The University of Melbourne
School of Computing and Information Systems
COMP10002 Foundations of Algorithms
Semester 1, 2024
Assignment 2
Due: 4pm Tuesday 21 May 2024
Version 1.0

1 Learning Outcomes

In this assignment you will demonstrate your understanding of structures, linked data structures, and algorithm time complexity analysis. You will further extend your skills in program design and implementation.

2 The Story...

In Assignment 1, we have extended your algorithmic searching capability to multidimensional numeric data. In this assignment, we will continue to add text searching to your arsenal. We continue using point of interest (POI) search as the background application to minimise context switching. However, the algorithms you will implement are generic to text search problems and serve as fundamental building blocks of search engines such as Google or Bing. If you have missed Assignment 1, you can still compete this assignment – it would be helpful to revisit the first two pages of the Assignment 1 specification to understand the context in this case.

In Assignment 1, we have assumed that all POIs given are relevant to the queries (e.g., they are all *cafes*), and we only need to filter them based on the POI locations which are numeric properties. In reality, there are POIs of many different *categories* in the same area, and only few are relevant to a query. See Figure 1 for example. There are not only cafes in the Melbourne CBD but also shops, supermarkets, hotels, post offices, a cathedral, etc. When a user queries for "cafes", none of the POIs in the other categories need to be considered. In this assignment, we will implement an algorithm to quickly filter out POIs of irrelevant categories (or web documents in search engines, users in social network searches, etc.).

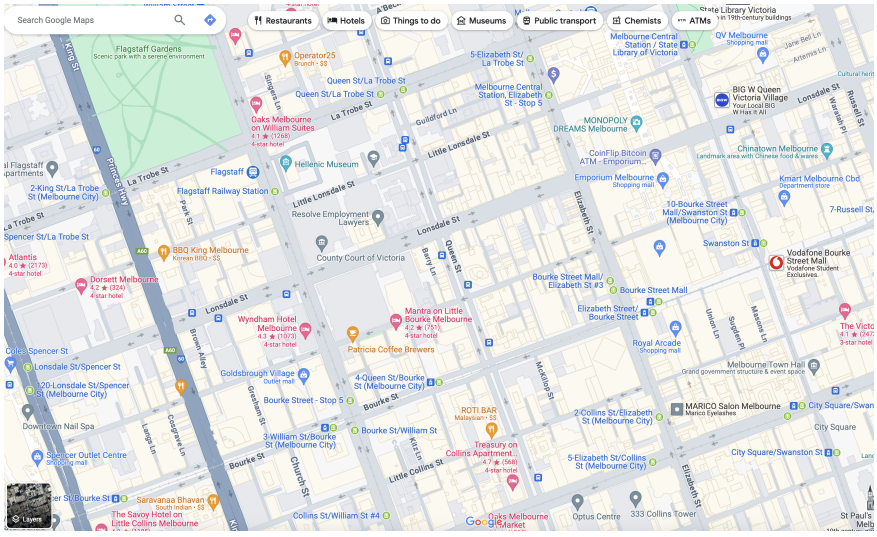

Figure 1: POIs of different categories in Melbourne CBD

3 Your Task

The input POI and query dataset in this assignment is expanded to include POI category information. The input still contains two sections, with a sample input shown below:

1. At least 1 and up to 50 lines of POI records. Each line represents a POI, which starts with an unique integer POI ID of up to two digits (the POI IDs are just the line numbers, to simplify the assignment). Each POI record then contains two real numbers representing the POI coordinates in the x- and y-dimensions. After that, each POI record contains at least 1 and up to 5 *category keywords* ("*category*" for short hereafter) separated by single whitespace characters. Each category is a string of at least 1 and up to 20 lower-case English letters. At the end of each line, there is a special character '`#`' to indicate the end of the line – this is used to simplify input processing for the assignment; it is not part of the POI categories.

   You may assume that there are no repetitions among the categories of a POI. In the sample input below, POI `#0` has coordinates $(16.4, 69.4)$, and its categories are `petrol`, `atm`, and `carwash`.

2. At least 1 and no predefined maximum number of lines of queries. Each line represents a query, which starts with four real numbers representing the query range $(x_{lb}, y_{lb}, x_{ub}, y_{ub})$ as in Assignment 1. Recall that x_{lb} and y_{lb} represent the lower bounds in the x- and y-dimensions of the query range, while x_{ub} and y_{ub} represent the respective upper bounds. Intuitively, each query range is a rectangle whose bottom left corner is at (x_{lb}, y_{lb}) and top right corner is at (x_{ub}, y_{ub}). You may assume that $x_{lb} < x_{ub}$ and $y_{lb} < y_{ub}$ always hold. Each query line is then followed by a *query category*, which is a string of at least 1 and up to 20 lower-case English letters.

