Assignment 1

General

General

You must read fully and carefully the assignment specification and instructions.

- **Course:** COMP20007 Design of Algorithms @ Semester 1, 2024
- **Deadline Submission:** Tuesday, 9th April 2024 @ 11:59 pm
- **Course Weight:** 10%
- **Assignment type:** individual
- **ILOs covered:** 2, 3, 4
- **Submission method:** via ED

Purpose

The purpose of this assignment is for you to:

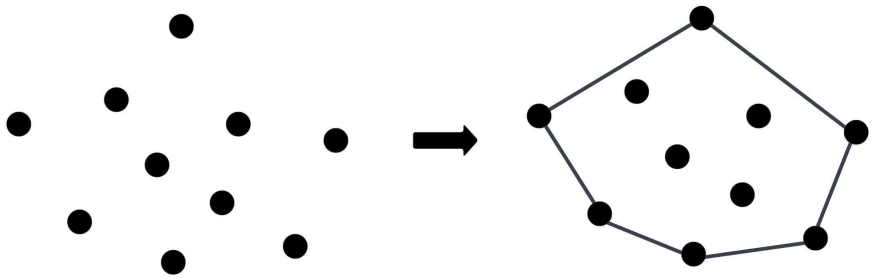
- Design efficient algorithms in pseudocode.
- Improve your proficiency in C programming and your dexterity with dynamic memory allocation.
- Demonstrate understanding of a representing problems as graphs and implementing a set of algorithms.

Convex Hull

What is a convex hull?

A convex hull is a concept from geometry and computational geometry that refers to the smallest convex set that contains a given set of points in a Euclidean space. Imagine you have a set of nails sticking out from a board; the convex hull is the shape you would get by stretching a rubber band around all the nails and letting it snap tight around them. The boundary created by the rubber band defines the convex hull.

More formally, a convex set is a set of points in which, for any two points within the set, the line segment connecting them lies entirely within the set. The convex hull of a set of points is then the smallest convex set that contains all those points. It is often visualized in two dimensions (2D) but can be extended to higher dimensions as well.



Computing the convex hull is a foundational problem in computational geometry because it is a precursor to solving many other problems, such as finding the closest pair of points, and solving various optimization problems. Several algorithms exist for computing the convex hull, with their efficiency depending on the specific dimensions and characteristics of the point set. Examples of such algorithms include Graham's scan and Jarvis's march (also known as the gift wrapping algorithm) among others.

Applications

Convex hulls have numerous real-world applications. Some of the typical applications include:

- **Image Processing and Computer Vision**: Convex hulls are used in image processing for object detection, recognition, and segmentation. They help in identifying the outer boundary of objects and separating foreground objects from the background.
- **Geographic Information Systems (GIS)**: In GIS, convex hulls are utilized for spatial analysis, such as identifying the boundary of a geographic region or finding the convex hull of a set of spatial data points.
- **Robotics and Path Planning**: Convex hulls play a vital role in robotics for path planning, obstacle avoidance, and navigation. They help robots efficiently navigate through complex

environments by identifying obstacles and determining safe paths.

- **Collision Detection in Physics Simulations**: In physics simulations and video games, convex hulls are used for collision detection between objects. They help determine if two objects intersect or collide with each other.
- **Pattern Recognition**: Convex hulls are employed in pattern recognition tasks, such as handwritten character recognition, signature verification, and fingerprint matching. They aid in extracting important features and reducing the dimensionality of data.

Convex Hull Computation Algorithms

Jarvis March

- **Initialization**:
  - Start by selecting the leftmost point among the given points as the starting point of the convex hull. If there are multiple leftmost points, choose the one with the lowest y-coordinate.
  - Add this point to the convex hull.
- **March**:
  - Iterate through all the points to find the next point on the convex hull.
  - To find the next point, start from the current point and choose the point that has the largest counterclockwise angle with respect to the current point.
  - This is done by comparing the slopes of the line segments formed by the current point and each of the other points.
  - The point with the largest counterclockwise angle is the next point on the convex hull.
  - Add this next point to the convex hull.
- **Termination**:
  - Repeat the main loop until the next point on the convex hull is the same as the starting point.
  - Once the next point becomes the starting point, the algorithm terminates.
- **Output**:
  - The output of the algorithm is the list of points that form the convex hull, ordered in counterclockwise order.

The time complexity of the Jarvis March algorithm is $O(nh)$, where n is the number of input points and h is the number of points on the convex hull. In the worst case, where all points are on the convex hull, the time complexity is $O(n^2)$.

