Lakehead
UNIVERSITY

COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

Jan. 16, 2023

Overview

When: *Mondays (except long weekends) + Wednesdays 7:00pm ~ 8:30pm*

Text book: Introduction to Algorithms, (3rd Edition)

(*by Cormen*, *Thomas H.*, Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford)

https://sd.blackball.lv/library/Introduction_to_Algorithms_Third_Edition_(2009).pdf

Instructor: Dr. Yilan Gu
Email: ygu16@lakeheadu.ca

Online Course Link: https://mycourselink.lakeheadu.ca

Q&A: via email or make a online meeting appointment via email

TA(s): TBD

Measurement: Homework (15%)+ Midterm (35%) +Final (50%)

Course Objectives

Course Description:

Algorithms are the soul of computing. Algorithmic thinking, unlike the very young electronic machinery it brings alive, is rooted in ancient mathematics. It can be roughly described as creating "recipes" (well defined sequences of computational steps) for getting "things" (computational problems specifying an input-output relation) "successfully" (correctly) "done" (in finite steps and time). This course introduces basic methods for the design and analysis of efficient algorithms emphasizing methods useful in practice. Different algorithms for a given computational task are presented and their relative merits evaluated based on performance measures.

Course Objectives:

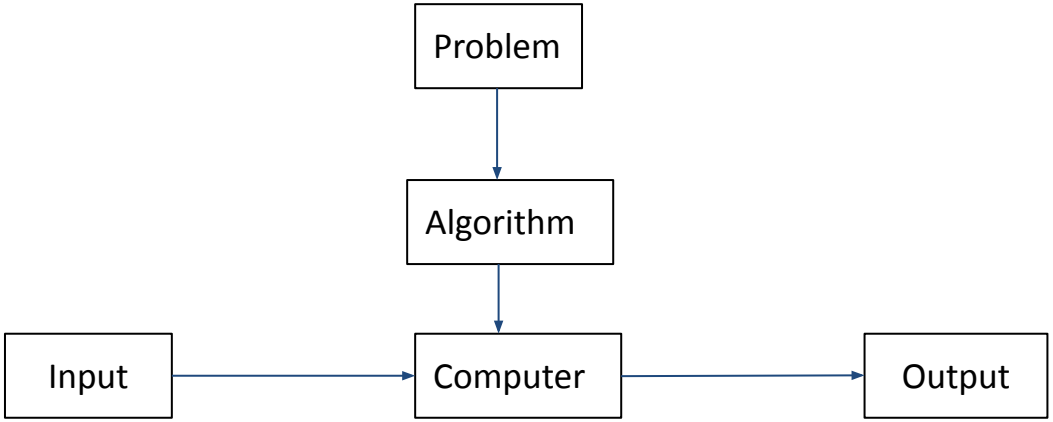
The goal of this course is to provide a solid background in the design and analysis of the major classes of algorithms. At the end of the course students will be able to understand various different algorithms for some important computational problems and to be able to compare and contrast their performance.

Introduction

Algorithm Definition

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.



A Simple Example

A **sorting algorithm** is a computing procedure which solves the following problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

