2019 Semester 1
COMP20007 Design of Algorithms
School of Computing and Information Systems
The University of Melbourne

Assignment 1: Convex Hull

Due: 11:59 PM Thursday April 11th

Introduction

A set S in the plane is called *convex* if it satisfies the following property: for every pair of points $p, q \in S$ the line segment between them $\overline{pq}$ is also in the set S (*i.e.*, $\overline{pq} \subseteq S$).

One problem that comes up time and time again in areas such as computational geometry is the problem of finding the smallest convex set S which contains all points in some other set X. This set is called the *convex hull* of X.

When given some set of points we can describe the convex hull S by listing the points which make up the vertices in the boundary of the convex hull.

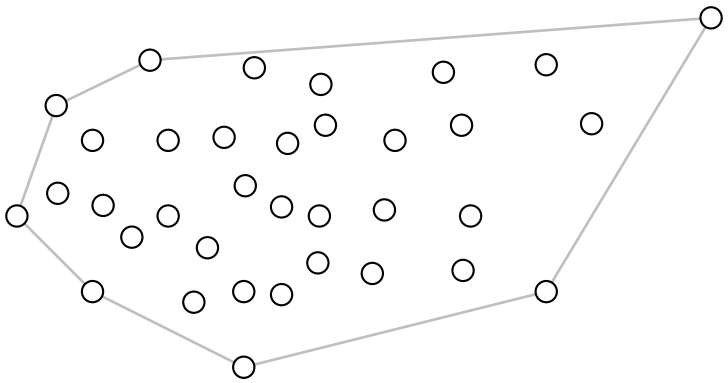


Figure 1: A set of points and the boundary of their corresponding convex hull, which is coloured in grey.

Levitin Section 3.3: Closest-Pair and Convex-Hull Problems by Brute Force provides a more comprehensive description and discussion of the convex hull problem.

In this assignment you will implement an algorithm to compute the convex hull of a given set of points.

The assignment consists of 3 (three) coding tasks (Part A, Part C and Part F) and 4 (four) analysis tasks (Part B, Part D, Part E and Part G).

Tasks

Part A: Point Orientation

Let p_0, p_1, p_2 be three points in the plane. If p_2 is left of the line segment $\overline{p_0p_1}$, then we write Left(p_0, p_1, p_2). If p_2 is right of the line segment $\overline{p_0p_1}$, then we write Right(p_0, p_1, p_2). If three points are on the same straight line, then we say that (p_0, p_1, p_2) are *collinear*.

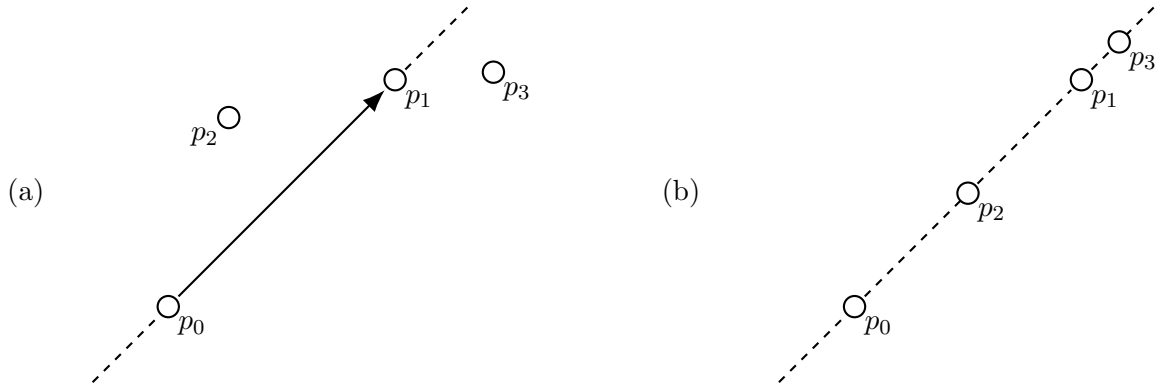


Figure 2: (a) In this example p_2 is Left of $\overline{p_0p_1}$, while p_3 is Right of $\overline{p_0p_1}$. (b) Here p_0, p_1, p_2 and p_3 are all collinear points, so we say that each 3-subset of the four points are collinear *e.g.,* $p0, p2$ and $p3$ are collinear.

