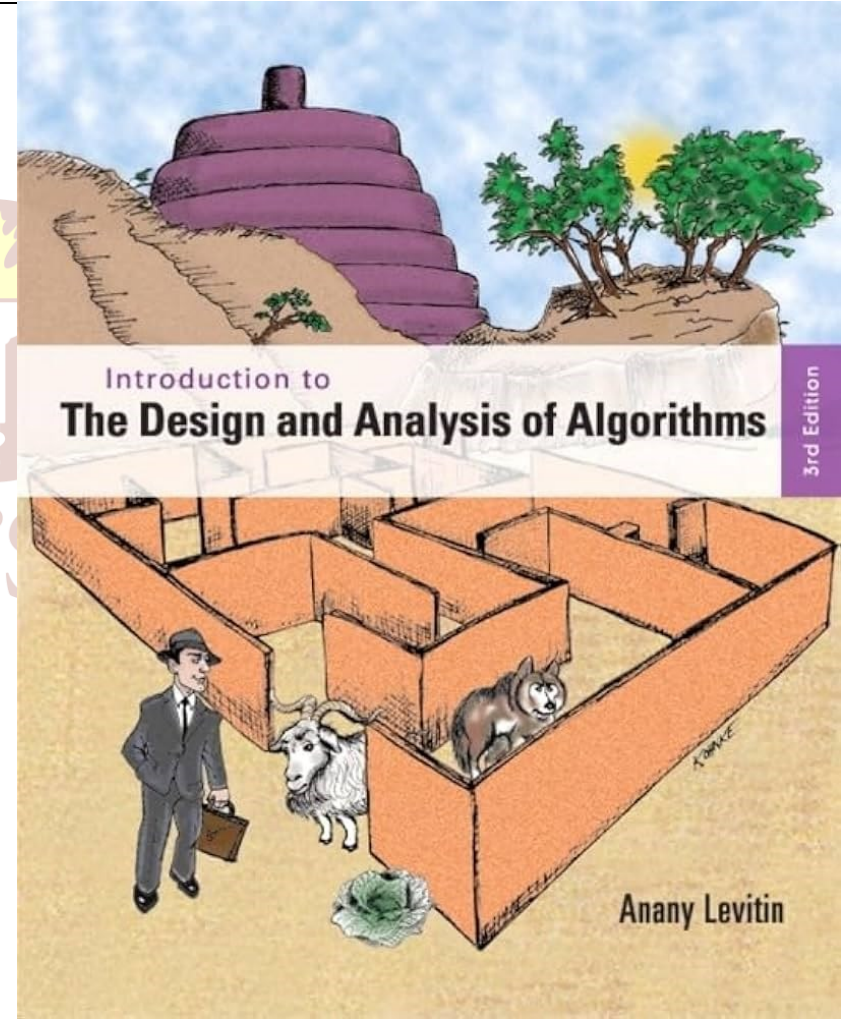



CS 07340 DESIGN AND ANALYSIS OF ALGORITHMS

WEEK 1 OVERVIEW


- Introduction
 - What is an algorithm?
 - Problem types
 - Data structures
 - Abstract Data Type (ADT)



INTRODUCTION

... an algorithm is a finite sequence of mathematically rigorous instructions, typically used to solve a class of specific problems or to perform a computation. Algorithms are used as specifications for performing calculations and data processing. ( Wikipedia)

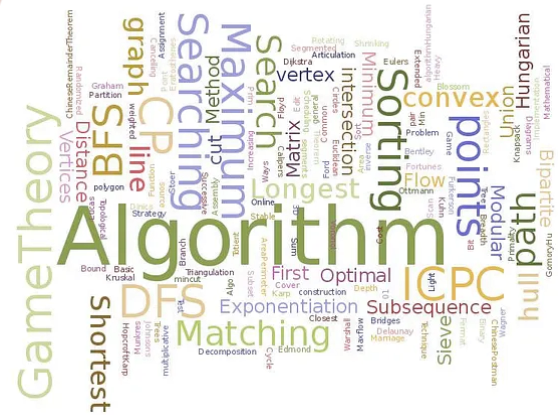
In this course, we want to analyze runtime behavior of some often used and important programs/algorithms.

