Engr 101 Project 6: gLyft

Program Design Due: Tuesday, November 30, 2021
Project Due: Tuesday, Tuesday, December 7, 2021

Engineering intersections: Industrial & Operations Engineering / Computer Science

Implementing this project provides an opportunity to practice file I/O, functions, loops, and data structures (vectors and structs).

The program design is worth 10 points, the autograded portion of the final submission is worth 100 points, and the style and comments grade is worth 10 points.

You may work alone or with a partner. Please see the syllabus for partnership rules.

Project Roadmap

project redos are NOT available for this project

This is a big picture view of how to complete this project. Most of the pieces listed here also have a corresponding section later on in this document that goes into more detail.

1. **Read through the project specification (DO THIS FIRST)**
2. **Complete the Program Design and submit it to Gradescope**
   Please bring your program design to office hours. We'll ask to see it before we help you with any coding or debugging questions.
3. **Write helper functions to handle the data input part of the program**
   Get these parts working first; verify you are reading in the data correctly!
4. **Create the location grid and determine the route**
   To debug your loops, use `cout` statements to print some intermediate variable values to the terminal; you can use these values to check your calculations. Once it's working, delete the `cout` statements and put in the code to print to the output file.
5. **Check that your program is sufficiently and correctly commented**
6. **Verify that you can reproduce the Test Cases in the Appendix**
7. **Submit `planRoute.cpp` to the Autograder**

Background

CEO Uberwine von Lyften is looking to expand the service of the galactic ride-sharing company, gLyft, headquartered on Proxima b. The von Lyften family can trace their roots back to the original gLyft crew from the 20th Century (Fig. 1), and they are very excited to expand gLyft's service throughout Sector 7-G of the Milky Way Galaxy.

Figure 1. The first gLyft crew: Neil Armstrong, Michael Collins, and Buzz Aldrin. Collins safely delivered Armstrong and Aldrin to the Moon but was ticketed by Luna PD for keeping the engine running while in a parking zone. Photo Source.

The gLyft planetary database has a list of coordinates corresponding to each planet's location. Expanding gLyft's service requires a computer program that will automatically plan routes based on the shortest path through the customers' destinations. Each driver is restricted to their own range, so the program must limit the destinations to only those planets within a maximum area defined by a grid.

Your Job

In this project, your job is to find the shortest path from each planet to the next within your gLyft driver's range.

Engineering Concepts

There are two high level engineering concepts that will be used in this project: the *traveling salesperson problem* and *data corruption*. These two terms are briefly explained in the next sections.

Traveling Salesperson Problem (TSP)

The traveling salesperson problem is a commonly-used optimization problem that asks,

> *Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?* [Wikipedia]

The "cities" and "distances" can have different definitions according to the application of the TSP algorithm. The problem has analogous questions in circuit design (shortest distance the laser needs to travel to create a circuit board), delivery logistics (shortest time to complete package delivery to homes and/or businesses), school bus routes, and similar engineering challenges.

There are many different algorithms that have been developed to solve this problem. In this project, we will use the nearest neighbor algorithm to determine which planet to go to next. The nearest neighbor algorithm we will use is defined here as:

1. Start at a given location
2. Find the closest location that has not already been visited
3. Go to that location
4. Mark that this location has been "visited"
5. Repeat steps 2-4 until all locations have been visited
6. Go to the specified end location

A sample path found using this algorithm is shown in Fig. 2 (click the link to see the path animated). Note that this is a "greedy" algorithm -- it picks the single closest location that hasn't already been visited, regardless of where the other yet-to-be-visited locations are, traffic in and out of those locations, or total distance traveled. Therefore, this greedy algorithm does not generally find the *best* path through all the required locations, but it often finds a path that is close to optimal. Step 6 above is required for our particular gLyft application -- it is not a necessary part of the nearest neighbor algorithm.

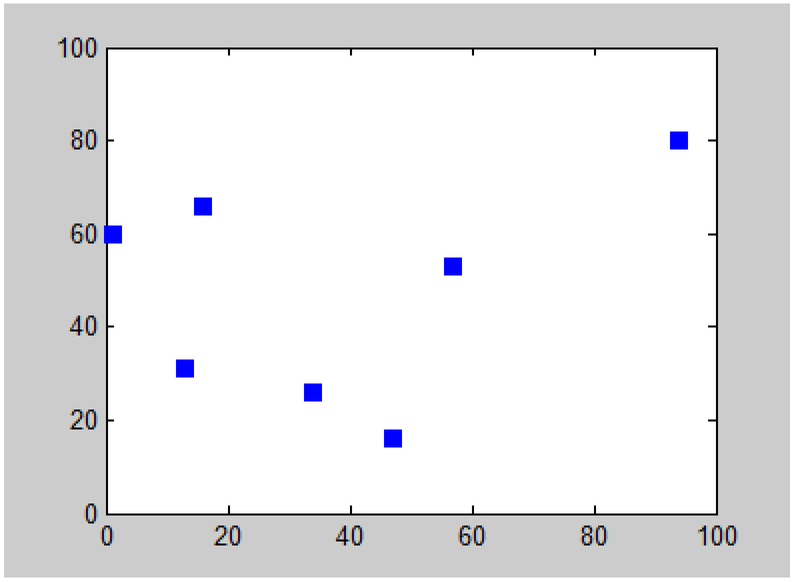


Figure 2. Example of how the nearest neighbor algorithm works. Note how the path changes based on the starting location! (This is a .gif so please do click the source: Photo Source)

