Chapter 1

The World of Problem-solving

1.1 Introduction

"The greatest challenge to any thinker is stating the problem in a way that will allow a solution."

– Bertrand Russell

"A great discovery solves a great problem but there is a grain of discovery in the solution of any problem. Your problem may be modest; but if it challenges your curiosity and brings into play your inventive faculties, and if you solve it by your own means, you may experience the tension and enjoy the triumph of discovery. Such experiences at a susceptible age may create a taste for mental work and leave their imprint on mind and character for a lifetime."

– George Pólya

The digital general-purpose computer stands out as one of the most significant technological advancements of the past century, marking the beginning of a transformative era that introduced us to the Internet, smartphones, tablets, and widespread computerization. To unlock the full potential of these computers, we rely on *programming*, which involves creating a sequence of instructions, or *code*, that a computer can execute to tackle computational problems. The language used to write this code is known as a *programming language*.

Embedded within this code is the concept of an *algorithm*, which represents the abstract method for solving a problem. The objective of *algorithmic problem-solving* is to develop an effective algorithm that addresses specific computational challenges. While it is not always necessary to write code to appreciate algorithmic problem-solving, engaging in the programming process often deepens understanding and helps in discovering more efficient and straightforward solutions to complex issues.

To get an idea about what we are going to discuss in this book, i.e., algorithmic thinking (algorithmic problem solving), let us go through some common problems you encounter as a student or as a common person.

Problem-1: Suppose that you are back from college, and discover that you have lost your wallet that is of great sentimental value to you. How do you regain your lost wallet?

In our daily life, we encounter problems that are big and small. Some are easy to solve and some are really tough to solve. Some call for a structured solution, whereas some require an unstructured approach. Some are interesting and some are not.

If we consider the nature of problems, they range from mathematical to philosophical. In computing, we deal with *algorithmic problems* whose solutions are expressed as algorithms. An algorithm is a set of well-defined steps and instructions that can be translated into a form, usually known as the code or program, that is executable by a computing device. You will learn more about algorithmic problem-solving in Section 1.3.

For the time being, we shall study problem-solving in general. However, we still want to confine our domain to the logical applications of scientific and mathematical methods. Therefore, we will exclude problems such as “How to become a millionaire in 50 days?”, “How to score full A+’s in examinations?”, and “How many years do computer scientists take to build a ”human-like” robot?”

Returning to the wallet problem, have you found a solution? If you have, then something is really missing. First of all, the problem statement is incomplete. Where did you actually drop the wallet? Did you assume that it was dropped at the college? Could it have happened on the way back home? If the wallet was indeed dropped at the school, where in the school? In the classroom or playground or the library or somewhere else? These questions must be answered before you can reach a complete understanding of the problem. Assumptions should not be made without basis. Irrelevant information (such as the sentimental value of the wallet) should not get in to the way. Incomplete information must be sought out. You could only fully understand the problem after all ambiguities are removed. Then only you should try to solve the problem. Otherwise, you might end up in a situation where you get stuck.

Now, suppose your wallet was lost in the library. How do you recover the wallet?

Problem-2: In a circle, a square with side length $2a$ is inscribed. What is the area of the circle?

This is a well-stated problem in geometry. After you have fully understood the problem, you need to devise a plan for solving it. You need to determine the input (the length of each side of the square) and the output (the area of the circle). You use suitable notation to represent the data, and you examine the relationship between the data and the unknown, which will often lead you to a solution. This may involve the calculation of some intermediate result, like the length of the diagonal of the square. In this case, you need to make use of the domain knowledge of basic geometry.

$n = 0$:					1				
$n = 1$:				1		1			
$n = 2$:			1		2		1		
$n = 3$:		1		3		3		1	
$n = 4$:	1		4		6		4		1

Table 1.1: Pascal's triangle

Problem-3: Suppose that you are to ship a wolf, a sheep, and a cabbage across the river. The boat can only hold the weight of two: you plus another item. You are the only one who can row the boat. The wolf must not be left with the sheep alone, or the wolf will eat the sheep. Neither should the sheep be left alone with the cabbage. How do you get them over to the other side of the river?

This type of problem involves logic. The required solution is not a computed value as in Problem 2, but a series of moves that represent the transition from the initial state (in which all are on one side of the bank) to the final state (where all are on the opposite side).

Problem-4: Let us consider the well-known Pascal's triangle shown in Table 1.1.

There are 1's on the boundaries. For the rest, each value is the sum of the nearest two numbers above it, one on its right and the other on its left. How do you compute the combination $\binom{n}{k}$ using Pascal's triangle? How are the values related to the coefficients in the expansion of $(x + y)^n$ for non-negative n?

Problem-5: Suppose that you are given a rectangular 3×5 grid. You are allowed to move from one intersection to another, governed by the rule that you can travel only upwards or to the right, along the grid lines. How many paths are there from each of the intersections to the top-right corner of the grid?

Are the above two problems related? If so, how is this problem related to the Pascal's triangle? Knowing that Problems 4 and 5 are related, how would you use Pascal's triangle to solve this problem?

This illustrates the often-encountered situation that calls for relating a problem at hand to one that we have solved before. Figuring out the similarity among problems, and performing problem transformation, help us to sharpen our problem-solving ability.

The problems that we discussed offer a quick look into the craft of problem-solving. The Hungarian mathematician George Pólya (1887-1985) conceptualized the process and captured it in the famous book ”How To Solve It”. In his book, Pólya enumerates four phases a problem solver has to go through. First, we need to understand the problem and clarify any doubts about the problem statement if necessary, as we have discussed in Problem 1. Second, we need to find the connection between the data and the unknown, to make out a plan for the solution. Auxiliary problems might be created in the process. Third, we are to carry out our plan, checking the validity of each step. Finally, we have to review the solution.

The four phases are outlined below.

- **Phase 1: Understanding the problem.**
  - *What is the unknown? What are the data?*
  - What is the condition? Is it possible to satisfy the condition? Is the condition sufficient to determine the unknown? Or is it insufficient? Or redundant? Or contradictory?
  - Draw a figure. Introduce suitable notation.
  - Separate the various parts of the condition. Can you write them down?
- **Phase 2: Devising a plan.**
  - Have you seen it before? Or have you seen the *same problem in a slightly different form?*
  - Do you know a related problem?
  - Look at the unknown! Try to think of a familiar problem having the same or similar unknown.
  - Split the problem into smaller, simpler sub-problems.
  - If you cannot solve the proposed problem try to solve first some related problem. Or solve a more general problem. Or a special case of the problem. Or solve a part of the problem.
- **Phase 3: Carrying out the plan.**
  - Carrying out your plan of the solution, check each step.
  - Can you see clearly that the step is correct?
  - Can you prove that it is correct?
- **Phase 4: Looking back.**
  - Can you check the result?
  - Can you derive the result differently?
  - Can you use the result, or the method, for some other problem?

You should note that asking questions is a running theme in all phases of the process. In fact, asking questions is the building block of problem-solving. So, you should start developing the art of asking questions from now on, if you have not done so.

1.2 Problem-Solving Strategies

Problem-solving strategies are essential tools that enable you to effectively tackle a wide range of challenges by providing structured methods to analyze, understand, and resolve problems. These strategies include systematic approaches such as Trial and Error, Heuristics, Means-Ends Analysis, and Backtracking, each offering unique benefits and applications. Understanding multiple problem-solving strategies is crucial as it allows for adaptability and flexibility, ensuring that one can choose the most efficient method for any given situation. This versatility enhances problem-solving efficiency and improves outcomes by offering diverse perspectives and potential solutions. Additionally, employing various strategies enriches cognitive skill development, critical thinking, and creativity. By integrating these strategies into everyday problem-solving, you can approach challenges with confidence and resilience, ultimately achieving more successful and innovative solutions

1.2.1 Importance of Understanding Multiple Problem-Solving Strategies

Understanding multiple problem-solving strategies is crucial because it equips individuals with a diverse toolkit to tackle a variety of challenges. Different problems often require different approaches, and being familiar with multiple strategies allows for greater flexibility and adaptability. For example, some problems might be best solved through a systematic trial and error method, while others might benefit from a more analytical approach like means-ends analysis. By knowing several strategies, one can quickly switch tactics when one method does not work, increasing the chances of finding a successful solution.

Additionally, having a collection of problem-solving strategies enhances critical thinking and creativity. It encourages thinking outside the box and considering multiple perspectives, which can lead to more innovative and effective solutions. This broad understanding also helps in recognizing patterns and similarities between different problems, making it easier to apply previous knowledge to new situations. In both academic and real-world scenarios, this versatility not only improves problem-solving efficiency but also boosts confidence and competence in facing various challenges. Some benefits are as follows

- **Adaptability:** Different problems require different approaches. Understanding multiple strategies allows for flexibility and adaptability in problem-solving.
- **Efficiency:** Some strategies are more effective for specific types of problems. Having a repertoire of strategies can save time and resources.
- **Improved Outcomes:** Diverse strategies offer multiple perspectives and potential solutions, increasing the likelihood of finding optimal solutions.
- **Skill Development:** Exposure to various strategies enhances cognitive skills, critical thinking, and creativity.

Let us look at some commonly used problem-solving strategies

1.2.2 Trial And Error Problem-Solving Strategy

The trial-and-error problem-solving strategy involves attempting different solutions and learning from mistakes until a successful outcome is achieved. It is a fundamental method that relies on experimentation and iteration, rather than systematic or analytical approaches.

Consider the situation where you have forgotten the password to your online account, and there is no password recovery option available. You decide to use trial and error to regain access:

1. **Initial Attempts**: You start by trying passwords you commonly use. For instance, you might first try ”password123,” ”Qwerty2024,” or ”MyDog-Tommy.”
2. **Learning from Mistakes**: None of these initial attempts work. You then recall that you sometimes use a combination of personal information. You try variations incorporating your birthdate, pet’s name, or favorite sports team.
3. **Refinement**: After several failed attempts, you remember you recently started using a new format for your passwords, combining a favorite quote with special characters. You attempted various combinations, such as”ToBeOrNotToBe!”, ”NeeNeeyaayirikkuka#,” and other combinations.
4. **Success**: Eventually, through persistent trial and error, you hit upon the correct password: ”NeeNeeyaayirikkuka#2024.”

In this scenario, trial and error involved systematically trying different potential passwords, learning from each failed attempt, and refining the approach based on what you remember about your password habits. This method is practical when there is no clear pathway to the solution and allows for discovering the correct answer through persistence and adaptability.

1.2.3 Algorithmic Problem-Solving Strategy

An algorithm is a step-by-step, logical procedure that guarantees a solution to a problem. It is systematic and follows a defined sequence of operations, ensuring consistency and accuracy in finding the correct solution.

When baking a cake, you follow a precise recipe, which acts as an algorithm:

1. **Gather Ingredients**: Measure out 2 cups of flour, 1 cup of sugar, 2 eggs, $\frac{1}{2}$ cup of butter, 1 teaspoon of baking powder, and 1 cup of milk.
2. **Preheat Oven**: Set the oven to 175°C.
3. **Mix Ingredients**: In a bowl, combine the flour, baking powder, and sugar. In another bowl, beat the eggs and then mix in the butter and milk. Gradually combine the wet and dry ingredients, stirring until smooth.