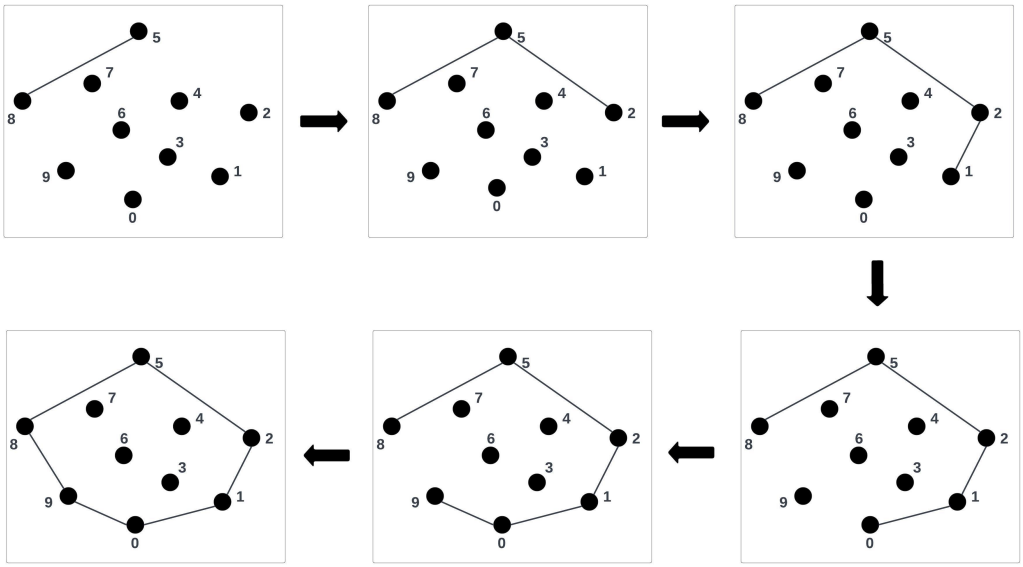
Pseudocode

```
JarvisMarch(points):
    // Ensure there are at least 3 points
    if length(points) < 3:
        return empty convex hull

    // Initialize an empty list to store convex hull points
    convexHull <- empty list

    // Find the leftmost point (pivot) among the given points
    leftmost <- leftmost_point(points)

    // Start from the leftmost point
    current <- leftmost
    repeat:
        // Add current point to the convex hull
        add current to convexHull

        // Find the next point 'nextPoint' such that it forms a counterclockwise turn
        // with the current point and any other point in the set
        nextPoint <- points[0]
        for each point in points:
            if nextPoint = current or orientation(nextPoint, current, point) = counterclockwise:
                nextPoint <- point

        // Set 'nextPoint' as the current point for the next iteration
        current <- nextPoint

    // Repeat until we return to the starting point (leftmost)
    until current = leftmost

    // Return the list of points in the convex hull
    return convexHull
```

Notes

- **`leftmost_point(points)`** returns the point with the lowest x-coordinate (or the bottom-most point satisfying this condition in case of ties).
- **`orientation(p, q, r)`** is a function that determines the orientation of three points. It returns:
    - 0 if the points are collinear.
    - 1 if they form a clockwise turn.
    - 2 if they form a counterclockwise turn.

Example



In the above example, the starting point is identified as point 8. We examine each point in relation to point 8, determining which one results in the greatest counterclockwise turn compared to the rest. To put it more straightforwardly, the goal is to locate a point that ensures the line segment from the previous to the current point remains on the left side, while all other points are positioned to the right. As observed, point 5 is the initial choice because the line segment from point 8 to point 5 generates the most significant counterclockwise angle in comparison to the others. Following this logic, when positioned at point 5, we evaluate all other points and select point 2 as the one that yields the most pronounced counterclockwise turn. The process continues in this manner until it returns to the starting point, which is point 8.

Graham's scan

- **Initialization**:
  - Start by selecting the point with the lowest y-coordinate (or the leftmost point in case of ties) as the pivot point.
  - Sort the remaining points based on their polar angles with respect to the pivot point. If two points have the same polar angle, the one closer to the pivot point should come first.
- **Scan**:
  - Iterate through the sorted points and add them to the convex hull one by one.
  - For each point, check if it forms a counterclockwise turn with the two previously added points on the convex hull.
  - If it forms a counterclockwise turn, add the point to the convex hull. Otherwise, remove the last point from the convex hull until a counterclockwise turn is formed.
  - Repeat this process until all points are scanned.
- **Termination**:
  - Once all points are scanned, the convex hull is complete.
- **Output**:

- The output of the algorithm is the list of points that form the convex hull, ordered in counterclockwise order.

The time complexity of Graham's scan algorithm is $O(n \log n)$, where n is the number of input points. This complexity arises from the sorting step required to order the points based on their polar angles. The scanning step, where each point is checked and added to the convex hull, takes linear time.

