The Design and Analysis of Algorithm

Introduction Class

Instructor: Sadullah Karimi

M.Sc. in CSE

Two ideas lie gleaming on the jeweler's velvet. The first is the calculus, the second, the algorithm. The calculus and the rich body of mathematical analysis to which it gave rise made modern science possible; but it has been the algorithm that has made possible the modern world.

—David Berlinski, *The Advent of the Algorithm*, 2000

دو ایده بر روی مخمل جواهر سازی می درخشد. اولی حساب دیفرانسیل و انتیگرل، دومی، الگوریتم است. محاسبات و حجم غنی از تجزیه و تحلیل ریاضی که باعث شد علم مدرن امکان‌پذیر شود. اما این الگوریتم بوده است که دنیای مدرن را ممکن ساخته است

Introduction

- Computer programs would not exist without algorithms.
- Another reason for studying algorithms is their usefulness in developing analytical skills.
- After all, algorithms can be seen as special kinds of solutions to problems—not just answers but precisely defined procedures for getting answers.

What Is an Algorithm?

- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate or legal input in a finite amount of time.
- We call this a “computer,” keeping in mind that before the electronic computer was invented, the word “computer” meant a human being involved in performing numeric calculations.
- Nowadays, of course, “computers” are those ubiquitous electronic devices that have become indispensable (must) in almost everything we do.

What is Algorithm (Continue)

- The nonambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

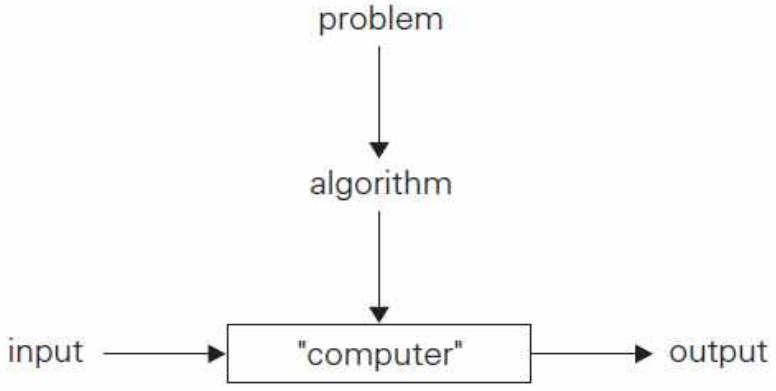

FIGURE 1.1 The notion of the algorithm.

Euclid’s algorithm

- gcd(m, n) = gcd(n, m mod n),
- Greatest Common Algroithm
- Where m mod n is the remainder of the division of m by n, until m mod n is equal to 0.
- gcd(60, 24) = gcd(24, 12) = gcd(12, 0) = 12.

Euclid’s algorithm

	12	33
Divisors	1,2,3,4,6,12	1,3,11,33
Common Divisors	1,3	
Greatest Common Divisor (GCD)	3	

- GCD(12,33) = 3

Example (2)

	25	150
Divisors	1,5,25	1,2,3,5,6,10,15,25,30,50,75, 150
Common Divisors	1,5,25	
Greatest Common Divisor (GCD)	25	

- GCD(25,150) = 3

Class Practice Examples

- GCD(20, 40) = ?
- GCD(10, 30) = ?
- GCD(50, 100) = ?

Euclidean Algorithm in Python

- a = 60 # first number
- b = 48 # second number

- # loop until the remainder is 0
- while b != 0:
- 	a, b = b, a % b
- print(a)
- Explanation: while loop runs until b becomes 0. In each iteration, a is updated to b and b is updated to a % b. When b becomes 0, the value of a is the GCD .

