CSC2710 Analysis of Algorithms — Fall 2022
Unit 1 - Introduction to Algorithms & Data Structures

1.1 What is an Algorithm

- What is an algorithm?

  Definition 1 (Algorithm)**.** *A well-defined computation procedure that takes some value, or set of values, as input and produces a value, or set of values, as output.*

  – A sequence of computational steps (instructions) that convert an input to an output.

- You can also think of an algorithm as tool for solving a computational problem.
- We will often write a computational problem as an input and output relations.
  – Example: Let's consider the Greatest Common Divisor (GCD) problem. It is generally expressed as:

    Input: Two non-negative integers $m \geq 0$ and $n > 0$ **Output**: The greatest common divisor, d, of m and n. In other words, d is the largest integer such that there exists two positive integers a and b where $m = ad$ and $n = db$.

  – You will often see me use the following notation
    * $n \in \mathbb{Z}^+$ to denote the fact that n is a positive non-zero integer.
    * $m \in \mathbb{Z}^*$ to denote the fact that m is a positive integer or zero.
    * $a \in \mathbb{Z}$ to denote a is an integer.

  – Notice that both the inputs and the outputs are clearly described.
- A specific set of inputs for a problem is called an **instance**.
- There are many possible ways to solve an algorithm. Here is one way for the greatest common divisor of m and n (denoted GCD (m, n)).

```python
# Input: m is an integer, n is a positive integer
# Output: The greatest common divisor of m and n is returned
def EUCLID(m, n):
    while n != 0:
        r = m % n
        m = n
        n = r
    return m
```

- **Note:** In this course, we will be using Python code to describe algorithms. This is a good opportunity to learn a very useful and widely used programming language.
- Is this algorithm mechanically correct? Why? In other words, does its execution conclude?
  – We have that each time through the loop the number n gets closer to zero but, never becomes negative. Why?
- **Practice:** Can you come up with another algorithm to solve the greatest common divisor problem?
- **Practice:** What happens when $m < n$?
- **Practice:** Why does at least one of the two input numbers (n in our case) need to be non-zero?

1.2 Important Problem Types

- While there are several problem types investigated in Computer Science perhaps the most common are:
  1. Sorting
  2. Searching
  3. String Processing
  4. Graph Problems
  5. Combinatorial Problems
  6. Geometric Problems
  7. Numerical Problems

1.2.1 The Sorting Problem

- The sorting problem is defined as follows:

  Definition 2 (Sorting Problem)**.**

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

  Output: *A permutation* $\langle a'_1, a'_2, \ldots, a'_n \rangle$ *of the input such that* $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- Algorithms for the sorting problems potentially exhibit two interesting properties:
  1. **Stable Sort**: A sort that preserves the relative order of two equal elements in the input. In other words, if $a_i = a_j$ and $i > j$ then, the output permutation will have a_i before a_j.
  2. **In-Place Sort**: A sort is said to be in-place if the algorithm does not require any space *beyond* the space required for the input.
- The sorting problem itself is *extremely* well studied.

1.2.2 The Searching Problem

- The search problem is defined as follows

  Definition 3 (Search Problem)**.**

  Input: *A sequence of n keys* $\langle k_1, k_2, \ldots, k_n \rangle$ *and a search key* a.

  Output: *The index* i*, of* a *in the input sequence if there exists a* $k_i = a$ *or* `void` *otherwise.*

- Like the sorting problem, the searching problem is also very well studied.

1.2.3 String Problems

- We will also be concerned about algorithms that operate on strings.
  - A string is a sequence of symbols where each symbol is drawn from an alphabet.
  - An alphabet is a *finite* set of symbols.
  - Examples
    * 1011 is a string using the alphabet $\{0, 1\}$.
    * 13 is a string using the alphabet $\{0, 1, 2, \ldots, 9\}$.
    * "taco" is a string using the alphabet $\{a, b, c, \ldots, z\}$.

- A common problem in the string space is that of string matching.
- String matching comes up everywhere. The problem is formally defined as:

  Definition 4 (String Matching Problem)**.**
  ***Input**: An n symbol string*
  $$\sigma = \sigma_1\sigma_2\cdots\sigma_n$$
  and an $m \leq n$ symbol search string
  $$s = s_1 s_2 \ldots s_m.$$
  ***Output**: If there exists an index i such that*
  $$\sigma_i = s_1, \sigma_{i+1} = s_2, \ldots, \sigma_{i+m} = s_m,$$
  output i; otherwise, output `void`*.*

