Lab #6: Strings, Recursion and Structures

CS1010 AY2017/8 Semester 1

Date of release: 25 October 2017, Wednesday, 5pm.

Submission deadline: 8 November 2017, Wednesday, 5pm.

School of Computing, National University of Singapore

0 Introduction

Important: Please read Lab Guidelines before you continue.

This is your final lab! *Hooray!* ☺ ☺ ☺

This lab consists of 3 exercises. You are required to submit 2 exercises. If you submit 3 exercises, the best 2 out of 3 exercises will be used to determine your attempt mark.

The main objective of this lab is on the use of characters and strings, recursion, and structures to solve problems.

The maximum number of submissions for each exercise is **10**.

If you have any questions on the task statements, you may post your queries **on the relevant IVLE discussion forum**. However, do **not** post your programs (partial or complete) on the forum before the deadline!

Important notes applicable to all exercises here:

- You should take the "Estimated Development Time" seriously and aim to complete your programming within that time. Use it to gauge whether your are within our expectation, so that you don't get surprised in your PE. We advise you to do the exercises here in a simulated test environment by timing yourself.

- Please do **not** use variable-length arrays. An example of a variable-length array is as follows:
  ```
  int i;
  int array[i];
  ```
  This is not allowed in ANSI C, as explained in Unit #8 Arrays. Declare an array with a known maximum size. We will tell you the maximum number of elements in an array.

- You can finally use recursion now!

- You are **NOT allowed to use global variables**. (A global variable is one that is not declared in any function.)

- You are free to introduce additional functions if you deem it necessary. This must be supported by well-thought-out reasons, not a haphazard decision. By now, you should know that you **cannot write a program haphazardly**.

- In writing functions, we would like you to include function prototypes before the main function, and the function definitions after the main function.

- As mentioned in Unit #11 UNIX I/O Redirection, you may consider entering the input data in a file and then use UNIX input redirection to feed the data into your programs.

1 Exercise 1: Prerequisites

1.1 Learning objectives

- Array of strings

1.2 Task

National Programming Academy (NPA) provides several modules for its students. All module codes are formatted as strings of length 6, with 2 characters followed by 4 digits. For instance, these are some examples of module codes: CX1010, BT1234, GG1001, IZ4239.

In NPA, a module A is a **prerequisite** of another module B if and only if

1. The first two characters of modules A and B are the same,
2. The first digit of module A is less than the first digit of B, and
3. The other digits of module A are not greater than the corresponding digits of B at the same position.

For example,

- CX1010 is a prerequisite of CX2010
- GG2001 is a prerequisite of GG3111
- IZ3139 is a prerequisite of IZ4239

But

- IZ1103 is not a prerequisite of CX2103
- CX1010 is not a prerequisite of CX1010
- CX1020 is not a prerequisite of CX2010

You are to write a program **`prerequisites.c`** to read a positive integer indicating the number of modules, and a list of module codes. The program also reads a particular module code and if that module code exists, it displays all the prerequisites of that module in the same order as the entry of module codes.

You may assume that there are at least 2 and at most 10 module codes.

1.3 Sample runs

Sample run using interactive input (user's input shown in blue; output shown in **bold purple**). Note that the first two lines (in green below) are commands issued to compile and run your program on UNIX.

Sample run #1:

```
$ gcc -Wall prerequisites.c -o prerequisites
$ prerequisites
Enter number of modules: 4
Enter 4 modules:
CX2010
CX3110
IZ2010
CX1010
Choose a module: CX3110
Prerequisites for CX3110: CX2010 CX1010
```

Sample run #2:

```
Enter number of modules: 4
Enter 4 modules:
CX2010
CX3110
IZ2010
CX1010
Choose a module: CX1020
No such module CX1020
```

Sample run #3:

```
Enter number of modules: 7
Enter 7 modules:
CX2101
CX4000
CX3311
GG1211
CX2112
CX3300
CX3221
Choose a module: CX3311
Prerequisites for CX3311: CX2101
```

Sample run #4:

```
Enter number of modules: 3
Enter 3 modules:
IZ1111
IZ2222
IZ3333
Choose a module: IZ1111
No prerequisites for IZ1111
```

1.4 Skeleton program and Test data

- The skeleton program is provided here: prerequisites.c
- Test data: Input files | Output files

1.5 Important notes

- There are at least 2 and at most 10 module codes. You may assume that all the module codes are valid and distinct.
- The following are given in the skeleton program. You should **NOT** modify them.
  - The **`main()`** function
  - The **`printPrereq()`** function
  - The function prototypes
- The function **`scanModules()`** is to return the number of modules read.
- The function **`computePrereq()`** is to return the number of prerequisites for the target module. If the target module is not in the list of modules (see sample run #2), the function should return -1.
- You may write additional function(s) you deem necessary.

1.6 Estimated development time

The time here is an estimate of how much time we expect you to spend on this exercise. If you need to spend way more time than this, it is an indication that some help might be needed.