To determine which new location is "closest" to the current location, we will assume that the planets all lie in the (x,y) galactic plane and that we can use the following formula for distance:

$$distance = \sqrt{\left(x_{current} - x_{potential}\right)^2 + \left(y_{current} - y_{potential}\right)^2}$$

where $x_{current}$ and $y_{current}$ are the coordinates of the current location and $x_{potential}$ and $y_{potential}$ are the coordinates of the location of a potential candidate that we wish to visit.

Data Corruption

Transmission of data is always subject to potential corruption. Data corruption occurs when data transmitted by one computer and received by a second computer does not match the actual data transmitted by the first computer. Data are stored in computers in binary form: a collection of 0's and 1's. When these data are transmitted from one computer to another over a long distance, it is possible for a 0 to be received as a 1 or vice-versa, due to noise in the system. This means some data may be *corrupted* and requires correction. There are many ways of checking for data corruption; error detection and correction is a critical component of modern computing. In this project, we will implement a simple error detection and correction based on known error effects. See below for further information.

Program Tasks

For this project, you will need to determine the shortest route for your gLyft driver. The route begins at a given starting point, proceeds to the customers' planets, and ends at a given endpoint. You will need to properly read data from input files, determine the route, and create an output file with a "map" of the planets and the route the gLyft driver needs to take to travel from the starting point to the end point. These tasks are detailed in the next sections.

Input Data

There are two input files that are required by your program: 1) a locations file, and 2) a planetary names file. The files themselves are defined in the Input Files Section. The sections below describe the procedures required to process the data within the input files.

Get Filenames

Your program must prompt the user for the names of the two input files (this makes the program flexible -- the person writing the files may name them whatever they want). Prompt the user for the locations file first using "**`Enter Locations Filename: `**"; the user then enters the `<location_file>.txt` name. Then, prompt the user for the planetary names filename, `<name_file>.txt` with "**`Enter Names Filename: `**".

Open Files

Once ***both*** input filenames have been entered by the user, the program must then open the files for reading. If either of the input files is not found or cannot be opened for reading, the program must output "**`Input file could not be opened`**" to the terminal followed immediately by (on a separate line) program exit with code `return 1;`. Note that you do not need to determine ***which*** file caused the failure; simply print the message and exit the program.

Note

The provided test cases do not test the error handling of input files. You must generate your own test case to ensure compliance. Do not submit your test case, just ensure your code satisfies this important required project specification.

Read Data

The locations file and the planetary names file are detailed in the Input Files section. Note that the locations file contains the range for that particular gLyft driver (i.e., number of rows and columns in their "map"), the starting and ending points for the journey, and the data for the planets. The names of the planets are stored separately in the names file. Input all the location and names data, and appropriately store the information for further use (you choose the storage method).

Recommendation: use structs!

We *strongly* recommend that you use a vector of structs to store the planetary data. See the Rover Mission and Containerships programs for coding inspiration.

As you check each planet in the file, you must **ensure that its coordinates lie within the grid** – if they do not, your code must output "`<Planet ID> out of range - ignoring`" to the terminal, where `<Planet ID>` is the ID of the Planet whose coordinates are not within the grid.

Correct Errors

Please look at the names files in the test cases. You will notice that some of the planet names have been corrupted during file transmission: there are "XX"s that have been erroneously added into the planet names. After reading in the data, you will need to correct errors by:

- Finding and erasing all occurrences of "XX" from each name
- Finding and replacing all underscore characters with spaces.

These corrections should happen to the data stored in variables within your program only; you should not modify the input files.

Create Location Grid

As part of the output file, `journey.txt`, you must provide a grid of characters representing the "map" of locations within your gLyft driver's range. The locations file indicates the number of rows and columns that must appear in the grid. The locations file also contains a 1-character symbol representing that planet.

Place each planet's symbol at its corresponding grid location (row, column). If a map location is not represented by a planetary symbol, then place a period (".") at that grid point. Also, place an "S" at the starting point and an "E" at the ending point.

Once you have created your grid point map, output the map to file `journey.txt`.

Watch your indexing!

On your grid point map, input row and column values start from 1, not 0. Row numbers increase from the top row to the bottom row and column numbers increase from the left column to the right column. In other words, the upper left corner of the grid map corresponds to (row 1, column 1). Note that if you store your map in a vector of vectors, C++ vector indices begin with index 0. **Check, then double check your indexing.**

Determine the Route

Acting as dispatcher for gLyft drivers, implement the nearest neighbor algorithm. To determine a route, begin at the (S)tart point (row & column) provided by the required locations file. From there, find the closest planet to the current location using the distance formula provided in the TSP algorithm (page 4). Once found, move to it – this means that your current coordinates must be updated to those of that closest planet. **Every time you visit a new planet, *mark that planet as Visited*, update current location to that planet, and use the TSP algorithm to determine the next planet that is nearest to your current location and *that has not yet been marked as Visited.*** After each move, you must check all remaining unvisited planets to determine which is nearest to your current location. Once all planets have been visited, your driver travels to the (E)nd point.