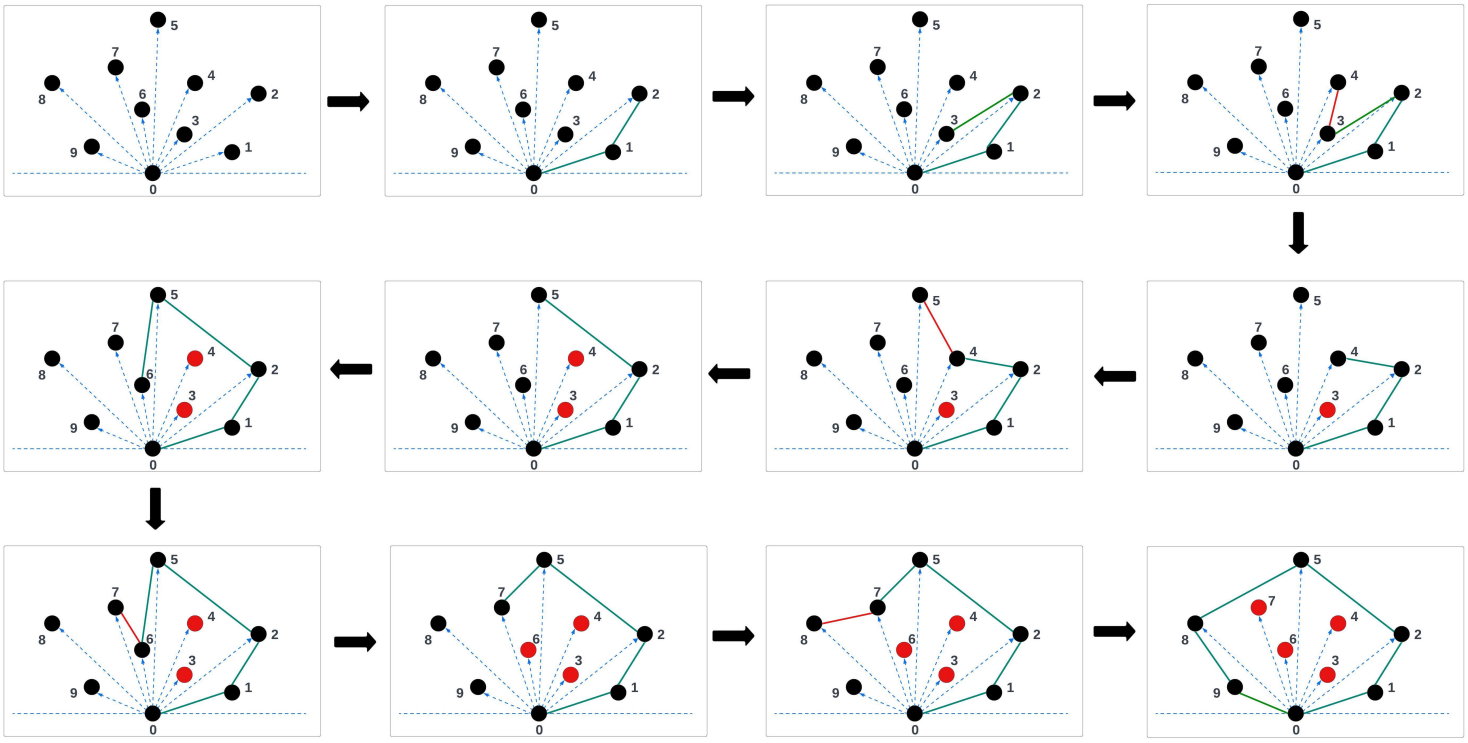
```
GrahamScan(points):
    // Ensure there are at least 3 points
    if length(points) < 3:
        return empty convex hull

    // Find the point with the lowest y-coordinate
    lowest = point with lowest y-coordinate in points
    // Sort the points based on their polar angles with respect to the lowest point
    sort points by polar angle with respect to lowest

    // Initialize an empty stack to store convex hull points
    stack = empty stack

    // Push the first three points to the stack
    push points[0] to stack
    push points[1] to stack
    push points[2] to stack

    // Iterate over the remaining points
    for i = 3 to length(points):
        // While the current point and the two points below the top of the stack
        // make a non-left turn, pop the top of the stack
        while orientation(second_top(stack), top(stack), points[i]) != counterclockwise:
            pop top of stack

        // Push the current point to the stack
        push points[i] to stack

    // The stack now contains the convex hull points
    return stack
```

Notes

- **`lowest`** represents the point with the lowest y-coordinate in the set of points.
- **`sort points by polar angle with respect to lowest`** sorts the points based on their polar angles with respect to the lowest point. If two points have the same polar angle, the one closer to the lowest point comes first.
- **`second_top(stack)`** returns the second topmost point in the stack.
- **`orientation(p, q, r)`** is a function that determines the orientation of three points. It returns:
  - 0 if the points are collinear.
  - 1 if they form a clockwise turn.

- 2 if they form a counterclockwise turn.

Example



In the above example, point 0 is identified as the point with the lowest y-coordinate. Subsequently, the other points are arranged based on their polar angles relative to point 0. Initially, points 0, 1, and 2 are placed into the stack. The addition of point 3 to the stack occurs because it facilitates a left turn in relation to the preceding two points (points 2 and 1). Conversely, point 4 induces a right turn compared to the last two points in the stack (points 3 and 2), leading to the removal of point 3 from the stack. This adjustment allows for a left turn when point 4 is added atop point 2 in the stack. The algorithm continues in this manner, examining each point in turn. Upon completion, the stack holds all points forming the convex hull.

Task 1: Convex Hull Computation with Doubly Linked Lists

Part A

Implement the **Jarvis' March** algorithm to compute the convex hull. Find the leftmost point. Then iterate over the remaining points, selecting the most counterclockwise point relative to the current point until you come back to the starting point.

Requirements

- **Data Structure**: Implement a doubly linked list to store the points of the convex hull. Each node should store the x and y coordinates of a point and pointers to the next and previous nodes.
- **Input Format**: The input will be a CSV file where the first line indicates the total number of points. Subsequent lines represent points with their x and y coordinates separated by a space (e.g., `x y`).
- **Output Format**: Your program should output two sequences of points forming the convex hull: one in clockwise order and the other in counterclockwise order. This should be done by traversing your doubly linked list to print the points in both clockwise and counterclockwise order, in both cases starting from the first point added to the hull. This may require starting from a specific point and moving in one direction for clockwise and the opposite direction for counterclockwise. This should be output to the console (stdout).