In computer science, the analysis of algorithms is the process of finding the computational complexity of algorithms – the amount of time, storage, or other resources needed to execute them. ( Wikipedia)

THE ART OF COMPUTER PROGRAMMING

Donald Knuth's [The Art of Computer Programming](#) is one of the most important modern works in Mathematics and Computer Science. In Volume 1 Section 1.1, he writes:

The modern meaning for algorithm is quite similar to that of recipe, process, method, technique, procedure, routine, rigmarole, except that the word “algorithm” connotes something just a little different. Besides merely being a finite set of rules that gives a sequence of operations for solving a specific type of problem, an algorithm has five important features:



- 1) *Finiteness. An algorithm must always terminate after a finite number of steps.*
- 2) *Definiteness. Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.*
- 3) *Input. An algorithm has zero or more inputs: quantities that are given to it initially before the algorithm begins, or dynamically as the algorithm runs. These inputs are taken from specified sets of objects.*
- 4) *Output. An algorithm has one or more outputs: quantities that have a specified relation to the inputs.*
- 5) *Effectiveness. An algorithm is also generally expected to be effective, in the sense that its operations must all be sufficiently basic that they can in principle be done exactly and in a finite length of time by someone using pencil and paper.*

EUCLIDIAN ALGORITHM

Here is an example of an algorithm (in fact the one Knuth uses for demonstration). Let us verify that it has the five features:

Algorithm E (*Euclid's algorithm*). Given two positive integers m and n , find their *greatest common divisor*, that is, the largest positive integer that evenly divides both m and n .

E1. [Find remainder.] Divide m by n and let r be the remainder. (We will have $0 \leq r < n$.)

E2. [Is it zero?] If $r = 0$, the algorithm terminates; n is the answer.

E3. [Reduce.] Set $m \leftarrow n$, $n \leftarrow r$, and go back to step E1. ■

While Knuth's notation highlights the step-by-step nature of algorithms, we will mostly use pseudocode that looks almost like code when developing algorithms (or use actual code).

ALGORITHM *Euclid*(m, n)

//Computes $\gcd(m, n)$ by Euclid's algorithm

//Input: Two nonnegative, not-both-zero integers m and n

//Output: Greatest common divisor of m and n

while $n \neq 0$ **do**

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return m



A computer algorithm is a detailed step-by-step set of instructions for solving a problem using a computer. There may exist several algorithms for solving the same problem. A program is an implementation of one or more algorithms.

SIEVE OF ERATOSTHENES

Another early example of a simple algorithm is the sieve of Eratosthenes, used for generating consecutive primes not exceeding any given integer $n > 1$. It was probably invented in ancient Greece (200 BC).



ALGORITHM *Sieve(n)*

//Implements the sieve of Eratosthenes

//Input: A positive integer $n > 1$

//Output: Array L of all prime numbers less than or equal to n

for $p \leftarrow 2$ **to** n **do** $A[p] \leftarrow p$

for $p \leftarrow 2$ **to** $\lfloor \sqrt{n} \rfloor$ **do** //see note before pseudocode