Your task is to complete the implementation of `orientation()` in `convex-hull.c` which takes three points (p_0, p_1 and p_2) and returns `'l'` if p_2 is Left of $\overline{p_0p_1}$, `'r'` if p_2 is Right of $\overline{p_0p_1}$ or `'c'` if p_0, p_1 and p_2 are Collinear.

Your function must run in $O(1)$ time.

Hint: check out Appendix A which describes a number of useful vector operations.

Part B: Checking for Convexity

Let $P = (p_0, \ldots, p_{n-1})$ be a sequence of n points in the Euclidean plane ($\mathbb{R}^2$). Assume you have been given the following algorithm:

```
function isConvex(p_0, ..., p_{n-1})
    for i ← 0 to n − 1 do
        if Right(p_i, p_{(i+1) mod n}, p_{(i+2) mod n}) then
            return false
        return true
```

Does the algorithm above correctly determine if P is the boundary of a convex polygon in counterclockwise order? Explain your answer.

The Inside Hull Algorithm

A polygon is called *simple* if it has no self intersections.

We describe a convex hull algorithm, called *InsideHull* and abbreviated with *IH*, that computes the convex hull of a simple polygon $P = (p_0, p_1, \ldots, p_{n-1})$, with points $p_i \in \mathbb{R}^2$ and its corresponding edges $\overline{p_ip_{i+1}}$. The points are ordered in counterclockwise order.

Although our definition of a simple polygon doesn't exclude successive points being collinear, InsideHull assumes that this is not the case.

InsideHull uses a *double ended queue* abstract data type to represent the convex polygon it constructs. A double ended queue is abbreviated to *deque* and it has the following operations:

- *push* to the top of the deque
- *pop* from the top of the deque
- *insert* to the bottom of the deque
- *remove* from the bottom of the deque

The top and bottom of a deque c are indexed by c_t and c_b respectively. In the following algorithm the deque c gives rise to the convex hull $C = \langle c_b, c_{b+1}, \ldots, c_{t-1}, c_t \rangle$.

The algorithm is as follows. Parts of the algorithm denoted by ... are not complete and will require you to fill them in.

```
function InsideHull(Polygon)
    ... confirm the input is valid ...
    if Left(p_0, p_1, p_2) then
        C ← new deque ⟨p_2, p_0, p_1, p_2⟩
    else
        C ← new deque ⟨p_2, p_1, p_0, p_2⟩
    ...
    while i < n do
        Get next point p_i
        if Left(c_{t-1}, c_t, p_i) and Left(c_b, c_{b+1}, p_i) then
            i ← i + 1
            Continue
        while Right(c_{t-1}, c_t, p_i) do
            Pop c_t from C
        Push p_i to C
        while Right(c_b, c_{b+1}, p_i) do
            Remove c_b from C
        Insert p_i into C
        i ← i + 1
    ...
```

Note that during the algorithm, the deque C contains the points of the partial convex hull (which is completed by the end of the algorithm) in counter-clockwise order. At the beginning of each outer while loop c_b and c_t refer to the same point, and this point will always be the most recent point added to C.

Part C: Deque Data Structure

Implement a `Deque` abstract data structure and complete the definitions of the following functions in `deque.c`:

- `new_deque()`
- `free_deque()`
- `deque_push()`

- `deque_insert()`
- `deque_pop()`
- `deque_remove()`
- `deque_size()`

It is up to you to decide on which underlying data structure will be used for your Deque. All of the operations described above must run in constant ($O(1)$) time[1].

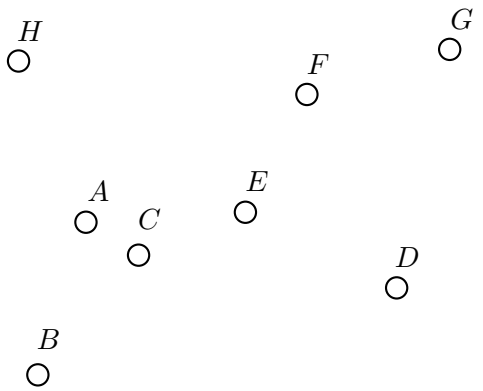
Part D: Deque Analysis

Explain your deque implementation and the decisions you made. Describe the time complexities of the four main operations (push, pop, insert, remove) and how these time complexities are achieved.