4. **Prepare Baking Pan**: Grease a baking pan with butter or cooking spray.

5. **Pour Batter**: Pour the batter into the prepared pan.

6. **Bake**: Place the pan in the preheated oven and bake for 30-35 minutes.

7. **Check for Doneness**: Insert a toothpick into the center of the cake. If it comes out clean, the cake is done.

8. **Cool and Serve**: Let the cake cool before serving.

By following this algorithm (the recipe), you systematically achieve the desired result — a perfectly baked cake.

1.2.4 Heuristic Problem-Solving Strategy

A heuristic is a practical approach to problem-solving based on experience and intuition. It does not guarantee a perfect solution but provides a good enough solution quickly, often through rules of thumb or educated guesses. When driving in a city with frequent traffic congestion, you might use a heuristic approach to find the fastest route to your destination:

1. **Rule of Thumb**: You know from experience that certain streets are typically less congested during rush hour.

2. **Current Conditions**: You use a traffic app to check current traffic conditions, looking for red or yellow indicators on major roads.

3. **Alternative Routes**: You consider side streets and shortcuts you have used before that tend to be less busy.

4. **Decision**: Based on the app and your knowledge, you decide to avoid the main highway (which shows heavy congestion) and take a series of back roads that usually have lighter traffic.

While this heuristic approach does not guarantee that you will find the absolute fastest route, it combines your experience and real-time data to make an informed, efficient decision, likely saving you time compared to blindly following the main routes.

1.2.5 Means-Ends Analysis Problem-Solving Strategy

Means-ends analysis is a strategy that involves breaking down a problem into smaller, manageable parts (means) and addressing each part to achieve the final goal (ends). It involves identifying the current state, the desired end state, and the steps needed to bridge the gap between the two.

Imagine you want to plan a road trip from Trivandrum to Kashmir. Here is how you might use means-ends analysis:

1. **Define the Goal**: Your ultimate goal is to drive from Trivandrum to Kashmir.
2. **Analyze the Current State**: You start in Trivandrum with your car ready to go.
3. **Identify the Differences**: The primary difference is the distance between Trivandrum and Kashmir, which is approximately 3,700 kilometers.
4. **Set Sub-Goals (Means)**:
   - **Fuel and Rest Stops**: Determine where you will need to stop for fuel and rest.
   - **Daily Driving Targets**: Break the trip into daily segments, such as driving 500-600 kilometers per day.
   - **Route Planning**: Choose the most efficient and scenic route, considering highways, weather conditions, and places you want to visit.
5. **Implement the Plan**:
   - **Day 1**: Drive from Trivandrum to Bangalore, Karnataka (approx. 720 km). Refuel in Madurai, Tamil Nadu. Overnight stay in Bangalore.
   - **Day 2**: Drive from Bangalore to Hyderabad, Telangana (approx. 570 km). Refuel in Anantapur, Andhra Pradesh. Overnight stay in Hyderabad.
   - **Day 3**: Drive from Hyderabad to Nagpur, Maharashtra (approx. 500 km). Refuel in Adilabad, Telangana. Overnight stay in Nagpur.
   - **Day 4**: Drive from Nagpur to Jhansi, Uttar Pradesh (approx. 580 km). Refuel in Sagar, Madhya Pradesh. Overnight stay in Jhansi.
   - **Day 5**: Drive from Jhansi to Agra, Uttar Pradesh (approx. 290 km). Refuel in Gwalior, Madhya Pradesh. Overnight stay in Agra. Visit the Taj Mahal.
   - **Day 6**: Drive from Agra to Chandigarh (approx. 450 km). Refuel in Karnal, Haryana. Overnight stay in Chandigarh.
   - **Day 7**: Drive from Chandigarh to Jammu (approx. 350 km). Refuel in Pathankot, Punjab. Overnight stay in Jammu.
   - **Day 8**: Drive from Jammu to Srinagar, Kashmir (approx. 270 km).
6. **Adjust as Needed**: Throughout the trip, you may need to make adjustments based on traffic, road conditions, or personal preferences.

By breaking down the long journey into smaller, achievable segments and addressing each part systematically, you can effectively plan and complete the road trip. This method ensures that you stay on track and make steady progress toward your final destination, despite the complexity and distance of the trip.

1.2.6 Problem decomposition

In the previous example of traveling from Trivandrum to Kashmir, breaking down the journey into smaller segments helped you come up with an effective plan for the completion of the trip. This is the key to solving complex problems. When the problem to be solved is too complex to manage, break it into manageable parts known as sub-problems. This process is known as **problem decomposition**. Here are the steps in solving a problem using the decomposition approach:

1. **Understand the problem**: Develop a thorough understanding of the problem.
2. **Identify the sub-problems**: Decompose the problems into smaller parts.
3. **Solving the sub-problems**: Once decomposition is done, you proceed to solve the individual sub-problems. You may have to decide upon the order in which the various sub-problems are to be solved.
4. **Combine the solution**: Once all the sub-problems have been solved, you should combine all those solutions to form the solution for the original problem.
5. **Test the combined solution**: Finally you ensure that the combined solution indeed solves the problem effectively.

1.2.7 Other Problem-solving Strategies

Here are some examples of problem-solving strategies that may equally be adopted to see which works best for you in different situations:

i. **Brainstorming**

   Brainstorming involves generating a wide range of ideas and solutions to a problem without immediately judging or analyzing them. The goal is to encourage creative thinking and explore various possibilities.

   In a team meeting to improve customer satisfaction, everyone contributes different ideas, such as enhancing product quality, improving customer service training, offering loyalty programs, and using customer feedback to make improvements. These ideas are later evaluated and the best ones are implemented.

ii. **Lateral Thinking**

   Lateral thinking is about looking at problems from new and unconventional angles. It involves thinking outside the box and challenging established patterns and assumptions.

   A company facing declining sales of a product might use lateral thinking to identify new uses for the product or new markets to target, rather than just trying to improve the existing product or marketing strategy.

iii. **Root Cause Analysis**

Root cause analysis involves identifying the fundamental cause of a problem rather than just addressing its symptoms. The goal is to prevent the problem from recurring by solving its underlying issues.

If a factory's production line frequently breaks down, rather than just repairing the machinery each time, the team conducts a root cause analysis and discovers that poor maintenance scheduling is the underlying issue. By implementing a better maintenance plan, they reduce the breakdowns.

iv. **Mind Mapping**

Mind mapping is a visual tool for organizing information. It helps in brainstorming, understanding, and solving problems by visually connecting ideas and concepts.

When planning a large event, an organizer creates a mind map with the event at the center, branching out into categories like venue, catering, entertainment, invitations, and logistics. Each category further branches into specific tasks and considerations.

v. **SWOT Analysis**

SWOT analysis involves evaluating the **S**trengths, **W**eaknesses, **O**pportunities, and **T**hreats related to a particular problem or decision. It helps in understanding both internal and external factors that impact the situation.

A business firm considering a new product launch performs a SWOT analysis. They identify their strengths (strong brand, good distribution network), weaknesses (limited R&D budget), opportunities (market demand, potential partnerships), and threats (competition, economic downturn). This analysis guides their decision-making process.

vi. **Decision Matrix**

A decision matrix, also known as a decision grid or Pugh matrix, helps in evaluating and prioritizing a list of options. It involves listing options and criteria, assigning weights to each criterion, and scoring each option based on the criteria.

A family deciding on a new car creates a decision matrix with criteria such as cost, fuel efficiency, safety features, and brand reputation. They rate each car option against these criteria and calculate a total score to make an informed choice.

vii. **Simulation**

Simulation involves creating a model of a real-world system and experimenting with it to understand how the system behaves under different conditions. It helps in predicting outcomes and identifying the best course of action.

Urban planners use traffic simulation software to model the impact of new road constructions on traffic flow. By testing different scenarios, they can design the most effective road network to reduce congestion.

viii. **Use Experience**

The use of experience as a problem-solving strategy involves drawing on previous knowledge and experiences to address current challenges. This strategy relies on the idea that similar problems often have similar solutions, and leveraging past experiences can lead to efficient and effective outcomes.

A company trying to market a new clothing line may consider marketing tactics they have previously used, such as magazine advertisements, influencer campaigns, or social media advertisements. By analyzing what tactics have worked in the past, they can create a successful marketing campaign again.

These strategies provide various approaches to problem-solving, each suitable for different types of challenges and contexts.

1.3 Algorithmic problem solving with computers

In today's digital era, the computer has become an indispensable part of our life. Down from performing simple arithmetic right up to accurately determining the position for a soft lunar landing, we rely heavily on computers and allied digital devices. Computers are potent tools for solving problems across diverse disciplines. Problem-solving is a systematic way to arrive at solutions for a given problem.

1.3.1 The Problem solving process

Let us now explore how computers can be put to solving problems:

1. **Understand the problem**: Effective problem-solving demands a thorough knowledge of the problem domain. Once you have identified the problem, its exact nature must be sought and defined. The problem context, objectives, and constraints if any are to be understood properly. Several techniques can be used to gather information about a problem. Some of these include conducting interviews and sending questionnaires to the stakeholders (people who are concerned with the problem). Segmenting a big problem into simple manageable ones often helps you to develop a clear picture of the problem.

2. **Formulate a model for the solution**: After the problem is thoroughly understood, the next step is to devise a solution. You should now identify

the various ways to solve the problem. Brainstorming is one of the most commonly used techniques for generating a large number of ideas within a short time. Brainwriting and Mind mapping are two alternative techniques that you can employ here. The generated ideas are then transformed into a conceptual model that can be easily converted to a solution. Mathematical modeling and simulation modeling are two popular modeling techniques that you could adopt. Whatever the modeling technique is, ensure the defined model accurately reflects the conceived ideas.

3. **Develop an algorithm**: Once a list of possible solutions is determined, they have to be translated into formal representations – *algorithms*. Obviously, you do not implement all the solutions. So the next step is to assess the pros and cons of each algorithm to select the best one for the problem. The assessment is based on considering various factors such as memory, time, and lines of code.

4. **Code the algorithm**: The interesting part of the process! Coding! After the *best algorithm* is determined, you implement it as an executable program. The program or the code is a set of instructions that is more or less, a concrete representation of the algorithm in some programming language. While coding, always follow the incremental paradigm – start with the essential functionalities and gradually add more and more to it.

5. **Test the program**: Nobody is perfect! Once you are done with the coding, you have to inspect your code to verify its correctness. This is formally called *testing*. During testing, the program is evaluated as to whether it produces the desired output. Any unexpected output is an *error*. The program should be executed with different sets of inputs to detect errors. It is impossible to test the program with all possible inputs. Instead, a smaller set of representative inputs called *test suite* is identified and if the program runs correctly on the test suite, then it is concluded that the program will probably be correct for all inputs. You can get the help of automated testing tools to generate a test suite for your code. Closely associated with testing is the process of *debugging* which involves fixing or resolving the errors (technically called bugs) identified during testing. Testing and debugging should be repeated until all errors are fixed.

