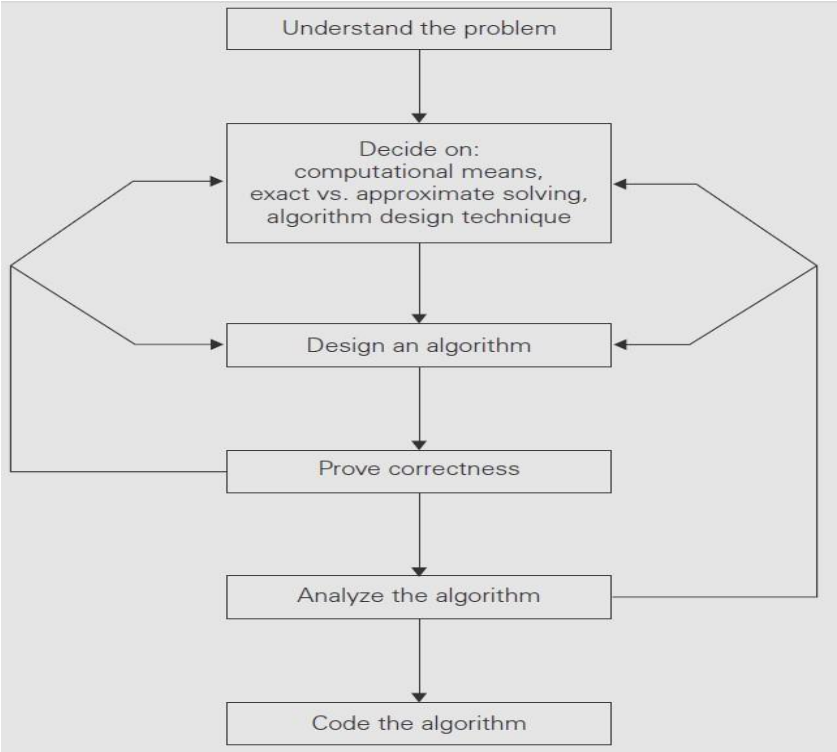
FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

A sequence of steps involved in designing and analyzing an algorithm is shown in the figure below.

FIGURE 1.2.1 Algorithm design and analysis process.



(i) Understanding the Problem

- This is the first step in designing of algorithm.
- Read the problem's description carefully to understand the problem statement completely.
- Ask questions for clarifying the doubts about the problem.
- Identify the problem types and use existing algorithm to find solution.
- Input (***instance***) to the problem and range of the input get fixed.

(ii) Decision making

The Decision making is done on the following:

a) Ascertaining the Capabilities of the Computational Device

In ***random-access machine*** (***RAM***), instructions are executed one after another (The central assumption is that one operation at a time). Accordingly,

algorithms designed to be executed on such machines are called ***sequential algorithms***.

In some newer computers, operations are executed **concurrently**, i.e., in parallel. Algorithms that take advantage of this capability are called ***parallel algorithms***.

Choice of computational devices like Processor and memory is mainly based on space and time efficiency

a) Choosing between Exact and Approximate Problem Solving:

The next principal decision is to choose between solving the problem exactly or solving it approximately.

An algorithm used to solve the problem exactly and produce correct result is called an **exact algorithm**.

If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an **approximation algorithm**. i.e., produces an

Approximate answer. E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

a) Algorithm Design Techniques

- An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different

  Algorithms*+ Data Structures =Programs*

  areas of computing.
- Though Algorithms and Data Structures are independent, but they are combined together to develop program. Hence the choice of proper data structure is required before designing the algorithm.
- **Implementation** of algorithm is possible only with the help of Algorithms and Data Structures
- **Algorithmic strategy / technique / paradigm** are a general approach by which many problems can be solved algorithmically. E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and soon.

(iii) Methods of Specifying an Algorithm

There are three ways to specify an algorithm. They are**:**

a. **Natural language**

b. **Pseudocode**

c. **Flowchart**

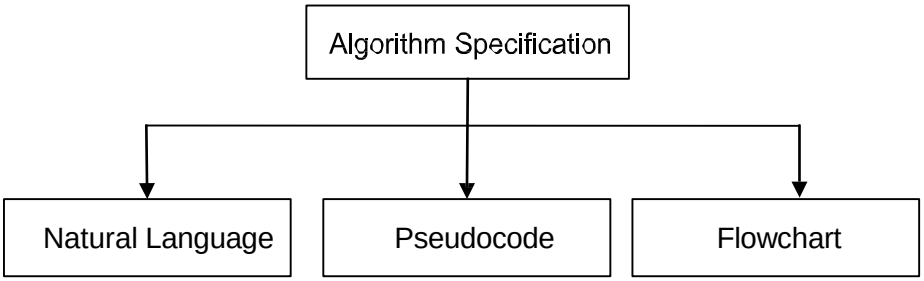


FIGURE 1.2.2 Algorithm Specifications

Pseudocode and flowchart are the two options that are most widely used nowadays for specifying algorithms.

a. Natural Language

It is very simple and easy to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear and thereby we get brief specification.

Example: An algorithm to perform addition of two numbers.

