Analysis of Algorithms

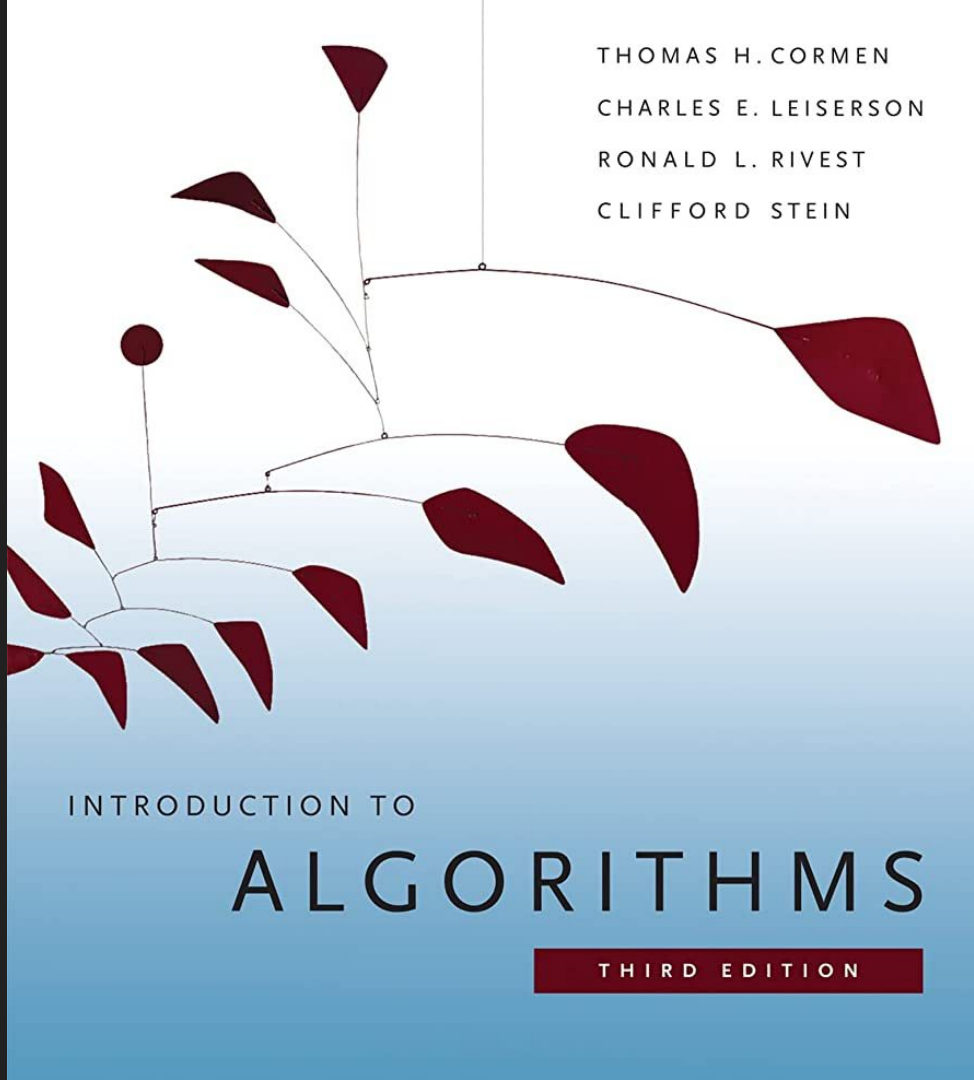
2023

Prepared by: Beimnet G.

