The University of Melbourne
School of Computing and Information Systems
COMP10002 Foundations of Algorithms
Semester 1, 2024
Assignment 1
Due: 4pm Tuesday 30 April 2024
Version 1.1

1 Learning Outcomes

In this assignment, you will demonstrate your understanding of arrays, pointers, input processing, and functions. You will also extend your skills in terms of code reading, program design, testing, and debugging.

2 The Story...

Given an array of n 1-dimensional values, we have learned that sorting the array will enable a fast $O(\log_2 n)$-time search to look up for a *search key* (that is, a number to be located in the array). In this assignment, we will extend our search capability to an array of 2-dimension values, where the ordering is less obvious.

Consider digital mapping services like Google Maps. Such services have millions of records of business and *points of interest* (POIs), if not more. A common query is to show the POIs in the mapping app view of a user. See Figure 1 for example, where a user is querying for *cafes* in the Melbourne CBD. Such queries happen frequently – Google Maps alone serves over 1 billion monthly active users (`https://zipdo.co/statistics/google-maps/`). For services of such a scale, a simple linear search over all POI records for each query is sub-optimal.

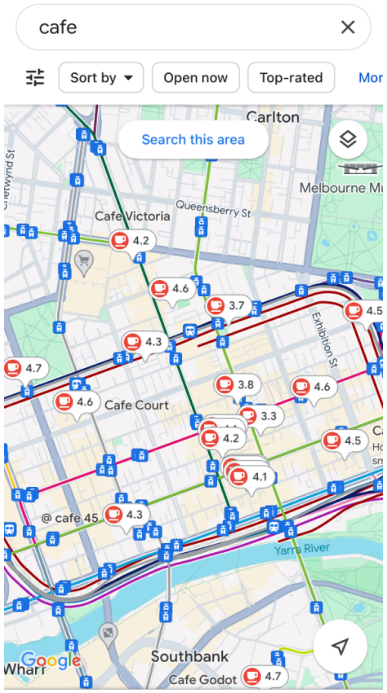

Figure 1: A map query example

In this assignment, we will design an algorithmic solution for more efficient query processing. Our aim is to achieve (approximately) an $O(\log_2 n + k)$-time search for 2-dimensional points on maps, where n represents the total number of POI records in a database, and k represents the number of POIs satisfying a query.

3 Your Task

You will be given a skeleton code file named `program.c` for this assignment. The skeleton code file contains a `main` function that has been completed (which you do *not* have to modify unless you want to). There are a few other functions which are incomplete. You need to add code to complete the functions for the following tasks.

The given input to the program consists of 15 lines of numbers as follows:

- The first 10 lines contains the IDs (unique integers) and coordinates (real numbers) of 50 POI records separated by '`;`', to run our query algorithms upon. The POI records are sorted by their IDs in ascending order, and their IDs are always from `0` to `49` to simplify the assignment. Each line contains

five POI records. For the sample input shown below, the first record has ID `0`, and its coordinates in the x- and y-dimensions are `16.4` and `69.4`, respectively.

- The next five lines represent the queries. Each line contains four real numbers representing a query range $(x_{lb}, y_{lb}, x_{ub}, y_{ub})$, where 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.