1.2.4 Other Problems

- Graph problems are problems that use a mathematical graph of vertices and edges. There are copious graph problems that come up all the time. Examples include:
  - Shortest distance between two points.
  - Planning a trip to visit a set of cities in the most efficient way possible (called the Traveling Salesman Problem).
  - Assigning a finite number of radio frequencies to regions such that neighboring regions don't use the same frequency (an application of a problem called vertex coloring).
- Combinatorial problems ask us to find "combinatorial objects"
  - Combinatorial objects are things like: permutations or subsets that satisfy some particular constraint.
- Geometric problems deal with points, lines, polygons, etc. We will briefly touch on these. Computer Graphics (CSC5210) makes heavy use of these.
- Numerical problems are those problems centered around solving problems in continuous mathematics.
  - Integration
  - Solving systems of equations.
  - Mathematica type problems.
- Numerical problems are investigated in depth in Numerical Analysis (MTH3725)

1.3 Fundamental Data Structures

- Data structures are *extremely* important in the study of algorithms.

  Definition 5 (Data Structure)**.** *A particular scheme for orgnazing related data items.*

- Algorithm efficiency is partly dictated by the way in which data is organized and accessed.
- We will breifly[1] review material from CSC2820.
- We will review
  - Linked lists
  - Stacks
  - Queues
  - Graphs
  - Trees
  - Dictionaries

1.3.1 Linked Lists

- A *list* is a finite sequence of data items arranged in a linear order.
- A linked list is a sequence of nodes that contain
  - data
  - pointer or pointers to other nodes in the list. If the pointer does not point to a node it is nil.
- There are two types of lists:
  - Singly Linked: The pointer in a nodes only points to its successor in the list.
  - Doubly Linked: The pointers in a node point to both the successor *predecesor* in the list.
- The first node of the list is a special node called the *head* [2].
- We may also optionally keep track of the tail of the list using a special *tail* node.
- **Practice:** How do you search for an item in the list?

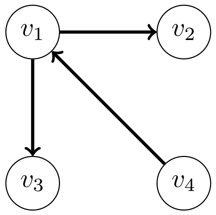
[1]Quickly – as I assume you are all comfortable with this material. We review just to clean out the cobwebs.
[2]In CSC2820 you made this a pointer.

1.3.2 Stack and Queues

- A stack is a data structure that mimics a stack of plates.
- Stacks support two operations:
  1. Push: Add an item to the top of the stack.
  2. Pop: Remove an item from the top of the stack.
- A stack is sometimes referred to as a LIFO structure.
  - LIFO stands for **L**ast **I**n **F**irst **O**ut.
- **Practice:** How would use a stack to reverse a string?
- A Queue is a data structure that mimics a line
- A queue supports two operations:
  1. Enqueue: Inserts an element at the end of the line (queue).
  2. Dequeue: Removes and element from the front of the line (queue).
- A queue is sometimes referred to as a FIFO structure.
  - FIFO stands for **F**irst **I**n **F**irst **O**ut.
- Queues also have a special form called a *priority queue* that comes up over and over again.
  - A priority queue is a queue that is ordered by a priority field as well as the normal FIFO ordering.
  - There are many technical applications but, you can think of it as a speed pass at amusement parks. If you hold the speed pass you are placed after all other existing speed pass holders but, before all of the other people waiting to ride the roller coaster.

1.3.3 Graphs

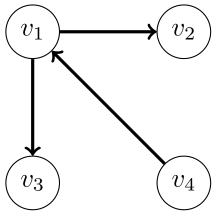
- A graph $G = (V, E)$ is a pair of two sets where,
  - V is the set of vertices (nodes) in the graph.
  - E is the set of edges in a graph. Every edge in the edgeset is of the form (u, v) where $u, v \in V$.
- The edgset indicates certain properties of the graph. For example,
  - If the edges $(u, v) \in E$ implies $(v, u) \in E$ then, the graph is called *undirected.*
    - This mean we can also say that u is *adjacent* to v.
  - If the edge $(u, v) \in E$ does not imply $(v, u) \in E$ then, the graph is called *directed*, or a *digraph* for short.
  - If $(v, u) \in E$ for all $v, u \in V$ then, the graph is called a *complete graph.*
  - A graph that is almost complete (missing a few edges) is called *dense.*
  - A graph that has very few edges is called *sparse.*
  - An edge $(u, u) \in E$ is called a *loop.*
- Graphs are commonly represented using two different possible data structures.
  1. Adjacency Matrix: A $|V| \times |V|$ matrix with a one in location that corresponds to $(u, v) \in E$. This is done fore every edge in E.
     - Example:



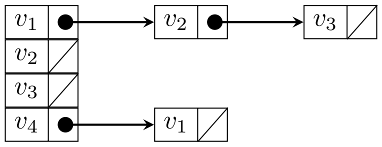
     - The adjacency matrix is:

$$\begin{array}{c c} & \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

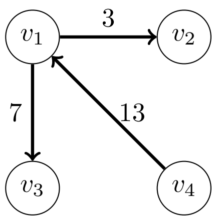
  2. Adjacency list: An array of $|V|$ linked lists (one for every element). Each element of the array represents the list of vertices adjacent to the element.
     - You may think of an adjacency list as representing the columns of the adjancency matrix where the entry for a given vertex is 1.
     - Example:



     - The adjacency list:



- **Practice:** What representation is better for a sparse graph?
- **Practice:** What representation is better for a complete, or almost complete, graph?
- Edges in graphs can also have weight. That is they can be labelled with a number.
  - Adjancecy matrix now uses the weight instead of a 1 when storing edite $(u, v) \in V$. If there is no edge between $(u, v) \in E$ then the adjancency matrix will use ∞ for the weight.
  - Example



  - The adjacency matrix is:

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} \infty & 3 & 7 & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ 13 & \infty & \infty & \infty \end{pmatrix} \end{array}$$

- We say there is a *path* from vertex u to vertex v in a graph if there is a sequence of connected edges that start with u and end with v.
  - Formally,
    $$\langle (u, x_1), (x_1, x_2), (x_2, x_3), \ldots (x_n, u) \rangle,$$
    where all $(x_i, x_j) \in E$.
  - A path is denoted as $u \leadsto v$.
  - If the graph is directed, the path is called a *directed path*.
- A path is called a *simple path* if all vertices along the path are distinct.
- The length of a path is the total number of vertices in the path minus one.
- A graph is *connected* if for every pair of vertices $u, v \in V$, $\exists$ either $u \leadsto v$ or $v \leadsto u$.
- A graph has a *cycle* if there is a path, of nonzero length, from a vertex back to itself.
- A graph is said to be *acyclic* if there are no cycles.

1.3.4 Rooted Trees

- Formally we define a (rooted) tree as a connected, directed, acyclic graph with exactly one vertex of indegree zero called the *root*.
- If we have a collection of directed, acyclic graphs we will call that set a *forest* with each element being a tree
- Tree terminology
  - ancestor: The ancestors of a vertex v are all vertices along the simple path from the root to v.
  - parent: The parent of a vertex v is the lowest ancestor in the simple path from the root to v.
  - child: Of a node v is any vertex w such that $(v, w) \in E$.
  - sibling: A vertex that shares a parent with at least one other vertex.
  - leaf: A vertex with no children.
  - descendant: A vertex v is a descendent of vertex u if u is an ancestor of v.
  - subtree: All the descendents of a vertex v in the tree.
  - depth: The depth of a vertex v is the length of the simple path from the root to v.
  - height: The height of a tree is the length of the longest simple path from root-to-leaf.
- We can impose more restrictions on a tree by creating ordered trees.
  - Ordered Tree: A rooted tree in which all the children of each vertex are ordered.
  - Example: A binary tree is orderd as all vertices have no more than two children either left or right.
    - We call a subtree that is rooted at the left child, the *left subtree*. We similarly define *right subtree*s.
    - There are also binary search trees which impose even more order namely
      - vertices in the left subtrees are always smaller than (or equal) to their parents.
      - vertices in the right subtrees are always larger than their parents.
    - One can generalize binary search tree to *multiway* search trees using other relationships. For example, greater than, less than, and equal.

1.3.5 Sets and Dictionaries

- A *set* is a collection of unordered distinct items called elements.
  - Example: $A = \{x \mid x^2 \bmod p = q\}$.
- Important set operations
  - Testing set membership.
  - Computing the union of two sets.
  - Computing the intersection of two sets.
- We define a *subset* A of set B ($A \subseteq B$) as a set of elements from B.
- We define a *proper subset* similiar to a subset with the added condition that $A \neq B$.
- If we let $S \subseteq U$ where U is a *universe* we can represent a set such that operations are quick. To do this we assign each element of U and index $1, 2, \ldots, |U|$. The set is then represented as a $|U|$-bit binary string. The i-th bit is 1 if and only if element i in U is in S.
- **Practice:** What other data structure can we use to represent a set?
- We define a dictionary data structure as any data structure that supports the following three operations
  1. Search
  2. Insert
  3. Delete
- We will encounter many dicitionary data structures in our travels.

Challenge Problem

- If you have not yet, install Python on your machine.
- Using Python lists (which are basically just arrays), implement a *binary search tree* (p. 286 in your textbook)
  - We will cover these in detail later in the semester, but for now simply implement
    * Insert
    * Remove
  - How can you use a list to represent a binary search tree? Given a root index `i`, which indices represent the left and right child? (you may want to Google this)
- Test your BST implementation by adding several integers into your tree, then removing a few of them. Print your list each time.