Course Outline

- I Foundation
  - Introduction
  - Algorithm Analysis
  - Correctness of Algorithms
- II Graphs
- III Algorithm Design
  - Divide and Conquer
  - Greedy Algorithms
  - Dynamic Programming

Course Materials



Course Materials

- Books
  - T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, **Introduction to algorithms** , 3rd edition , MIT Press / McGraw-Hill, 2009.
  - A. Drozdek, Data structures and algorithms in C++, 4th edition, Cengage Learning, 2013.
- Other resources
  - Geeks for geeks
  - Youtube

Objective

Upon completion of the course, you should: -

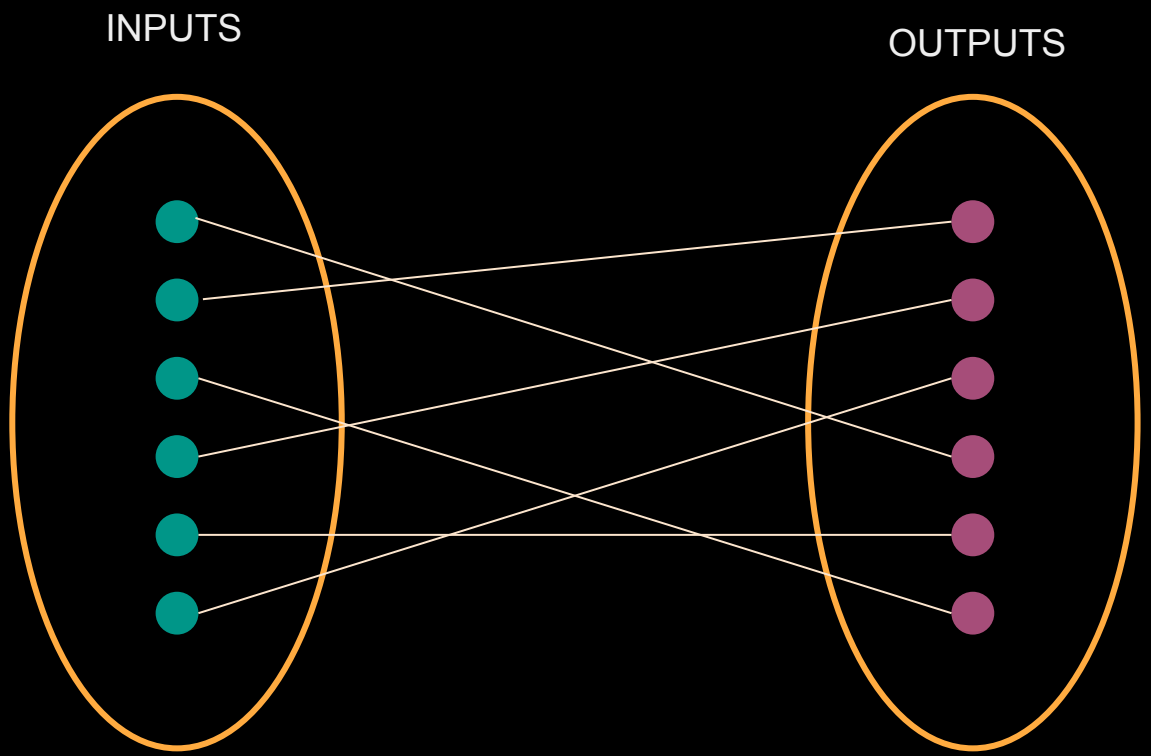
- Be able to analyze different algorithms: proof correctness and argue efficiency.
- Understand and be able to implement different algorithmic design techniques: **divide and conquer**, **dynamic** and **greedy**.
- Understand properties of graph, graph algorithm and different searching techniques in graphs.

Evaluation

- Class activities- 20%
- Test 1- 20%
- Test 2- 20%
- Exam- 40%

Chapter 1- Introduction

What is a problem?



A binary relation from problem inputs to correct outputs

We deal with problems with large general input spaces

We can’t specify every correct output for all possible inputs

Instead, provide a property that all correct outputs must satisfy.

What is an algorithm?

Algorithm *(n)*:

A procedure mapping each input to a single output (deterministic).

An algorithm solves a problem if it returns a correct output for every problem input

What are algorithms?

An algorithm is any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**.

It’s a tool for solving a well-specified computational problem.

For example: you might need to sort a sequence of numbers in a non-decreasing order.

So given the sequence (24,13,18, 21, 11, 9), a sorting algorithm will produce the sequence (9,11,13,18,21,24).

This specific input is referred to as an **instance**.