6. **Evaluate the solution**: This final step is crucial to ensure that the program effectively addresses the problem and attains the desired objectives. You have to first define the evaluation criteria. These could include metrics like efficiency, feasibility, and scalability, a few to mention. The potential risks that could arise with the program's deployment are also to be assessed. Collect quantitative and qualitative feedback from the stakeholders. Based on the feedback, you have to work on making necessary improvements to the program. Nevertheless, the refined code should also be subject to rigorous testing.

1.3.2 A case study - The Discriminant calculator

Tell me and I forget. Teach me and I remember. Involve me and I learn. - Benjamin Franklin

How about going around the problem-solving process a second time? But this time, with a real problem at hand – determining the discriminant of a quadratic equation. Roll up your sleeves!

1. **Understand the problem**: Here we formally define the problem by specifying the inputs and output.

   ***Input*:** The three coefficients a, b and c of the quadratic equation

   ***Output*:** The discriminant value D for the quadratic equation

2. **Formulate a model for the solution**: Develop a mathematical model for the solution, that is identify the mathematical expression for the quadratic equation discriminant D:

   $$D = b^2 - 4ac$$

3. **Develop an algorithm**: A possible algorithm (actually, a pseudocode) for our discriminant problem is given below:

   ```
   1  Start
   2  READ($a, b, c$)
   3  $d = b * b - 4 * a * c$
   4  PRINT($d$)
   5  Stop
   ```

   You will learn more about pseudocodes in Chapter 2.

4. **Code the algorithm**: The Python program to calculate the discriminant is as follows:

   ```
   #Input the coefficients
   a = int(input("Enter the value of first coefficient"))
   b = int(input("Enter the value of second coefficient"))
   c = int(input("Enter the value of third coefficient"))
   #Find the discriminant
   d = (b**2) - (4*a*c)
   #Print the discriminant
   print(d)
   ```

   Completely puzzled about the code? Don't worry! We will start with Python programming in Chapter 4.

5. **Test the program**: You create a test suite similar to the one shown in Table 1.2. Each row denotes a set of inputs (a, b, and c) and the expected output (d) with which the actual output is to be compared.

Table 1.2: A test suite for the discriminant calculator

Sl. No.	a	b	c	d
1	10	2	5	-196
2	5	7	1	29
3	2	4	2	0
4	1	1	1	-3
5	3	2	5	-56
6	2	8	2	48

1.4 Conclusion

In this chapter, we've explored the diverse landscape of problem-solving, highlighting the importance of understanding and applying various strategies. By delving into methods like trial and error, heuristics, and means-ends analysis, we've gained insight into how different approaches can be leveraged depending on the nature of the problem at hand. The discussion on algorithmic problem-solving has further illustrated how structured processes can be applied, particularly when using computers to tackle complex issues.

The case study on the Discriminant calculator effectively bridges theory with practice, showing that these problem-solving strategies are not just abstract concepts but have real-world applications. This chapter provides a foundational toolkit for tackling a variety of challenges and enhancing your confidence, creativity, and precision in both computational and practical scenarios.

1.5 Exercises

1. A bear, starting from the point P, walked one mile due south. Then he changed direction and walked one mile due east. Then he turned again to the left and walked one mile due north, and arrived at the point P he started from. What was the color of the bear?

2. Two towns A and B are 3 kilometers apart. It is proposed to build a new school serving 100 students in town A and 50 students in town B. How far from town A should the school be built if the total distance travelled by all 150 students is to be as small as possible?

3. A traveller arrives at an inn. He has no money but only a silver chain consisting of 6 links. He uses one link to pay for each day spent at the inn, but the innkeeper agrees to accept no more than one broken link.

How should the traveller cut up the chain in order to settle accounts with the innkeeper on a daily basis?

4. What is the least number of links that have to be cut if the traveller stays 100 days at the inn and has a chain consisting of 100 links? What is the answer in the general case (n days and n links)?

5. The minute and hour hands of a clock coincide exactly at 12 o'clock. At what time later do they first coincide again?

6. Six glasses are in a row, the first three full of juice, the second three empty. By moving only one glass, can you arrange them so that empty and full glasses alternate?

7. You throw away the outside and cook the inside. Then you eat the outside and throw away the inside. What did you eat?

8. Rearrange the letters in the words new door to make one word.

9. A mad scientist wishes to make a chain out of plutonium and lead pieces. There is a problem, however. If the scientist places two pieces of plutonium next to each other, BOOM! The question is, in how many ways can the scientist safely construct a chain of length n?

10. Among 12 ball bearings, one is defective, but it is not known if it is heavier or lighter than the rest. Using a traditional balance (with two pans hanging down the opposite ends of a lever supported in the middle), how do you determine which is the defective ball bearing, and whether it is heavier or lighter than the others, within three attempts?

Bibliography

[1] George Pólya and John Conway, How to Solve It: A New Aspect of Mathematical Method, Princeton University Press, 2014.

[2] Maureen Sprankle and Jim Hubbard, Problem Solving and Programming Concepts, Pearson, 2011.

[3] Donald E. Knuth, The Art of Computer Programming (Volume 1), Addison-Wesley, 1997.

[4] Nell Dale and John Lewis, Computer Science Illuminated, Jones and Bartlett Publishers, 2019.

[5] R G Dromey, How To Solve It By Computer, Pearson, 2008.

Chapter 2

Algorithms, pseudocodes, and flowcharts

"When you choose an algorithm, you choose a point of view"
– Anonymous

"An Algorithm is a storytellers' script while Pseudocode is the builders' blueprint"
– Anonymous

2.1 Algorithms and pseudocodes

An **algorithm** describes a systematic way of solving a problem. It is a step-by-step procedure that produces an output when given the necessary inputs. An algorithm uses pure English phrases or sentences to describe the solution to a problem. A **Pseudocode** is a high-level representation of an algorithm that uses a mixture of natural language and programming language-like syntax. It is more structured than an algorithm in that it uses mathematical expressions with English phrases to capture the essence of a solution concisely. You can also use programming constructs (See Section 2.1.2 below) in a pseudocode which are not permitted in an algorithm in a strict sense.

As the name indicates, pseudocode is not a true program and thus is independent of any programming language. It is not executable rather helps you understand the flow of an algorithm.

Confused between algorithms and pseudocodes? Let us take an example. We will now write an algorithm and a pseudocode to evaluate an expression, say $d = a + b * c$. Here $a, b, c,$ and d are known as *variables*. Simply put, a variable is a name given to a memory location that stores some data value. First, let us look at the algorithm for the expression evaluation.

```
Evaluate-Algo
1  Start
2  Read the values of $a, b$ and $c$.
3  Find the product of $b$ and $c$.
4  Store the product in a temporary variable $temp$.
5  Find the sum of $a$ and $temp$.
6  Store the sum in $d$.
7  Print the value of $d$.
8  Stop.
```

The following is the pseudocode to evaluate the same expression.

```
Evaluate-Pseudo
1  Start
2  Read($a, b, c$)
3  $d = a + b * c$
4  Print($d$)
5  Stop
```

In a pseudocode, Read is used to read input values. Print is used to print a message. The message to be printed should be enclosed in a pair of double quotes. For example,

Print(“Hello folks!!”)

prints

```
Hello folks!!
```

To print the value of a variable, just use the variable name (without quotes). In the above example, Print(d) displays the value of the variable d.

Although pseudocode and algorithm are technically different, these words are interchangeably used for convenience.

2.1.1 Why pseudocodes?

Wondering why pseudocodes are important? Here are a few motivating reasons:

1. **Ease of understanding**: Since the pseudocode is programming language independent, novice developers can also understand it very easily.

2. **Focus on logic**: A pseudocode allows you to focus on the algorithm's logic without bothering about the syntax of a specific programming language.

3. **More legible**: Combining programming constructs with English phrases makes pseudocode more legible and conveys the logic precisely.

4. **Consistent**: As the constructs used in pseudocode are standardized, it is useful in sharing ideas among developers from various domains.

Table 2.1: Relational operators

Operator	**Meaning**
$>$	greater than
$<$	less than
$==$	equal to
$>=$	greater than or equal to
$<=$	less than or equal to
$!=$	not equal to

5. **Easy translation to a program**: Using programming constructs makes mapping the pseudocode to a program straightforward.

6. **Identification of flaws**: A pseudocode helps identify flaws in the solution logic before implementation.

2.1.2 Constructs of a pseudocode

A good pseudocode should follow the structured programming approach. Structured coding aims to improve the readability of pseudocode by ensuring that the execution sequence follows the order in which the code is written. Such a code is said to have a *linear flow of control*. Sequencing, selection, and repetition (loop) are three programming constructs that allow for linear control flow. These are also known as *single entry – single exit* constructs.

☞ When it is said that "the pseudocode is executed", it just means that the pseudocode instructions are interpreted. It doesn't denote the actual execution on a computer.

In the sequence structure, all instructions in the pseudocode are executed (exactly) once without skipping any. On the other hand, with selection and loop structures, it is possible to execute certain instructions repeatedly or even skip some. In such structures, the decision as to which statements are to be executed or whether the execution should repeat will be determined based on the outcome of testing a condition. We use special symbols called *relational operators* to write such conditions. The various relational operators are listed in Table 2.1. It is also possible to combine two or more conditions using *logical operators* like "**AND**" (&&), "**OR**" (||). Eg: $a > b$ **AND** $a > c$.

2.1.2.1 Sequence

This is the most elementary construct where the instructions of the algorithm are executed in the order listed. It is the logical equivalent of a straight line. Consider the code below.

```
S1
S2
S3
.
.
Sn
```

The statement `S1` is executed first, which is then followed by statement `S2`, so on and so forth, `Sn` until all the instructions are executed. No instruction is skipped and every instruction is executed only once.

2.1.2.2 Decision or Selection

A selection structure consists of a test condition together with one or more blocks of statements. The result of the test determines which of these blocks is executed. There are mainly two types of selection structures, as discussed below:

A if structure

There are three variations of the if-structure:

A.1 if structure

The general form of this structure is:

```
if (condition)
    TRUE__INSTRUCTIONS
endif
```

If the test condition is evaluated to TRUE, the statements denoted by TRUE__INSTRUCTIONS are executed. Otherwise, those statements are skipped.

Example 2.1. The pseudocode CHECKPOSITIVE(x) checks if an input value x is positive.

```
CHECKPOSITIVE(x)
1  if (x > 0)
2      PRINT(x,“ is positive”)
3  endif
```

A.2 if else structure

The general form is given below:

```
if (condition)
    TRUE__INSTRUCTIONS
else
    FALSE__INSTRUCTIONS
endif
```

This structure contains two blocks of statements. If the test condition is met, the first block (denoted by TRUE__INSTRUCTIONS) is executed and the algorithm skips over the second block (denoted by FALSE__INSTRUCTIONS). If the test condition is not met, the first block is skipped and only the second block is executed.