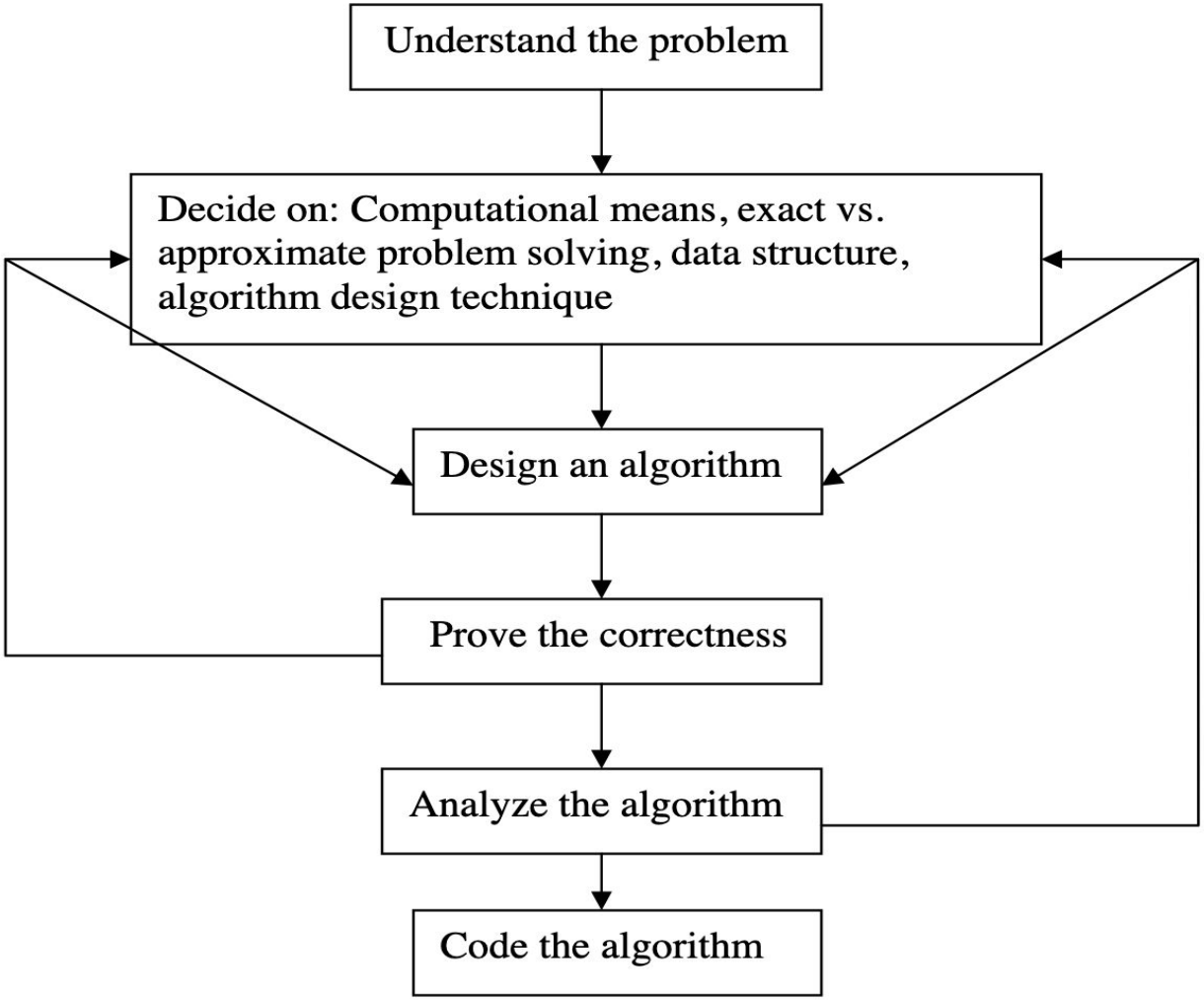
Output: A reordered sequence $\langle a_1', a_2', \ldots, a_n' \rangle$ of the input sequence such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

When people use the algorithm, actually they use it to solve an instance of a problem. One might input $\langle 31, 41, 59, 26, 41, 58 \rangle$ and get the output $\langle 26, 31, 41, 41, 58, 59 \rangle$. An algorithm is not for a instance problem, but for a class of instance problems.

There are many different ways (algorithms) to solve this problem.

Some Well-Known Computational Problems

- Sorting
- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Chess
- Towers of Hanoi
- Primality testing
- Traveling salesman problem
- Knapsack problem


Understand the problem
Decide on: Computational means, exact vs. approximate problem solving, data structure, algorithm design technique
Design an algorithm
Prove the correctness
Analyze the algorithm
Code the algorithm