To establish the drivers route, you can make use of two loops:

- The inner loop should iterate through the locations yet to be visited to determine which planet is the closest to the current location – solely based on the nearest neighbor algorithm.

> ***Really Important Note!!!***
>
> It may very well be that when you are deciding on the next planet to visit, there is more than one planet at exactly the same distance from your current location. If this case is TRUE, then you must choose the planet with the lower ID value.

- The outer loop iterates while there are planets that have yet to be visited. It should also be the place where you output the necessary text to `journey.txt` indicating where you are going to next. Refer to the description of the `journey.txt` file and the test cases for more details.

Program Description

In general, you need to write a C++ program that:

1. Reads in the locations and planet names
2. Processes the locations and planet names
3. Determines the route to take
4. Outputs a file summarizing the route

Note: The above list can be considered to be the *start* of a program design for this project. **You will be required to submit a full program design as part of this project.** Refer to the Program Design Section for more details for what submission requirements. Refer to the Project 6 Overview slides on the Google Drive for some inspiration on how to structure your program.

Terminology

In this program description, the following terms shall be used:

- **Input Files:** Files that will be used by your function, such as a `.txt` file.
- **Output Files:** Files that will be produced by your function, such as a `.txt` file.

Required Program

There is one required program file for this project. The C++ file for this program must be named `planRoute.cpp`. There are no helper functions required for this project but we *highly recommend* you create some to help abstract away some of the complexity of this project.

Input Files

There are two input data files that your code must read: 1) a file containing the locations your gLyft driver is required to visit, and 2) a file containing the names of the planets with their ID numbers. These two files are defined in the next sections. Snapshots of each file can be seen in the Appendix; these input files are also located in the Project 6 Google Drive folder.

Recall that the names of the input files will be entered by the user via a terminal. Refer back to Get Filenames for details of how to handle this requirement.

Required Locations File

This file contains information about the locations that your gLyft driver must visit. The first line in the file gives the number of rows and the number of columns, respectively, defining the limits of the grid in which you will dispatch. Lines two and three contain the coordinates of your starting point and the ending point, in that order. Next is a list of planet locations to which the gLyft customers are traveling.

Data describing each planet location includes row and column coordinates, a symbol representing the surcharge type, and a location ID number . **Some planets will have coordinates that lie outside the dispatch grid dimensions – refer back to Create Location Grid for handling this situation.**

You are guaranteed that the input will be structured correctly and all IDs will be unique integer numbers. You are also guaranteed that the start and end points will be within range and that the routes' planets will not correspond to either the route start point or the route end point.

INPUT: `<location_file>.txt`

```
<Number of Grid Rows> <Number of Grid Columns>
<Starting Row> <Starting Column>
<Ending Row> <Ending Column>
<Planet 1 Row> <Planet 1 Column> <Planet 1 Symbol> <Planet 1 ID>
<Planet 2 Row> <Planet 2 Column> <Planet 2 Symbol> <Planet 2 ID>
...
```

The values in this file are explained below:

- **<Number of Grid Rows>** - the number of rows there are in your grid
- **<Number of Grid Columns>** - the number of columns there are in your grid
- **<Starting Row>** - the starting point's row number (not index number)
- **<Starting Column>** - the starting point's column number (not index number)
- **<Ending Row>** - the ending point's row number
- **<Ending Column>** - the ending point's column number
- **<Planet Row>** - the planet's row number
- **<Planet Column>** - the planet's column number
- **<Planet Symbol>** - the symbol associated with the planet's surcharge type (this will always be one character in length)
- **<Planet ID>** - a unique integer ID that is associated with the planet

Planet Names File

This file contains a list of planet IDs and names. The file will contain the same location IDs given in `<location_file>.txt`, **not necessarily in the same order and not *only* these location IDs**, followed by the name of each location. During transmission, the file data may have been corrupted. It may contain meaningless pairs of X's which will require removal. You must also replace underscores ("_") in the names with spaces in your output file. You are guaranteed that all of the location IDs in the location file will be present in this planet names file, and each location ID will have a corresponding planet name. The planet names file may have location IDs and planet names that you don't need for a given route, though.

INPUT: `<name_file>.txt`

```
<Planet 1 ID> <Planet 1 Name>
<Planet 2 ID> <Planet 2 Name>
...
```

The values in this file are explained below:

- **<Planet ID>** - a unique ID that is associated with the planet
- **<Planet Name>** - a name that is a contiguous set of characters (i.e. no spaces)

Output File

There is one output file for this program: `journey.txt`. This file contains a grid map of the route and a list of all the planets that have been visited in the correct order based on the following rules. Below is the structure for the file:

OUTPUT: `journey.txt`

```
<Map>
Start at <Starting Row> <Starting Column>
Go to <Planet Name> at <Planet Row> <Planet Column>
...
End at <Ending Row> <Ending Column>
```

The values in this file are explained below (see the test cases in the Appendix for an example of `journey.txt`):

- **<Map>** - the grid with all the valid planets' symbols on it
- **<Starting Row>** - the starting point's row number
- **<Starting Column>** - the starting point's column number
- **<Planet Name>** - a name that is a contiguous set of characters
- **<Planet Row>** - the planet's row number
- **<Planet Column>** - the planet's column number
- **<Ending Row>** - the ending point's row number
- **<Ending Column>** - the ending point's column number

Note: there should be a newline at the end of the last line of this file.