What are algorithms?

Computational problems often have more than one correct solution (i.e algorithm)

To determine which algorithm is best we need to determine which algorithm is most efficient when it comes to limited resources.

An algorithm is considered to be a correct, if for **every** input instance it terminates with the correct output.

Real world applications

The Internet

E-commerce

Manufacturing

Navigation

In this class, is there one or more pair of students with the same birthdays?

Solution?

Proposed Algorithm

Is it correct?

Is it efficient?

Proposed Algorithm

Maintain a record of names and birthdays (initially empty)

Interview each student in some order

    If birthday exists in record, return found pair!

    Else add name and birthday to record

Return None if last student interviewed without success

Characteristics of Algorithms

- **Finiteness**: An algorithm must always terminate after a finite number of steps.
- **Definiteness/Precision**: Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
- **Input**: An algorithm has zero or more inputs, i.e, quantities which are given to it initially before the algorithm begins.
- **Output**: An algorithm has one or more outputs i.e, quantities which have a specified relation to the inputs.

Characteristics of Algorithms

- **Effectiveness**: An algorithm is also generally expected to be effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time.
- **Uniqueness**: Results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- **Generality**: The algorithm applies to a set of inputs

Classification of Algorithms

By implementation

- Recursive **vs** iterative
- Serial **vs** parallel **vs** distributive
- Deterministic **vs** nondeterministic
- Exact **vs** Approximate

Classification of Algorithms

By design paradigm:

- Brute-force or exhaustive search
- **Divide and Conquer**
- **Greedy Algorithms**
- **Dynamic Programming**

Classification of Algorithms

By complexity:

- **Constant time**: if the time needed by the algorithm is the same, regardless of the input size. E.g. an access to an array element.
- **Linear time**: if the time is proportional to the input size. E.g. the traversal of a list.
- **Logarithmic time**: if the time is a logarithmic function of the input size. E.g. binary search algorithm.
- **Polynomial time**: if the time is a power of the input size. E.g. the bubble sort algorithm has quadratic time complexity.
- **Exponential time**: if the time is an exponential function of the input size. E.g. Brute-force search.

Hard Problems

This course focuses on efficient algorithms.

Efficiency is measured in regards to different computational resources.

Some problems, however, that have no known efficient solution. These problems are known as NP-complete.

What makes NP complete problems unique?

Hard Problems: why the interest?

First, although no efficient algorithm for an NP-complete problem has ever been found, nobody has ever proven that an efficient algorithm for one cannot exist. In other words, no one knows whether or not efficient algorithms exist for NP-complete problems.

Second, the set of NP-complete problems has the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them. This relationship among the NP-complete problems makes the lack of efficient solutions all the more tantalizing.

Third, several NP-complete problems are similar, but not identical, to problems for which we do know of efficient algorithms.

The Millennium Prize Problems- Million $ Questions

1. Birch and Swinnerton-Dyer Conjecture
2. Hodge Conjecture
3. Navier-Stokes Equations
4. **P versus NP**
5. Poincaré Conjecture
6. Riemann Hypothesis
7. Yang-Mills Theory

The Millennium Prize Problems- Million $ Questions

1. Birch and Swinnerton-Dyer Conjecture
2. Hodge Conjecture
3. Navier-Stokes Equations
4. **P versus NP**
5. ~~Poincaré Conjecture~~
6. Riemann Hypothesis
7. Yang-Mills Theory

