ROB 502 FA24: Homework 5

See Canvas for due dates.

Rules:

1. All HW must be done individually, but you are encouraged to post questions on Piazza and discuss during OHs (recall: do not email assignment questions to teaching staff!)
2. Late work adheres to the late grading policy (see syllabus, or use late-day tokens)
3. Submit your code on Autograder.io
4. Remember that copying-and-pasting code from other sources is not allowed

Code Download:
The code is available on the ROB 502 FA24 Git repo in the appropriate HW folder.

We prefer that you use the '`fetch`'/'`pull`' Git methods for the HWs, but we know that Git can be hard/confusing; if necessary, you can always just **download** the code from the repo link. However, learning to use Git will be to your benefit for multiple reasons:

1. You can get some easy extra credit if you properly pull and version-control (and comment) your work for the HW assignments!
2. ROB 550 will require you to use Git in a much more complex capacity, and without much learning time (large code databases, lots of teamwork with multiple branches, etc.).
3. If you ever need to setup a new VM or use your code on another device (new computer, VM crashes, etc.) you can easily restore your code progress with Git.
4. Many robotics-related industries and research labs rely on Git for robust version-controlling of their work… if mastered, you can put this on your resumé!

Open the directory `hw5` in VSCode. You can do this by `cd`-ing to the `hw5` directory and running `code .` . Do not run VSCode from subdirectories of `hw5` .

Instructions:

See previous homework assignments for notes about using Autograder.io, but you should be an expert now!

Each problem will give you a file with some template code, and you need to fill in the rest. Make sure to only put your code in the areas that start with `// --- Your code here` and end with `// ---`. **Do not edit code outside these blocks!**

Sample input and output are given as `input.txt` and `output.txt` respectively. Getting your program to successfully compile and correctly reproduce the sample output will get you some points, while the remaining points will be from hidden inputs that test some edge cases. Think about valid inputs that could break your logic. **Note that the templates given will read from `input.txt` and output to `output.txt`, so it might be a good idea to make a copy of the sample output to avoid overriding it.**

Problem 1: Binary Search [10 points]

A very important task is to be able to search for items of interest. By sorting a sequence, we can dramatically speed up search. This is the famous *binary search* algorithm. It is termed binary because at each iteration it divides the sequence in two: one half that might contain the element, and the other that definitely does not have the element. Your task is to implement the iterative and recursive versions of binary search in `hw5/binary_search/binary_search.cpp`.

See https://en.wikipedia.org/wiki/Binary_search_algorithm for pseudocode.

The content of `input.txt` will be a sequence of two lines, the first one being a **sorted** sequence of integers (in ascending order), and the second line being the element to lookup. The elements are guaranteed to be unique. For example,

```
-6 -3 0 1 2 5 8 9 12
2
```

looks for the value 2 in the sorted sequence. For each of these pairs of lines, you should output a line to `output.txt` in the format:

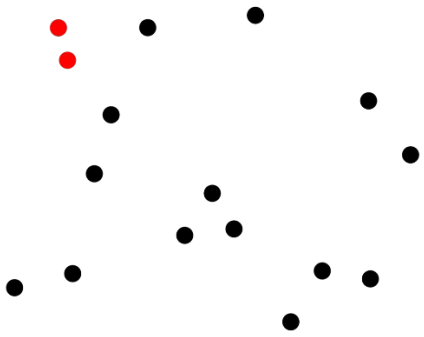
```
<index> <numIterativeCalled> <numRecursiveCalled>
```

where `numIterativeCalled` counts the number of iterations the main loop in the iterative version is run, while `numRecursiveCalled` counts the number of times the recursive version is called. Note that **binary search only works on sorted sequences**, therefore, if the element cannot be found in the sequence, you should output `ERROR` for that line (no other numbers, just `ERROR`).

Note that the number of calls scales roughly with the logarithm (base 2) of the length of the sequence, so we say the runtime has complexity `O(log n)` . In the above example, the index for 2 would be 4, and both the iterative and recursive loops would only be called once each, so the output should be `4 1 1` (as seen in the sample `output.txt` file).

Problem 2: Closest Pair of Points [14 points]

Simulators often have to do collision checking of objects. To speed up this process, one potential sub-task is to first find what objects could be potentially in collision with each other. A way to do this filtering is to find the pairwise distance between the object centers. This problem is a variant of finding the closest pair of any 2 points within a set, and finding the distance between them. You will be given `N` points in 2D (problem extends to 3D, but for simplicity we will only do 2D), and you need to find the closest pair of points, as seen in an example below:

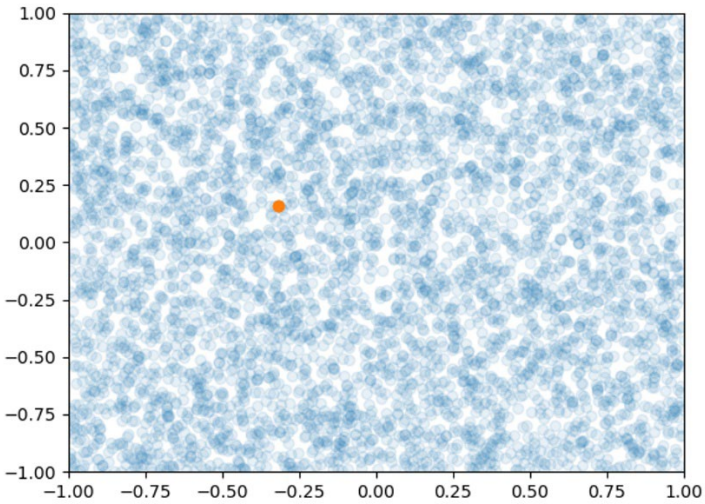


Your task is to implement `closest_pair` in `hw5/closest/closest_pair.h` . **The element with the lower ID should be the first element in the returned pair.** Each line of `input.txt` is `<num> <seed>`, where `<num>` is the number of points, and `<seed>` is a seed for a random number generator. Generating and outputting the points will be taken care of for you in `hw5/closest/closest_pair.cpp` **which you should not modify**.

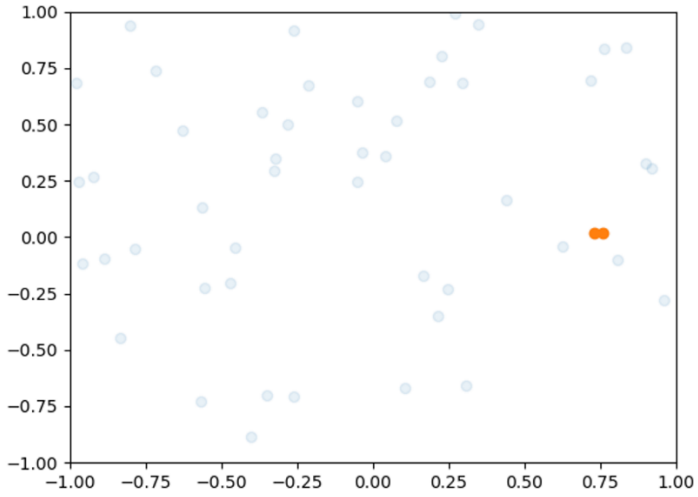
You can visualize the points using the provided `viz_points.py` script, by calling it as:

```
python3 viz_points.py points0.txt --output output.txt --index 0
```

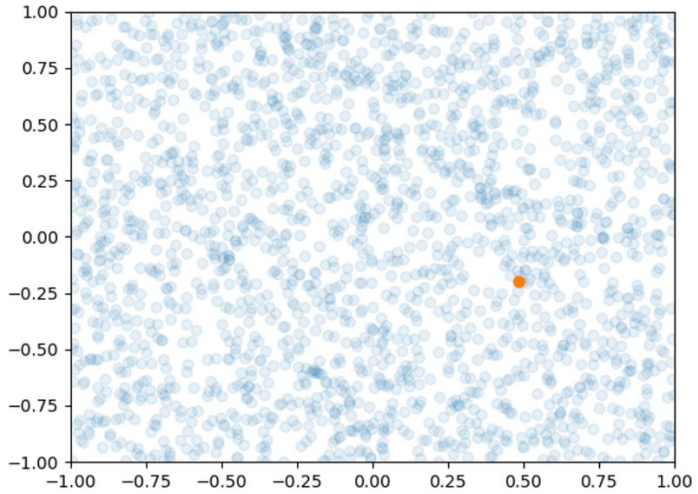
This will visualize all the points, along with the solution points given by the first line (index 0) in the `output.txt` file. You can omit `--output` and `--index` to just show the points. To generate the points, compile with `export_points = true` in `hw5/closest/closest_pair.cpp` (don't worry about remembering to turn this back off when submitting, since you are only submitting the header).



Correct solution for the first set of given points and the first given input.



Correct solution for the second set of given points and the second given input.



Correct solution for the third set of given points and the third given input.

Note that with the naïve $O(N^2)$ implementation of simply computing pairwise distances between all points and taking the minimum will allow you to pass the sample test cases, but this method will fail the hidden test cases due to timing out. You need to implement at least a $O(N(\log N)^2)$ time algorithm to pass.

Hint: You should use recursion. Sorting the list of points can be done in $O(N \log N)$ time and this kind of structure may be necessary for speedups. Consider each x,y dimension separately and whether you can break the problem up into simpler, smaller sub-problems. If so, how do the solutions to the sub-problem relate to the bigger problem? Note that the naïve $O(N^2)$ method be good enough for *small-enough sub-problems*.

Problem 3: git Conflict Resolution [?? points]

We there will be an additional question uploaded as a separate document soon, related to a git merge conflict. We will announce this in class and on Canvas when the problem is ready. Thanks for your patience!