ECE2810J — Data Structures and Algorithms

Programming Assignment 1

Instructor: Yutong Ban

— UM-SJTU-JI (Summer 2024)

Notes

- Due Date: June 20, 2024
- Submission: on JOJ

1 Introduction

In this first project, you will review some of the algorithms we have previously discussed in this course and utilize them to solve a real-life problem with some hints.

In part one, you are asked to implement six sorting algorithms: bubble sort, insertion sort, selection sort, merge sort, quick sort with extra-space partitioning, and quick sort with in-place partitioning.

In part two, you are given an actual problem to deal with. But don't worry, we will help you to step by step implement an algorithm based on sorting algorithms: Graham's scan.

You will recall the knowledge about templates and function objects in this project and have a simple practice with STL(Standard Template Library).

A copy of the handout and the template of `<sort.hpp>` can be found on Canvas.

2 Programming Assignment

2.1 Sorting Algorithms

In `sort.hpp`, you need to write six functions corresponding to the six sorting algorithms:

- `bubble_sort`
- `insertion_sort`
- `selection_sort`
- `merge_sort`
- `quick_sort_extra` (quick sort using extra array for partitioning)
- `quick_sort_inplace` (quick sort with in-place partitioning)

You are allowed to add more helper functions in this file, but you're not allowed to change the definitions of these six functions defined by us.

Below is an example function `std_sort`, which use the `std::sort` function in the `<algorithm>` C++ standard library. However, do not copy and paste it into any other functions (you are prohibited from including additional libraries, such as `<algorithm>` in `<sort.hpp>`). Only use the below code in your main function to test your comparison function objects in your `<main.cpp>`.

```
template<typename T, typename Compare>
void std_sort(std::vector<T> &vector, Compare comp = std::less<T>()) {
```

```
    std::sort(vector.begin(), vector.end(), comp);
}
```

Recall what you've learned about STL containers and templates in VE280, here `T` is an arbitrary type which can be compared with the `comp` function (or *function object*). Each of your sorting functions should provide the same result as the standard sorting function, which means for each pair of neighboring elements *a* and *b* in the sorted vector, `a==b` or `comp(a,b)==true`.

The `std::less<T>()` function is equivalent to the following function:

```
template<typename T>
bool compare_less(const T &lhs, const T &rhs) {
    return lhs < rhs;
}
```

Alternatively, if `T` is restricted, i.e., you already know the first argument is `vector<int>`, you can define the compare function without using the template:

```
bool compare_int_less(const int &lhs, const int &rhs) {
    return lhs < rhs;
}
```

It can also be written in the form of *function object*:

```
template<class T>
struct CompareLess {
    bool operator()(const T &lhs, const T &rhs) const {
        return lhs < rhs;
    }
};
```

Then you can call the sorting function with those methods (which are all equivalent):

```
std::vector<int> vector = {3, 2, 1};

std_sort(vector, std::less<T>());
std_sort(vector, compare_less<T>);
std_sort(vector, compare_int_less);
std_sort(vector, CompareLess<T>());
```

Note: `std::less` and `CompareLess` are *function objects*, so that they need to be initialized with `()`.

Important: do not include the compare function objects in `sort.hpp`. You are also not allowed to include additional libraries (in addition to the `vector` and `functional` STL library) in `sort.hpp`. Remove all customized `include` statements from `sort.hpp` (including the ones that are commented out) before submitting to JOJ. Hint: include additional libraries in `main.cpp`.

Only submit the `sort.hpp` file (do not submit your main.cpp) to JOJ at `https://joj.sjtu.edu.cn`.

2.2 Graham's Scan Algorithm[1]

2.2.1 Problem Background

Imagine a JI student named Alex who is passionate about planting trees. Alex has a small piece of land in her hometown in the countryside where she wants to create a tree garden. What Alex is concerned about is that the saplings need to be protected from passersby. To achieve this, Alex needs to enclose the entire tree garden with a fence. But here's the catch: Alex wants to minimize the length of the fence needed.