Hints for a Successful Program

This project is less structured than the previous projects because we want you to have a chance to design and implement a program that makes sense *to you*. But that doesn't mean we don't have some helpful hints to share about how to go about setting up your program! Take a look at these sections as you start to design your program for this project.

Getting Started

There are many, many different ways to successfully approach this project. We encourage you to develop a good program design before composing your code. If you are stuck with getting started, refer back to what we've covered in Runestone and labs: string manipulation, vectors, structs, vectors of structs, functions, and input/output.

Storing Planetary Data: Use a Vector of Structs

We strongly recommend using structs to store the planetary data. Structs are a logical choice since each planet has the same set of data (name, ID, location, etc.) and because one naturally thinks about the planet first and then all the data associated with that planet (as opposed to thinking of a bunch of names, then a bunch of IDs, then a bunch of row numbers, etc.). A vector of structs will conveniently store all planetary data for use later in your program.

Getting the Warning About "comparison of integers of different signs"

We strongly recommend that you continue to compile using the `-Wall` and `-pedantic` flags to help you catch bugs in your program. Sometimes, these warnings are important to fix, and sometimes the warnings are more in the way of a "heads up" that you can ignore if you want to. For example, if you have some code that uses a vector of `int`s like this:

```
vector <int> test_vector;
for (int i = 0; i < test_vector.size(); ++i) {
      // do vector stuff
}
```

and you compile using the `-Wall` and `-pedantic` flags, you might see an error that looks something like this:

```
warning: comparison of integers of different signs: 'int' and
'std::__1::vector<int, std::__1::allocator<int> >::size_type' (aka
'unsigned long') [-Wsign-compare]
      for (int i = 0; i < test_vector.size(); ++i){
```

This error message is trying to warn you that you are comparing integers whose values might represent different things: a *signed* integer vs. an *unsigned* integer. Remember how we have different variable types in C++ like `int`, `double`, `bool`, `string`, and `vector`? Well, there are [different types of variables](#) that can represent integers with different ranges and/or use different amounts of memory in the computer. This warning is telling us that we're comparing `i` (a "regular" signed integer) to the value that is returned by the vector's `.size()` function (an "unsigned long" integer which has a different range of values than "regular" integers).

We don't really get into the different kinds of integers (or `doubles`) in ENGR 101 because our applications don't require this level of memory control and instruction… and because we're covering quite enough content as it is! **In this case of the "traversing a vector" pattern, you can safely ignore this warning because our vectors are small enough that comparing a signed integer to an unsigned integer will not cause us any problems.** However, if you would like to resolve this error so that you don't have to look at it every time you re-compile, here are a couple of options you can use.

Option 1: The .size() vector function returns a value with type 'size_t' (this means it is an unsigned integer). Therefore, to make the warning go away, we need to compare the value returned by .size() to an unsigned integer, rather than our usual signed integer. So, we can declare i to be a size_t variable instead of an int variable:

```
vector <int> test_vector;
for (size_t i = 0; i < test_vector.size(); ++i) {
      // do vector stuff
}
```

Option 2: We could also *cast* the value returned by `.size()` as an `int` so that we are comparing `ints`. *Casting* is explicitly converting a variable from one type to another.

```
vector <int> test_vector;
for (size_t i = 0; i < int(test_vector.size()); ++i) {
      // do vector stuff
}
```

Either one of these options will resolve the "comparison of integers of different signs" warning.

Passing Parameters to Functions

Recall that when you write functions which pass in large data structures, such as vectors and structs, it is best to use pass-by-reference. Refer back to Runestone to remind you when to use pass-by-value, pass-by-reference, and `const` pass-by-reference.

Submission and Grading

This project has two deliverables: the program design and the `planRoute.cpp` file. Grades are divided into 3 categories: the program design, the autograder, and the style and comments.

Grades are broken down into three categories: the program design, the autograder, and the style and comments. First, a grader will evaluate the *general completeness* of your program design that is submitted to Gradescope and will provide a maximum score of 10 points. Second, the Autograder will be responsible for evaluating your submission. Third, one of our graders will evaluate your submission for style and commenting and will provide a maximum score of 10 points. Thus the maximum total number of points on this project is 120 points. Each test case on the Autograder is worth the same amount. Some test cases are public (meaning you can see your output compared to the solution as well as the inputs used to test your submission) while many are private (you are simply are told pass/fail and the reason). The breakdown of points for each category is described in the next sections.

Program Design

Maximum Score: 10 points

This project requires a complex program that performs two major tasks that need to be broken down into several smaller sub-tasks. Therefore, a program design document is a required deliverable. A good program design helps to find logic errors (those are the hard ones to debug, remember!) and helps immensely when describing your program to someone. We will ask to see this program design document if you come into office hours, so keep it handy.