Step 1: Read the first number, say a. Step 2: Read the first number, say b. Step 3: Add the above two numbers and store the result in c. Step 4: Display the result from c.

Such a specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of Pseudocode.

b) **Pseudocode:**

- Pseudocode is a mixture of a natural language and programming language constructs. Pseudocode is usually more precise than natural language.
- For Assignment operation left arrow **"←"**, for comments two slashes **"//",if** condition**, for, while** loops are used.

ALGORITHM *Sum(a,b)*

```
//Problem Description: This algorithm performs addition of two numbers
//Input: Two integers a and b
//Output: Addition of two integers
c←a+b
returnc
```

This specification is more useful for implementation of any language.

c) Flowchart

- In the earlier days of computing, the dominant method for specifying algorithms was a ***flowchart***, this representation technique has proved to be inconvenient.
- ***Flowchart*** is a graphical representation of an algorithm. It is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

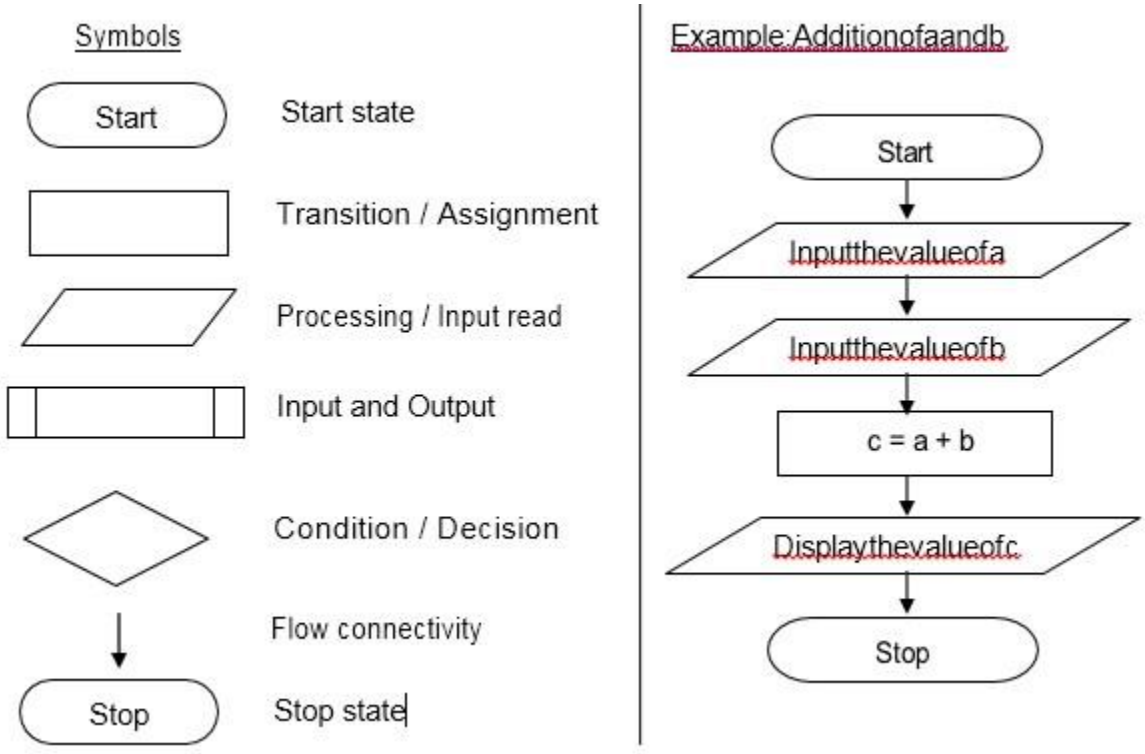


FIGURE 1.2.3 Flowchart symbols and Example for two integer addition.

(iv) Proving an Algorithm's Correctness

- Once an algorithm has been specified then its ***correctness*** must be proved.
- An algorithm must yield a required **result** for every legitimate input in a finite amount of time.
- For Example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd*(m, n)* = gcd*(n, m* mod *n).*
- A common technique for proving correctness is to use **mathematical induction** because an algorithm's iterations provide a natural sequence of steps needed for such proofs.
- The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The **error** produced by the algorithm should not exceed a predefined limit.

(v) Analyzing an Algorithm

- For an algorithm the most important is ***efficiency***. In fact, there are two kinds of algorithm efficiency.

They are:

- ***Time efficiency***, indicating how fast the algorithm runs, and
- ***Space efficiency***, indicating how much extra memory it uses.
- The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.
- So factors to analyze an algorithm are:
    - Time efficiency of an algorithm
    - Space efficiency of an algorithm
    - Simplicity of an algorithm
    - Generality of an algorithm

(vi) Coding an Algorithm

- The coding / implementation of an algorithm is done by a suitable programming language like C, C++,JAVA.
- The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The Algorithm power should not reduce by in efficient implementation.
- Standard tricks like computing a **loop's invariant** (an expression that does not change its value) outside the loop, collecting **common subexpressions**, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.
- Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by **orders of magnitude**. But once an algorithm is selected, a 10–50% speedup may be worth an effort.
- It is very essential to write an **optimized code (efficient code)** to reduce the burden of compiler.