- Devising and writing the algorithm (pseudo-code): 30 minutes
- Translating pseudo-code into code: 15 minutes
- Typing in the code: 15 minutes
- Testing and debugging: 30 minutes
- **Total: 1 hour 30 minutes**

2 Exercise 2: The Driver's Problem

2.1 Learning objectives

- Recursion

2.2 Task

Bill owns a car and wishes to drive on a long straight road for a total distance of *D* kilometres. Along the road, there are *N* gas stations. Each gas station *i* is at distance d_i from the start point and has f_i decalitres of fuel available.

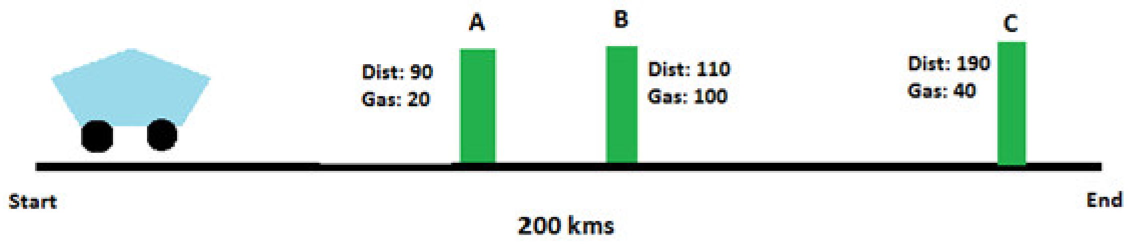
(The above symbols such as *D* and *N* are used for convenience in this write-up. In your program, you should use **descriptive variable names** for them.)

To make the problem simpler, you may assume that the initial fuel in the car is always 100 decalitres. One decalitre of fuel provides enough fuel to cover a distance of one kilometre.

Write a program `driver.c` to read the total distance, the number of gas stations, and for each gas station, its distance from the start and the amount of fuel available. All values are positive integers. You may assume that there are at most 20 gas stations and no two gas stations are at the same distance from the start. Also, the gas stations are entered in increasing order of their distance from the start. If Bill stops by a gas station, he will take all the available fuel in that station.

Your program is to compute the total number of possible routes Bill can take to complete his journey.

For example, refer to the following diagram.



In this example, Bill wishes to cover 200km. There are 3 gas stations:

- Gas station A has distance of 90km from starting point and 20 dal of gas available.
- Gas station B has distance of 110 from starting point and 100 dal of gas available.
- Gas station C has distance of 190 from starting point and 40 dal of gas available.

In this case, there are two possible routes for Bill:

- Route 1: Start → A → B → C → End
- Route 2: Start → A → B → End

Therefore the answer is 2.

2.3 Sample runs

Sample run using interactive input (user's input shown in blue; output shown in **bold purple**). Note that the first two lines (in green below) are commands issued to compile and run your program on UNIX.

Sample run #1:

```
$ gcc -Wall driver.c -o driver
$ driver
Enter total distance: 200
Enter number of gas stations: 3
Enter distance and amount of fuel for each gas station:
90 20
110 100
190 40
Total distance = 200
Number of gas stations = 3
```

```
Gas stations' distances:   90  110  190
Gas stations' fuel amt :   20  100   40
Possible number of routes = 2
```

Sample run #2:

```
Enter total distance: 300
Enter number of gas stations: 3
Enter distance and amount of fuel for each gas station:
90 20
110 100
190 40
Total distance = 300
Number of gas stations = 3
Gas stations' distances:   90  110  190
Gas stations' fuel amt :   20  100   40
Possible number of routes = 0
```

2.4 Skeleton program and Test data

- The skeleton program is provided here: driver.c
- Test data: Input files | Output files

2.5 Important notes

- Initial fuel is 100 decalitres.
- There are at most 20 gas stations, listed in increasing order of their distances from the start. So there is **no need** to sort them by distance from the start.
- All gas stations are located at a distance in the range (0, *D*), where *D* is the total distance to be travelled by Bill.
- The fuel amount in a gas station is non-negative.
- The following are given in the skeleton program. You should **NOT** modify them.
  - The **`readStations()`** function
  - The **`printStations()`** function
- You must complete the recursive function **`calcPossibleRoutes()`**. You may write additional function(s).
- This exercise might be a little challenging. You may want to try the practice exercises on recursion first to brush up your skills on recursion before attempting this exercise.

2.6 Estimated development time

The time here is an estimate of how much time we expect you to spend on this exercise. If you need to spend way more time than this, it is an indication that some help might be needed.

- Devising and writing the algorithm (pseudo-code): 40 minutes
- Translating pseudo-code into code: 10 minutes
- Typing in the code: 10 minutes
- Testing and debugging: 1 hour
- **Total: 2 hours**

3 Exercise 3: Elevators

3.1 Learning objectives

- Structures

- Characters and strings
- Arrays

3.2 Task statement

Brusco is a big fan of a vintage PC game called SimTower. He is particular fascinated by how the game simulates the elevators which transport the residents of the tower to different floors.

After learning more about structures in CS1010, he realized that they can be used to model almost everything in real-life, including elevators. Therefore, he decided to write a a simple program to simulate the running of elevators.