P vs NP problem

A million dollar question

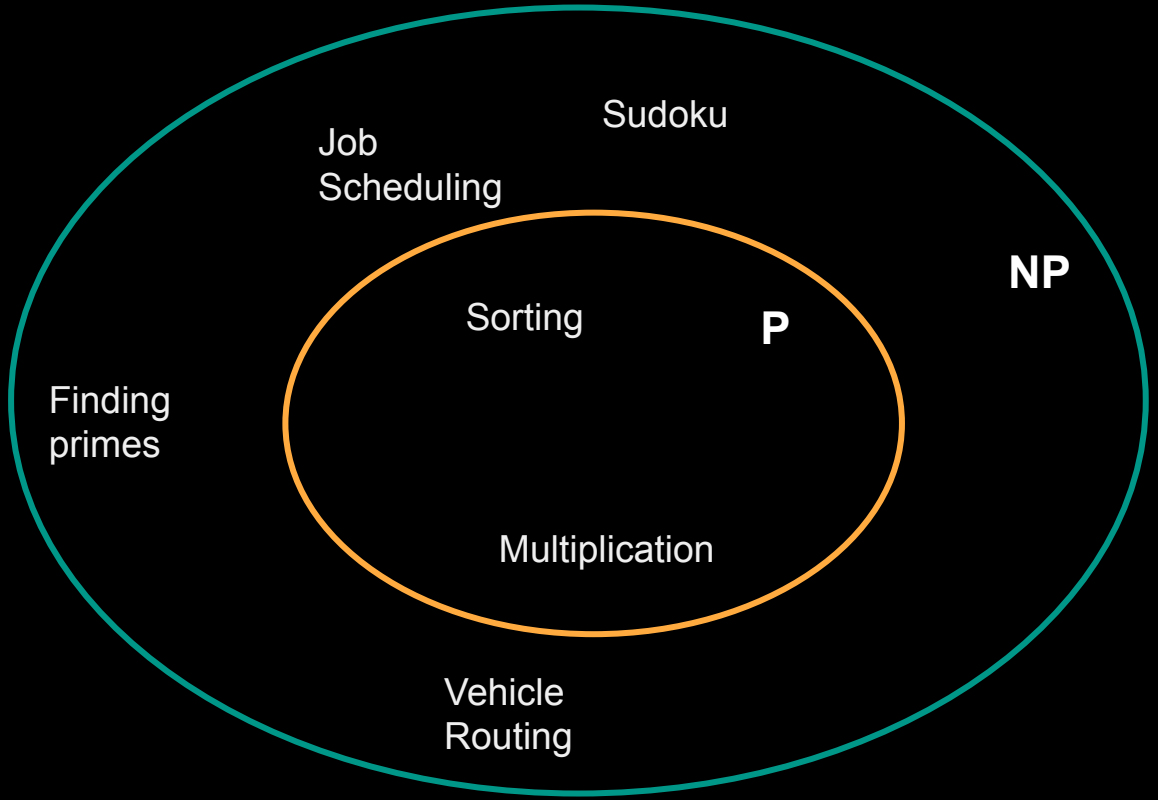
It asks whether every problem whose solution can be quickly verified can also be solved quickly.

How do we define quickly?

