CSD 301
Final Project

Aims:

- *Understanding real used cases of data structures/algorithms in the course.*
- *Practicing three essential skills related to algorithms: design, analyze, and implement.*
- *Making decisions (with tradeoffs) to choose the optimal solution based on the actual conditions.*

Description:

There are five topics with vague statements. Each group (5-6 students) chooses one topic to research during the course. First, the topic needs to restate in a more appropriate form (re-define input, output, constraints, scope, etc.). Then the groups propose at least two (should be three) solutions. Finally, the solutions should be analyzed and compared to show their advantages/disadvantages.

Requirements:

1. *Submit: (a) Member list with each member's work assignment, (b) a complete report, (c) source code of the proposed solutions.*
2. *Report: 12-15 pages, including (a) introduction, (b) problem statement with clear input, output, constraints, scope, (c) proposed algorithms with flowchart, pseudo-code, computational complexity, (d) comparing algorithms, showing advantages/disadvantages, (e) when/how each proposed algorithm should be chosen (which types of data, applications).*
3. *Source code: clean OOP code with explicit description/comments. The solutions should be in a visual form with an acceptable interface (should not be the console or command line).*
4. *Presentation: each group presents for a maximum of 60 minutes (should be 40min for presentation, 5min for demo/test, and 15min for Q&A)*

Timeline:

- *Week #3: First meeting and choose the topic.*
- *Week #5: Second meeting, supporting, and/or changing/updating topics.*
- *Week #7: Third meeting, supporting and discussing.*
- *Week #9, 10: Presentation, demo, and submit works.*

Topic 1: Knapsack Problem

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
The knapsack problem is a combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to complex problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications

In many cases of resource allocation, along with some constraints, the problem can be derived similarly to the Knapsack problem. Following is a set of examples:

- *Finding the least wasteful way to cut raw materials*
- *Cutting stock problems*

Problem Scenario

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store, and the weight of ith item is wi and its profit is pi. What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize profit.

Topic 2: Finding way for shipper

Problem Statement

A shipper needs to visit all the customers from a list, where distances between all the customers are known and each customer should be visited just once. What is the shortest possible route that he visits each customer exactly once and returns to the origin place (his house or company)?

Approach

This problem is the most notorious computational problem. We can use a brute-force approach to evaluate every possible tour and select the best one. For n number of vertices in a graph, there are (n - 1)! number of possibilities.
Instead of brute force using a dynamic programming approach, the solution can be obtained in less time, though there is no polynomial time algorithm.

Let us consider a graph G = (V, E), where V is a set of places and E is a set of weighted edges. An edge e(u, v) represents that vertices u and v are connected. The distance between vertex u and v is d(u, v), which should be non-negative.

Suppose we have started at place #1 (corresponding with customer #1), and after visiting some customers, now we are in place #j. Hence, this is a partial tour. We certainly need to know j, since this will determine which customer is most convenient to visit next. We also need to know all the customers visited so far so we do not repeat them. Hence, this is an appropriate sub-problem.

Topic 3: Convex hull

Problem statement*: Given a set of points in the plane. The convex hull of the set is the smallest convex polygon that contains all its points.*

Input
The input contains an integer n (1≤n≤1000). Then follow a list of n points, one per line, each of the form x y. Coordinates are integers with absolute values bounded by 10000. The input is terminated by a case beginning with 0.
Output
output is a polygon giving the convex hull of the input points in the same format as the input. This polygon should not contain any collinear edges and should be given in counterclockwise order. Any cyclic shift of the convex hull is acceptable.

Topic 4: Place Autocomplete

The Place Autocomplete service is a web service that returns place predictions in response to an HTTP request. The request specifies a textual search string and optional geographic bounds. The service can be used to provide autocomplete functionality for text-based geographic searches by returning places such as businesses, addresses and points of interest as user types.

The Place Autocomplete service can match full words and substrings, resolving place names, addresses, and numbers. Applications can therefore send queries as the user types to provide on-the-fly place predictions.
The returned predictions are designed to be presented to the user to aid them in selecting the desired place.

Certain parameters are required to initiate a Place Autocomplete request. As is standard in URLs, all parameters are separated using the ampersand (&) character. The list of parameters and their possible values are enumerated below.

Required parameters
Input: *The text string on which to search.*
Output: *The Place Autocomplete service will return matched candidate(s)*

Dataset: https://batch.openaddresses.io/location/4135

Example 1:(input, output) = ("227 Nguyễn Văn", "227 Nguyễn Văn Cừ, Quận 5, TPHCM")
Example 2:(input, output) = ("Đại học F", "Đại học FPT, Lô E2a-7, Đường D1, Đ. D1, Long Thạnh Mỹ, Thành Phố Thủ Đức, Thành phố Hồ Chí Minh 700000")
Example 3: (input, output) = ("96 Nguyễn Thị Minh Kggai", "96 Nguyễn Thị Minh Khai, Quận 3, TPHCM")

Topic 5: String searching algorithm

String-searching algorithms, sometimes called string-matching algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text.

A basic example of string searching is when the pattern and the searched text are arrays of elements of an alphabet (finite set) Σ. Σ may be a human language alphabet, for example, the letters A through Z and other applications may use a binary alphabet (Σ = {0,1}) or a DNA alphabet (Σ = {A,C,G,T}) in bioinformatics.

In practice, the method of feasible string-search algorithms may be affected by the string encoding. In particular, if a variable-width encoding is in use, then it may be slower to find the Nth character, perhaps requiring time proportional to N. This may significantly slow some search algorithms. One of many possible solutions is to search for the sequence of code units instead, but doing so may produce false matches unless the encoding is specifically designed to avoid it.

***Hints** (3 Algorithms):*
Rabin–Karp
Knuth–Morris–Pratt
Boyer–Moore