Given the coordinates of the tree planting locations (as a set of points), Alex needs to find the smallest possible perimeter for the fence that encloses all the trees. She searched on the Internet and found that this perimeter corresponds to the convex hull of the tree locations.

The convex hull of a set of points is the smallest convex polygon that encloses all the points. In other words, it's the fence that wraps around the outermost trees, forming a closed shape with straight sides, such that all its interior angles are no larger than 180 degrees.

Alex wants to determine the optimal placement of the fence posts to minimize the total length of the fence while ensuring that no tree lies outside the enclosed area. Now she must find an efficient algorithm to compute the convex hull of the given tree locations since her time is limited.

2.2.2 Introduction of Graham's Scan

A set of points in an Euclidean space is defined to be *convex* if it contains the line segments connecting each pair of its points. The *convex hull* of a given set X may be defined as The (unique) minimal convex set containing X. For example, on a 2D plane, Figure 1 shows the convex hull of a set of points.

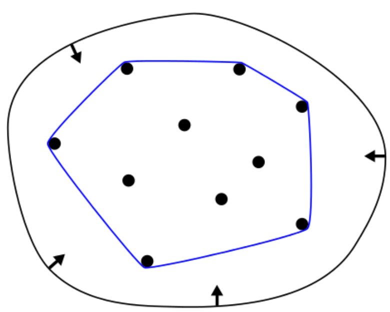

Figure 1: Convex hull of points on a 2D plane[2].

Graham's scan is a method of finding the convex hull of a finite set of points in the 2D plane. It is named after Ronald Graham, who published the original algorithm in 1972. The algorithm finds all vertices of the convex hull ordered along its boundary. It uses a stack to detect and remove concavities in the boundary efficiently.

To ensure numeric precision, we use the sign of the cross product to determine relative angles when sorting and performing the algorithm. For three points $P_1(x_1, y_1)$, $P_2(x_2, y_2)$ and $P_3(x_3, y_3)$, compute the z-coordinate of the cross product of the two vectors $\overline{P_1P_2}$ and $\overline{P_1P_3}$, we define

$$ccw(P_1, P_2, P_3) = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1). \quad (1)$$

If the result is 0, the points are collinear; if it is positive, the three points constitute a "left turn" or counter-clockwise orientation, otherwise a "right turn" or clockwise orientation (for counter-clockwise numbered points).

2.2.3 Algorithm in Pseudo-code

The Graham's scan algorithm is defined in Algorithm 1.

Input : A finite set of points X

Output: A vector of points on the convex hull of X

$S \leftarrow$ empty stack;

$P_0 \leftarrow$ the point in X with the lowest y-coordinate (if the lowest y-coordinate exists in more than one point in X, the point with the lowest x-coordinate out of the candidates should be chosen);

sort the points in X by the polar angle with P_0 (if several points have the same polar angle then only keep the farthest);

foreach *P in X* **do**

  while *size(S) > 1 and* `ccw`(*next_to_top(S), top(S), P*) $\leq$ *0* **do**

    pop from S;

  end

  push P to S;

end

Algorithm 1: Graham's scan.

Now the stack contains the convex hull, where the points are oriented counter-clockwise and P_0 is the first point.

The polar angle of a point P with P_0 is the angle $\overline{P_0P}$ forms with the x axis. In the sorting process, we compare the polar angle of two points P_1 and P_2 by computing $ccw(P_0, P_1, P_2)$. If it is positive, the polar angle of P_2 is greater than P_1. So the *ccw* function won't introduce any precision loss compared to computing the cosine values of the angles.

In each iteration, we also use the *ccw* function, since we are popping the last point from the stack if we turn clockwise ($ccw < 0$) to reach this point in the while loop.