Continue

- import math

- a = 60 # first number
- b = 48 # second number

- print(math.gcd(a, b))

Fundamentals of Algorithmic Problem Solving

- We can consider algorithms to be procedural solutions to problems.
- These solutions are not answers but specific instructions for getting answers.
- It is this emphasis on precisely defined constructive procedures that makes computer science distinct from other disciplines.
- In particular, this distinguishes it from theoretical mathematics, whose practitioners are typically satisfied with just proving the existence of a solution to a problem and, possibly, investigating the solution's properties.

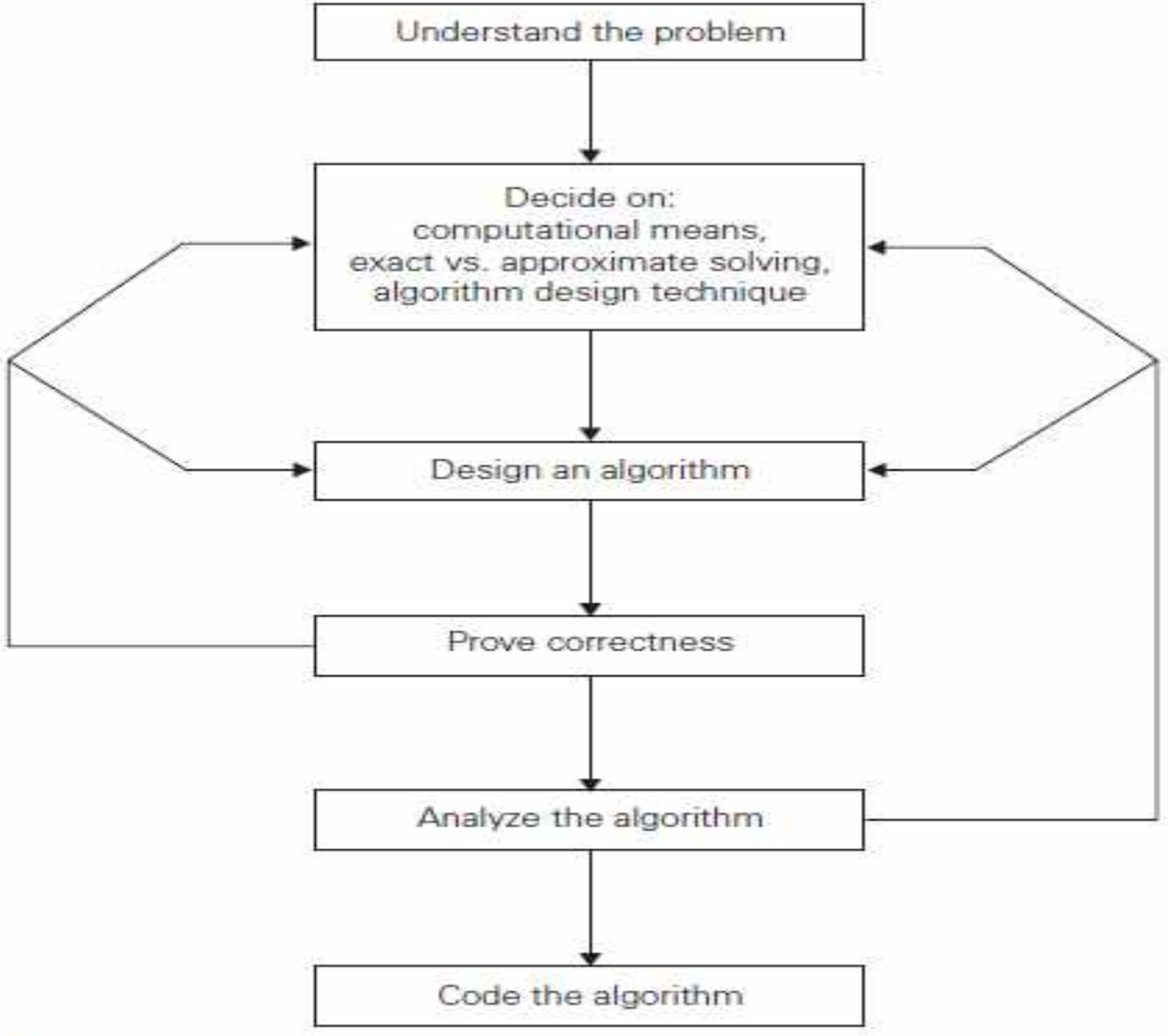

FIGURE 1.2 Algorithm design and analysis process.