Project 4 used a flowchart and Project 5 used pseudocode for their program designs. You may use either of these approaches, both of these approaches, a mash-up of the approaches, or something else entirely. The choice is yours. If you choose to use a flowchart at any point, you can make a flowchart using PowerPoint, Google Drawings, and many other programs that have the flowchart shapes built-in. If you choose to use pseudocode at any point, make sure that it is a detailed list of steps/algorithms/abstractions (helper functions).

Your program design should describe how you are breaking down the major tasks into smaller subtasks (consider how the Program Tasks section is broken into subsections for inspiration). There must be a “Start” and an “End” to the program design, with a clear control flow from start to end, but there should not be actual code in the program design (pseudocode is okay). For example, if you are removing an element from a vector within a loop, you have to show this in the program design but not write the actual code you would use. If you have questions, show your GSI your program design and get feedback!

The Project 6 program design will be graded for general completeness and quality, i.e. that the major steps required are documented and described and that the design is clear and easy to understand and not scribbled on the back of a napkin with a sharpie. It will *not* be graded on style, graphic design, or "prettiness," so don't spend a lot of time worrying about your color scheme!

Submitting your assignment to Gradescope

This Gradescope assignment has two questions: the pseudocode and the Honor Pledge. Create a separate document that states the Honor Pledge and includes your signature below the Honor Pledge (if your signature does not look like your name, also print your name).

If working alone:

I have neither given nor received unauthorized aid on this assignment, nor have I concealed any violations of the Honor Code on this assignment.

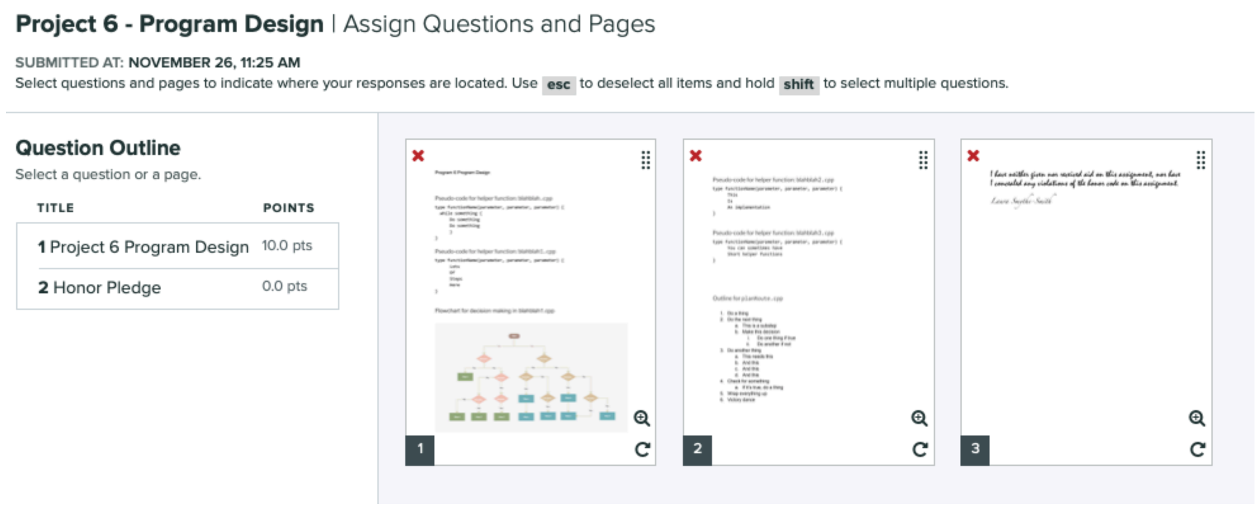
If working with a partner:

We have neither given nor received unauthorized aid on this assignment, nor have we concealed any violations of the Honor Code on this assignment.

The Honor Pledge and your signature must be *written*, not typed; if you are working in a partnership, both partners must sign the Honor Pledge. To create this written document, you have two options. Option 1: write the Honor Pledge and sign it on a piece of paper and then scan it to create an electronic copy; we recommend the Scannable app to get a high quality scan (low quality scans will not be accepted). Option 2: on a tablet or touchscreen, write the Honor Pledge, sign it, and save the file electronically. Please come to Office Hours if you need help with either of these options.

Create a document that contains both the Project 6 program design and a signed copy of the Honor Pledge. Include your name, partner uniqname (or "none"), lab section number, and date submitted at the top at the top of the program design. Save this document as a single `.pdf` (the name of this file does not matter).