if $A[p] \neq 0$ // p hasn't been eliminated on previous passes

$j \leftarrow p * p$

while $j \leq n$ **do**

$A[j] \leftarrow 0$ //mark element as eliminated

$j \leftarrow j + p$

//copy the remaining elements of A to array L of the primes

$i \leftarrow 0$

for $p \leftarrow 2$ **to** n **do**

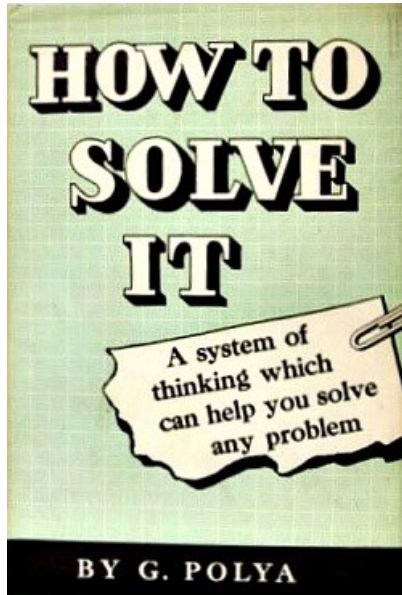
if $A[p] \neq 0$

$L[i] \leftarrow A[p]$

$i \leftarrow i + 1$

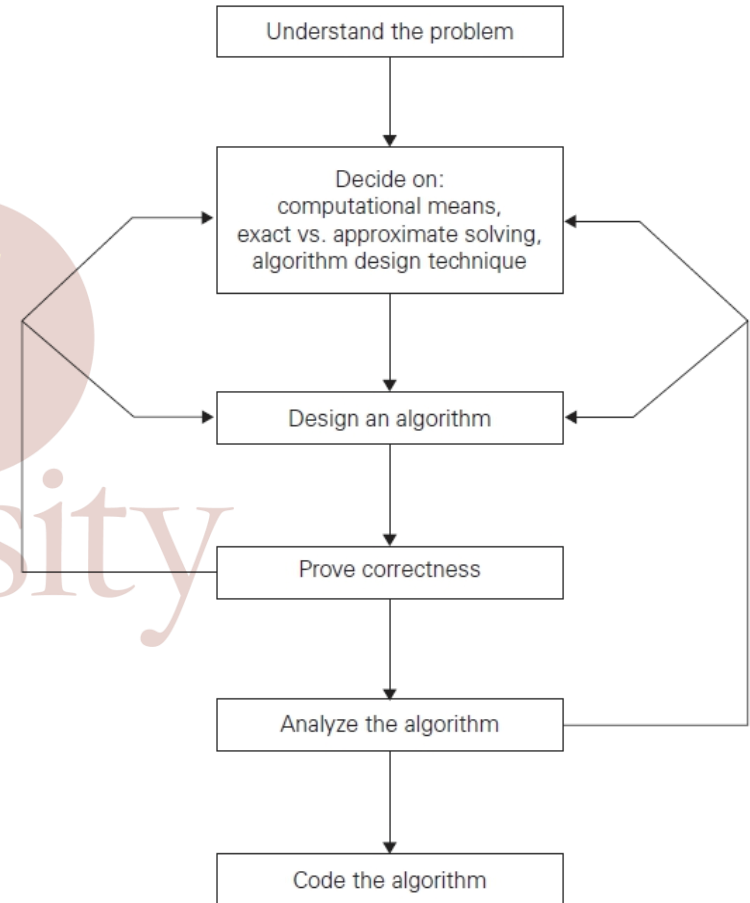
return L

FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

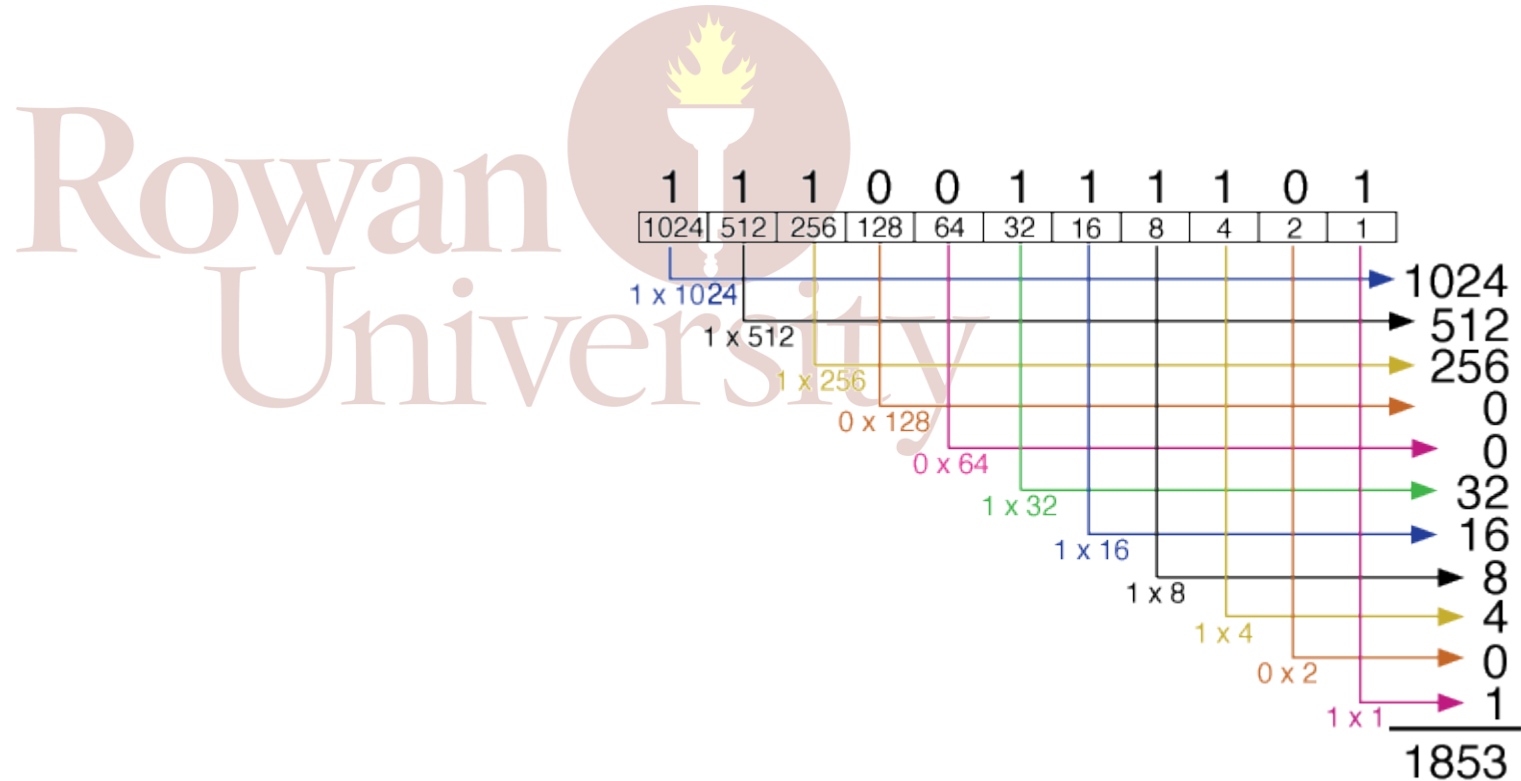


A variation of George Pólya's "How to Solve It" (1945) (a four-step method for solving problems) gives us a roadmap for our design and analysis of algorithms. Note that designing an

algorithm often comes with the necessity for designing data structures that go with it.