Part B

Implement the **Graham's Scan** algorithm to compute the convex hull. Find the bottom-most point, sort the points, and proceed with the scan, pushing and popping points from the stack (which will be your doubly linked list in this case) to ensure you maintain a convex hull throughout.

Requirements

- **Data Structure**: Implement a stack with doubly linked lists. Utilize this stack in Graham's Scan algorithm to store the convex hull points. Each node should store the x and y coordinates of a point and pointers to the next and previous nodes.
- **Input Format**: The input will be a CSV file where the first line indicates the total number of points. Subsequent lines represent points with their x and y coordinates separated by a space (e.g., `x y`).
- **Output Format**: Your program should output two sequences of points forming the convex hull: one in clockwise order and the other in counterclockwise order. This should be done by traversing your doubly linked list to print the points in both clockwise and counterclockwise

order, in both cases starting from the first point added to the hull. This may require starting from a specific point and moving in one direction for clockwise and the opposite direction for counterclockwise. This should be output to the console (stdout).

Part C

Evaluate both algorithms through experimental analysis by quantifying the total basic operations across various input scales and configurations. Consider creating input sets of at least three distinct sizes, each under three differing distribution conditions: random, points on a circle, and random points contained within a set of points making up a simple hull.

You should use the following basic operations for each algorithm:

- Jarvis' March - A comparison between the angles of points.
- Graham's Scan - A comparison between the angles of points during the sort.

Reporting: Write a report including a discussion on the choice of data structure, the experimental evaluation, and conclusions drawn from the comparisons. Include any assumptions or simplifications made in your implementations.

Submission Guidelines

- Submit your C source code files with appropriate comments explaining the algorithms and data structures used.
- Your report should be in PDF format, including your findings from the experimental evaluation and any observations regarding the performance of the two algorithms.

Grading Criteria

- Correctness of the implemented algorithms and adherence to the requirements.
- Efficiency and proper use of the doubly linked list for storing the convex hull.
- Clarity of the report, including the depth of the experimental evaluation and the analysis of the results.
- Code readability, structure, and documentation.

Task 1: Assignment Submission

Upload your solution to Task 1!

Input Formats

The test cases for each problem are present in the `test_cases` folder, and the expected answers for each of these test cases are present in the `test_case_answers` folder.

Part A

The input for 1A is the number of points, followed by each point.

Example

```
3 (Number of points we are constructing the hull for)
0 0 (Point 1)
5 6 (Point 2)
2 3 (Point 3)
```

Part B

The input for Part B is the same as Part A.

Part C

Add your answers to Part C as a pdf called `written-tasks.pdf`. Submit your assignment using the "Mark" button. You may submit as many times as you like.

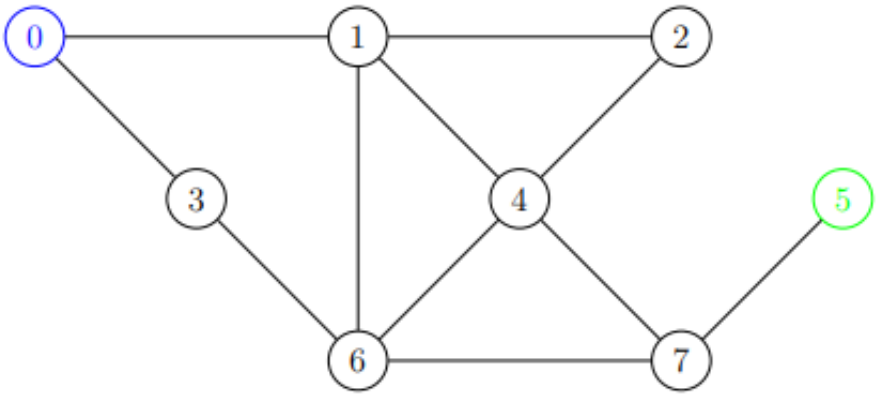
Task 2: Baldur's Door

Baldur's Door is a new computer role-playing game based on the setting of a popular table-top game called Delves & Drakes, the game features an extensive variety puzzles and mechanics which critics have lauded as interesting and allowing for diverse modes of play. The game's characters take on quests which lead to puzzles.

Part A

In the early quests of the game, the town's tavern is advertising exciting new flavours for their soups. The cook has promised 500 silver pieces for each novel ingredient on their list gathered. Though appearing generous for ingredient gathering in the nearby forest, the party cleric noticed that each ingredient requires crawling between poison bushes. The party rogue has the smallest build and will likely be able to crawl through the bush without being scratched too badly - but will surely be afflicted. Once the poison takes hold, each step taken will passively do a fixed amount of damage. The cleric's healing spell can only be casted once the party rogue has exited the narrow space. Since each casting of the healing spell requires expensive magical materials, you will have to find the route that reaches the exit in the least possible steps.

An example of the forest pathways are shown below, where each edge represents a step and each node represents a location the rogue can step to:



The forest layout above would be represented by the input to your program:

```
8
11
0
5
0 1
0 3
1 2
1 4
1 6
2 4
```

```
3 6
4 6
4 7
5 7
6 7
```

Where:

- the first line (`8`) is the number of locations the rogue can possibly step,
- the following line (`11`) is the number of connections between locations,
- the third line (`0`) is the location the rogue starts at,
- the fourth line (`5`) is the location the rogue exits the space and is in range to be healed, and
- all following lines are pairs, indicating an undirected connection between the two locations.

