To find the greatest common divisor of a pair of numbers we can use Euclid's algorithm.

- Divide and conquer is a problem solving strategy where the given problem is reduced to smaller problems. The solution to the smaller problems is combined to get a solution to the original problem.

Exercises

1.1.1 List the characteristics of a program module.

1.1.2 What are the benefits of modular program design?

1.1.3 What are the advantages of using

i) Flowcharts

ii) Pseudocode

1.1.4 Develop a hierarchy chart, pseudocode and flowchart to calculate and print the average grade of three tests for an entire class.

1.1.5 Draw a flowchart for factorial computation.

1.1.6 Write the algorithm in pseudocode for a simple calculator. Also draw a hierarchy chart for the same.

1.2.1 Design an algorithm that needs only n-1 additions to compute the average of n numbers.

1.2.2 Write a program to generate the Lucas sequence:

1 3 4 7 11 18 29

1.2.3 Starting with a single pair, how many pairs of rabbits will there be at the beginning of each month given that - A pair of rabbits, one month old, is too young to reproduce. Suppose that in their second month, and every month thereafter, they produce a new pair. Each new pair of rabbits does the same, and none of the rabbits dies.

(**Hint**: Try to determine the sequence for number of pairs of rabbits at each month end. At the end of the first month, they mate, but there is still one only 1 pair. At the end of the second month the female produces a new pair, so now there are 2 pairs of rabbits in the field. At the end of the third month, the original female produces a second pair, making 3 pairs in all in the field.)

1.2.4 Write a program that counts the number of digits in a number.

1.2.5 Write a program to generate a number given the digits. For example: If the given digits are 4, 2, 1, and 9 - Output should be 4219.

1.2.6 A three digit number is Armstrong if the sum of cubes of its digits is equal to the number itself. Input: 153 (is Armstrong) as $153 = 1^3 + 5^3 + 3^3$ Input: 456 (is not Armstrong)). Write a program to determine whether a three digit number is Armstrong number or not.

1.3.1 Write a program that converts a decimal number to its corresponding hexadecimal number.

1.3.2 Write a program to evaluate the function cos(x) as defined by the following infinite series expansion:

$$\text{Cos(x)} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^6}{8!} - \frac{x^{10}}{10!} + \ldots$$

The acceptable error for the computation is 10^{-6}.

1.3.3 The exponential growth constant e is characterized by the expression

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

Write a program to compute e up to n terms.

1.3.4 Write a program that generates a random number 1 through 100. The program then asks the user to guess the number. If the user guesses too high or too low then the program should output "too high" or "too low" accordingly. The program must let the user continue to guess until the user correctly guesses the number. Also display how many guesses it took the user to correctly guess the right number.

1.3.5 Modify exercise 3.4 so that the player picks the number, and the computer has to guess it. The user must tell the computer whether it guessed too high or too low.

1.4.1 Write a program which finds the roots of a quadratic equation from the three coefficients (a, b, c) using the formula roots= $(-b \pm \sqrt{(b^2 - 4ac)}) / 2a$.

1.4.2 Write a program which takes a rational number from the user as a numerator and a denominator. The program should remove common factors from this number, for example, 15/24 = 5/8. To do this you will have to find the GCD of the two numbers and then divide the original values by this. For example, GCD of 15 & 24 is 3, 15/3=5, 24/3=8.

1.4.3 Write a program to add two rational numbers, for example, 2/7+1/6=12/42+7/42=19/42.

1.4.4 Write a program which takes an integer from the user and then offers a choice between the displaying the smallest, the largest or all the factors of the integer. Use a switch statement.

1.5.1 Write a program that takes as input an integer N and that indicates if N is a prime number or not.

1.5.2 Write a program that takes as input an integer N and determine the number of prime numbers below N.

1.5.3 Write a program that takes as input an integer N and compute the N^{th} prime number.

1.5.4 Raising a number to a large power n will need n-1 multiplications if a simple loop to accumulate products is used. What is the number of multiplications needed when you use the algorithm given in section 5.2?

1.5.5 Write a program that takes less than n-1 multiplications to compute a^n.

Glossary

Algorithm: a precise step-by-step sequence of instructions that usually begins with an input value and produces an output value in a finite number of steps.

Divide and conquer: a problem solving strategy where the given problem is reduced to smaller problems. The solution to the smaller problems is combined to get a solution to the original problem.

Flowchart: A program design tool that uses special symbols to display pictorially the flow of execution within a program or a program module.

Heron's Principle: If you divide m by a number n_0 which is not the square root, you will get the quotient (q) different from the square root. However the average of n_0 and q is closer to the actual root than the starting number n_0.

Hierarchy Chart: A program design tool that helps us keep track of the relationships between modules in a visual way.

Hypothesis Testing: a problem solving strategy where first a solution to the problem is assumed and then tested to determine if it is the required solution. A new guess is made iteratively until the guess converges to the solution.

Modular Programming: Identifying the tasks and various subtasks involved in the program design.

Pseudocode: A program design tool that uses short English like phrases to describe the outline of a program giving specific instructions to accomplish each task.

Reduction: a problem solving strategy where we reduce a given problem to a smaller problem successively until we reach a solution.

Stepwise Refinement: The process of breaking down a problem into simpler and smaller sub problems repeatedly until no further break down is possible,

Top-down Design: The process of breaking down a problem into simpler and smaller sub problems repeatedly until no further break down is possible.

References

1. Suggested Reading

1. B. A. Forouzan and R. F. Gilberg, Computer Science, A structured Approach using C++, Cengage Learning, 2004.
2. R.G. Dromey, How to solve it by Computer, Pearson Education
3. E. Balaguruswamy, Object Oriented Programming with C++ , 4th ed., Tata McGraw Hill
4. G.J. Bronson, A First Book of C++ From Here to There, 3rd ed., Cengage Learning.
5. Graham Seed, An Introduction to Object-Oriented Programming in C++, Springer
6. J. R. Hubbard, Programming with C++ (2nd ed.), Schaum's Outlines, Tata McGraw Hill
7. D S Malik, C++ Programming Language, First Indian Reprint 2009, Cengage

Learning

8. R. Albert and T. Breedlove, C++: An Active Learning Approach, Jones and Bartlett India Ltd.

2. Web Links

1.1 http://www.cse.iitd.ernet.in/~suban/CSL102/lecture/node36.html
1.2 http://www.indiastudychannel.com/resources/104906-Modular-Programming.aspx
1.3 http://www.nos.org/htm/basic2.htm
1.4 http://www.unf.edu/~broggio/cop2221/2221pseu.htm
1.5 http://cnx.org/content/m18682/latest/
1.6 http://www.cplusplus.com/