L1 - Introduction

Textbook Sections 1.1- 1.3

Welcome to CSE 318 virtual-live section!

I’m so happy you’re here. Let me introduce myself and get to you each of you a little better as well.

What have you heard about this course?

What are you excited about?

What are your fears?

What kind of learner are you?

Let’s talk about the structure of this course

- CSE 381 virtual live meets via Team/Zoom twice per week
- There will be quite a lot of group work in class
- We will have short student lectures on given topics
- Group work will be presented on the second lecture during the week
- No graded assignments or quizzes
- There will be 3 grade claims

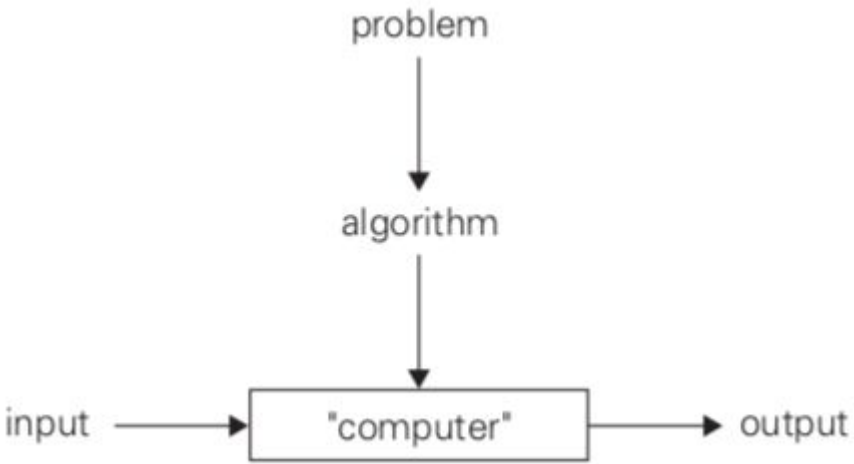
Each class period will be broken into 3 parts

1. Lecture materials
2. Group work/group discussion
3. Group presentation and class discussion

Syllabus Review

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 input in a finite amount of time.



What is an algorithm?

- The non-ambiguity 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.

Solve for the Greatest Common Divisor

The Euclidean algorithm is a way to find the *greatest common divisor* (gcd) of two positive integers, m and n.

gcd(m, n) = gcd(n, m mod n),

where m mod n is the remainder of the division of m by n, until m mod n is equal to 0.

For example, gcd(60, 24) can be computed as follows: gcd(60, 24) = gcd(24, 12) = gcd(12, 0) = 12.

Solve for the Greatest Common Divisor

The Euclidean algorithm is a way to find the *greatest common divisor* (gcd) of two positive integers, m and n.

Let me show the computations for m=210 and n=45.

Solve for the Greatest Common Divisor

The Euclidean algorithm is a way to find the *greatest common divisor* (gcd) of two positive integers, m and n.

First let me show the computations for m=210 and n=45, that is gcd(210, 45)

- Divide 210 by 45, and get the result 4 with remainder 30, so 210=4·45+30.
  - gcd(45, 30)
- Divide 45 by 30, and get the result 1 with remainder 15, so 45=1·30+15.
  - gcd(30, 15)
- Divide 30 by 15, and get the result 2 with remainder 0, so 30=2·15+0.
  - gcd(15, 0)
- The greatest common divisor of 210 and 45 is 15.

Solve for the Greatest Common Divisor

Please take a moment to use this method to solve for **gcd(31412, 14142)**

Solve for the Greatest Common Divisor

Please take a moment to use this method to solve for **gcd(31412, 14142)**

Solution:

gcd(31412, 14142)

gcd(14142, 3128)

gcd(3128, 1630)

gcd(1630, 1498)

gcd(1498, 132)

gcd(132, 46)

gcd(46, 40)

gcd(40, 6)

gcd(6, 4)

gcd(4, 2)

gcd(2, 0)

GCD(31412, 14142) = 2

Euclid's algorithm for computing gcd(m, n)

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide *m* by *n* and assign the value of the remainder to *r*.

Step 3 Assign the value of *n* to *m* and the value of *r* to *n*. Go to Step 1.

Euclid's algorithm for computing gcd(m, n), another way

Step 1 If $m < n$, exchange m and n.

Step 2 Divide m by n and get the remainder, r. If $r = 0$, report n as the GCD of m and n.

Step 3 Replace m by n and replace n by r. Repeat Step 2.