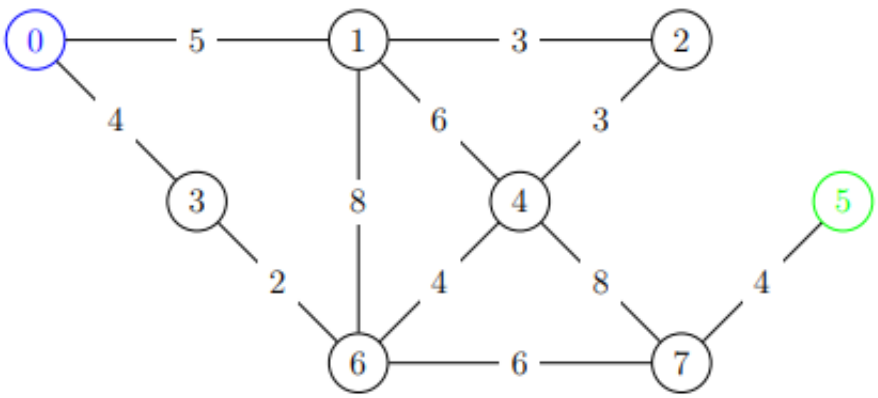
You may assume the list is sorted numerically by the first location and then (where the first location is equal) by the second location. You may also assume locations are always equal to or greater than 0 and numbered less than the number of locations.

The output should be the damage taken - assuming one point of damage is taken per step.

Part B

After hearing the ease with which our protagonists were able to deal with the poison forest, a local merchant staying at the tavern asks if the party would be interested in a challenge. The merchant had come into possession of a number of lair maps that showed the locations of all the traps in each lair. The merchant confessed they did not have any non-mercantile skills but had managed to purchase information on how to disarm each trap. After discussing the skills the party held, the merchant explained the cost of the materials they'd need to disarm each trap they had the skills to disarm and marked this total cost on each pathway. To preserve the most treasure, you'll need to find the cheapest path through the lair.

Here is an example of one of these maps:



This map would be represented using the following format:

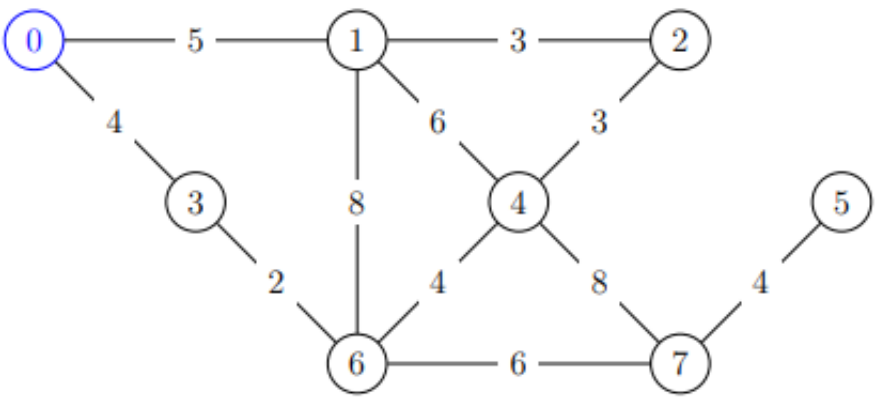
```
8
11
0
5
0 1 5
0 3 4
1 2 3
1 4 6
1 6 8
2 4 3
3 6 2
4 6 4
4 7 8
5 7 4
6 7 6
```

Which is the same as the format for Part A, but each edge specified includes an additional third number describing the cost of disarming the trap. You may also assume that all costs will be non-negative.

Part C

To reward the adventurers for their extensive help connecting the merchant with their extensive treasures, they share their own prior connections. The documents they share detail artisans who will offer ongoing services for a particular price - each artisan on the map is able to create one material from another and reverse the process. Each document is limited to the connections for a particular set of materials which are related in some way. Since the general store merchants offer a new daily special periodically, having a network which allows the creation of all materials from any material in the documents will save a lot of money in future adventures. You will have to find the gold you'll need to build the network which allows any material to be reached from any other material in the document.

Here is an example of one of these documents:



This document would be represented using the following format:

```
8
```

```
11
0
0 1 5
0 3 4
1 2 3
1 4 6
1 6 8
2 4 3
3 6 2
4 6 4
4 7 8
5 7 4
6 7 6
```

Which is similar to the format for Part B, but excludes the fourth line that would normally specify the final destination.

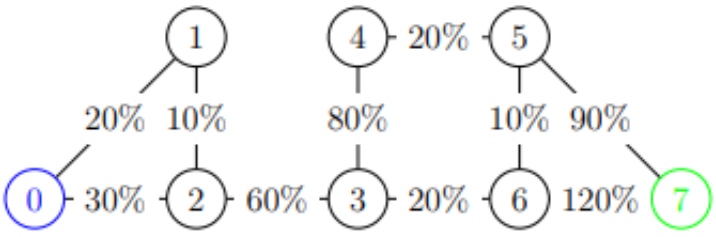
The output will be the minimum cost required to pay across all artisans such that each material in the document can be made from any other material.

