CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, MARCH 4, 2022 AT 11:30 PM VIA COLLAB

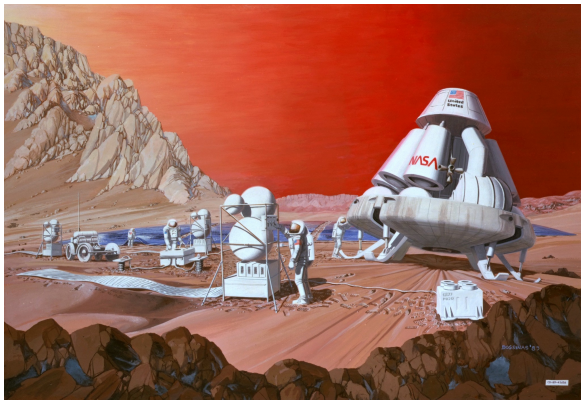
A PROGRAM

Collaboration Policy: You are encouraged to collaborate with up to 3 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in a comment at the top of your files. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite by naming the book etc. or listing a website's URL. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

Collaborators: List computing ids in comments at the top of your file

Sources: List URLs etc. in comments at the top of your file

PROBLEM 1 *Choosing Mars Pioneers*



A rich entrepreneur is financing a space exploration program that will culminate is a mission to Mars. There's a lot of excitement because he's going to choose four ordinary citizens to be Mars "pioneers." After arriving on Mars, the pioneers will work in pairs; each pair will focus on one particular Mars pioneering kind of task. (You know, like digging wells, oppressing the natives, growing potatoes, etc.) Thus it is critically important to find two pairs of citizens where each pair is highly compatible under stressful working conditions.

Luckily for this rich entrepreneur, our course instructors have a side hustle. Using both their algorithmic and considerable life-skills, they founded a company called *Bipartite Matchmakers* that uses a secret and highly effective means of matching up pairs of people. While initially designed to help connect students during the semesters of online teaching, they quickly adapt it to identify compatible space pioneers and win the contract to choose the two pairs of people who will travel to Mars.

Their match-making system works by first getting a lot of relevant info about match candidates. Then this complex and multi-dimensional data about each candidate is mapped to a point on a two-dimensional grid in such a way that the distance between any two points indicates a higher degree of compatibility. *(Note: how this mapping is done uses a highly-secret technique, which isn't relevant here. Really, it's none of your business! But rest assured we never pair up dog-people with cat-people.)*

Millions of people have applied to be Mars pioneers. We need an efficient algorithm to process the millions of two-dimensional data points representing each person and find a group with the two most compatible pairs. That is, the pair that is most compatible and the pair that is second most compatible. Specifically, this means finding the pair of points that has the minimum distance between them, and also the pair of points that has the second-minimum distance. *There may be overlap in our pairings, where one person is in both pairs, and that's okay!* Succeeding in this task is crucial for the mission to Mars as well as the professors' plans to fund their retirement to a warm tropical island courtesy of the rich entrepreneur's money.

Implementation Note: As the problem has been explained above, we would need to find actual pairs. But for this assignment we are asking you only to find the distances and not the pairs of points associated with those distances. It would be an easy change to find both, but you'll just need to find and return the two distances and not the points themselves.

Results Calculated and Returned

You'll write a function (method) that finds and returns the minimum distance and the second-minimum distance. Both of these will be floating point values. The wrapper code (described below) shows how your function will return these two values to our driver that will call your function. **Note:** If there are two pairs of points that are equally close, and that distance is less than or equal to the distance between any other pair of points, then the two values you return will be the same.

Input

The input will be a sequence of points, which will be passed to your function as a list of strings. Each string will be an (x, y) pair separated by whitespace. Each x and y value will represent a real number with a minimum value of $-50,000.0$ and a maximum value of $50,000.0$.

Other Constraints

Your program should have the following properties:

- Your algorithm must be written in Python (2 or 3), Java (11), or C++.
- You must download the appropriate wrapper code from Collab based on the language you choose: `Main.py` and `ClosestPair.py` for Python, `Main.java` and `ClosestPair.java` for Java, `Main.cpp` and `ClosestPair.cpp` for C++.
- The `compute()` method in `ClosestPair` receives as input the body of the input file as a list of strings. You must have `compute()` return the results described above. An example input file is shown in Figure 1, which has answer ≈ 4.12311 and ≈ 5.65685.
- You should write **another** function in the `ClosestPair` file to implement the Closest Pair algorithm. Your `compute()` method will then invoke this function. Do **not** modify the method definition for `compute`.
- You should think about what intermediate representation you want to use to represent the list of coordinates. *You will probably prefer a different representation than a list of strings.* We suggest you also write a helper method to parse the input into your chosen representation.
- Two additional test files are provided, `test1.txt` and `test2.txt`, both with answer $\approx$ 1.4142135623 and $\approx$ 2.2360679774. You **should** produce additional tests, including edge cases.
- You *may* modify the `Main` files to test your algorithm, but they **will not** be used during grading.
- You must submit your `ClosestPair` file to Gradescope. Do **not** submit `Main.java`, `Main.py`, `Main.cpp`, or any test files.
- You may submit your code a **maximum of 10 times** to Gradescope. After the 10th submission, Gradescope will not run your code.
- A few other notes:
  - Your code will be run as:
    `python Main.py` or `python3 Main.py` for Python,
    `javac *.java && java Main` for Java,
    or `g++ ClosestPair.cpp Main.cpp && ./a.out`.

- You may upload multiple files if you need additional classes (such as Java or C++), but **do not assign packages** to the files.
- You may use any editor or IDE you prefer as long as the code can be executed as stated above.
- We strongly encourage you to practice good development as you write code: use meaningful variable names, break your code into functions, "code a little then test a little", use the debugger, etc.

As noted earlier, input sizes could be very large. Therefore, we need this algorithm to be very efficient. If we put in the values for thousands of people, the algorithm should still run in a reasonable amount of time. For this reason, an $\Omega(n^2)$ algorithm is too slow; to be efficient enough, your algorithm **must** run in $O(n \log n)$ time.

```
4 13
12 10
8 9
1 1
42 108
```

Figure 1: Example input, for which your algorithm should return ≈ 4.12311 and ≈ 5.65685. Note that while all of the coordinates in this example are integer-valued, in our actual test cases, the coordinates may also be floating point values. Your implementation must support **both** integer and floating point values.