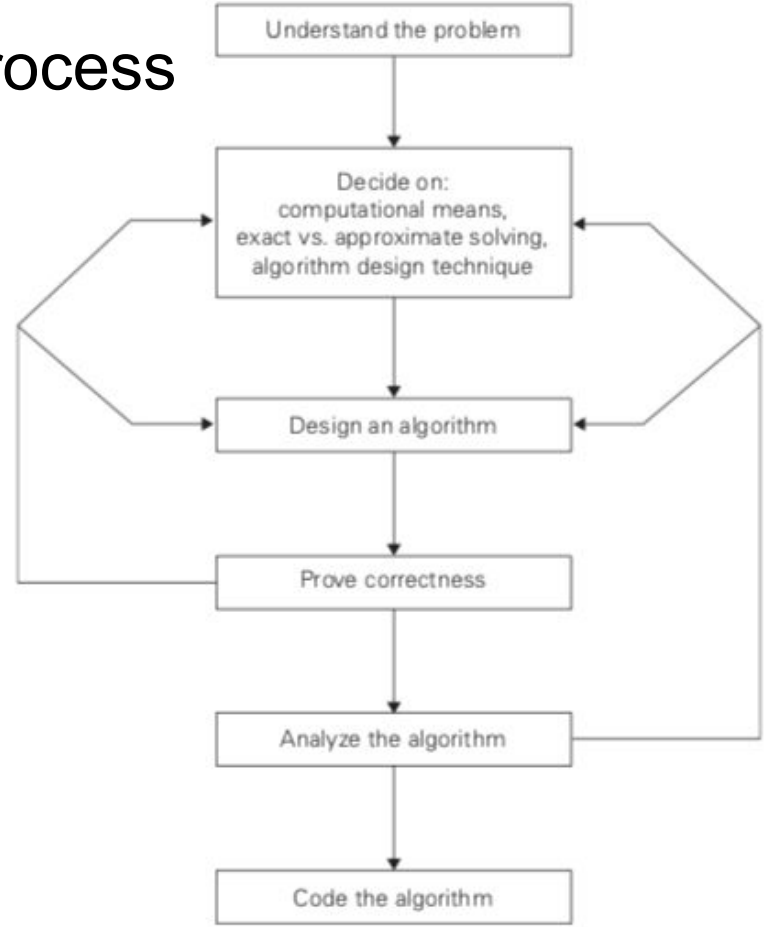
Euclid's algorithm for computing gcd(m, n)

Why does the algorithm stop?

At each step, the remainder, r, decreases by at least 1. Therefore r must eventually be 0. A formal *proof* would use mathematical induction.

Algorithm design and analysis process

1. Understand
2. Decide
3. Design
4. Prove
5. Analyze
6. Code



Understand the Problem

Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

Sometimes, we can use a known algorithm. Most often, we have to write our own

An input to an algorithm specifies an ***instance*** of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle.

Decide on Computational Means

Most computers are going to execute your algorithms without any problems

Decide on Exact vs Approximate Problem Solving

Why would one opt for an approximation algorithm?

Decide on Exact vs Approximate Problem Solving

1. 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).
2. Available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity. (This happens, in particular, for many problems involving a very large number of choices)
3. An approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

Decide on 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.

1. they provide guidance for designing algorithms for new problems, i.e., problems for which there is no known satisfactory algorithm
2. algorithms are the cornerstone of computer science.

Design the Algorithm

Even with the algorithm design techniques, designing an algorithm for a problem may be challenging!

Flowcharts aren’t used often anymore, but sometimes a flowchart or a list of steps can be helpful when designing

Pseudocode is used most often to describe algorithms. There are many “dialects” of pseudocode, but many are similar and very readable

Prove the Algorithm’s correctness

We have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.

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.

For an approximation algorithm, we usually would like to be able to show that the error produced by the algorithm does not exceed a predefined limit.

We will learn more about this in Chapter 12

Analyze the Algorithm

We usually want our algorithms to possess several qualities.

1. Efficiency
2. Simplicity
3. Generality

If you are not satisfied with the algorithm's efficiency, simplicity, or generality, you must return to the drawing board and redesign the algorithm. In fact, even if your evaluation is positive, it is still worth searching for other algorithmic solutions.

Code the Algorithm

Coding can be (is!) difficult!

Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril and an opportunity. The peril lies in the possibility of making the transition from an algorithm to a pro- gram either incorrectly or very inefficiently.

Note that throughout the book, we assume that inputs to algorithms belong to the specified sets and hence require no verification. When implementing algorithms as programs to be used in actual applications, you should provide such verifications.

Important Problem Types

Sorting

Searching

String processing

Graph problems

Combinatorial problems

Geometric problems

Numerical problems