The University of Melbourne
School of Computing and Information Systems
COMP10002 Foundations of Algorithms
Semester 1, 2017
Assignment 1
Due: 4pm Wednesday 26th April 2017

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...

Recommender engines are used in various systems to predict what a user may like and to make recommendations accordingly. For example, amazon recommends relevant products at its product pages; YouTube recommends relevant videos for every video played. A good recommender engine is crucial for such systems to boost product sale and/or to keep users in. Companies invest heavily on their recommender engines. Netflix, for example, used to host an open competition for recommender algorithms to predict user ratings on films. On September 21, 2009, Netflix awarded the Grant Prize (US$1,000,000)[1] to a team that came up with an algorithm that reduces the prediction errors by 10.06% comparing with Netflix's own algorithm.

Recommender engines often make recommendations based on user history data such as purchase records. However, user history data are not always available, especially when a new service has just been launched. Making recommendations without user history data – a.k.a. the *cold start* problem – is challenging.

In this assignment, we will try to tackle the cold start problem with a technique called the *skyline operator*. The name of this technique comes from the city skyline. Looking out of the window, we see a skyline formed by a series of buildings. Notice that a building can only appear in view if it is either close or tall enough to *not* be blocked by any other building. A building behind Eureka Tower will be blocked by it and will not be seen. This observation applies in making recommendations as well. Ideally, a product p_1 (e.g., a laptop computer) should be recommended only if there does *not* exist any other product p_2 that is better than p_1 in all possible measures (e.g., higher CPU frequency, larger storage, etc.). Otherwise, no reasonable consumer would consider p_1, and there is no point recommending it.

Consider a set P of n products ($n > 0$). Each product p is represented by a d-dimensional point ($d > 0$):

$$p = \langle c_1, c_2, ..., c_i, ..., c_d \rangle$$

Here, c_i is the coordinate of p in dimension i ($0 < i <= d, c_i > 0$). The d coordinates represent p's value in different measures, e.g., CPU frequency, disk size, etc. Table 1 shows an example of such a product set.

The skyline operator returns the subset $P' \subseteq P$ of all the points that are *not dominated* by any point in P. A point p_x is dominated by another point p_y if the coordinates of p_x are **less than or equal to** those of p_y in all dimensions[2], that is, p_x is not better than p_y in any dimension. In Table 1, p_1 is dominated by p_3, as p_1 has smaller values in all of CPU frequency, disk size, and memory size. Similarly, p_5 is also dominated by p_3, while both p_5 and p_6 are dominated by p_4. All other points (p_2, p_3, p_4, p_7) are not dominated by any

[1] https://www.netflixprize.com
[2] Assume that there are no two points with exactly the same coordinates in all dimensions.

Table 1: A Set of Laptop Computers

Laptop	CPU frequency (c_1)	Disk size (c_2)	Memory size (c_3)
p_1	2.3	300	20
p_2	1.7	1000	16
p_3	2.8	500	32
p_4	2.1	750	16
p_5	2.1	500	12
p_6	1.2	600	10
p_7	0.9	900	99

point. The non-dominated points are called the *skyline points*, and are returned as the set P'. Note that p_4 does not have the largest value in any dimension, but it is still not dominated as it has a higher CPU frequency than p_2 and p_7 and a larger disk size than p_3.

3 Your Task

You will be given a list of input in the following format. The first input line contains a positive integer representing the number of dimensions d. You may assume that $0 < d \leq 10$. Each following line contains d positive real numbers separated by whitespace characters, representing the coordinates of a point (product). *You may assume that the input is always valid and correctly formated.* There are at least one and at most 99 point lines. A sample input is shown below.

```
3
2.3 300 20
1.7 1000 16
2.8 500 32
2.1 750 16
2.1 500 12
1.2 600 10
0.9 900 99
```

Your task is to write a program that implements the skyline operator. You will be given a skeleton code file named "`assmt1.c`" for this assignment in LMS. The skeleton code file contains a `main` function that has been completed already. There are a few other functions which are incomplete. You need to add code into them for the following tasks (Stage 5 is for a challenge and is optional). **Note that you should *not* change the `main` function, but you are free to modify any other parts of the skeleton code (including adding more functions)**.

3.1 Stage 1 - Reading in a Point (Up to 3 Marks)

Your first task is to understand the skeleton code. Note the use of the type `point_t` in the skeleton code, which is essentially a `double` type array. Each `point_t` variable stores the coordinates of one point.

You need to add code to the `stage_one` function to call the `read_one_point` function to read the first point, and print out its coordinates (print two digits after the decimal point, same for Stages 4 and 5). The output of this stage given the above sample input should be (where "`mac:`" is the command prompt):

```
mac: ./assmt1 < test0.txt
Stage 1
==========
Point 01: <2.30, 300.00, 20.00>
```

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, `jEdit` 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. 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 make sure its correctness.

