COMP9024 25T1

Assignment
Trip Planner

Data Structures and Algorithms

Change Log

We may make minor changes to the spec to address/clarify any outstanding issues. These may require minimal changes in your design/code, if at all. Students are strongly encouraged to check the online forum discussion and the change log regularly.

Version 1.0 (2025-03-16 13:00)	• Initial release.

Objectives

The assignment aims to give you more independent, self-directed practice with

- advanced data structures, especially graphs
- graph algorithms
- asymptotic runtime analysis

Admin

Marks	2 marks for stage 1 (correctness) 2 marks for stage 2 (correctness) 4 marks for stage 3 (correctness) 2 marks for stage 4 (correctness) 1 mark for complexity analysis 1 mark for style Total: 12 marks
Due	16:59:59 on **Tuesday** 22 April (week 10)
Late	Reduction by 5% of maximum mark per day late, capped at 5 days (= 120 hours) E.g. if you are 25 hours late, your mark will be reduced by 1.2 (= 10% of max mark)

Aim

You're in a new city and would like to plan a walking tour of some of its landmarks. Luckily we've seen a number of algorithms in class to help with path planning. Unluckily, the city isn't necessarily connected, at least not on foot. To help, there are ferries to help people get around.

Your objective is to write a program `tripPlan.c` that generates an optimal trip between landmarks, based on user preferences.

Input

Landmarks

The first input to your program consists of an integer $l > 0$, indicating the number of landmarks in the city, followed by l lines of the form:

```
landmark-name
```

Here is an example:

```
prompt$ ./tripPlan
Number of landmarks: 4
TheRocks
CircularQuay
ManlyWharf
ManlyBeach
```

You may assume that:

- The input is syntactically correct.
- The maximum length (`strlen()`) of the name of a landmark is 31 and will not use any spaces.
- Names are case sensitive.
- No name will be input more than once.

Hint:

To read a single line with a landmark name you should use:

```
scanf("%s", name);
```

where `name` is a string, i.e. an array of `char`s.

Walking Links

The next input to your program is an integer $w \geq 0$, indicating the number of walking links between landmarks, followed by $w \times 3$ lines of the form:

```
landmark-1
landmark-2
walking-time
```

where the third line denotes the time, in minutes, it takes to walk between the two landmarks. Note, this link may be walked in either direction, from `landmark-1` to `landmark-2`, or from `landmark-2` to `landmark-1`.

Here is an example:

```
Number of walking links: 2
TheRocks
CircularQuay
6
ManlyWharf
ManlyBeach
8
```

You may assume that:

- The input is syntactically correct.
- Only landmarks that have been input earlier will be used.

- The walking time will be a strictly positive integer.

Ferry Schedules

The next input to your program is an integer $f \geq 0$, indicating the number of ferries on any day, followed by $f \times 4$ lines of the form:

```
departing-landmark
departing-time
arriving-landmark
arriving-time
```

Here is an example:

```
Number of ferry schedules: 2
CircularQuay
0930
ManlyWharf
0952
ManlyWharf
1000
CircularQuay
1022
```

You may assume that:

- The input is syntactically correct.
- Only landmarks that have been input earlier will be used.
- All times are given as 4 digits in `hhmm` format (`hh` – hour, `mm` – minute), and are valid, ranging from `0000` to `2359`.
- The arrival time is strictly later than the time of departure.
- All ferries reach their destination before midnight.

Trip Plan

The final input to your program are user queries:

```
From: TheRocks
To: ManlyBeach
Departure time: 0915
```

You may assume that:

- The input is syntactically correct.
- Only landmarks that have been input earlier will be used.
- Two different landmarks will be given.
- No expected plan will roll over into the following day.

Your program should terminate when the user enters "`done`" when prompted with `From:`

```
From: done
Happy travels!
prompt$
```

Stage 1 (2 marks)

Stage 1 requires you to generate a suitable data structure from the input.

Test cases for this stage will only use queries `FromLandmark`, `ToLandmark`, `DepartureTime` such that:

- there exists at most one direct connection between `FromLandmark` and `ToLandmark`;
- should such a connection exist, this connection is the shortest path between `FromLandmark` and `ToLandmark`; and
- should such a connection not exist, no other path between `FromLandmark` and `ToLandmark` will exist.