When you upload your assignment, you will see something similar to what is shown below:



Project 6 - Program Design | Assign Questions and Pages

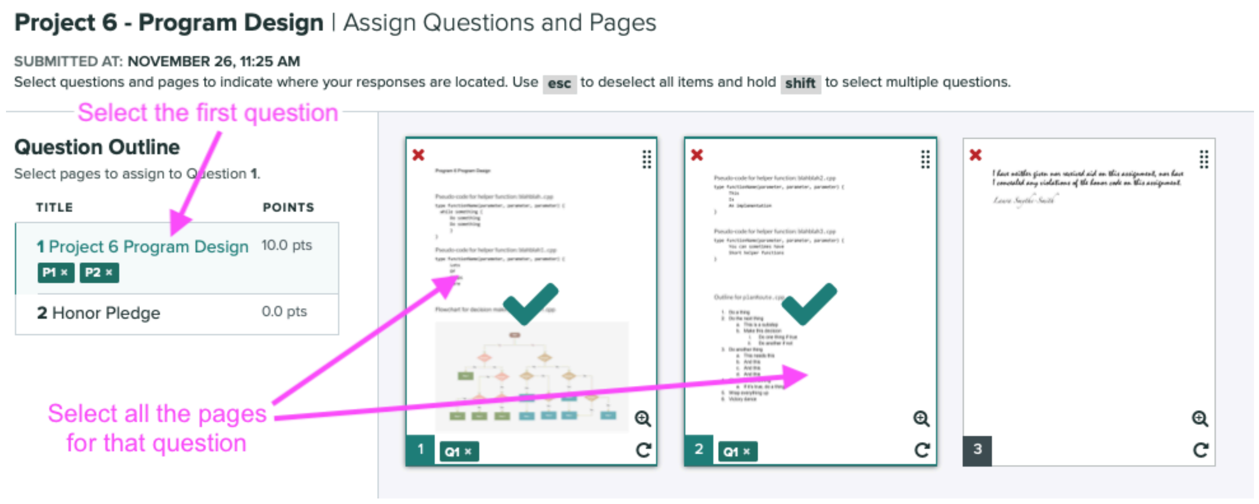
SUBMITTED AT: **NOVEMBER 26, 11:25 AM**
Select questions and pages to indicate where your responses are located. Use **esc** to deselect all items and hold **shift** to select multiple questions.

Question Outline
Select a question or a page.

TITLE	POINTS
1 Project 6 Program Design	10.0 pts
2 Honor Pledge	0.0 pts

This is a "variable length" assignment. Your program design can be as many pages as you need it to be: there is no minimum and no maximum on length. If you are looking for guidance, we would expect the pseudocode portion of the assignment to be at least 2 pages and probably not more than 5 or 6 pages, but fewer pages is perfectly fine as long as your document sufficiently details your algorithms and program design.

However many pages your pseudocode is, assign all of those pages to the Project 6 Program Design question, as shown below:



Project 6 - Program Design | Assign Questions and Pages

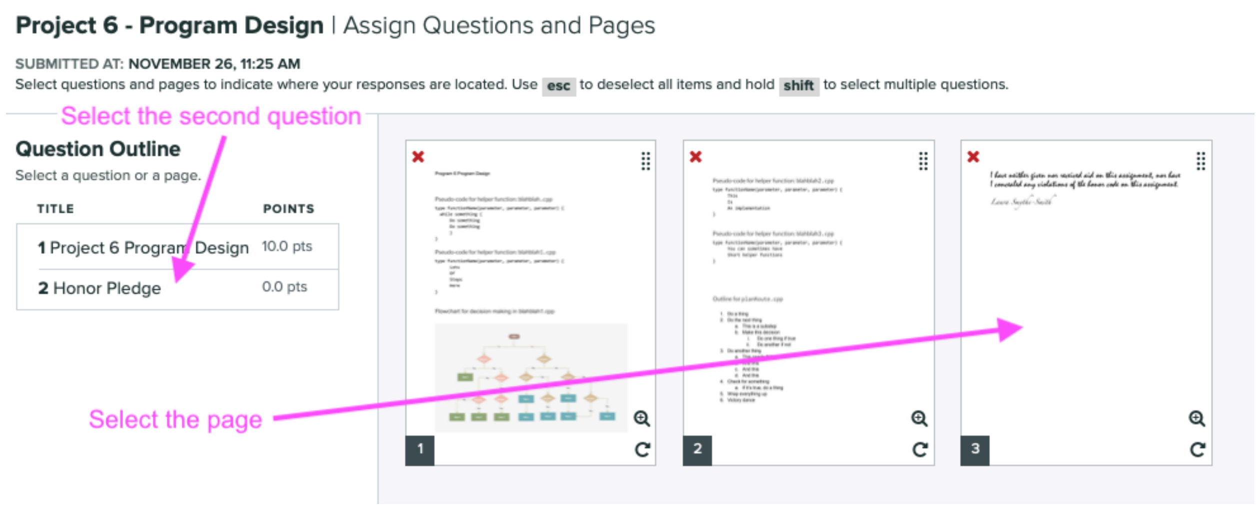
SUBMITTED AT: **NOVEMBER 26, 11:25 AM**
Select questions and pages to indicate where your responses are located. Use **esc** to deselect all items and hold **shift** to select multiple questions.

Question Outline
Select pages to assign to Question **1**.

TITLE	POINTS
1 Project 6 Program Design P1 × P2 ×	10.0 pts
2 Honor Pledge	0.0 pts

Select the first question
Select all the pages for that question
1 Q1 ×
2 Q1 ×
3

Then, assign the page that has your signed honor pledge to the Honor Pledge question, as shown below:



Project 6 - Program Design | Assign Questions and Pages

SUBMITTED AT: NOVEMBER 26, 11:25 AM

Select questions and pages to indicate where your responses are located. Use esc to deselect all items and hold shift to select multiple questions.