Part D

The final battle with Balgor, the Ruler of the Delve requires venturing into the eponymous mythic delves. Mythic delves are typically reserved for the strongest of adventurers, as each room applies a further unique condition which increases the amount of damage done. Those who came before you have managed to map the damage multiplier that each room applies, and have sold these precious maps to you for a hefty sum. Since one of Balgor's minions lies at the end of each mythic delve, you will have to reach the end of the delve with the lowest multiplier to have any chance of besting them.

The game uses the mathematical floor of the calculated multiplier when calculating the final damage multiplier (as this multiplier is displayed to the player as a whole number) - however this is only applied to the final value, intermediate calculations retain the fractional elements.

Here is an example of a map of a mythic delve:



This map would be given in the following format:

```
8
10
0
7
0 1 20
```

```
0 2 30
1 2 10
2 3 60
3 4 80
3 6 20
4 5 20
5 6 10
5 7 90
6 7 120
```

This is similar to the format of Part A and B, but the weight of each edge is the percentage increase (e.g. a value of 20 implies a 20% increase in the damage taken in each subsequent room). For this problem, successive rooms multiply by the weight of the previous rooms, so in the above map, a path which takes the edge between 0 and 1, followed by the edge between 1 and 2, would apply a total increase of 32%.

The output should be the final percentage increase (without the percent sign) with any fractional part of the percentage omitted.

Task 2: Assignment Submission

Upload your solution to Task 2!

Input Formats

The test cases for each problem are present in the `test_cases` folder (using the format `2x-in-y.txt`, where x is the part and y is the test case number) and the expected answers for each of these test cases are present in the same folder (using the format `2x-out-y.txt`, where x is the part and y is the test case number).

Part A

The details of the input to Part A are described in the previous slide:

```
8 (Number of lucations)
11 (Connections between locations)
0 (Start location)
5 (End location)
0 1 (Connection 1)
0 3 (Connection 2)
1 2 (Connection 3)
1 4 (Connection 4)
1 6 (Connection 5)
2 4 (Connection 6)
3 6 (Connection 7)
4 6 (Connection 8)
4 7 (Connection 9)
5 7 (Connection 10)
6 7 (Connection 11)
```

Part B

The details of the input to Part B are described in the previous slide:

```
8 (Number of lucations)
11 (Connections between locations)
0 (Start location)
5 (End location)
0 1 5 (Connection 1)
0 3 4 (Connection 2)
1 2 3 (Connection 3)
1 4 6 (Connection 4)
1 6 8 (Connection 5)
2 4 3 (Connection 6)
3 6 2 (Connection 7)
```

```
4 6 4 (Connection 8)
4 7 8 (Connection 9)
5 7 4 (Connection 10)
6 7 6 (Connection 11)
```

Part C

The details of the input to Part C are described in the previous slide:

```
8 (Number of lucations)
11 (Connections between locations)
0 (Start location)
0 1 5 (Connection 1)
0 3 4 (Connection 2)
1 2 3 (Connection 3)
1 4 6 (Connection 4)
1 6 8 (Connection 5)
2 4 3 (Connection 6)
3 6 2 (Connection 7)
4 6 4 (Connection 8)
4 7 8 (Connection 9)
5 7 4 (Connection 10)
6 7 6 (Connection 11)
```

Part D

The details of the input to Part D are described in the previous slide:

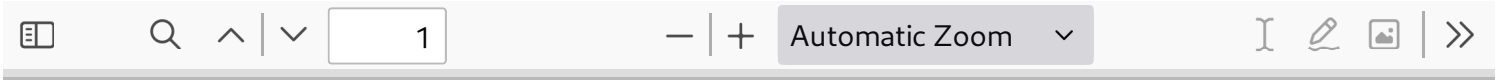
```
8 (Number of lucations)
10 (Connections between locations)
0 (Start location)
7 (End location)
0 1 20 (Connection 1 - +20%)
0 2 30 (Connection 2 - +30%)
1 2 10 (Connection 3 - +10%)
2 3 60 (Connection 4 - +60%)
3 4 80 (Connection 5 - +80%)
3 6 20 (Connection 6 - +20%)
4 5 20 (Connection 7 - +20%)
5 6 10 (Connection 8 - +10%)
5 7 90 (Connection 9 - +90%)
6 7 120 (Connection 10 - +120%)
```

Submitting

When you're happy with your solution for - you should click the "Test" button (you can click this as many times as you like), if it comes back as expected, click the blue "Submit" button in the top-right corner to submit both tasks.

Submit

Task 2: Test Cases



Academic Honesty

This is an individual assignment. The work must be your own work.

While you may discuss your program development, coding problems and experimentation with your classmates, you must not share files, as doing this without proper attribution is considered plagiarism.

If you have borrowed ideas or taken inspiration from code and you are in doubt about whether it is plagiarism, provide a comment highlighting where you got that inspiration.

If you refer to published work in the discussion of your experiments, be sure to include a citation to the publication or the web link.

“Borrowing” of someone else’s code without acknowledgment is plagiarism. Plagiarism is considered a serious offense at the University of Melbourne. You should read the University code on Academic integrity and details on plagiarism. Make sure you are not plagiarizing, intentionally or unintentionally.