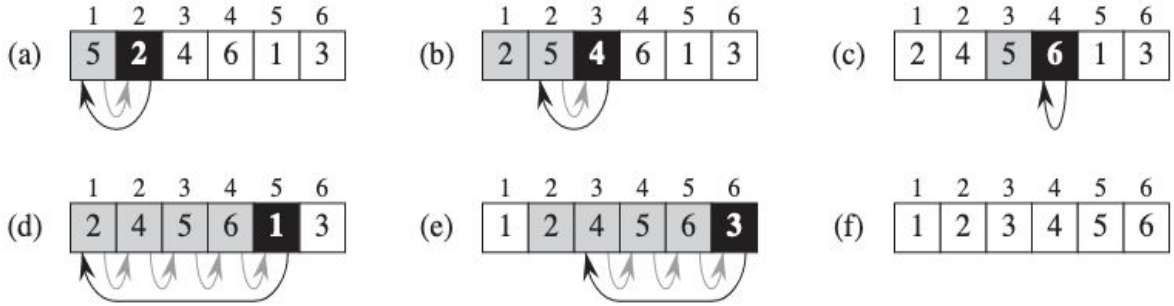
Notes about Algorithms

- Some algorithms may not always output correct answers. Such an algorithm is called incorrect algorithm. Sometimes, an incorrect algorithm is also useful, if the probability of the wrong output is small.
- But in general, we consider correct algorithms.
- Not every problem has an efficient solution (of cause there are still many problems have no solution). So sometimes, we need to prove that a problem does not have an efficient solution. Or we need to prove that an existing algorithm is optimal, i.e., one cannot design a better algorithm.
- Algorithms are usually written in **pseudocode**.

Pseudocode

- Pseudocode is a human readable description of an algorithm leaving many details of it. Writing a pseudocode has no restriction of styles and its only objective is to describe the high level steps of algorithm.
- What separates pseudocode from computer code (i.e., “real” code) is that we can employ whatever expressive method is most clear and concise to specify a given algorithm. Sometimes, the clearest method is English, so do not be surprised if you come across an English phrase or sentence embedded within a section of “real” code.
- Another difference between pseudocode and real code is that pseudocode is not typically concerned with issues of software engineering.
- Issues of data abstraction, modularity, and error handling are often ignored in order to convey the essence of the algorithm more concisely.

An Example of Pseudocode



INSERTION-SORT(A)