Example 2.2. The pseudocode PersonType(*age*) checks if a person is a major or not.

```
PersonType(age)
1  if (age >= 18)
2      Print("You are a major")
3  else
4      Print("You are a minor")
5  endif
```

A.3 if else if else structure

When a selection is to be made out of a set of more than two possibilities, you need to use the *if else if else structure*, whose general form is given below:

```
if (condition_1)
    TRUE__INSTRUCTIONS_1
else if (condition_2)
    TRUE__INSTRUCTIONS_2
else
    FALSE__INSTRUCTIONS
endif
```

Here, if $condition_1$ is met, TRUE__INSTRUCTIONS$_1$ will be executed. Else $condition_2$ is checked. If it evaluates to True, TRUE__INSTRUCTIONS$_2$ will be selected. Otherwise FALSE__INSTRUCTIONS will be executed.

Example 2.3. The pseudocode CompareVars(x, y) compares two variables x and y and prints the relation between them.

```
CompareVars($x, y$)
if ($x > y$)
    Print($x$,“is greater than”,$y$)
else if ($x < y$)
    Print($y$,“is greater than”,$x$)
else
    Print(“The two values are equal”)
endif
```

There is no limit to the number of **else if** statements, but in the end, there has to be an **else** statement. The conditions are tested one by one starting from the top, proceeding downwards. Once a condition is evaluated to be True, the corresponding block is executed, and the rest of the structure is skipped. If none of the conditions are met, the final **else** part is executed.

B Case Structure

The **case** structure is a refined alternative to **if else if else** structure. The pseudocode representation of the **case** structure is given below.

The general form of this structure is:

```
caseof (expression)
case 1 $value_1$:
    BLOCK$_1$
case 2 $value_2$:
    BLOCK$_2$
    ⋮
default :
    DEFAULT__BLOCK
endcase
```

The case structure works like this: First, the value of *expression* (you can also have a single variable in the place of *expression*) is compared with $value_1$. If there is a match, the first block of statements denoted as BLOCK$_1$ will be executed. Typically, each block will have a `break` at the end which causes the case structure to be exited.

If there is no match, the value of the expression (or of the variable) is compared with $value_2$. If there is a match here, BLOCK$_2$ is executed and the structure is exited at the corresponding `break` statement. This process continues until either a match for the expression value is found or until the end of the cases is encountered. The DEFAULT__BLOCK will be executed when the expression does not match any of the cases.

If the `break` statement is omitted from the block for the matching case, then the execution continues into subsequent blocks even if there is no match in the subsequent blocks, until either a `break` is encountered or the end of the case structure is reached.

Example 2.4. The pseudocode PrintDirection(*dir*) prints the direction name based on the value of a character called *dir*.

```
PrintDirection(dir)
caseof (dir)
    case 'N':
        Print("North")
        break
    case 'S':
        Print("South")
        break
    case 'E':
        Print("East")
        break
    case 'W':
        Print("West")
        break
    default :
        Print("Invalid direction code")
endcase
```

2.1.2.3 Repetition or loop

When a certain block of instructions is to be repeatedly executed, we use the repetition or loop construct. Each execution of the block is called an **iteration** or a **pass**. If the number of iterations (how many times the block is to be executed) is known in advance, it is called **definite iteration**. Otherwise, it is called **indefinite** or **conditional iteration**. The block that is repeatedly executed is called the *loop body*. There are three types of loop constructs as discussed below:

A while loop

A **while** loop is generally used to implement indefinite iteration. The general form of the **while** loop is as follows:

```
while (condition)
    TRUE_INSTRUCTIONS
endwhile
```

Here, the loop body (TRUE_INSTRUCTIONS) is executed repeatedly as long as *condition* evaluates to True. When the condition is evaluated as False, the loop body is bypassed.

B repeat-until loop

The second type of loop structure is the **repeat-until** structure. This type of loop is also used for indefinite iteration. Here the set of instructions constituting the loop body is repeated as long as *condition* evaluates to FALSE. When the condition evaluates to TRUE, the loop is exited. The pseudocode form of **repeat-until** loop is shown below.

```
repeat
    FALSE__INSTRUCTIONS
until (condition)
```

There are two major differences between **while** and **repeat-until** loop constructs:

1. In the **while** loop, the pseudocode continues to execute as long as the resultant of the condition is TRUE; in the **repeat-until** loop, the looping process stops when the resultant of the condition becomes TRUE.
2. In the **while** loop, the condition is tested at the beginning; in the **repeat-until** loop, the condition is tested at the end. For this reason, the while loop is known as an *entry controlled loop* and the repeat-until loop is known as an *exit controlled loop.*

You should note that when the condition is tested at the end, the instructions in the loop are executed at least once.

C for loop

The **for** loop implements definite iteration. There are three variants of the **for** loop. All three **for** loop constructs use a variable (call it the *loop variable*) as a counter that starts counting from a specific value called *begin* and updates the loop variable after each iteration. The loop body repeats execution until the loop variable value reaches *end*. The first **for** loop variant can be written in pseudocode notation as follows:

```
for var = begin to end
    LOOP__INSTRUCTIONS
endfor
```

Here, the loop variable (*var*) is first assigned (initialized with) the value *begin*. Then the condition *var* <= *end* is tested. If the outcome is TRUE, the loop body is executed. After the first iteration, the loop variable is incremented (increased by 1). The condition *var* <= *end* is again tested with the updated value of *var* and the loop is entered (loop body is executed), if the condition evaluates to TRUE. This process of updating the loop variable after an iteration and proceeding with the execution if the condition (tested with the updated value of the loop variable) evaluates to TRUE continues until the counter value becomes greater than *end*. At that time, the condition evaluates to FALSE and the loop execution stops.

In the second **for** loop variant, whose pseudocode syntax is given below, the loop variable is decremented (decreased by 1) after every iteration. And the condition being tested is *var* >= *end*. Here, *begin* should be greater than or equal to *end*, and the loop exits when this condition is violated.

```
for var = begin downto end
    LOOP__INSTRUCTIONS
endfor
```

It is also possible to update the loop variable by an amount other than 1 after every iteration. The value by which the loop variable is increased or decreased is known as *step*. In the pseudocode shown below, the *step* value is specified using the keyword **by** .

```
for var = begin to end by step
    LOOP__INSTRUCTIONS
endfor
```

Table 2.2 lists some examples of **for** loops. In these examples, *var* is the loop variable.

Table 2.2: **for** loop examples

Loop construct	**Description**	**Values taken by** *var*
for *var* = 1 **to** 10	*var* gets incremented by 1 till it reaches 10	$1, 2, \cdots 9, 10$
for *var* = 10 **downto** 1	*var* gets decremented by 1 till it reaches 1	$10, 9, \cdots 2, 1$
for *var* = 2 **to** 20 **by** 2	*var* gets increased by 2 till it reaches 20	$2, 4, \cdots 18, 20$
for *var* = 20 **downto** 2 **by** 2	*var* gets decreased by 2 till it reaches 2	$20, 18, \cdots 4, 2$

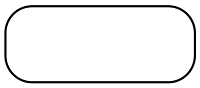
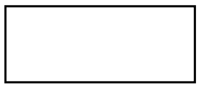
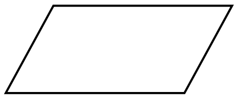
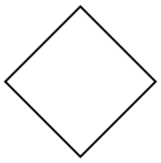
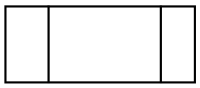
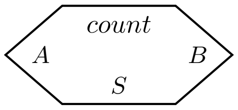
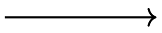
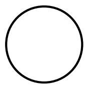
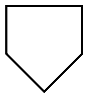
2.2 Flowcharts

A flowchart is a diagrammatic representation of an algorithm that depicts how control flows in it. Flowcharts are composed of various *blocks* interconnected by *flow-lines*. Each block in a flowchart represents some stage of processing in the algorithm. Different types of blocks are defined to represent the various programming constructs of the algorithm.

Flow lines indicate the order in which the algorithm steps are executed. The flow lines entering a block denote data (or control) flow into the block and the flow lines emerging from a block denote data (control) outflow. Most blocks have only single incoming and outgoing flow lines. The exception is for blocks representing selection and loop constructs. Such blocks have multiple exits, one for each possible outcome of the condition being tested and each such outcome is called a *branch*.

Table 2.3 lists some commonly used flowchart symbols and their descriptions.

Table 2.3: The different flowchart symbols

Flowchart symbol	**Description**
	Flattened ellipse indicates the start and end of a module.
	Rectangle is used to show arithmetic calculations.
	Parallelogram denotes an input/output operation.
	Diamond indicates a decision box with a condition to test. It has two exits. One exit leads to a block specifying the actions to be taken when the tested condition is TRUE and the other exit leads to a second block specifying the actions for FALSE case.
	Rectangle with vertical side-lines denotes a module. A module is a collection of statements written to achieve a task. It is known by the name *function* in the programming domain.
	Hexagon denotes a **for** loop. The symbol shown here is the representation of the loop: **for** *count* = *A* **to** *B* **by** *S*.
	Flowlines are indicated by arrows to show the direction of data flow. Each flowline connects two blocks.
	This indicates an on-page connector. This is used when one part of a long flowchart is drawn on one column of a page and the other part in the other column of the same page.
	This indicates an off-page connector. This is used when the flowchart is very long and spans multiple pages.

2.3 Solved problems - Algorithms and Flowcharts

Problem 2.1 To find simple interest.

Solution:

See Figure 2.1 for the algorithm and flowchart.

```
SimpleInterest
1  Start
2  Read($principal, rate, years$)
3  $SI = (principal * rate * years)/100$
4  Print($SI$)
5  Stop.
```

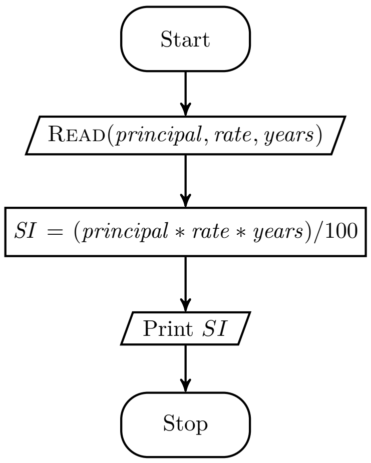

Start
Read(principal, rate, years)
SI = (principal * rate * years)/100
Print SI
Stop

Figure 2.1: To find simple interest

- Read the principal amount, interest rate, and period values. Store them as three variables: *principal*, *rate*, and *years* respectively.
- Multiply *principal*, *rate* and *years* and divide the result by 100 to obtain the simple interest, which is stored in the variable *SI*.
  - Division by 100 is necessary since *rate* is input as a percentage(7% instead of 0.07).
- Finally, the value of *SI* is displayed on the terminal.

Problem 2.2 To determine the larger of two numbers.