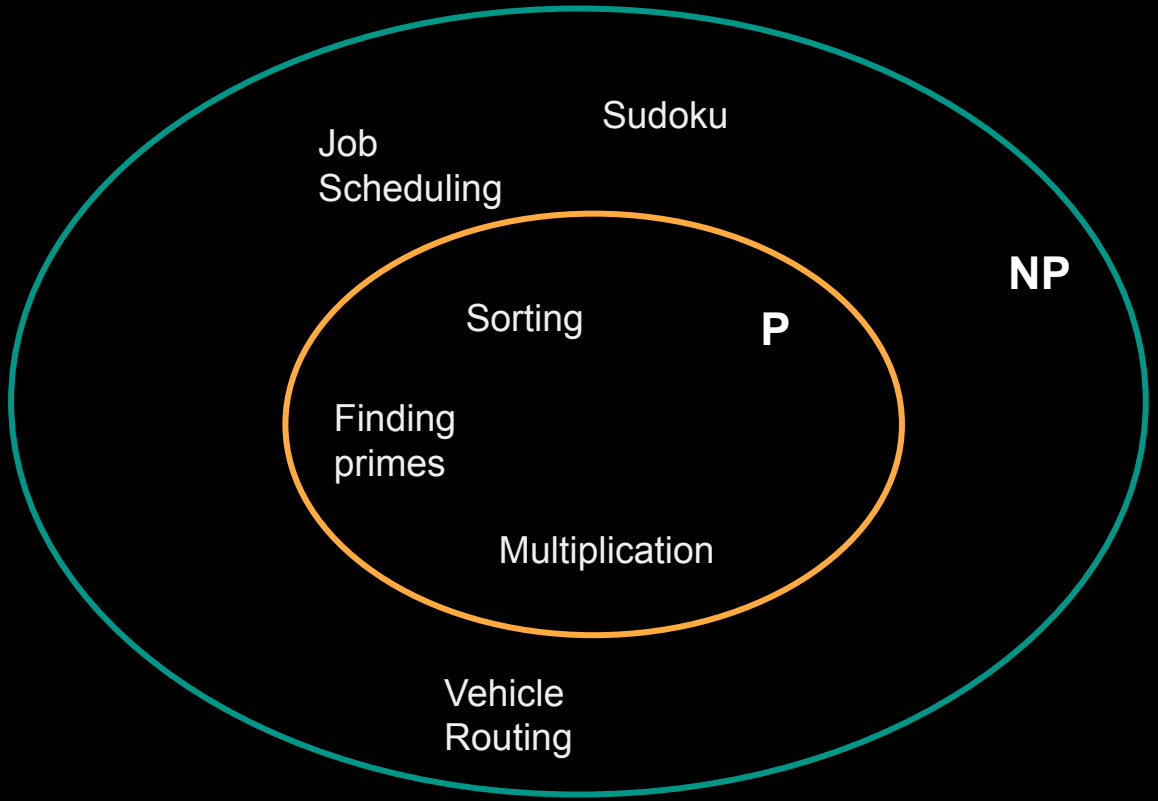
		P	NP
Problems	Easy to solve	✓	?
	Easy to verify	✓	✓

Problems: Sorting, GDC, prime?, factorization, jigsaw puzzle, chess

P vs NP problem



P vs NP problem



NP ?= P

Common Algorithms

- Find the GCD of 2 numbers

  Problem Statement
  Input: Two integers of a and b
  Output: An integer i such that i ≤ min(a, b) and i is the biggest number that can divide both a and b

- Find the LCM of 2 numbers

  Problem Statement
  Input: Two integers of a and b
  Output: An integer i such that i is the smallest number that is divisible by both a and b

Common Algorithms

- Linear search

  Problem Statement
  Input: A collection of n numbers {a0, a1, a2...an−1} and a single item key
  Output: The first index i such that ai = key. If no such item is found within the collection return −1

- Binary search

  Problem Statement
  Input: A sorted collection of n numbers {a1, a2, a3...an−1} and a single item key
  Output: The first index i such that ai = key. If no such item is found within the collection return −1

Common Algorithms

- **Optimum finding**
  - Find max

    Problem Statement
    Input: A collection of n numbers {a0, a1, a2...an−1}
    Output: The first index j such that $a_j \geq a_i \ \forall i \in 1..n$

  - Find min

Common Algorithms

- **Sorting**- insertion sort, bubble sort, selection sort ...

  Problem Statement
  Input: A sequence of n numbers {a1, a2, a3...an}
  Output: A permutation (reordering) {ai, aii, aiii, ..., am} of the input sequence such that {ai ≤ aii ≤ aiii ≤ ... ≤ am}

- **Peak finding**

  Input: A sequence of n numbers {a1, a2, a3...an}

  Output: An element a_i such that $a_{i-1} \leq a_i \geq a_{i+1}$

Exercise: Ethiopian Multiplication

- Ethiopian multiplication is a method of multiplying integers using only addition, doubling, and halving.

Method:

Take two numbers to be multiplied and write them down at the top of two columns.

In the left-hand column repeatedly halve the last number, discarding any remainders, and write the result below the last in the same column, until you write a value of 1.

In the right-hand column repeatedly double the last number and write the result below. stop when you add a result in the same row as where the left hand column shows 1.

Examine the table produced and discard any row where the value in the left column is even.

Sum the values in the right-hand column that remain to produce the result of multiplying the original two numbers together

Important Concepts

- Arithmetic and geometric sequences
- Complexity time (linear, quadratic, logarithmic…)