Question Outline

Select a question or a page.

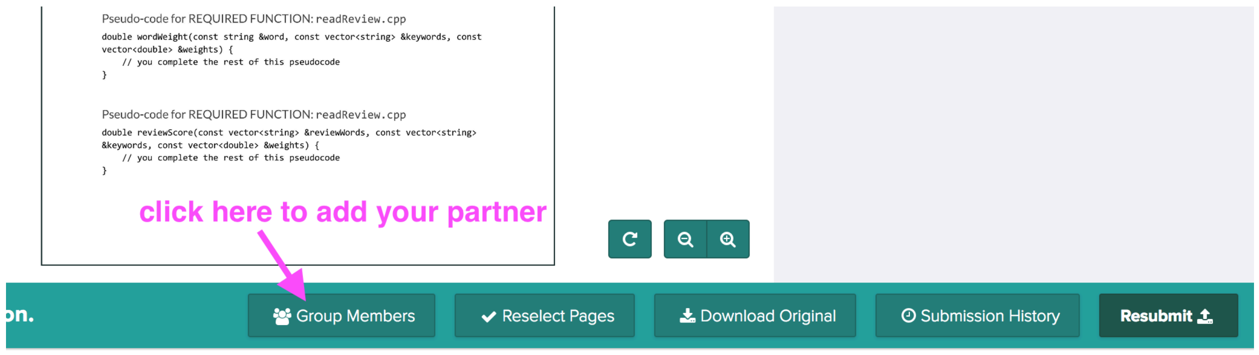
TITLE	POINTS
1 Project 6 Program Design	10.0 pts
2 Honor Pledge	0.0 pts

After you have assigned the correct pages to the correct questions, click the "Submit" button in the bottom right corner of the page.

Note: The program design is worth 9 points, and the signed Honor Pledge is worth 1 point (we haven't updated these screenshots yet).

Registering your partnership on Gradescope

If you are working with a partner, use "group submission" on Gradescope. Have one member of the partnership submit the program design and honor pledge as described above. After you submit the `.pdf` file, click "Group Members" to add your partner, as shown below:



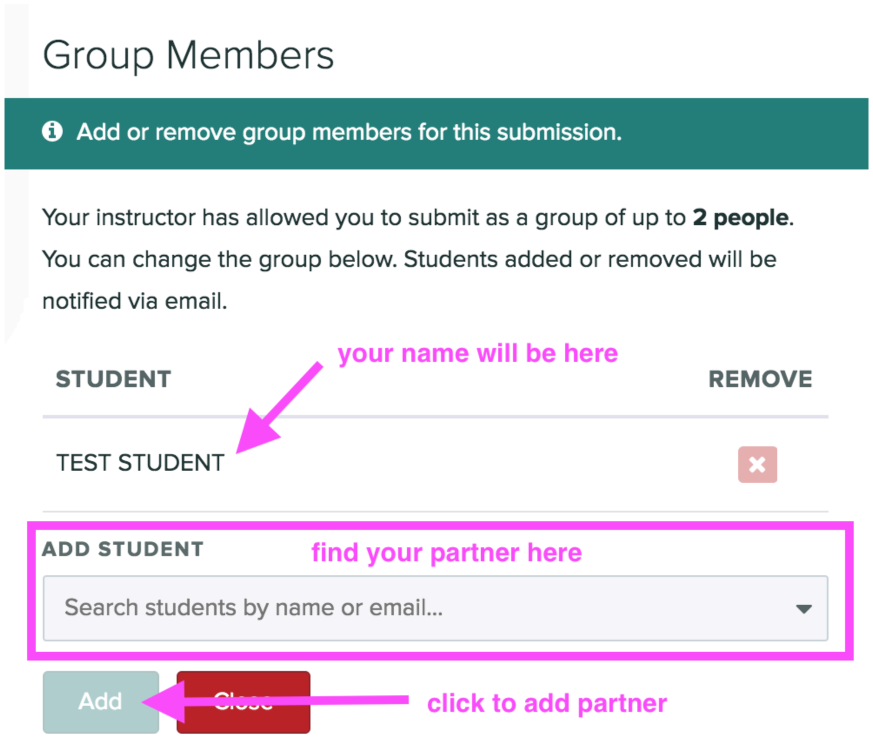
Pseudo-code for REQUIRED FUNCTION: `readReview.cpp`

```
double wordWeight(const string &word, const vector<string> &keywords, const
vector<double> &weights) {
    // you complete the rest of this pseudocode
}
```

Pseudo-code for REQUIRED FUNCTION: `readReview.cpp`

```
double reviewScore(const vector<string> &reviewWords, const vector<string>
&keywords, const vector<double> &weights) {
    // you complete the rest of this pseudocode
}
```

This will pop up a little box where you can search for your partner's name and then add them:


Group Members

Add or remove group members for this submission.

Your instructor has allowed you to submit as a group of up to **2 people**. You can change the group below. Students added or removed will be notified via email.

STUDENT	REMOVE
TEST STUDENT	✖

ADD STUDENT

Search students by name or email...

Your partner will receive an email verifying that they have been added to your group. Make sure your partner gets this email!