Here is an example to demonstrate the expected behaviour of your program for a stage 1 test:

```
prompt$ ./tripPlan
Number of landmarks: 3
CircularQuay
ManlyWharf
TheRocks
Number of walking links: 1
CircularQuay
TheRocks
8
Number of ferry schedules: 1
CircularQuay
1130
ManlyWharf
1152

From: TheRocks
To: CircularQuay
Departure time: 0000

Walk 8 minute(s):
  0000 TheRocks
  0008 CircularQuay

From: CircularQuay
To: ManlyWharf
Departure time: 0100

Ferry 22 minute(s):
  1130 CircularQuay
  1152 ManlyWharf

From: done
Happy travels!
prompt$
```

If there is no connection that satisfies the requirements, then the output should be: `No route.`

```
From: CircularQuay
To: ManlyWharf
Departure time: 1200

No route.
```

Stage 2 (2 marks)

Stage 2 requires you to implement a basic path finding algorithm.

Test cases for this stage will only use queries `FromLandmark`, `ToLandmark`, `DepartureTime` such that:

- there exists one, and only one, simple path that connects `FromLandmark` and `ToLandmark`;
- this path does not involve any ferries; and
- the `DepartureTime` is `0000`.

Here is an example to demonstrate the expected behaviour of your program for a stage 2 test:

```
prompt$ ./tripPlan
Number of landmarks: 7
Barrangaroo
CircularQuay
FingerWharf
RoyalBotanicGardens
SydneyHarbourBridge
SydneyOperaHouse
TheRocks
Number of walking links: 6
Barrangaroo
TheRocks
17
TheRocks
SydneyHarbourBridge
16
TheRocks
CircularQuay
8
CircularQuay
SydneyOperaHouse
6
SydneyOperaHouse
RoyalBotanicGardens
9
RoyalBotanicGardens
FingerWharf
11
Number of ferry schedules: 0

From: Barrangaroo
To: SydneyHarbourBridge
Departure time: 0000

Walk 17 minute(s):
  0000 Barrangaroo
  0017 TheRocks
Walk 16 minute(s):
  0017 TheRocks
  0033 SydneyHarbourBridge

From: SydneyHarbourBridge
To: RoyalBotanicGardens
Departure time: 0000

Walk 16 minute(s):
  0000 SydneyHarbourBridge
  0016 TheRocks
```

```
Walk 8 minute(s):
  0016 TheRocks
  0024 CircularQuay
Walk 6 minute(s):
  0024 CircularQuay
  0030 SydneyOperaHouse
Walk 9 minute(s):
  0030 SydneyOperaHouse
  0039 RoyalBotanicGardens

From: done
Happy travels!
prompt$
```

Stage 3 (4 marks)

For the next stage, your program should find and output a route from `FromLandmark` to `ToLandmark` that:

- may involve one or more ferries;
- departs at `DepartureTime`.

Note that to board a ferry, it is necessary to arrive at the departing landmark no later than the time the ferry departs.

In all test scenarios for this stage there will be at most one route that satisfies all requirements.

Here is an example to demonstrate the expected behaviour of your program for stage 3:

```
prompt$ ./tripPlan
Number of landmarks: 4
TheRocks
CircularQuay
ManlyWharf
ManlyBeach
Number of walking links: 2
TheRocks
CircularQuay
6
ManlyWharf
ManlyBeach
8
Number of ferry schedules: 2
CircularQuay
1130
ManlyWharf
1152
CircularQuay
1200
ManlyWharf
1222

From: TheRocks
To: ManlyBeach
Departure time: 1125

Walk 6 minute(s):
  1125 TheRocks
  1131 CircularQuay
Ferry 22 minute(s):
  1200 CircularQuay
  1222 ManlyWharf
Walk 8 minute(s):
  1222 ManlyWharf
  1230 ManlyBeach

From: TheRocks
To: ManlyBeach
Departure time: 1200

No route.

From: done
Happy travels!
prompt$
```

Stage 4 (2 marks)

For the final stage, if there are multiple possible routes, your program should take into account the additional user preference that:

- of all possible routes, choose the one with the **shortest overall travel time**.

You may assume that there will never be more than one route with the shortest overall travel time. Note also that travel time includes any time that may be spent waiting for a ferry.