Part E: Inside Hull Tracing

Run the following example by hand and provide the points contained in the deque at each step of the algorithm. Be clear about which end of your deque is the top and which is the bottom.

The points are provided to InsideHull in alphabetic order, starting with point A.



Part F: Inside Hull Implementation

Implement the C function `inside_hull()` that takes a polygon consisting of n points and computes the points of its convex hull.

The polygon will be provided as an array of `n` points named `points`, and the points will appear in counter-clockwise order.

Your function should store the points which make up the convex hull in the array `hull`, which will have enough memory for at least `n` points. These points should be stored in counter-clockwise order, and the first and last points should **not** be the same. This will mean the number of points in `hull` will be one fewer than the size of the deque C at the end of the algorithm.

The number of points stored in `hull` should be returned. If an error occurs return `INSIDE_HULL_ERROR`. If three consecutive points are collinear return `COLLINEAR_POINTS` and don't complete the algorithm (make sure you free any memory you have allocated before you return).

[1]We require constant case on average, *i.e.,* constant amortised runtime. This means you may choose to use some sort of resizable array, as long as the resizing isn't being performed during each operation

Note: You may assume that the input we will use to test your program will either have collinear points occurring consecutively or no collinear points at all. Note that the points across the array boundary (*e.g.,* $Polygon[n-1], Polygon[0]$, and $Polygon[1]$) are considered consecutive.

Part G: Inside Hull Analysis

The best algorithms we have for computing convex hulls for a general set of n points are $O(n \log n)$, if they are not *output sensitive*[2].

Identify the basic operations in the InsideHull algorithm and use this to determine the time complexity of InsideHull, give your answer in Big-Oh notation. Explain how you arrived at this answer.

Does this contradict the statement about runtimes above? Explain why it does or does not.

Completing the Coding Tasks

We have provided the skeleton for this project which includes the following files:

- `main.c` – The entry point to the program, and the functionality which deals with command line input and output. **Do not change this file.**
- `point.h`, `point.c` – Provides the `Point` struct and related functions. **You may add to this module if you need additional functionality, but don't change existing code.**
- `convex-hull.h`, `convex-hull.c` – Here is where you will implement `orientation()` and `inside_hull()`. **You may add to the header file don't change existing function signatures.**
- `deque.h deque.c` – Here is where you will implement your `Deque` data structure. **You may add to the header file don't change existing function signatures.**
- `Makefile` – provides make targets `all` (to compile the program), `clean` (to delete the `.o` files and the executable) and `submit` (which creates `submission.zip` including all files in the directory). **You should add to this Makefile if you want us to compile your program in another way, or if you add other files.**

To compile the program run `make`. This will create an executable `a1`.

Feel free to add any additional `.c`/`.h` files, but make sure you add these to your `Makefile` as well.

For information about how the program `a1` runs, and the command line options, please see *Appendix B: Using The Command Line Tool*.

Your code will be compiled and tested on the School of Engineering's Linux Servers so make sure you compile and test your code thoroughly on `dimefox` before submission. For information on how to login to `dimefox` see the Workshops section of the LMS. Please try this early and often, being unable to access `dimefox` is not grounds for an extension or other considerations.

Completing the Analysis Tasks

You should create a PDF file `analysis.pdf` with your answers to Parts B, D, E and G. You should aim to write no more than 2 pages.

[2] An output sensitive algorithm's runtime depends on the size of the *output* as well as the size of the input, in this context it would mean that the runtime is dependent on the number of points which are part of the convex hull.

Justify all your answers and provide answers as asymptotic complexity classes (*e.g.,* O, Ω, Θ). To receive full marks the bounds should be tight, for example if an algorithm is an n^2 algorithm $O(n^2)$ will receive full marks, while $O(n^3), O(2^n), O(n!)$, etc. will not.

Marking

The total number of available marks for this assignment is 10 (ten), and this assignment is worth 10% of your grade in this subject.