Solution:

See Figure 2.2 for the algorithm and flowchart.

```
LargerTwo
1 Start
2 Read($a, b$)
3 if ($a > b$)
4         $large = a$
5 else
6         $large = b$
7 endif
8 Print($large$)
9 Stop.
```

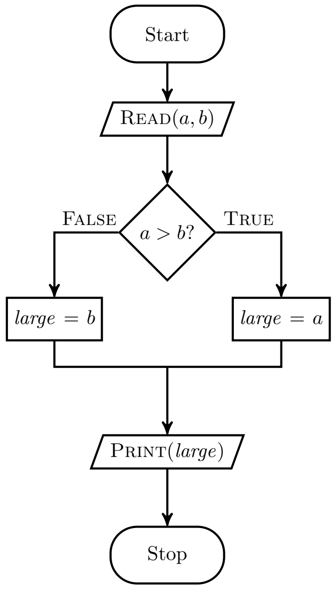


Figure 2.2: To find the larger of two numbers

- First, read two numbers from the user and store them in the variables a and b respectively.
- Next, compare these two numbers using an if-else statement.
- Check whether a is greater than b.
  - If this condition is True, assign the value of a to the variable *large*.
- Otherwise, the control moves to the else part, where the value of b is assigned to *large*.
- Finally, the value of *large* is printed, which is the largest of the two numbers.

Problem 2.3 To determine the smallest of three numbers.

Solution:

See Figure 2.3 for the algorithm and flowchart.

```
SMALLESTTHREE
1  Start
2  READ($a, b, c$)
3  if ($a < b$)
4        $small = a$
5  else
6        $small = b$
7  endif
8  if ($c < small$)
9        $small = c$
10 endif
11 PRINT($small$)
12 Stop.
```

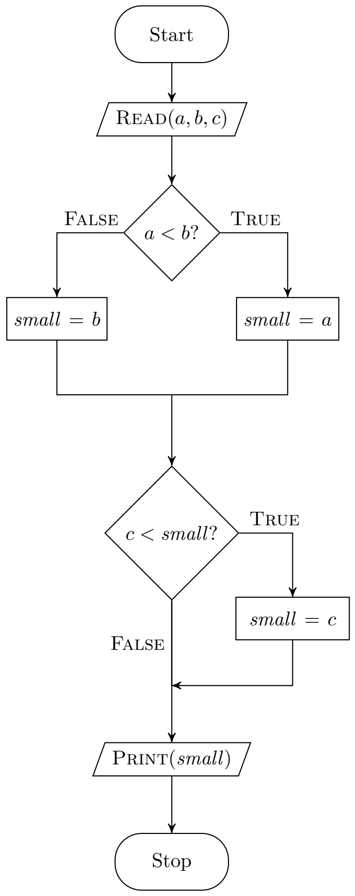


Figure 2.3: To find the smallest of three numbers

- Similar to the previous problem, input three numbers and store them in variables a, b, and c respectively.
- To solve this problem, first find the smaller of the two numbers a and b and then compare that smaller number with the third variable c.
- The first if statement determines the smaller between a and b and keeps it in *small*.
- Using a second if statement, check whether c is less than *small*.
  - If so, the value of c is assigned to *small*.
- Finally, *small* is printed.

Problem 2.4 To determine the entry-ticket fare in a zoo based on age as follows:

Age	Fare
< 10	7
$>= 10$ and < 60	10
$>= 60$	5

Solution:

See Figure 2.4 for the pseudocode and flowchart.

```
TICKETFARE
1   Start
2   READ(age)
3   if (age < 10)
4         fare = 7
5   else if (age < 60)
6         fare = 10
7   else
8         fare = 5
9   endif
10  PRINT(fare)
11  Stop
```

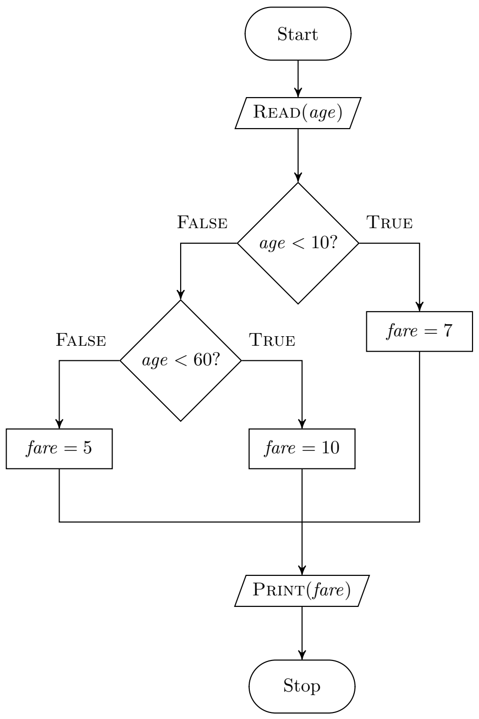


Figure 2.4: To determine the entry fare in a zoo

- Accept *age* from the user.
- First check whether *age* is less than 10. If so, *fare* is assigned the value 7.
- If the condition is FALSE, the next condition is checked. If *age* $>=$ 10 and *age* $<$ 60, *fare* gets the value 10.
- If the above condition also turns out to be FALSE (i.e. *age* $>=$ 60), else statement is executed, and *fare* gets the value 5.
- Finally, the *fare* value is printed.

Problem 2.5 To print the colour based on a code value as follows:

Grade	Message
R	Red
G	Green
B	Blue
Any other value	Wrong code

Solution:

Figure 2.5 for the pseudocode and flowchart.

```
PrintColour
Start
Read(code)
caseof (code)
    case 'R':
        Print("Red")
        break
    case 'G':
        Print("Green")
        break
    case 'B':
        Print("Blue")
        break
    default :
        Print("Wrong code")
endcase
Stop
```

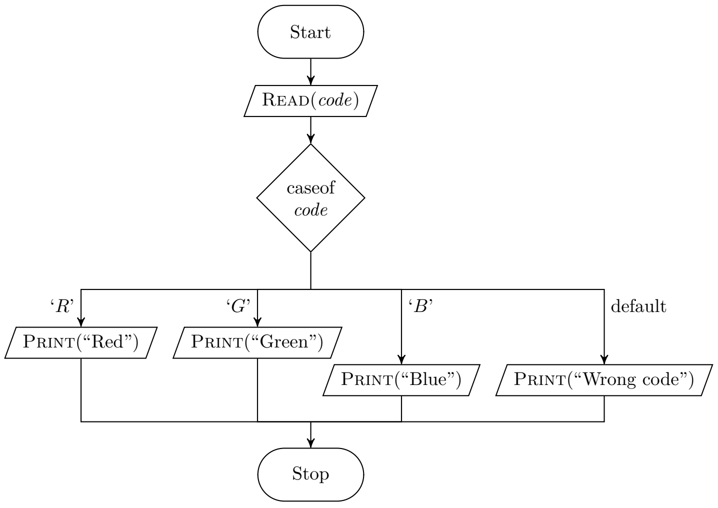


Figure 2.5: To print colors based on a code value

- Read the *code* (a character constant) from the user.
- The value of *code* is matched against a number of case constants (*R*, *G*, *B* as per the question).
- If the *code* is '*R*', the statements associated with it are executed until the **break** statement is encountered (i.e. "Red" is printed here). A **break** statement moves the control out of the case structure.
- A similar execution happens if the value of *code* is *G* or *B*. If *code* is '*G*', "Green" is printed, and "Blue" is printed if *code* is '*B*'.
- If the user inputs any other character other than '*R*', '*G*', or '*B*', the default statement is executed.

Problem 2.6 To print the numbers from 1 to 50 in descending order.

Solution:

See Figure 2.6 for pseudocode and flowchart.

```
PrintDown
1 Start
2 for count = 50 downto 1
3         Print(count)
4 endfor
5 Stop
```

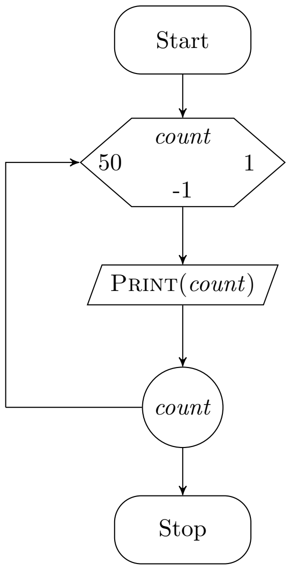


Figure 2.6: To print numbers in descending order

- The *count* variable is initially assigned 50 and the condition, *count* >= 1 (i.e. 50 >= 1) is checked.
- Since it evaluates to TRUE, print statement is executed (Body of for loop in this question).
- '**downto**' decrements the value of *count* by 1 i.e. *count* now becomes 49.

- The condition, *count* $>= 1$ is checked, and since it evaluates to TRUE, the body of the loop is executed (49 is printed), and again *count* is decremented.
- The above step repeats until *count* becomes 0 (in which case, the condition evaluates to FALSE) and you stop. The sequence thus printed is 50 49 48 $\cdots$ 1.

Problem 2.7 To find the factorial of a number.

Solution: The factorial of a number n is defined as $n! = n \times n - 1 \times \cdots \cdots \times 2 \times 1$.

See Figure 2.7 for the pseudocode and flowchart.

```
FACTORIAL
1  Start
2  READ(n)
3  fact = 1
4  for var = n downto 1
5          fact = fact * var
6  endfor
7  PRINT(fact)
8  Stop
```

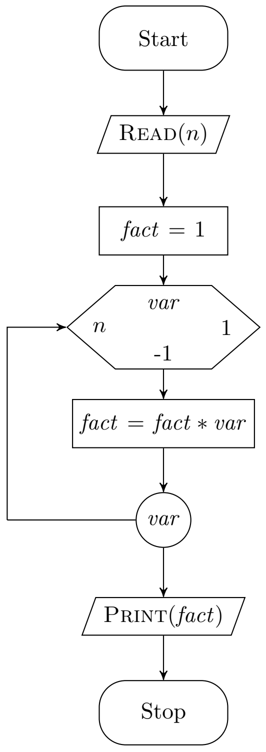


Figure 2.7: To find the factorial of a number

- Read n from the user, whose factorial is to be calculated.
- Initialize *fact* to 1
- Write a loop, with the loop control variable *var* initialized to n.
  - For each iteration of the loop, multiply the value inside *fact* variable with *var* and store it in*fact*.

- This is continued until the value of *var* becomes greater than or equal to 1, with *var* being decremented by 1 after every iteration.
- At the end of all iterations, *fact* is printed.

Problem 2.8 To determine the largest of n numbers.