Here is an example to demonstrate the expected behaviour of your program for stage 4:

```
prompt$ ./tripPlan
Number of landmarks: 9
Barrangaroo
CircularQuay
DarlingHarbour
FingerWharf
RoyalBotanicGardens
SydneyHarbourBridge
SydneyOperaHouse
TheRocks
WatsonsBay
Number of walking links: 6
Barrangaroo
TheRocks
17
CircularQuay
RoyalBotanicGardens
9
DarlingHarbour
Barrangaroo
8
RoyalBotanicGardens
FingerWharf
11
TheRocks
```

```
CircularQuay
8
TheRocks
SydneyHarbourBridge
16
Number of ferry schedules: 1
Barrangaroo
1600
CircularQuay
1611

From: DarlingHarbour
To: FingerWharf
Departure time: 1552

Walk 8 minute(s):
  1552 DarlingHarbour
  1600 Barrangaroo
Ferry 11 minute(s):
  1600 Barrangaroo
  1611 CircularQuay
Walk 9 minute(s):
  1611 CircularQuay
  1620 RoyalBotanicGardens
Walk 11 minute(s):
  1620 RoyalBotanicGardens
  1631 FingerWharf

From: DarlingHarbour
To: FingerWharf
Departure time: 1600

Walk 8 minute(s):
  1600 DarlingHarbour
  1608 Barrangaroo
Walk 17 minute(s):
  1608 Barrangaroo
  1625 TheRocks
Walk 8 minute(s):
  1625 TheRocks
  1633 CircularQuay
Walk 9 minute(s):
  1633 CircularQuay
  1642 RoyalBotanicGardens
Walk 11 minute(s):
  1642 RoyalBotanicGardens
  1653 FingerWharf

From: DarlingHarbour
To: WatsonsBay
Departure time: 1600

No route.

From: done
Happy travels!
prompt$
```

Complexity Analysis (1 mark)

You should include a time complexity analysis for the asymptotic worst-case running time of your program, in Big-Oh notation, depending on the size of the input:

1. the number of landmarks, l
2. the number of walking links, w
3. the number of ferry schedules, f.

Hints

If you find any of the following ADTs from the lectures useful, then you can, and indeed are encouraged to, use them with your program:

- linked list ADT : `List.h`, `List.c`
- stack ADT : `Stack.h`, `Stack.c`
- queue ADT : `Queue.h`, `Queue.c`
- priority queue ADT : `PQueue.h, PQueue.c`
- graph ADT : `Graph.h`, `Graph.c`
- weighted graph ADT : `WGraph.h, WGraph.c`

You are free to modify any of the six ADTs for the purpose of the assignment (*but without changing the file names*). If your program is using one or more of these ADTs, you **must** submit both the header and implementation file, even if you have not changed them.

Your main program file `tripPlan.c` should start with a comment: `/* … */` that contains the time complexity of your program in Big-Oh notation, together with a short explanation.

Testing

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` program that corresponds to this assignment. It expects to find, in the current directory, the program `tripPlan.c` and any of the admissible ADTs (`Graph, WGraph, Stack, Queue, PQueue, List`) that your program is using, even if you use them unchanged. You can use dryrun as follows:

```
prompt$ 9024 dryrun tripPlan
```

Please note: Passing dryrun does not guarantee that your program is correct. You should thoroughly test your program with your own test cases.

Submission

For this assignment you will need to submit a file named `tripPlan.c` and, optionally, any of the ADTs named `Graph, WGraph, Stack, Queue, PQueue, List` that your program is using, even if you have not changed them. You can either submit through WebCMS3 or use the command line. For example, if your program uses the `Graph` ADT and the `Queue` ADT, then you should submit:

```
prompt$ give cs9024 assn tripPlan.c Graph.h Graph.c Queue.h Queue.c
```

Do not forget to add the time complexity to your main source code file `tripPlan.c`.

You can submit as many times as you like — later submissions will overwrite earlier ones. You can check that your submission has been received on WebCMS3 or by using the following command:

```
prompt$ 9024 classrun -check assn
```

Marking

This project will be marked on functionality in the first instance, so it is very important that the output of your program be ***exactly*** correct as shown in the examples above. Submissions which score very low on the automarking will be looked at by a human and may receive a few marks, provided the code is well-structured and commented.

Programs that generate compilation errors will receive a very low mark, no matter what other virtues they may have. In general, a program that attempts a substantial part of the job and does that part correctly will receive more marks than one attempting to do the entire job but with many errors.

Style considerations include:

- Readability
- Structured programming
- Good commenting

Collection

Once marking is complete you can collect your marked submission using the following command:

```
prompt$ 9024 classrun -collect assn
```

You can also view your marks using the following command:

```
prompt$ 9024 classrun -sturec
```

You can also collect your marked submission directly through WebCMS3 from the "Collect Submission" tab at the top of this page.

Plagiarism

Group submissions will not be allowed. Your programs must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for similar assessments in previous years, if applicable) and serious penalties will be applied, including an entry on UNSW's plagiarism register.

You are not permitted to use code generated with the help of automatic tools such as GitHub Copilot, ChatGPT, Google Bard.

- ***Do not copy ideas or code from others***
- ***Do not use a publicly accessible repository or allow anyone to see your code***
- ***Code generated by GitHub Copilot, ChatGPT, Google Bard and similar tools will be treated as plagiarism.***

Please refer to the on-line sources to help you understand what plagiarism is and how it is dealt with at UNSW:

- Plagiarism and Academic Integrity
- UNSW Plagiarism Policy
- UNSW Plagiarism Management Procedure

Finally …

Have fun!