You are also advised that there will be a C programming component (on paper, not on a computer) in the final examination. Students who do not program their own assignments will be at a disadvantage for this part of the examination.

Late Policy

The late penalty is 20% of the available marks for that project for each working day (or part thereof) overdue. Requests for extensions on medical grounds will need to be supported by a medical certificate. Any request received less than 48 hours before the assessment date (or after the date!) will generally not be accepted except in the most extreme circumstances. In general, extensions will not be granted if the interruption covers less than 10% of the project duration. Remember that departmental servers are often heavily loaded near project deadlines, and unexpected outages can occur; these will not be considered as grounds for an extension.

Students who experience difficulties due to personal circumstances are encouraged to make use of the appropriate University student support services, and to contact the lecturer, at the earliest opportunity.

Finally, we are here to help! Frequently asked questions about the project will be answered on Ed.

Requirements: C Programming

The following implementation requirements must be adhered to:

- You must write your implementation in the C programming language.
- Your code should be easily extensible to multiple data structure instances. This means that the functions for interacting with your data structures should take as arguments not only the values required to perform the operation required, but also a pointer to a particular data structure, e.g. `search(dictionary, value)`.
- Your implementation must read the input file once only.
- Your program should store strings in a space-efficient manner. If you are using `malloc()` to create the space for a string, remember to allow space for the final end of string character, '`\0`' (`NULL`).
- Your approach should be reasonably *time efficient*.
- Your solution should begin from the provided scaffold.

Hints:
• If you haven't used `make` before, try it on simple programs first. If it doesn't work, read the error messages carefully. A common problem in compiling multifile executables is in the included header files. Note also that the whitespace before the command is a tab, and not multiple spaces.
• It is not a good idea to code your program as a single file and then try to break it down into multiple files. Start by using multiple files, with minimal content, and make sure they are communicating with each other before starting more serious coding.

Programming Style

Below is a style guide which assignments are evaluated against. For this subject, the 80 character limit is a guideline rather than a rule — if your code exceeds this limit, you should consider whether your code would be more readable if you instead rearranged it.

```
/** ***********************
* C Programming Style for Engineering Computation
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au) 13/03/2011
* Definitions and includes
* Definitions are in UPPER_CASE
* Includes go before definitions
* Space between includes, definitions and the main function.
* Use definitions for any constants in your program, do not just write them
* in.
*
* Tabs may be set to 4-spaces or 8-spaces, depending on your editor. The code
* Below is ``gnu'' style. If your editor has ``bsd'' it will follow the 8-space
* style. Both are very standard.
*/

/**
* GOOD:
*/

#include <stdio.h>
#include <stdlib.h>
#define MAX_STRING_SIZE 1000
#define DEBUG 0
int main(int argc, char **argv) {
 ...

/**
* BAD:
*/

/* Definitions and includes are mixed up */
#include <stdlib.h>
#define MAX_STING_SIZE 1000
/* Definitions are given names like variables */
#define debug 0
#include <stdio.h>
/* No spacing between includes, definitions and main function*/
int main(int argc, char **argv) {
 ...

/** ****************************
* Variables
* Give them useful lower_case names or camelCase. Either is fine,
```

```
* as long as you are consistent and apply always the same style.
* Initialise them to something that makes sense.
*/

/**
* GOOD: lower_case
*/

int main(int argc, char **argv) {

  int i = 0;
  int num_fifties = 0;
  int num_twenties = 0;
  int num_tens = 0;

 ...
/**
* GOOD: camelCase
*/

int main(int argc, char **argv) {

  int i = 0;
  int numFifties = 0;
  int numTwenties = 0;
  int numTens = 0;

 ...
/**
* BAD:
*/

int main(int argc, char **argv) {

  /* Variable not initialised - causes a bug because we didn't remember to
  * set it before the loop */
  int i;
  /* Variable in all caps - we'll get confused between this and constants
  */
  int NUM_FIFTIES = 0;
  /* Overly abbreviated variable names make things hard. */
  int nt = 0

  while (i < 10) {
    ...
    i++;
  }

 ...

/** ********************
* Spacing:
* Space intelligently, vertically to group blocks of code that are doing a
* specific operation, or to separate variable declarations from other code.
```

```
* One tab of indentation within either a function or a loop.
* Spaces after commas.
* Space between ) and {.
* No space between the ** and the argv in the definition of the main
* function.
* When declaring a pointer variable or argument, you may place the asterisk
* adjacent to either the type or to the variable name.
* Lines at most 80 characters long.
* Closing brace goes on its own line
*/

/**
* GOOD:
*/

int main(int argc, char **argv) {

 int i = 0;

 for(i = 100; i >= 0; i--) {
   if (i > 0) {
     printf("%d bottles of beer, take one down and pass it around,"
     " %d bottles of beer.\n", i, i - 1);
   } else {
     printf("%d bottles of beer, take one down and pass it around."
     " We're empty.\n", i);
   }
 }

 return 0;
}

/**
* BAD:
*/

/* No space after commas
* Space between the ** and argv in the main function definition
* No space between the ) and { at the start of a function */
int main(int argc,char ** argv){
 int i = 0;
 /* No space between variable declarations and the rest of the function.
 * No spaces around the boolean operators */
 for(i=100;i>=0;i--) {
 /* No indentation */
 if (i > 0) {
 /* Line too long */
 printf("%d bottles of beer, take one down and pass it around, %d
bottles of beer.\n", i, i - 1);
 } else {
 /* Spacing for no good reason. */

 printf("%d bottles of beer, take one down and pass it around."
 " We're empty.\n", i);
```

```
 }
 }
 /* Closing brace not on its own line */
 return 0;}

/** ****************
* Braces:
* Opening braces go on the same line as the loop or function name
* Closing braces go on their own line
* Closing braces go at the same indentation level as the thing they are
* closing
*/

/**
* GOOD:
*/

int main(int argc, char **argv) {

  ...

  for(...) {
    ...
  }

  return 0;
}

/**
* BAD:
*/

int main(int argc, char **argv) {

  ...

  /* Opening brace on a different line to the for loop open */
  for(...)
  {
    ...
    /* Closing brace at a different indentation to the thing it's
    closing
    */
           }

 /* Closing brace not on its own line. */
       return 0;}

/** **************
* Commenting:
* Each program should have a comment explaining what it does and who created
* it.
* Also comment how to run the program, including optional command line
```

```
* parameters.
* Any interesting code should have a comment to explain itself.
* We should not comment obvious things - write code that documents itself
*/

/**
* GOOD:
*/

/* change.c
*
* Created by Aidan Nagorcka-Smith (aidann@student.unimelb.edu.au)
13/03/2011
*
* Print the number of each coin that would be needed to make up some
change
* that is input by the user
*
* To run the program type:
* ./coins --num_coins 5 --shape_coins trapezoid --output blabla.txt
*
* To see all the input parameters, type:
* ./coins --help
* Options::
*   --help                Show help message
*   --num_coins arg       Input number of coins
*   --shape_coins arg     Input coins shape
*   --bound arg (=1)      Max bound on xxx, default value 1
*   --output arg          Output solution file
*
*/

int main(int argc, char **argv) {

  int input_change = 0;

  printf("Please input the value of the change (0-99 cents
  inclusive):\n");
  scanf("%d", &input_change);
  printf("\n");

  // Valid change values are 0-99 inclusive.
  if(input_change < 0 || input_change > 99) {
    printf("Input not in the range 0-99.\n")
  }

  ...

/**
* BAD:
*/

/* No explanation of what the program is doing */
int main(int argc, char **argv) {
```

```
  /* Commenting obvious things */
  /* Create a int variable called input_change to store the input from
  the
  * user. */
  int input_change;

  ...

/** ****************
* Code structure:
* Fail fast - input checks should happen first, then do the computation.
* Structure the code so that all error handling happens in an easy to read
* location
*/

/**
* GOOD:
*/
if (input_is_bad) {
  printf("Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}

/* Do computations here */
...

/**
* BAD:
*/

if (input_is_good) {
  /* lots of computation here, pushing the else part off the screen.
  */
  ...
} else {
  fprintf(stderr, "Error: Input was not valid. Exiting.\n");
  exit(EXIT_FAILURE);
}
```