You may assume that the test data always follow the format above, and that each coordinate value and query range bound is in the range of $(0, 100)$. *No input validity checking is needed.* See below for a sample input.

```
 0 16.4 69.4;  1 88.7 13.3;  2  1.8 98.8;  3 85.1 96.1;  4 15.4 22.3;
 5 79.4 61.9;  6 97.3 68.1;  7 68.3  9.3;  8 46.8 43.3;  9 87.3 42.3;
10 38.9 46.5; 11 87.5 34.6; 12 34.7 40.9; 13  5.8 37.4; 14 28.6 55.8;
15 60.7 70.4; 16 73.5 63.1; 17 76.8 92.1; 18 24.7  4.4; 19 15.3 46.4;
20 15.5 51.7; 21 55.1 99.5; 22 95.0 13.2; 23 54.0 97.1; 24  6.4 37.8;
25 66.0 16.4; 26 59.2 88.2; 27 59.4 81.6; 28 79.9 68.5; 29 61.5  0.5;
30  5.2 69.1; 31 76.9 74.9; 32 29.7 50.3; 33 19.5 78.5; 34 12.1 83.4;
35 35.6  1.2; 36 98.5 97.5; 37 95.4 78.4; 38 80.0 32.7; 39 22.2 73.5;
40 80.7 42.0; 41  9.8 10.9; 42 81.0 93.4; 43 97.5 30.4; 44 28.2 44.1;
45 37.2 97.0; 46 73.6  9.8; 47 68.5 17.9; 48 65.5 50.0; 49 21.0 48.0;
11.8  3.5 53.5 28.5
19.7 58.6 47.1 66.8
16.9 67.6 74.8 93.4
49.0 70.7 54.9 74.9
75.1 25.1 99.9 49.9
```

Figure 2 plots the 50 POIs and the five query ranges in the sample input, where the bottom left point is POI `#41` at `(9.8, 10.9)`, and the bottom left rectangle (the one in red) is the first query range.

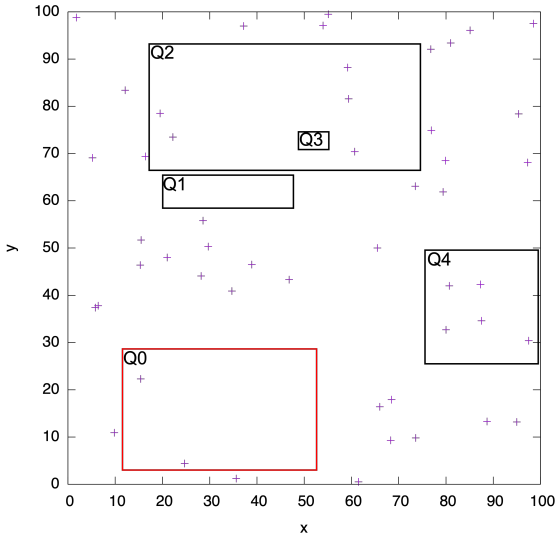

Figure 2: The sample POIs and query ranges

3.1 Stage 1: Read the Input and Answer the First Query (Up to 5 Marks)

Your first task is to understand the skeleton code. Then, you should add code to the `stage_one` function to (1) read input POI IDs and coordinates into the arrays `ids` and `coordinates`, respectively, and read query ranges into `queries`, (2) identify all POIs within the first query range using linear search, and (3) print out such POIs (print "`none`" if no such POIs can be found).

The output for this stage given the above sample input should be (where "`mac:`" is the command prompt):

```
mac: ./program < test0.txt
Stage 1
==========
POIs in Q0: 4 18
```

Here, `Q0` refers to the first input query, that is, we use `Qi` to refer to the `i`-th input query.

Hint: To test if a point at (x, y) is within a query range $(x_{lb}, y_{lb}, x_{ub}, y_{ub})$, we test the following inequalities:

$$x_{lb} \leq x \leq x_{ub} \text{ and } y_{lb} \leq y \leq y_{ub} \quad (1)$$

In the sample input above, for POI `#4` at `(15.4, 22.3)` and the first query range `(11.8, 3.5, 53.5, 28.5)`, we have:

$$11.8 \leq 15.4 \leq 53.5 \text{ and } 3.5 \leq 22.3 \leq 28.5$$

Thus, POI `#4` is within the first query range and is part of the Stage 1 output. The same applies for POI `#18`.

As this example illustrates, the best way to get data into your program is to edit it in a text file (with a "`.txt`" extension, any text editor can do this), and then execute your program from the command line, feeding the data in via input redirection (using `<`). In the program, we will still use the standard input functions such as `scanf` to read the data fed in from the text file. Our auto-testing system will feed input data into your submissions in this way as well. You do not need to (and *should not*) use any file operation functions such as `fopen` or `fread`. To simplify the assessment, your program should not print anything except for the data requested to be output (as shown in the output example).