Understanding the Problem

- From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given.
- Do not skimp on this first step of the algorithmic problem-solving process; otherwise, you will run the risk of unnecessary rework.

Ascertaining the Capabilities of the Computational Device

- Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for.

Choosing between Exact and Approximate Problem Solving

- The next principal decision is to choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an exact algorithm; in the latter case, an algorithm is called an approximation algorithm.
- Why would one opt for an approximation algorithm? First, there are important problems that simply cannot be solved exactly for most of their instances; examples include extracting square roots, solving nonlinear equations, and evaluating definite integrals.
- Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity.

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 areas of computing
- Second, algorithms are the cornerstone of computer science.

Designing an Algorithm and Data Structures

- While the algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem may still be a challenging task.
- Some design techniques can be simply inapplicable to the problem in question.
- Sometimes, several techniques need to be combined, and there are algorithms that are hard to pinpoint as applications of the known design techniques.
- Even when a particular design technique is applicable, getting an algorithm often requires a nontrivial (significant) ingenuity on the part of the algorithm designer.
- With practice, both tasks—choosing among the general techniques and applying them—get easier, but they are rarely easy

Methods of Specifying an Algorithm

- Using a natural language has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult.

- Nevertheless, being able to do this is an important skill that you should strive to develop in the process of learning algorithms

- Pseudocode is a mixture of a natural language and programming language like constructs.

- Pseudocode is usually more precise than natural language, and its usage often yields more succinct (brief) algorithm descriptions. Surprisingly, computer scientists have never agreed on a single form of pseudocode, leaving textbook authors with a need to design their own “dialects.”

- Fortunately, these dialects are so close to each other that anyone familiar with a modern programming language should be able to understand them all.

Pseudo-code Example

- Pseudocode Example: Checks if a Number is Even or Odd
- Start

    Input number

    If number mod % (remainder) 2 == 0 then

        Print “even”

    Else

        Print “Odd”

- End

Methods of Specifying an Algorithm

- In the earlier days of computing, the dominant vehicle for specifying algorithms was a flowchart, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.
- This representation technique has proved to be inconvenient for all but very simple algorithms; nowadays, it can be found only in old algorithm books.

Proving an Algorithm's Correctness

- Once an algorithm has been specified, you have to prove its correctness.
- That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.
- For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex.
- A common technique for proving correctness is to use mathemati_x005F_x0002_cal induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.
- It might be worth mentioning that although tracing the algorithm's performance for a few specific inputs can be a very worthwhile activ_x005F_x0002_ity, it cannot prove the algorithm's correctness conclusively.
- But in order to show that an algorithm is incorrect, you need just one instance of its input for which thealgorithm fails

Analyzing an Algorithm

- We usually want our algorithms to possess several qualities.
- After correctness, by far the most important is efficiency.
- In fact, there are two kinds of algorithm efficiency:
- Time efficiency, indicating how fast the algorithm runs.
- Space efficiency, indicating how much extra memory it uses
- Another desirable characteristic of an algorithm is simplicity
- Yet another desirable characteristic of an algorithm is generality.
- There are,in fact, two issues here:
- generality of the problem the algorithm solves.
- The set of inputs it accepts.

Coding an Algorithm

- Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril (خطر) and an opportunity.
- As a practical matter, the validity of programs is still established by testing.
- Testing of computer programs is an art rather than a science, but that does not mean that there is nothing in it to learn.
- Look up books devoted to testing and debugging; even more important, test and debug your program thoroughly whenever you implement an algorithm.

Thanks for your attention

Any question are appreciating