2.2.4 Input / Output Requirements

In `p1.cpp`, you need to implement this algorithm. You are allowed to use any of the sorting functions defined in `sort.hpp` and `std::sort` from the `<algorithm>` library.

The input (from standard input) will contain $N + 1$ lines. The first line is an integer N. Each of the next N lines consists of a space-separated pair of integers x and y, representing a point $P(x, y)$. All of the coordinates are integers in the range $[-2^{31}, 2^{31} - 1]$. The input would look like:

```
5
-2 -1
0 0
2 1
-3 4
5 -4
```

The output (to standard output) should be the points outputted by the algorithm, which consist of a convex hull of the input points. Each line should contain two integers representing a point $P(x, y)$, which has the same format as the input. In particular, the first line should be P_0 as described above, and these points should be in oriented counter-clockwise order on the convex hull. This point sequence is unique, which is guaranteed by the algorithm. A sample output would look like:

```
5 -4
2 1
-3 4
-2 -1
```

You can assume the input is valid for the problem and this algorithm.

2.2.5 Hints

You may test your sorting algorithms with the main program, but in your final submission, you are recommended to use `std::sort` for safe.

For the stack S, since you need to access the "next to top" element, you can use a `std::vector` instead of a `std::stack` for simplicity.

For the compare function, if you don't use a "function object"-based one, global variables may be introduced, which is not a good idea. A better solution is to define a `struct` with an attribute P_0, and the `operator()` overload function.

When testing your algorithm, don't forget to include some extreme cases such as duplicated or collinear points.

3 Implementation Requirements and Restrictions

3.1 Requirements

- You must make sure that your code compiles successfully on a Linux operating system with `g++` and the options `-std=c++1z -Wconversion -Wall -Werror -Wextra -pedantic`.
- You should not change the definitions of the functions in `sort.hpp`.
- You should only hand in two files `sort.hpp` and `p1.cpp`.
- You can use any header file defined in the `C++17` standard. You can use cppreference as a reference.

3.2 Memory Leak

You may not leak memory in any way. To help you see if you are leaking memory, you may wish to call Valgrind, which can tell whether you have any memory leaks. (You need to install Valgrind first if your system does not have this program.) The command to check memory leak is:

```
valgrind --leak-check=full <COMMAND>
```

You should replace `<COMMAND>` with the actual command you use to issue the program under testing. For example, if you want to check whether running program

```
./main < input.txt
```

causes memory leak, then `<COMMAND>` should be "`./main < input.txt`". Thus, the command will be

```
valgrind --leak-check=full ./main < input.txt
```

4 Grading

Your program will be graded along five criteria:

4.1 Functional Correctness

Functional Correctness is determined by running a variety of test cases against your program and checking your solution using our automatic testing program.

4.2 Implementation Constraints

We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. In this project, we will also check whether your program has memory leaks. For those programs that behave correctly but have memory leaks, we will deduct some points.

4.3 General Style

General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. Part of your grade will also be determined by the performance of your algorithm.

4.4 Performance

We will test your program with some large test cases. If your program is not able to finish within a reasonable amount of time, you will lose the performance score for those test cases.

5 Honor Code

Please **DO NOT** violate the honor code.
Some behaviors that are considered as cheating:

- Reading another student's answer/code, including keeping a copy of another student's answer/code
- Copying another student's answer/code, in whole or in part
- Having someone else write part of your assignment
- Using test cases of another student
- Testing your code with another one's account
- Post your code to GitHub

6 Acknowledgement

The programming assignment is co-authored by Yihao Liu, an alumnus of JI and the chief architect of JOJ.
This programming assignment is designed based on Weikang Qian's VE281 programming assignment 1.
This manual is designed based on Hongyi Xin's VE281 programming assignment 1.

References

[1] Graham scan - Wikipedia: https://en.wikipedia.org/wiki/Graham_scan

[2] Convex hull - Wikipedia: https://en.wikipedia.org/wiki/Convex_hull