You should plan carefully, rather than just leaping in and starting to edit the skeleton code. Then, before moving through the rest of the stages, you should test your program thoroughly to ensure its correctness.

You should create sub-functions to complete the tasks. As stated in the marking rubric, you need to create at least two self-defined functions in your final submission for the assignment.

3.2 Stage 2: Sort and Query POIs by the x-coordinates (Up to 10 Marks)

In Stage 1, we have examined all POI records to identify those within a query range. In the following stages, we aim to establish some ordering over the POIs, such that the unpromising POIs can be filtered without being examined, and query processing can be accelerated. For Stage 2, we will start by ordering the POIs by their x-coordinates.

Stage 2.1. Modify the given `sort_by_x` function, such that it takes the arrays of `ids` and `coordinates` as the input, and sorts both arrays based on the x-coordinates of the POIs in ascending order (that is, increasing order) with the *insertion sort* algorithm. For simplicity, you may assume that the x-coordinates of all input POIs are unique (in reality, if there are repeating x-coordinates, we can further sort by the y-coordinates). The `stage_two` function calls the modified `sort_by_x` function and performs the sorting.

Stage 2.2. Now further add code to the `stage_two` function to process all the input queries. For each query, we start with a linear scan (**Scan 1**) to find the POI with the smallest x-coordinate that is greater than or equal to x_{lb} of the query range. Let us call this POI the *lower bound POI*. From this POI, we continue with the linear scan (**Scan 2**), until we reach a POI whose x-coordinate exceeds x_{ub} of the query range. For each POI visited during Scan 2, we examine whether it is within the query range in the y-dimension (we call this a *POI-in-query test*), if so, it is part of the query answer and is outputted.

The output for this stage is the IDs of the POIs within each query, and the number of POI-in-query tests run for each query. For example, given the above sample input, the sample output of this stage is:

```
Stage 2
==========
POIs in Q0: 4 18; No. POI-in-query tests: 17
POIs in Q1: none; No. POI-in-query tests: 11
POIs in Q2: 33 39 26 27 15; No. POI-in-query tests: 24
POIs in Q3: none; No. POI-in-query tests: 1
POIs in Q4: 38 40 9 11 43; No. POI-in-query tests: 16
```

For example, as shown in Figure 2, for `Q0`, there are 17 points between x_{lb} and x_{ub} (that is, the two vertical edges) of `Q0`. Thus, 17 POI-in-query tests need to be run to find the two POIs `#4` and `#18` within the query range. Now the number of POI-in-query tests per query is just up to 24 (for the five input queries), that is, fewer than half of all POIs, rather than examining all POIs for a query as done in Stage 1.

For a challenge and ***not*** *for submission, see if you can replace Scan 1 with a binary search.*

3.3 Stage 3: Sort POIs by Coordinates of Both Dimensions (Up to 15 Marks)

Stage 3 further incorporates the y-coordinates into POI ordering. Figure 3 illustrates the idea. We partition the space with an $m \times m$ regular grid. In this assignment, $m = 8$. A curve (the black dotted line in the figure) goes through each cell in the grid exactly once. The order that the curve goes through the cells gives a *curve value* for each cell. For example, as shown in the figure, the bottom left cell has a curve value of 0. The cell to its right has a curve value of 1, while the cell above it has a curve value of 2. The top right cell has a curve value of 63 (as there are $8 \times 8 = 64$ cells in total).