All coordinate values are in $(0, 100)$. Note that there is a line with 10 '`#`'s to separate the two input sections.

```
0  16.4 69.4 petrol atm carwash #
1  88.7 13.3 atm #
2  19.5 78.5 shop supermarket atm coles woolworths #
3  85.1 96.1 supermarket #
4  15.4 22.3 cinema movie theatre #
5  22.2 73.5 gas petrol carwash #
6  97.3 68.1 petrol carwash #
7  80.0 32.7 carpark cafe shop #
8  46.8 43.3 hotel atm restaurant carpark #
9  87.3 42.3 cafe atm #
10 87.5 34.6 atm cafe shop #
11 24.7  4.4 theatre cinema atm carpark movie #
##########
11.8  3.5 53.5 28.5 cinema
19.7 58.6 47.1 66.8 atm
16.9 67.6 74.8 93.4 petrol
49.0 70.7 54.9 74.9 supermarket
75.1 25.1 99.9 49.9 cafe
```

You may assume that the test data always follows the format above. No input validity checking is needed.

You will be given a skeleton code file named `program.c` for this assignment on LMS. The skeleton code file contains a `main` function that has been partially completed. There are a few other functions which are incomplete. You need to add code to all the functions including the `main` function for the following tasks.

3.1 Stage 1: Read the POIs (Up to 5 Marks)

Your first task is to add code to the `stage_one` function to read the POI records. You need to define a `struct` named `poi_t` to represent a POI, and an array of this `struct` type to store all POI records. This stage outputs (1) the number of POI records read, (2) the POI with the largest number of categories, and (3) the categories of this POI. If there is a tie, print out the POI with the smallest ID among the tied ones.

Hint: Note the newline character '`\n`' at the end of each input line if you use `getchar` to read the categories.

The output for this stage given the above sample input is shown in the next page (where "`mac:`" is the command prompt).

```
mac: ./program < test0.txt
Stage 1
==========
Number of POIs: 12
POI #2 has the largest number of categories:
shop supermarket atm coles woolworths
```

Like in Assignment 1, we will again use input redirection to feed test data into your program. Thus, you should still use the standard input functions such as `scanf` or `getchar` to read the data. You do not need to (and *should not*) use any file operation functions such as `fopen` or `fread`. Your program should not print anything except for the data requested to be output (as shown in the output example).

You can also modify the `stage_one` function to read all input in one go. You can (and should) create further functions to complete the tasks when opportunities arise.

3.2 Stage 2: Read and Process the Queries (Up to 10 Marks)

Add code to the `stage_two` function to read the queries. You will need to define another `struct` named `query_t` to represent a query, and a linked list to store the input list of queries. You should adapt the linked list structure and code given in the skeleton code for this purpose. Note: If you are not confident with linked lists, you may use an array of the `query_t` type instead, assuming up to 20 queries. This will cost a **2-mark** deduction but will not impact the rest of your assignment implementation.

Then, add code to the `process_queries` function to go through the list (or array) of queries. For each query, calculate and output the IDs of the POIs that are within the query range *and* match the query category. You may use linear search to go through all POIs and their categories to process each query, and do not need to use the advanced search algorithms described in Assignment 1.

We say that a POI at (x, y) is within a query range $(x_{lb}, y_{lb}, x_{ub}, y_{ub})$ if the following inequalities hold:

$$x_{lb} \leq x \leq x_{ub} \text{ and } y_{lb} \leq y \leq y_{ub} \tag{1}$$

Further, we say that a POI *matches* a query category if one of the categories of the POI is *exactly the same* as the query category. The output for this stage given the above sample input should be:

```
Stage 2
==========
POIs in Q0: 4 11
POIs in Q1: none
POIs in Q2: 5
POIs in Q3: none
POIs in Q4: 7 9 10
```

Here, `Qi` to refer to the `i`-th input query. If there is no POI that satisfies a query, we output "`none`". As an example, for `Q2`, even though both POIs `#2` and `#5` are within the query range, only POI `#5` matches the query category `petrol`. Thus, only POI `#5` is outputted.

3.3 Stage 3: Compute the Unique POI Categories (Up to 15 Marks)

Next, we consider a smarter strategy to search the POIs based on their categories. Stage 3 is a preparation step for this purpose. Add code to the `stage_three` function to identify and output the list of all unique POI category strings that have appeared in the input POI records. The POI categories should be outputted in the order that they appear in the input. To simplify the assignment, you may assume that there are at most 50 unique POI categories in the input, and you may use an array of strings to store the unique POI categories. Given the sample input above, the output of this stage is shown below (5 categories per line).

```
Stage 3
==========
15 unique POI categories:
petrol, atm, carwash, shop, supermarket
coles, woolworths, cinema, movie, theatre
gas, carpark, cafe, hotel, restaurant
```

Hint: Study `words.c` (`https://people.eng.unimelb.edu.au/ammoffat/ppsaa/c/words.c`) for how to identify all unique words from input data and store them into an array.

3.4 Stage 4: Construct an Inverted Index (Up to 20 Marks)

Finally, add code to the `stage_four` function to construct an *inverted index* over the POI as follows:

1. Define a `struct` type named `index_t` of three fields:
   - `category`, which is a string of up to 20 lower-case English letters;
   - `pois`, which is an `int` array to store the IDs of all POIs that match the `category`; and
   - `num_matched_pois`, which is the number of POIs that match the `category`.