To keep things simple, he has decided to model elevators with the following three pieces of information:

- **Floor**: the floor the elevator is at.
- **Passenger**: the number of passengers in the elevator.
- **Usage**: the number of passengers who have finished using (i.e., exited) the elevator.

In addition, he has also set the following rules on the running of the elevators:

- Initially, all elevators are at the 1st floor with 0 passenger inside and 0 usage (i.e., no one has finished using them yet).
- Given a sequence of floor numbers, an elevator goes to the corresponding floors one by one.
- If an elevator goes from a lower floor to a higher floor *m*, *m* passengers will attempt to enter the elevator. However, once the number of passengers in the elevator reaches 15, which is the capacity of the elevator, no additional passengers will be able to enter the elevator.
- If an elevator with *p* passengers goes from a higher floor to a lower floor *n*, *n* (or *p*, whichever is smaller) passengers will exit the elevator. The actual number of passengers who exit the elevator should be counted towards the number of passengers who have finished using the elevator.

For example, if the sequence of floors is 7->9->8->6->3->5->..., then the movement of the elevator is as follows:

- Moving from 1st floor to 7th floor: 7 passengers enter the elevator. The number of passengers becomes 7. (Note that the elevator is at the 1st floor initially.)
- Moving from 7th floor to 9th floor: 9 passengers attempt to enter the elevator but only 8 of them are able to due to the capacity of the elevator. The number of passengers becomes 15.
- Moving from 9th floor to 8th floor: 8 passengers exit the elevator. The number of passengers becomes 7. The number of passengers who have finished using the elevator becomes 8.
- Moving from 8th floor to 6th floor: 6 passengers exit the elevator. The number of passengers becomes 1. The number of passengers who have finished using the elevator becomes 14.
- Moving from 6th floor to 3rd floor: 1 passenger exits the elevator (since there is only 1 passenger in the elevator). The number of passengers becomes 0. The number of passengers who have finished using the elevator becomes 15.
- Moving from 3rd floor to 5th floor: 5 passengers enter the elevator. The number of passengers becomes 5.
- ...

Last but not least, Brusco has also decided that, at the end of the simulation, all information (i.e., floor/passenger/usage) of all the elevators, as well as the elevator number of the most used elevator (see Section 2.5 on how to break ties), should be printed.

Brusco has managed to follow the first two stages of the problem-solving process to come up with a skeleton for his program; however, since he is not very familar with the syntax of structures yet, there is still quite a big chunk of his program which is yet to be completed.

You are to help Brusco complete the program `elevator.c`. This program takes in an integer *i*, which represent the number of elevators to be simulated, as well as *i* strings of digits, which represent the sequences of floors. The program then simulates the running of the elevators and prints the required outputs in the end.

3.3 Sample runs

Sample runs using interactive input (user's input shown in blue; output shown in **bold purple**). Note that the first two lines (in green below) are commands issued to compile and run your program on UNIX.

Sample run #1:

```
$ gcc -Wall elevator.c -o elevator
$ elevator
Enter number of elevators: 1
Enter sequence for elevator 1: 24653
Elevator 1:
Floor: 3
Number of passengers: 4
Usage: 8
Most used elevator: 1
```

Sample run #2:

```
Enter number of elevators: 1
Enter sequence for elevator 1: 798635
Elevator 1:
Floor: 5
Number of passengers: 5
Usage: 15
Most used elevator: 1
```

Sample run #3:

```
Enter number of elevators: 2
Enter sequence for elevator 1: 24653
Enter sequence for elevator 2: 798635
Elevator 1:
Floor: 3
Number of passengers: 4
Usage: 8
Elevator 2:
Floor: 5
Number of passengers: 5
Usage: 15
Most used elevator: 2
```

3.4 Skeleton program and Test data

- The skeleton program is provided here: elevator.c
- Test data: Input files | Output files

3.5 Important notes

- There is at least 1 elevator and at most 5 elevators.

- The floor number ranges from 1 to 9.
- There are at least 1 floor number and at most 20 floor numbers in a sequence.
- If there are more than 1 elevator with the same usage, report the one with more passengers in it after the simulation. If it is still a tie, report the one with a smaller elevator number.
- You must complete the program by filling in the parts which have been marked as "Incomplete" in the skeleton file. Do **not** modify other parts of the program (except for adding necessary comments for Style concerns).
- Do **not** implement additional functions.
- Do **not** use any additional array.

3.6 Estimated development time

The time here is an estimate of how much time we expect you to spend on this exercise. If you need to spend way more time than this, it is an indication that some help might be needed.

- Devising and writing the algorithm (pseudo-code): 20 minutes
- Translating pseudo-code into code: 20 minutes
- Typing in the code: 15 minutes
- Testing and debugging: 15 minutes
- **Total: 1 hour 10 minutes**

4 Deadline

The deadline for submitting all programs is **8 November 2017, Wednesday, 5pm**. Late submission will NOT be accepted.

Last updated: 20 October 2017