Example: Devise pseudocode for an algorithm that finds the binary representation of a positive decimal integer.



Example: Devise pseudocode for an algorithm that sorts a given list of integers by finding the smallest integer, setting it aside, reducing the size of the list, and repeating the process.



Example: Can the problem of computing the number π be solved exactly?



PROBLEM TYPES

- Sorting
- Searching
- String processing
- Graph problems
 - Shortest paths in a graph
 - Minimum spanning tree
- Combinatorial problems
 - Knapsack problem
- Geometric problems
- Numerical problems

and many, many more...



ALGORITHM DESIGN STRATEGIES

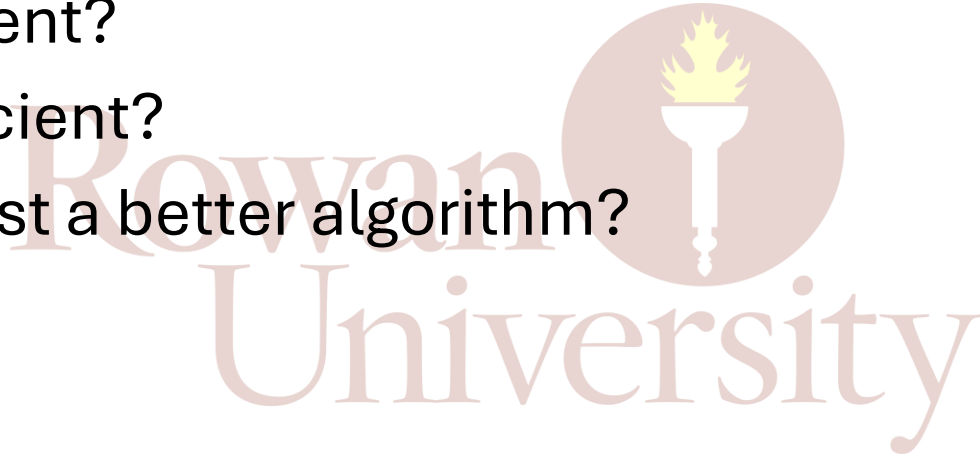
- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Greedy approach
- Dynamic programming
- Backtracking and Branch and bound
- Space and time tradeoffs