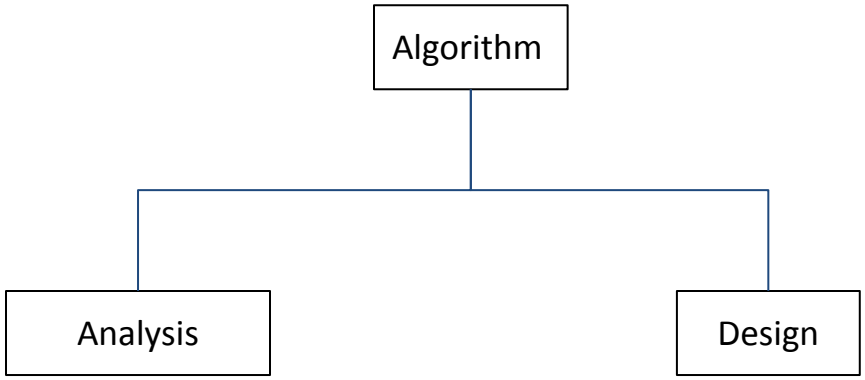
```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Pseudocode Conventions (In this Course)

- Indentation indicates block structure.
- The looping constructs while, for, and repeat-until and the if-else conditional construct have interpretations similar to those in C, C++, Java, Python, and Pascal.
- The symbol "//" indicates that the remainder of the line is a comment.
- Variables (such as i, j , and key) are local to the given procedure. We shall not use global variables without explicit indication.
- We access array elements by specifying the array name followed by the index in square brackets.
- We typically organize compound data into objects, which are composed of attributes. To specify the number of elements in an array A, we write A.length. We treat a variable representing an array or object as a pointer to the data representing the array or object. Sometimes, a pointer will refer to no object at all. In this case, we give it the special value NIL .
- We pass parameters to a procedure by value.
- The boolean operators "and" and "or" are short circuiting.

Other conventions will be discussed along the way in the course.

Key Aspects of Algorithms



- **Analysis:** predict the cost of an algorithm in terms of resources and performance.
- **Design:** creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.

Correctness and Efficiency

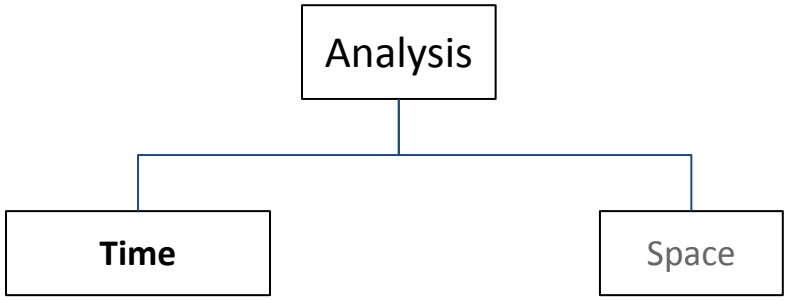
An algorithm is said to be **correct** if, for each input instance, it always finds out the correct output. To prove the correctness of an algorithm usually is not easy.

Note that to try many instances is not a proof of the correctness of the according algorithm. (This is called algorithm test.)

Usually, for one problem, there will be many correct algorithms to solve the problem. So we need to consider the **efficiency** of the algorithms.

This leads to ……

Algorithm Analysis



- **Time Complexity** is a function describing the amount of time required to run an algorithm in terms of the size of the input.

Computer – How "fast" is the algorithm?

- **Space Complexity** is a function describing the amount of memory an algorithm takes in terms of the size of input to the algorithm.

Computer – How much memory is used?

- **With the current hardware technology, we care more about time complexity than space complexity.**

RAM Model

- We need to consider machine-independent efficiency.
- Machine-independent algorithm design depends upon a hypothetical computer called the Random Access Machine or RAM.

Under RAM:

- ➢ Each simple operation (arithmetic operation (+, −, ×, /, =), if, call) takes exactly one time step.
- ➢ Loops and subroutines are not considered simple operations. Instead, they are the composition of many single-step operations. The time it takes to run through a loop or execute a subprogram depends upon the number of loop iterations or the specific nature of the subprogram.
- ➢ Each memory access takes exactly one time step. Further, we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk.

Notes about RAM

- RAM is not a real computer that the different steps may take different time. So the RAM model will not give the exact time calculation.
- But that model simplifies the calculation and it still gives us a very good way to understanding the efficiency of the algorithms.
- More importantly, the model gives us a method to estimate the run time of the algorithm, that is machine independent.

Runing Time Function

Since algorithm is used to treat the input, the running time depends on the length of input data. To estimate the running time, we define a function, T(n), of the data to represent the general running time, where n is the length of the data.

Insertion Sort Example

Example: Insertion Sort

Insertion	cost	times
for $j = 2$ to n	c_1	n
key $= A[j]$	c_2	$n-1$
$i = j - 1$	c_3	$n-1$
while $(i > 0)$ and $A[i] >$ key	c_4	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_5	$\sum_{j=2}^{n}(t_j - 1)$
$i = i - 1$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] =$ key //insert the key	c_7	$n-1$

Example (Continue)

In the algorithm, we use the summation formulas. In general,

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) + ... + f(n)$$

For examples

$\sum_{i=1}^{n} 1 = n,$

$\sum_{i=1}^{n} i = n(n+1)/2.$

$\sum_{i=0}^{n} i^2 = n(n+1)(2n+1)/6.$

Example (Continue)

In the example, t_j means the number of times the `while` loop test is executed for that value of j. c_i denotes the execute constant time for that step. We have

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j \\ &\quad + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1). \end{aligned}$$

Example (Continue)

In the example, t_j means the number of times the `while` loop test is executed for that value of j. c_i denotes the execute constant time for that step. We have

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j \\ &\quad + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1). \end{aligned}$$

Example (Continue)

Best case scenario: the numbers are sorted already in the right order,

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

We can express it as: $T(n) = an + b$, where a and b are constants.
It is thus a linear function of n.

Example (Continue)

Worst case scenario: the numbers are in reverse order, then $t_j = j$ and

$\sum_{j=2}^{n} j = n(n+1)/2 - 1$ and $\sum_{j=2}^{n}(j-1) = n(n-1)/2,$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1) \\ &= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n \\ &\quad - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

So the worst case running time is $T(n) = an^2 + bn + c$, where a, b and c are constants. It is a quadratic function.

General Rule

In general, we will focus on the complexity of **worst** case. The main reasons are

- The worst-case running time of an algorithm gives us an upper bound on the running time.
- For some algorithms, the worst case occurs fairly often.
- The “average” case is often roughly as bad as the worst case.The average case is not easy to define. Usually a probabilistic analysis is applied.

Order of Growth

In the example, the running time of the algorithm in best case is a linear function of n, while in worst case is quadratic function of n.

Now we want to make further simplifying abstraction. We call it the *rate of growth* or *order of growth* of the running time.

We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.

We write that insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced "theta of n-squared").

We usually consider one algorithm to be more efficient than another if its worst case running time has a lower order of growth.

Formal definition of order of growth will be discussed.

After Class

- Part I, Chapter 1 and Chapter 2
- Self Practice Exercises: 2.1-2; 2.1-4; 2.2-1; 2.2-2