Solution: See Figure 2.8 for pseudocode and flowchart.

```
LargeN
1   Start
2   Read($n$, $num$)
3   $large = num$
4   for $count = 1$ to $n - 1$
5       Read($num$)
6       if ($num > large$)
7           $large = num$
8       endif
9   endfor
10  Print($large$)
11  Stop
```

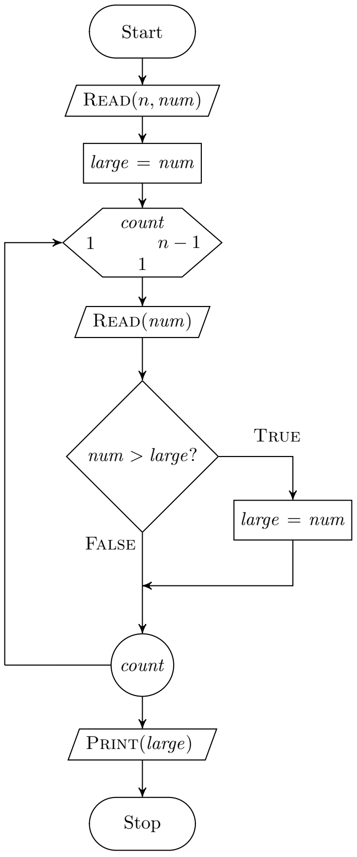


Figure 2.8: To find the largest of n numbers

- Read n from the user.
- Along with that, read a single number *num* from the user and assign a variable *large* with *num*. That is the first one among the set of input integers is assumed to be the largest.
- Initialize a loop, with the loop control variable *count* being assigned the value of 1.

- During every iteration of the loop,
  - obtain a number from the user and keep it in *num* variable
  - *num* is checked against *large*
  - If $num > large$, update *large*to *num*
- This is repeated as long as the value of *count* is less than or equal to *n* - 1.
  - You need to iterate the loop only *n* - 1 times since you received the first integer before entering the loop.
- After all iterations, the largest value is displayed.

Problem 2.9 To determine the average age of students in a class. The user will stop giving the input by giving the age as 0.

Solution: See Figure 2.9 for pseudocode and flowchart.

```
AverageAgev1
1  Start
2  $sum = 0$
3  $count = 0$
4  Read($age$)
5  while ($age$!=0)
6      $sum = sum + age$
7      $count = count + 1$
8      Read($age$)
9  endwhile
10 $average = sum/count$
11 Print($average$)
12 Stop
```

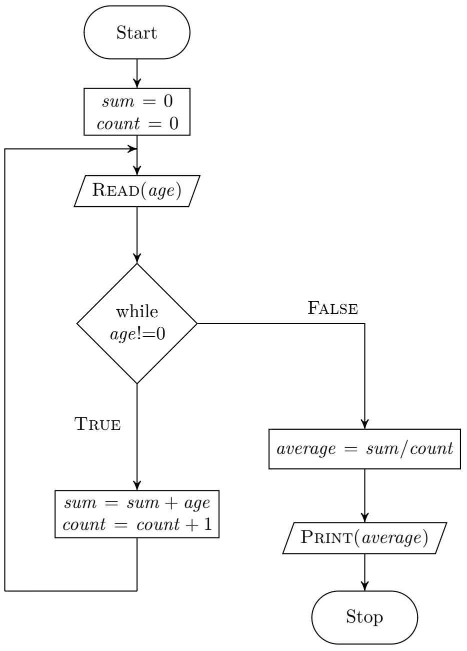


Figure 2.9: To determine the average age using **while** loop

- Initialize the variables *sum* and *count* to 0.
- Read the age of the first student and keep it in *age* variable.

- Write a while loop that will run until *age* is not equal to zero.
- In every iteration,
  - add the *age* value to *sum* and increment *count* by 1
  - read the next value of *age*
- After getting out of the loop, determine the average age by dividing *sum* by *count*.
- Print the average value.

Problem 2.10 Redo Problem 2.9 using repeat-until loop construct.

Solution: See Figure 2.10 for flowchart and pseudocode.

```
AVERAGEAGEV2
1   Start
2   $sum = 0$
3   $count = 0$
4   READ($age$)
5   repeat
6       $sum = sum + age$
7       $count = count + 1$
8       READ($age$)
9   until ($age == 0$)
10  $average = sum/count$
11  PRINT($average$)
12  Stop
```

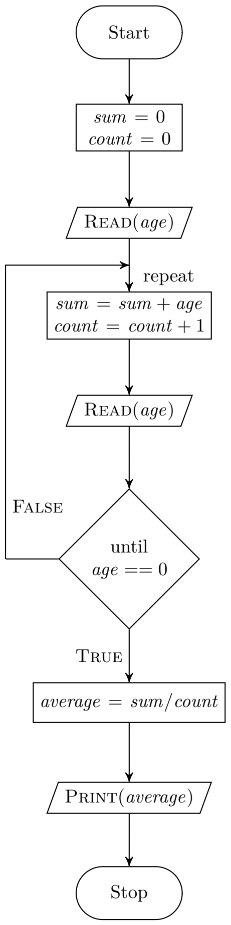


Figure 2.10: To determine the average age using **repeat until** loop

- Initialize *sum, count* to 0.
- Read *age* from the user.
- Write the repeat-until loop:
- Inside the loop:
  - add *age* value to *sum*
  - increment *Count*
  - read the next *age* value
- Continue the loop until the user inputs a 0 for *age*.
- After getting out of the loop, determine the average age by dividing *sum* by *count*.
- Print the average value.

Problem 2.11 To find the average height of boys and average height of girls in a class of n students.

Solution: See Figure 2.11.

```
AVERAGEHEIGHT
Start
READ($n$)
$btotal = 0$
$bcount = 0$
$gtotal = 0$
$gcount = 0$
for $var = 1$ to $n$
    READ($gender, height$)
    if ($gender ==$ 'M')
        $btotal = btotal + height$
        $bcount = bcount + 1$
    else
        $gtotal = gtotal + height$
        $gcount = gcount + 1$
    endif
endfor
$bavg = btotal/bcount$
$gavg = gtotal/gcount$
PRINT($bavg, gavg$)
Stop
```

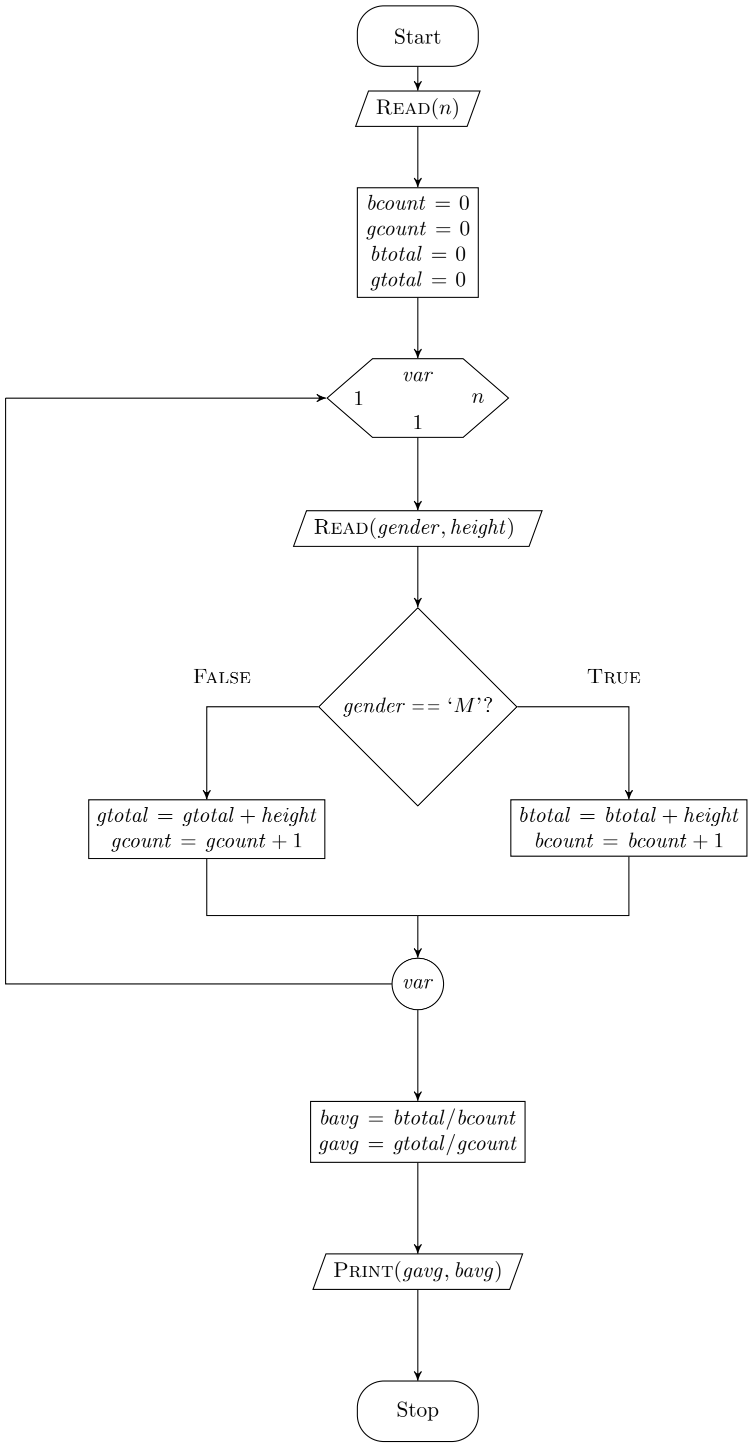


Figure 2.11: To find the average height of boys and girls

- Input n from the user.
- Define variables *btotal*, *bcount*, *gtotal*, and *gcount* to store the total height of boys, number of boys, total height of girls and number of girls respectively. Initialize all the four variables to 0
- Write a loop that prompts the user for the relevant inputs for all the n students.
- During each iteration:
  - Read *gender* and *height*
  - If the gender is '*M*', add the input height to *btotal* and increment *bcount*
  - Otherwise, add the input height to *gtotal* and increment *gcount*
- After getting out of the loop, determine the average height of the boys, *bavg* by dividing *btotal* with *bcount*. In a similar way, calculate the average height of the girls, *gavg* by dividing *gtotal* with *gcount*.
- Finally print the average values.

2.4 Conclusion

In this chapter, we've explored the foundational elements of algorithms, flowcharts, and pseudocode — the essential tools in the development of efficient and effective programs. Algorithms provide a clear, step-by-step approach to problem-solving, while flowcharts offer a visual representation that aids in understanding the logical flow. Pseudocode serves as an intermediary step, bridging the gap between the abstract algorithm and the actual code, making it easier to translate ideas into executable code. Together, these tools help in breaking down complex problems into manageable parts, ensuring clarity and precision in programming. Mastering these concepts is crucial for anyone aiming to develop robust software solutions. As you move forward, these skills will serve as a solid foundation for tackling more complex problems.

2.5 Exercises

1. Write algorithms for the following:
   1. to find the area and circumference of a circle.
   2. to find the area of a triangle given its three sides.
   3. to find the area and perimeter of a rectangle.
   4. to find the area of a triangle given its length and breadth.

2. If the three sides of a triangle are input, write an algorithm to check whether the triangle is isosceles, equilateral, or scalene.

3. Write a switch statement that will examine the value of `flag` and print one of the following messages, based on the value assigned to the flag.

Flag value	Message
1	HOT
2	LUKE WARM
3	COLD
Any other value	OUT OF RANGE

4. Write algorithms for the following:

   (a) to display all odd numbers between 1 and 500 in descending order.

   (b) to compute and display the sum of all integers that are divisible by 6 but not by 4 and that lie between 0 and 100.