ANALYSIS OF ALGORITHMS

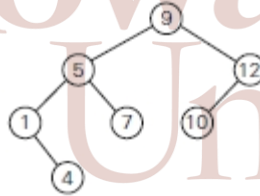
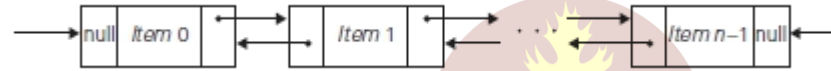
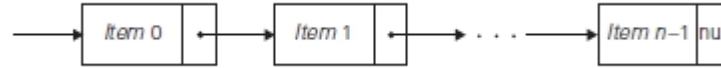
Once we created an algorithm, we need to ask ourselves:

- Is it correct?
- Is it time efficient?
- Is it space efficient?
- Does there exist a better algorithm?



FUNDAMENTAL DATA STRUCTURES

- Array
- Linked list
- Doubly linked list
- Graphs
- Trees
- Set
- Dictionary

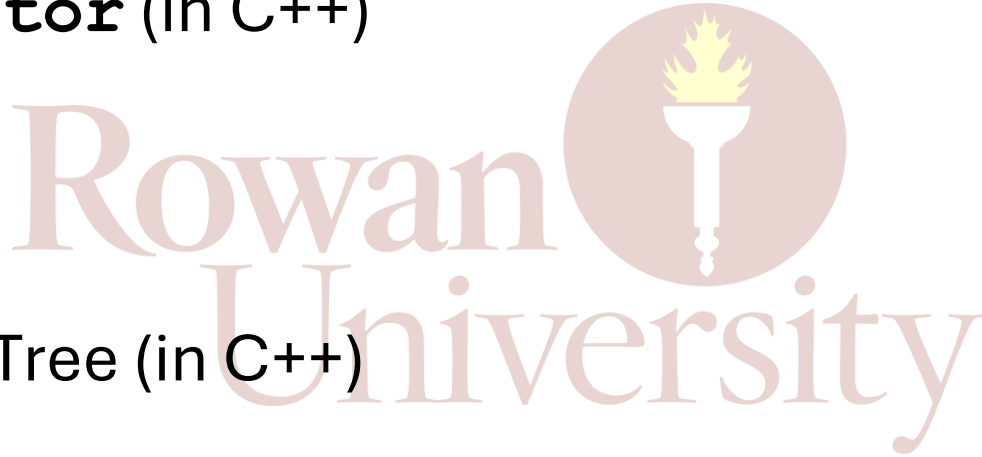


ABSTRACT DATA TYPE

An ***abstract data type*** (ADT) is a mathematical model for data types. We can think of an ADT as the definition of an interface (the name for “abstract class” in Java): it is a set of objects together with a set of operations. An ADT does not define how the set of operations is implemented. Implementations are data structures. Any given ADT may have several data structures associated with it. For instance, a search tree may be implemented as a Scapegoat tree, 2-3 tree, AVL tree, red-black tree, B-tree, etc.

Examples for ADTs and **a** corresponding data structure are

- List
 - **ArrayList** (in Java)
 - **std::vector** (in C++)
- Tree
 - AVL Tree
- Set
 - Red-Black Tree (in C++)
- Integer
 - **int** (tricky!)
- Floating Point Number
 - **float** (IEEE 754)



QUIZ TOPICS

- Computability
- ADT