Note on Quality of Submission

Your program design must be a document that is of professional quality. This means text is typed and flowcharts are done using some kind of software program (Google Draw, Powerpoint, etc.). **No handwritten submissions will be accepted.**

Autograder

Maximum Score: 100 points

Submit the `planRoute.cpp` file to the Autograder (autograder.io) for grading. The autograder provides you with a final score (out of 100 points) as well as information on *a subset of its test cases*. We provide this autograder to give you a sense of your current progress on the project.

Register Your Partnership Before the Deadline!

Don't forget to register your partnership on the Autograder *before* the deadline. See the course website for instructions on how to register your partnership on the Autograder.

Submitting files to the Autograder

The Autograder will look for one file: `planRoute.cpp`.

Note: You don't *have* to wait until your program is completely done before submitting to the Autograder. You can submit a partially complete program and receive feedback on the test cases and output available on the autograder. However, **to receive full credit, you need to submit all files and pass all test cases within a single submission.**

Project scoring and feedback on the Autograder

You are limited to 5 submissions on the Autograder per day. After the 5th submission, you are still able to submit, but all feedback will be hidden other than confirmation that your code was submitted. The autograder will only report a subset of the tests it runs. It is up to you to develop tests to find scenarios where your code might not produce the correct results.

You will receive a score for the autograded portion equal to the score of your best submission. Your latest submission with the best score will be the code that is style graded. For example, let's assume that I turned in the following:

Submission #	1	2	3	4
Score	50	100	100	50

Then my grade for the autograded portion would be 100 points (the best score of all submissions) and Submission #3 would be style graded since it is the latest submission with the best score.

Please refer to the syllabus on more information regarding partner groups and general information about the Autograder.

Style and Commenting

Maximum Score: 10 points

2 pts - Each submitted file has Name, Partner Uniqname (or "none"), Lab Section Number, and Date Submitted included in a comment at the top
2 pts - Comments are used appropriately to describe your code (e.g. major steps are explained)
2 pts - Indenting and whitespace are appropriate (including functions are properly formatted)
2 pts - Variables are named descriptively
2 pts - Other factors (Variable names aren't all caps, etc…)

Appendix: Test Cases

Here are two test cases for you to use in designing, writing, and testing your code.

Test Case 1

TERMINAL:

```
bash-4.1$  g++ planRoute.cpp -o planRoute
bash-4.1$  ./planRoute
Enter Locations Filename: location_1_test.txt
Enter Names Filename: name_1_test.txt
8434 out of range - ignoring
6341 out of range - ignoring
7632 out of range - ignoring
1111 out of range - ignoring
bash-4.1$
```

INPUT: *location_1_test.txt*

```
 13  5
  2  5
  5  1
  2  2 X 7924
 13  1 T 5555
  5  2 Q 8753
 19  4 Q 8434
  8  3 P 2341
  7  1 X 2523
  1 19 T 6341
  2  3 X 5125
 -2  2 T 7632
  1  0 Q 1111
  6  3 A 9349
```

INPUT: *name_1_test.txt*

```
9349 The_Library
8753 New_Earth_(Planet_Bob)
5555 Fhloston_Paradise
2523 Jupiter_Two
7924 Domeo_And_James_Juliett
5125 Etheria_and_Eternia
6341 Planet_X
1111 Jaros_II_and_The_Kyln
8434 Polyphemus_and_Pandora
2341 The_Discworld
7632 Planet_Drool
```

OUTPUT: *journey.txt*

```
.....
.XX.S
.....
.....
EQ...
..A..
X....
..P..
.....
.....
.....
.....
T....
Start at 2 5
Go to Etheria and Eternia at 2 3
Go to Domeo And James Juliett at 2 2
Go to New Earth (Planet Bob) at 5 2
Go to The Library at 6 3
Go to The Discworld at 8 3
Go to Jupiter Two at 7 1
Go to Fhloston Paradise at 13 1
End at 5 1
```

Test Case 2

TERMINAL:

```
bash-4.1$  g++ planRoute.cpp -o planRoute
bash-4.1$  ./planRoute
Enter Locations Filename: location_2_test.txt
Enter Names Filename: name_2_test.txt
1231 out of range - ignoring
bash-4.1$
```

INPUT: *location_2_test.txt*

```
14 12
13 11
 1  2
 4 11 T 4497
13  3 P 7932
 1  9 Q 7999
-1 -1 X 1231
14 11 T 1542
12 11 T 6543
 5  3 A 7543
```

INPUT: *name_2_test.txt*

```
7932 JakkXXu
1542 MXXagrathea
7543 GethXXen
6543 ThuXXXXndera
4497 SupercalifragilisticeXXXpialidocious
7999 LittXXleBigPlaneXXt
1231 LV-XX426
```

OUTPUT: *journey.txt*

```
.E......Q...
............
............
..........T.
..A.........
............
............
............
............
............
............
..........T.
..P.......S.
..........T.
Start at 13 11
Go to Magrathea at 14 11
Go to Thundera at 12 11
Go to SupercalifragilisticeXpialidocious at 4 11
Go to LittleBigPlanet at 1 9
Go to Gethen at 5 3
Go to Jakku at 13 3
End at 1 2
```