   (c) to read a value, and do the following: If the number is even, halve it; if it's odd, multiply by 3 and add 1. Repeat this process until the value is 1, printing out each value.

5. Write an algorithm that inputs two values a and b and that finds a^b. Use the fact that a^b is multiplying a with itself b times.

6. You visit a shop to buy a new mobile. In connection with the festive season, the shop offers a 10% discount on all mobiles. In addition, the shop also gives a flat exchange price of ₹ 1000 for old mobiles. Draw a flowchart to input the original price of the mobile and print its selling price. Note that all customers may not have an old mobile for exchange.

7. Draw flowcharts for the following:

   (a) to find the volume of a hemisphere by inputting the radius.

   (b) to find the profit or loss incurred by getting the cost price and selling price of an item. Note that you are not asked to determine whether profit or loss is incurred but rather the value of profit or loss. Assume cost price $\neq$ selling price.

   (c) to find the average of a list of numbers entered by the user. The user will stop the input by giving the value -999.

Bibliography

[1] Maureen Sprankle and Jim Hubbard, Problem Solving and Programming Concepts, Pearson, 2011.

[2] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest and Clifford Stein, Introduction to Algorithms, The MIT Press, 2009.

[3] Michael T. Goodrich and Roberto Tamassia, Algorithm Design and Applications, Wiley, 2015.

[4] Richard F. Gilberg and Behrouz A. Forouzan, Data Structures: A Pseudocode Approach With C, Cengage Learning, 2004.

[5] Donald E. Knuth, The Art of Computer Programming (Volume 1), Addison-Wesley, 1997.

Chapter 3

Foundations of computing

"The problem of computing is not about speed, it's about understanding."
– Donald E Knuth

3.1 Introduction

The computer is an advanced electronic device that takes raw data as *input* from the user and processes this data under the control of a set of instructions (called *program*) and gives the result (output) and saves the output for future use. It can perform numerical (arithmetic) and non-numerical (logical) calculations.

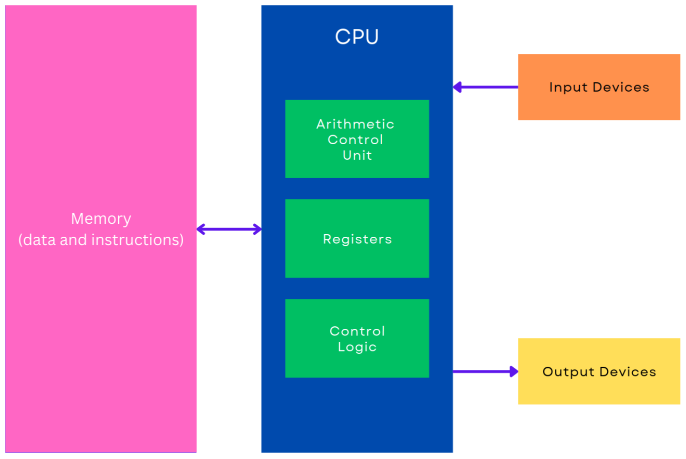


Figure 3.1: A typical computer architecture

3.2 Architecture of a computer

A typical computer architecture comprises of three main components:

1. Input/Output (I/O) Unit,
2. Central Processing Unit (CPU), and
3. Memory Unit.

The I/O unit consists of the input unit and output devices. The input devices accept data from the user, which the CPU processes. The output devices transfer the processed data (information) to the user. The memory unit is used to store the input data, the instructions required to process the input, and also the output information. Figure 3.1 illustrates a typical architecture of a computer.

3.2.1 Input/output unit

The user interacts with the computer via the I/O unit.

3.2.1.1 Input devices

Input devices allow users to input data into the computer for processing. They are necessary to convert the input data into a form that can be understood by the computer. The data input to a computer can be in the form of text, audio, video, etc. Input devices are classified into two categories:

1. Human data entry devices – the user enters data into the computer by typing or pointing a device to a particular location. Some examples are
   (a) Keyboard – the most common typing device.
   (b) Mouse – the most common pointing device. You move the mouse around on a *mouse pad* and a small pointer called *cursor* follows your movements on the computer screen.
   (c) Track ball – an alternative to a mouse which has a ball on the top. The cursor on the computer screen moves in the direction in which the ball is moved.
   (d) Joystick – has a stick whose movement determines the cursor position.
   (e) Graphics tablet – converts hand-drawn images into a format suitable for computer processing.
   (f) Light pen – can be used to "draw" on the screen or to select options from menus presented on the screen by directly pointing the pen on the screen.
2. Source data entry devices –They use special equipment to collect data at the source, create machine-readable data, and feed them directly into the computer. This category comprises:

(a) Audio input devices – It uses human voice or speech to give input. Microphone is an example.

(b) Video input devices – They accept input in the form of video or images. Video cameras, webcams are examples.

(c) Optical input devices – They use optical technology (light source) to input the data into computers. Some common optical input devices are scanners and bar code readers.

> ☞ Other examples include Optical Character Recogniser (OCR), Magnetic Ink Character Recogniser (MICR), and Optical Mark Recogniser (OMR).

3.2.1.2 Output devices

An output device takes processed data from the computer and converts them into information that can be understood by humans. The output could be on paper or a film in a tangible form or an intangible form like audio, video, etc. Output devices are classified as follows:

1. Hard copy devices – The output obtained in a tangible form on paper or any surface is called hard copy output. The hard copy can be stored permanently and is portable. The hard copy output can be read or used without a computer. Examples in this category include:

   (a) Printer – prints information on paper. The information could be textual or even images. Drum printers, laser printers, and inkjet printers are some commonly used printer types.

   (b) Plotter – used to produce very large drawings on paper sizes up to A0 (16 times as big as A4). A plotter draws onto the paper using very fine pens. Flatbed plotters and drum plotters are two types of plotters.

   (c) COM (Computer Output on Microfilm) – stores the output (mostly as images) on a microfilm.

2. Soft Copy Devices – Generates a soft copy of the output (output obtained in an intangible form) on a visual display, audio unit, or video unit. The soft copy can be stored and sent via e-mail to other users. It also allows corrections to be made. The soft copy output requires a computer to be read or used. This category comprises:

   (a) Monitor – the primary output device of a computer. Monitors can be of two types: monochrome (black and white) or color. It forms images from tiny dots, called *pixels*, that are arranged in a rectangular form.

   (b) Video output devices – produce output in the form of video or images. An example is a screen image projector or data projector that displays information. from the computer onto a large white screen.

(c) Audio output devices – speakers, headsets, or headphone, are used for audio output (in the form of sound) from a computer. The signals are sent to the speakers via a *sound card* that translates the digital sound back into analog signals.

3.2.2 Central Processing Unit

The Central Processing Unit (CPU) or the processor, is often known as the *brain* of a computer. It consists of an Arithmetic Logic Unit (ALU) and a Control Unit (CU). In addition, the CPU also has a set of *registers* which are temporary storage areas for holding data and instructions.

3.2.2.1 Arithmetic Logic Unit

This unit consists of two sub-units, namely arithmetic and logic units.

1. **Arithmetic unit** – performs arithmetic operations like addition, subtraction, multiplication, and division, on the data.

2. **Logic unit** – performs logical operations like comparisons of data values.

3.2.2.2 Registers

Registers are high-speed storage areas within the CPU but have the least storage capacity. They store data, instructions, addresses, and intermediate results of processing. Some of the commonly used registers are:

- Accumulator (ACC) – stores the result of arithmetic and logic operations.

- Instruction Register (IR) – holds the instruction that is currently being executed.

- Program Counter (PC) – holds the address of the next instruction to be processed.

- Memory Address Register (MAR) – contains the address of the data to be fetched.

- Memory Buffer Register (MBR) – temporarily stores data fetched from memory or the data to be sent to memory.

  ☞MBR is also called a Memory Data Register (MDR).

- Data Register (DR) stores the operands and any other data.

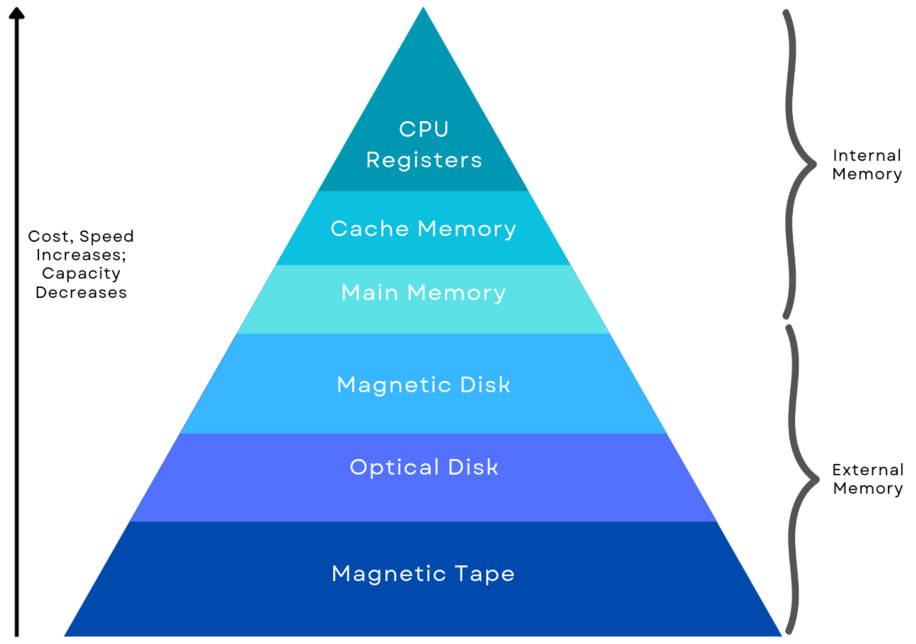


Figure 3.2: Memory hierarchy

3.2.2.3 Control Unit

This unit manages and coordinates the operations of all parts of the computer but does not carry out any actual data processing operations. The functions of this unit are:

1. generate *control signals* that controls various operations of the computer.
2. obtain instructions from the memory, interpret them, and then direct the ALU to execute those instructions.
3. communicate with I/O devices for transfer of data or results from/to memory.
4. decides when to fetch the data and instructions, what operation to perform, where to store the results, the ordering of various events during processing etc.

3.2.3 Memory unit

Memory is the storage space in a computer where data to be processed and instructions required for processing are stored. The various memories can be organized hierarchically called *memory hierarchy* as shown in Figure 3.2. As we ascend the memory hierarchy,

- Capacity in terms of storage decreases.
- Cost per bit of storage increases.
- Frequency of access of the memory by the CPU increases.
- Access time decreases.

Memory is primarily of two types :

1. Internal Memory: memories that reside on the motherboard.
2. External memory: memories that are outside the motherboard.

3.2.3.1 Internal Memory

Internal memory includes:

1. Registers – high-speed storage areas within the CPU.
2. Primary memory – main memory of the computer. It is categorized into two:

   (a) Random Access Memory (RAM) – used for storing data and instructions during the operation of a computer. Data to be processed are brought to RAM from input devices or secondary memory. After processing, the results are stored in RAM before being sent to the output device.