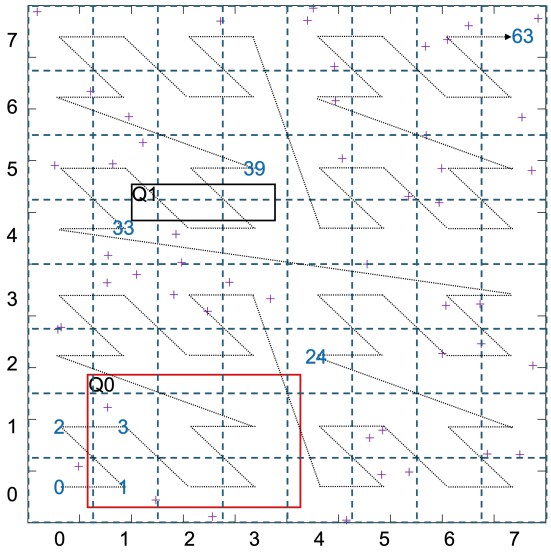

Figure 3: Curve-based POI ordering

For a POI with coordinates (x, y), we can calculate the curve value corresponding to the cell that encloses the POI ("the curve value of the POI" for short hereafter) as follows. First, we calculate the column number (col_num) and row number (row_num) of the POI using the following equations.

$$col\_num = \lceil \frac{x}{12.5} \rceil - 1; \quad row\_num = \lceil \frac{y}{12.5} \rceil - 1 \tag{2}$$

Here, $\lceil z \rceil$ is the ceiling function. It returns the smallest integer greater than or equal to z. In `C`, `ceil()` from `math.h` provides an implementation of this function. The constant $12.5 = \frac{100}{8}$ is the width (height) of each column (row) – recall that 100 is the coordinate value range and 8 is the number of columns (rows) of the grid. For example, for the bottom left point POI `#41` at `(9.8, 10.9)`:

$$col\_num = \lceil \frac{9.8}{12.5} \rceil - 1 = 0; \quad row\_num = \lceil \frac{10.9}{12.5} \rceil - 1 = 0 \tag{3}$$

Then, the curve value of the POI can be calculated by calling the `cal_curve_value(col_num, row_num)` function given as part of the skeleton code (*you should* ***not*** *modify the given* `cal_curve_value` *function*; if you are interested in what the function does, read Section 13.2 of the textbook on bitwise operators).

In this stage, you will add code to the `stage_three` function to calculate the curve values for the POIs (by calling the given `cal_curve_value` function with proper values of col_num and row_num), and sort the POIs by their curve values in ascending order – if there is a tie in the curve values, the POI with a smaller ID should be listed earlier. You may create another sorting function again based on the insertion sort algorithm. *Hint:* You may need to use the array `curve_values` to store the curve values for the POIs.

The output of your code should be the first five POIs after ordering by their curve values. To align the output, use `%02d` for the POI IDs and `%04.1` for the coordinates. See a sample output below.

```
Stage 3
==========
POI #41 @ (09.8, 10.9), curve value: 0
POI #18 @ (24.7, 04.4), curve value: 1
POI #04 @ (15.4, 22.3), curve value: 3
POI #35 @ (35.6, 01.2), curve value: 4
POI #13 @ (05.8, 37.4), curve value: 8
```

3.4 Stage 4: Query POIs by Curve Values (Up to 20 Marks)

This stage implements a query algorithm with the curve values, by adding code to the `stage_four` function.

Our query algorithm is based on the following observation. Consider a query range $(x_{lb}, y_{lb}, x_{ub}, y_{ub})$. Let the curve value of the bottom left corner point (x_{lb}, y_{lb}) be v_{lb}, and that of the top right corner point (x_{ub}, y_{ub}) be v_{ub} (these curve values can be calculated in the same way as in Stage 3 using the corner point coordinates). Then, for any POI within the query range, its curve value must be in the range of $[v_{lb}, v_{ub}]$. For example, given query `Q0` in Figure 3, $v_{lb} = 0$ and $v_{ub} = 24$. The curve values of the two POIs in `Q0` are 1 and 3, which are both in $[0, 24]$.