3.2 Stage 2 - Processing the Rest of the Points (Up to 8 Marks)

Now add code to the `stage_two` function so that the sum of coordinates of each input point is computed. This sum is then divided by 100 and visualised. You may assume that the sum of coordinates is within the range of (0, 10000). On the same sample input data the additional output for this stage should be:

```
Stage 2
==========
Point 01, sum of coordinates (/100):  3.22 |----
Point 02, sum of coordinates (/100): 10.18 |---------+-
Point 03, sum of coordinates (/100):  5.35 |------
Point 04, sum of coordinates (/100):  7.68 |--------
Point 05, sum of coordinates (/100):  5.14 |------
Point 06, sum of coordinates (/100):  6.11 |-------
Point 07, sum of coordinates (/100): 10.00 |---------+
```

For example, the coordinates of the first point are 2.30, 300.00, and 20.00. The sum of the coordinates is $2.30 + 300.00 + 20.00 = 322.3$, which yields 3.223 when divided by 100. Only the first two digits after the decimal point are printed out, that is, 3.22 is printed out instead of 3.223. This can be done via using "`%5.2f`" in the `printf` function. You need to work out how the rest of the visualisation works based on this example. You should write additional functions to process this stage where appropriate.

Note that the `stage_two` function has been partially completed as a hint for you. You may choose to complete the rest of the function, or to rewrite the whole function body, although you should *not* change the function signature.

You need to use the `-lm` flag in the compilation if you use any library functions from `math.h`.

3.3 Stage 3 - Overall Reporting (Up to 10 Marks)

Further add code to the `stage_three` function. The additional output is the total number of points, the point number with the largest sum of coordinates, and the largest sum of coordinates (print two digits after the decimal point). Again, only the first two digits after the decimal point should be output. In the case of ties of sum of coordinates, the smallest point number should be output.

```
Stage 3
==========
Total: 7 points
Point of largest sum of coordinates: 02
Largest sum of coordinates: 1017.70
```

3.4 Stage 4 - Finding the Skyline Points (Up to 15 Marks)

Now modify the `stage_four` function to find the skyline points from the input and print them out. The additional output for this stage given the sample input above is as follows.

```
Stage 4
==========
Skyline points:
Point 02: <1.70, 1000.00, 16.00>
Point 03: <2.80, 500.00, 32.00>
Point 04: <2.10, 750.00, 16.00>
Point 07: <0.90, 900.00, 99.00>
```

3.5 Stage 5 - Sorting the Skyline Points (Up to 15 Marks)

This stage is for a challenge. Do not start on this stage until your program through to Stage 4 is (in your opinion) perfect, and is submitted, and is verified, and is backed up somewhere.

Add code to the `stage_five` function to sort and output the skyline points in the descending order of the number of points they dominate. When there is a tie, the point appears earlier in the input should be output earlier. This will enable recommender engines to return a few top skyline points dominating the most points, which is a useful feature. The additional output given the sample input above is as follows.

```
Stage 5
==========
Sorted skyline points:
Point 03: <2.80, 500.00, 32.00>
Number of points dominated: 2
Point 04: <2.10, 750.00, 16.00>
Number of points dominated: 2
Point 02: <1.70, 1000.00, 16.00>
Number of points dominated: 1
Point 07: <0.90, 900.00, 99.00>
Number of points dominated: 0
```

You may use any sorting algorithm, library function, or sample code from the textbook. Make sure to attribute the source if you use any sample code. If you are successful in this stage, you can earn back 1 mark you lost in the earlier stages (assuming that you lost some, your total mark will not exceed 15).

4 Submission and Assessment

This assignment is worth 15% of the final mark. A detailed marking scheme will be provided on the LMS.

You need to submit all your code in one file named `assmt1.c` for assessment; detailed instructions on how to submit will be posted on the LMS once submissions are opened. Submission will NOT be done via the LMS; instead you will need to log in to a Unix server and submit your files to a software system known as `submit`. You can (and should) use submit both **early and often** - to get used to the way it works, and also to `verify` that *your program compiles correctly on our test system, which has some different characteristics (compiler version) to the lab machines and your own computers.* Only the last submission made before the deadline will be marked.

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: ./assmt1 < test0.txt    /* Here '<' feeds the data from test0.txt into assmt1 */
```

You may discuss your work with others, but what gets typed into your program must be individual work, **not** copied from anyone else. Do **not** give hard copy or soft copy 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 Wednesday 26th April 2017** will lose penalty marks at the rate of 2 marks per day or part day late. Late submissions after 4pm Friday 28th April 2017 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 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!*