Some automatic evaluations of your code style may be performed where they are reliable. As determining whether these style-related issues are occurring sometimes involves non-trivial (and sometimes even undecidable) calculations, a simpler and more error-prone (but highly successful) solution is used. You may need to add a comment to identify these cases, so check any failing test outputs for instructions on how to resolve incorrectly flagged issues.

Mark Breakdown

There are a total of 10 marks given for this assignment.

Your C programs for Task 1 and 2 should be accurate, readable, and observe good C programming structure, safety and style, including documentation. Safety refers to checking whether opening a file returns something, whether mallocs do their job, etc. The documentation should explain all major design decisions, and should be formatted so that it does not interfere with reading the code. As much as possible, try to make your code self-documenting, by choosing descriptive variable names. The remainder of the marks will be based on the correct functioning of your submission.

Note that marks related to the correctness of your code will be based on passing various tests. If your program passes these tests without addressing the learning outcomes (e.g. if you fully hard-code solutions or otherwise deliberately exploit the test cases), you may receive less marks than is suggested but your code marks will otherwise be determined by test cases. For questions with both a written component and a C code component, part of the mark will be given for the passing of test cases, with the remainder from the correctness of the written answer.

Task 1 will be marked out of 5 marks, Task 2 will be marked out of 4 marks and C code quality will comprise the final mark.

Additional Support

Your tutors will be available to help with your assignment during the scheduled workshop times. Questions related to the assignment may be posted on the Ed discussion forum, using the folder tag Assignments for new posts. You should feel free to answer other students' questions if you are confident of your skills.

A tutor will check the discussion forum regularly, and answer some questions, but be aware that for some questions you will just need to use your judgment and document your thinking.

If you have questions about your code specifically which you feel would reveal too much of the assignment, feel free to post a private question on the discussion forum.

Most students find Academic Skills' Research Report Guide extremely valuable in constructing a well formed and sensible analysis that makes good use of relevant material taught so far in the subject.

Acknowledgements

ChatGPT was used to brainstorm ideas and character names for this assignment - the names and some very general ideas in the introduction may potentially derive from or be found similar to any of the work in the InstructGPT training corpus or other ChatGPT training data.