   ☞RAM is often referred to as *volatile memory*, since the data stored in RAM are lost when you switch off the computer or if there is a power failure.

   ☞Dynamic RAM (DRAM) and Static RAM (SRAM) are the two types of RAM

   .

   (b) Read Only Memory (ROM) – a non-volatile primary memory. It does not lose its content when the power is switched off. ROM, as the name implies, has only read capability and no write capability. After the information is stored in ROM, it is permanent and cannot be modified. Therefore, ROM is used to store the data that do not require a change, for example, the *boot information* (information required while starting the computer when it is switched on).

☞PROM (Programmable Read Only Memory), EPROM (Erasable and Programmable Read Only Memory), EEPROM (Electrically Erasable and Programmable Read Only Memory), and UVEPROM (Ultra-Violet Erasable and Programmable Read Only Memory) are the various ROM types.

3. Cache memory – placed between RAM and CPU and stores the data and instructions that are frequently used . During data processing, the CPU first checks the cache for the required data or instruction. If found in the cache, the data or instructions are retrieved from the cache itself. Otherwise, they are then retrieved from RAM.

3.2.3.2 External Memory

External memory includes:

1. Magnetic tape – a plastic tape with magnetic coating mounted on a reel or in a cassette. It is a sequential access device, meaning that the data can be read only in the order in which they are stored.
2. Magnetic disk – a thin plastic or metallic circular plate coated with magnetic oxide and encased in a protective cover. Hard disk is an example.

   ☞A magnetic disk is a direct-access secondary storage device, ie., you can directly access the location you want, without accessing the previous locations.

3. Optical disk – a flat circular disk coated with reflective plastic material that can be altered by laser light. The data bits 1 and 0 are stored as spots that are relatively bright and light, respectively. CDs and DVDs are examples.

3.3 Von Neumann architecture

If you want to perform some processing on data, they must be stored in computer memory. Similarly, the instructions that process the data must also be stored in the memory. This concept of storing programs in computer memory is known as **stored program concept**. A computer based on *Von Neumann architecture* stores data and instructions in the same memory.

☞As an alternative to *Von Neumann architecture*, you have the *Harvard architecture*, wherein there are separate memories for storing programs and data (code memory for programs and data memory for data). It also utilises *stored program concept.*

3.4 Instruction execution

An instruction 'instructs' the processor to perform an elementary operation. Curious to know what an instruction looks like and how it is executed? Read on!

3.4.1 Instruction format

An instruction format defines the layout of an instruction, in terms of its constituent parts. The set of instructions that a computer processor can understand and execute is called its *Instruction Set Architecture* or ISA in short. Each processor has its own ISA. Irrespective of the ISA, an instruction can be thought to be divided into various parts called *fields*. The most common fields of instruction are *Operation code (opcode)* and *Operand code* as shown below.

opcode	operand code

The remainder of the instruction fields differ from one ISA to another. The operation code represents the action the processor must perform. The operand code defines the data (operands) on which the operations are to be performed. It directly specifies the value of the operands or tells the locations where to fetch the operands from.

3.4.2 Instruction execution cycle

CPU executes an instruction in a series of steps called *instruction execution cycle.* An instruction cycle (sometimes called a fetch–decode–execute cycle) is the process by which a computer retrieves a program instruction from its memory, determines what actions to perform, and carries out those actions to produce meaningful output. The instruction cycle which starts with fetching the instruction itself is shown in Figure 3.3.

The cycle comprises the following steps:

1. Fetch the instruction

    1.1 Fetch the instruction from memory whose address is currently held in the program counter.

    1.2 Store the fetched instruction in the instruction register.

    1.3 Increment the program counter so that it has the address of the next instruction to be fetched.

2. Decode the instruction
    - 2.1 Based on the instruction set architecture, the instruction is broken into opcode and operand codes.
    - 2.2 The operation to be performed is thus identified.
3. Execute the instruction
    - 3.1 The operation identified in the decode step is now performed by the CPU.
4. Store the result
    - 4.1 The result generated by the operation is stored in the main memory, or sent to an output device.

This cycle is then repeated for the next instruction.

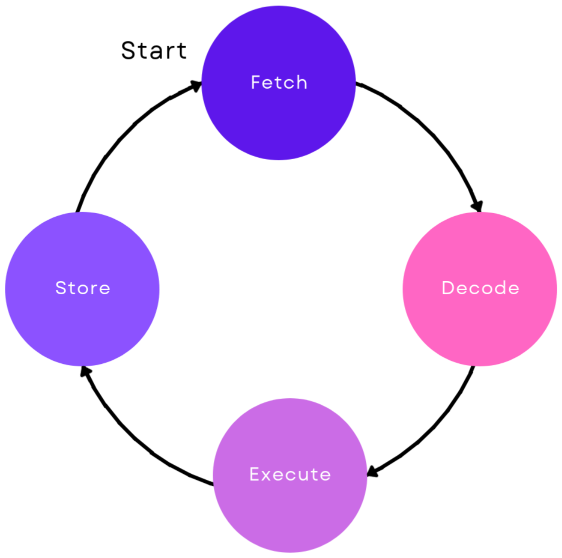


Figure 3.3: Instruction execution cycle[1]

3.5 Programming languages

Programming languages are used to write programs that are precise representations of algorithms and control the behaviour of a computer. Each language has a unique set of keywords (words that it understands) and syntax (set of rules) to organize the program instructions.

Programming languages fall into three categories:

[1]Image courtesy: Slightly adapted from *Computer Fundamentals, Anita Goel* and redrawn.

1. Machine language:- is what the computer can understand, but it is difficult for the programmer to understand.

   ☞Machine languages consist of binary numbers only.

2. High-level language:- is easier to understand and use for the programmer but difficult for the computer.

   ☞The programs written in high-level languages contain English-like statements as well as programming constructs specific to the language.

3. Assembly language:- falls in between machine language and high-level language. It is similar to machine language, but easier to write code because it allows the programmer to use symbolic names (like `ADD`, `SUB`) for operations.

   ☞Machine languages and assembly languages are also called **low-level languages**.

3.5.1 Machine Language

A program written in machine language is a collection of binary digits or bits (strings of 0's and 1's) that the computer reads and interprets. It is also referred to as *machine code* or *object code*. Some features of a program written in machine language are:

1. The computer can understand the programs written in machine language directly. No translation of the program is needed.

2. Program written in machine language can be executed very fast (Since no translation is required).

3. Machine language is defined by the hardware of a computer. It depends on the underlying processor, memory arrangement, operating systems, and peripheral devices; and is thus machine-dependent. A machine-level program written for one type of computer may not work on another type.

4. Each object code instruction has an **opcode** field that specifies the actual operation (such as add or compare) and some other fields for the operands.

   Example 3.1. Assume that the opcode and operands occupy 4 bits each and that the opcode for addition is 1010, then to add 3 and 6, the binary code would be $\underbrace{1010}_{add}\underbrace{0011}_{3}\underbrace{0110}_{6}$

3.5.2 Assembly Language

A program written in assembly language uses a symbolic representation of machine codes. The opcodes are replaced by symbolic names called **mnemonic codes** that are much easier to remember. The mnemonics for various operations are decided by the processor manufacturer and cannot be changed by the programmer. Some of the features of assembly code (program written in assembly language) are:

1. Assembly language programs are easier to write than machine language programs since assembly language programs use short, English-like representations of machine code.

   Example 3.2. If the mnemonic code for addition is `ADD`, then to add 3 and 6, the assembly level code will be `ADD 3,6`.

2. Assembly language programs are also machine-dependent.

3. Although assembly language programs use symbolic representation, they are still difficult to write. They are generally employed when efficiency matters.

3.5.3 High-level Language

A high-level language is a programming language such as C, FORTRAN, or Pascal that enables a programmer to write programs that are more or less independent of a particular type of hardware. Such languages are considered high-level because they stand closer to humans but farther from machine languages. Some of the features of programs written in high-level language are as follows:

1. Programs are easier to write, read, and understand in high-level languages than in machine language or assembly language.

2. Most of the operations like arithmetic and logical operations are denoted by symbols called *operators*.

   Example 3.3. To add 3 and 6, the high level language code will be `3+6`. Here '`+`' is the operator for addition.

3. The programs written in high-level languages are easily portable from one computer to another, since they are not machine-dependent

3.6 Translator software

The computer can understand only machine code (strings of 0's and 1's). Thus when the program is written in a language other than machine language (assembly or high-level languages), the program is to be converted to machine code. This conversion is called **translation** and is performed by the translator software.

☞In the translation process, the original program is called *source code*, and the translated code (object code) is the *target code*.

There are three types of translator software as discussed below.

3.6.1 Assembler

The assembler converts a program written in assembly language into machine code. There is usually a one-to-one correspondence between the assembly statements and the machine language instructions.

☞The machine language is dependent on the processor architecture. Thus, the converted assembly language programs also differ for different computer architectures.

3.6.2 Compiler

The compiler is a software that translates programs written in high-level language to object code, which can be then executed independently. Each programming language has its compiler. The compilation process generally involves breaking down the source code into small pieces creating an intermediate representation, and then constructing the object code from the intermediate representation.

3.6.3 Interpreter

The interpreter also converts the high-level language program into machine code. However, the interpreter functions in a different way than a compiler. An interpreter reads the source code line-by-line, converts it into machine-understandable form, executes the line, and then proceeds to the next line. This is unlike a compiler that takes the entire source code and converts it to object code. The key differences between a compiler and an interpreter are shown in Table 3.1.

Table 3.1: Comparison of compiler and interpreter

Compiler	**Interpreter**
1. converts the entire source code into object code which is then executed by the user.	1. translates the source code line-by-line – takes a line of source code, converts it into machine executable form, executes it, and proceeds with the next line.
2. Once the object code is created, it can be executed multiple times without the need to compile during each execution.	2. During each execution, the source code is first interpreted and then executed.
3. During execution of an object code, neither the source nor the compiler is required	3. Both interpreter and source code are required during execution.
4. faster	4. slower
5. Languages like C, C++ and Java are compiled.	5. Languages like BASIC, Pascal and Python are interpreted.

3.7 Conclusion

Through the exploration of computer architecture and computation, we've unraveled the essential components that form the backbone of modern computing. By examining the architecture of a computer, including the input/output units, central processing unit (CPU), and memory unit, we've gained insights into data flows and how data are processed within a computing system. The Von Neumann architecture, a foundational concept in computer science, was discussed as the blueprint that guides the organization of these components, highlighting the interaction among them.

The journey continued with a detailed look at how instructions are formatted and executed, providing clarity on how computers systematically perform tasks. We explored the various types of programming languages, from machine language to assembly and high-level languages, illustrating the evolution of languages that facilitate human-computer interaction. Additionally, the role of translator software viz. assemblers, compilers, and interpreters was emphasized as crucial for bridging the gap between human-readable code and machine-executable instructions.

This comprehensive overview of computer architecture and computation lays a strong foundation for understanding how computers function as powerful tools for solving complex problems. By grasping these concepts, you're better equipped to appreciate the intricacies of programming and the computational processes that drive the digital world.