Based on the observation, for each query range, the query algorithm runs as follows:

- Step 1. First, calculate and output v_{lb} and v_{ub} (use `%02d` for formatting). Then, run a binary search over the `curve_values` array produced in Stage 3 to locate the curve value greater than or equal to v_{lb} that is ranked the earliest in the array. Let the POI corresponding to this curve value be the *curve value lower bound POI*.
- Step 2. Run a linear scan over the array of POI coordinates (which has been sorted by the curve values in Stage 3) starting from the curve value lower bound POI, until reaching a POI whose curve value exceeds v_{ub}. For each POI visited during the scan, we examine whether it is within the query range (that is, to run POI-in-query tests – this time, we need to test the coordinates in both dimensions), and if so, it is part of the query answer and its ID is outputted. At the end of this step, we also output the number of POI-in-query tests run, like we did in Stage 2.

Note: You should adapt the `binary_search` function included in the skeleton code for Step 1 above, to output the `curve_values` array elements that have been compared with during the search process. *For marking purposes, you are* not *allowed to use binary search code from other sources, or to create your own binary search functions from scratch.* If you are not confident with your binary search implementation, you can replace it with a linear search for the same purpose. This will cost a **2-mark** deduction (the full mark of the assignment becomes 18) but will not impact the rest of your assignment implementation.

The output for this stage given the sample input above is shown below (note the final newline '`\n`').

```
Stage 4
==========
Q0: [00, 24]; curve values compared: 33 16 11 4 1 0
POIs in Q0: 18 4; No. POI-in-query tests: 19
Q1: [33, 39]; curve values compared: 33 16 27 29 30
POIs in Q1: none; No. POI-in-query tests: 6
Q2: [35, 59]; curve values compared: 33 54 41 36 35 34
POIs in Q2: 39 33 15 27 26; No. POI-in-query tests: 18
Q3: [39, 50]; curve values compared: 33 54 41 36 40 36
POIs in Q3: none; No. POI-in-query tests: 5
Q4: [28, 31]; curve values compared: 33 16 27 29 28 28
POIs in Q4: 11 38 43 9 40; No. POI-in-query tests: 5
```

Take `Q0` in Figure 3 as an example. At Step 1, we calculate $v_{lb} = 0$ and $v_{ub} = 24$. The binary search over the array `curve_values` will examine `33`, `16`, `11`, `4`, `1`, and `0` in the array – the last element examined, `0`, is the element ranked the earliest in `curve_values` that is greater than or equal to v_{lb}. This curve value `0` is the *first* element in the `curve_values` array. Thus, we perform a linear scan over the POI `ids` and `coordinates` arrays starting from their *first* elements. A total of 19 POIs are visited in the process, until we reach a POI whose curve value exceeds $v_{ub} = 24$ (see the POIs in Figure 3 in cell 0 to cell 24).

As you may have observed, even using the ordering based on curve values, there are POIs examined that are not exactly within the query ranges (these are called *false positives*, which need to be filtered during query processing). While the curve-based ordering helps reduce the number of POI-in-query tests in general, there is no guarantee that this is always the case, and there is no known solution with such a guarantee. The state-of-the-art solution for the problem offers $O(\sqrt{n} + k)$ query time in the worst case. Refer to `https://people.eng.unimelb.edu.au/jianzhongq/papers/TODS2020_RtreeRankSpaceSFC.pdf` if you are interested in learning more about the problem and solution.

Open challenge (no answer needed in your assignment submission, but you are welcomed to share your thoughts on Ed with *private* posts): Can you think of other strategies to make 2-dimensional range queries even more efficient? How about extending to d-dimensional range queries for any integer $d > 2$?

4 Submission and Assessment

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

Submitting your code. To submit your code, you will need to: (1) Log in to LMS subject site, (2) Navigate to "Assignment 1" in the "Assignments" page, (3) Click on "Load Assignment 1 in a new window", and (4) follow the instructions on the Gradescope "Assignment 1" 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 30 April 2024** will lose penalty marks at the rate of 3 marks per day or part day late. Late submissions after 4pm Friday 3 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!*