   For example, as shown in the sample output below, category `cafe` matches with POIs `#7`, `#9`, and `#10`. These three POI IDs are supposed to be stored in the `pois` array corresponding to category `cafe`, while `num_matched_pois` is 3.
2. Create an array of `index_t` type. Let us name this array `index`. This array will have the same size as the array of unique POI categories created in Stage 3. Each element of the `index` array stores a unique POI category in its `category` field.
3. Sort the `index` array created in Step 2, in ascending alphabetical order (that is, dictionary order) of the POI categories stored in the `category` field of each array element.

   Hint: You may adapt the insertion sort code from Assignment 1 for this step, or use the `qsort` function from `stdlib.h` (see `https://people.eng.unimelb.edu.au/ammoffat/ppsaa/c/callqsort.c` for a code example).
4. Go through the array of all POIs constructed in Stage 1, and each POI's categories. For each category (denoted by `cat`) of a POI, find the element of the `index` array whose `category` field matches `cat`. Once such an element is found, add the ID of the POI to the end of the `pois` array of this element.

   Hint: You may use either linear search or binary search for this step. If you use binary search, you may adapt the binary search code from Assignment 1, or use the `bsearch` function from `stdlib.h`.

The output of this stage is the content of the `index` array, where each array element is printed in one line. Given the sample input above, the output of this stage is as follows.

```
Stage 4
==========
atm: 0 1 2 8 9 10 11
cafe: 7 9 10
carpark: 7 8 11
carwash: 0 5 6
cinema: 4 11
coles: 2
gas: 5
hotel: 8
movie: 4 11
petrol: 0 5 6
restaurant: 8
shop: 2 7 10
supermarket: 2 3
theatre: 4 11
woolworths: 2
```

Query algorithm (for analysis and *not* for implementation). Once the inverted index is constructed, when a query comes, we can run a binary search over the inverted index (that is, the `index` array) to find the array element whose `category` field matches the query category. Then, we can perform a linear search over the `pois` array of the found `index` array element to identify the POIs in the query range. For example, for query `Q4`, we only need to examine POIs `#7`, `#9`, and `#10`.

At the end of your submission file, you need to add a comment that states:

1. the worst-case time complexity to process a *single* query using the linear search algorithm of Stage 2,
2. the worst-case time complexity to process a *single* query using the query algorithm described in the paragraph above (without actually implementing it), and **(continued on next page)**

3. the reason why the two algorithms have those time complexities.

In your analysis, use N to denote the number of POIs, C to denote the maximum number of categories per POI, L to denote the maximum length of a category, and U to denote the number of all unique POI categories.

4 Submission and Assessment

This assignment is worth 20% of the final mark (5% per stage). A detailed marking scheme will be provided on LMS.

Submitting your code. You should put all your code for the assignment into a single file named `program.c`. To submit your code, you will need to: (1) Log in to LMS subject site, (2) Navigate to "Assignment 2" in the "Assignments" page, (3) Click on "Load Assignment 2 in a new window", and (4) follow the instructions on the Gradescope "Assignment 2" page and click on the "Submit" link to make a submission. You can submit as many times as you want to. *Only the last submission made before the deadline will be marked.* Submissions made after the deadline will be marked with late penalties as detailed at the end of this document. Do *not* submit after the deadline unless a late submission is intended. Two hidden tests will be run for marking purposes. Results of these tests will be released after the marking is done.

You can (and should) submit both **early and often** – to check that your program compiles correctly on our test system, which may have some different characteristics to your own machines.

Testing on your own computer. You will be given a sample test file `test0.txt` and the sample output `test0-output.txt`. You can test your code on your own machine with the following command and compare the output with `test0-output.txt`:

```
mac: ./program < test0.txt    /* Here '<' feeds the data from test0.txt into program */
```

Note that we are using the following command to compile your code on the submission testing system (we name the source code file `program.c`).

```
gcc -Wall -std=c17 -o program program.c -lm
```

The flag "`-std=c17`" enables the compiler to use a modern standard of the `C` language – `C17`. To ensure that your submission works properly on the submission system, you should use this command to compile your code on your local machine as well.

You may discuss your work with others, but what gets typed into your program must be individual work, **not** from anyone else. Do **not** give (hard or soft) copies of your work to anyone else; do **not** "lend" your memory stick to others; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "no" when they ask for a copy of, or to see, your program, pointing out that your "no", and their acceptance of that decision, is the only thing that will preserve your friendship. *A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode.* See `https://academichonesty.unimelb.edu.au` for more information.

Deadline: Programs not submitted by **4pm Tuesday 21 May 2024** will lose penalty marks at the rate of 3 marks per day or part day late. Late submissions after 4pm Friday 24 May 2024 will **not** be accepted. Students seeking extensions for medical or other "outside my control" reasons should email the lecturer at jianzhong.qi@unimelb.edu.au. If you attend a GP or other health care professional as a result of illness, be sure to take a Health Professional Report (HRP) form with you (get it from the Special Consideration section of the Student Portal), you will need this form to be filled out if your illness develops into something that later requires a Special Consideration application to be lodged. You should scan the HPR form and send it in connection with any non-Special Consideration assignment extension requests.

And remember, *Algorithms are fun!*