There are 5 available marks for the coding tasks, and 5 available marks for the analysis tasks. The breakdown is as follows:

- Coding Tasks
  - Part A – 1 Mark
  - Part C – 2 Mark
  - Part F – 2 Mark
- Analysis Tasks
  - Part B – 1 Mark
  - Part D – 1 Mark
  - Part E – 1 Mark
  - Part G – 2 Mark

C code with poor coding style, lack of comments, poor functional decomposition, memory leaks and/or large chunks of duplicate code may have up to 1 mark deducted.

Compilation warnings may result in up to 1 mark being deducted.

We will compile and test your code automatically, so failure to compile on `dimefox` (see the *Submission* section) may result in an overall mark of 0 for the coding tasks.

To receive full marks your analysis must be consistent with your C implementation.

To receive full marks for the coding tasks the algorithms must meet the time complexity requirements (*i.e.*, all operations in Parts A and C must be constant time) and follow the prescribed algorithm correctly (in the case of Part F).

Submission

The `Makefile` provides the `make submit` command which zips up the assignment directory and creates `submission.zip`.

Make sure that all C code required to compile and execute your program are included.

We will open assignment submissions next week.

Code Usage and Academic Honesty

You may make use of code provided as part of this subject's workshops or their solutions (with proper attribution), however you **may not** use code sourced from the Internet or elsewhere. Using code from the Internet is grounds for academic misconduct.

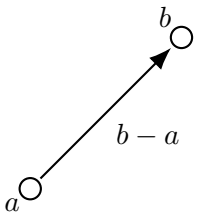
All work is to be done on an individual basis. All submissions will be subject to automated similarity detection. Where academic misconduct is detected, all parties involved will be referred to the School of Engineering for handling under the University Discipline procedures. Please see the Subject Guide and the "Academic Integrity" section of the LMS for more information.

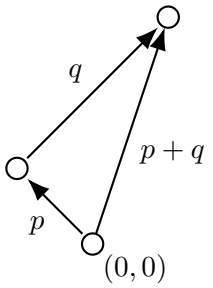
Appendix A: Vector Operations

In this assignment we will be dealing with points in the plane, *i.e.*, vectors in $\mathbb{R}^2$.

Sometimes we view a pair of points (x, y) as a position vector and sometimes as a direction vector.

The first two operations we will describe are vector addition and subtraction, as in the figure below.





As we can see, the line segment $\overline{ab}$ is given by $b - a$.

For $a = (x_a, y_a), b = (x_b, y_b)$ addition and subtraction are given by

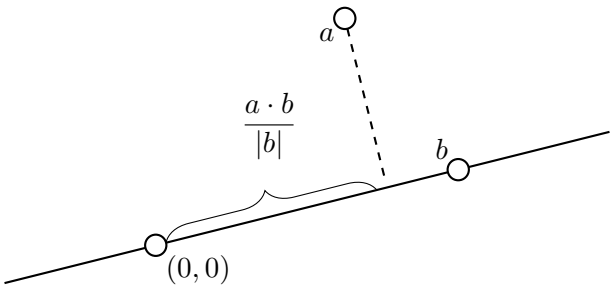
$$a + b = (x_a + x_b, y_a + y_b),$$
$$a - b = (x_a - x_b, y_a - y_b).$$

The dot product $a \cdot b$ is given by

$$a \cdot b = x_a x_b + y_a y_b.$$

The result of a dot product is a scalar.

The dot product $a \cdot b$ can give us the magnitude of the projection of a onto the line between the origin $(0, 0)$ and b.

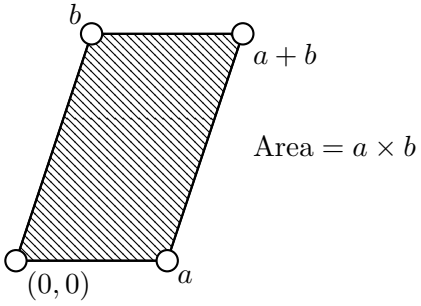


If a projected onto b was in the opposite direction of the line from the origin to b then the dot product will be negative.

The cross product between two vectors a and b is given by

$$a \times b = x_a y_b - y_a x_b.$$

The cross product gives the signed area of the parallelogram with vertices $(0, 0), a, b$ and $a + b$.



The sign of the area is given by the right hand rule, if you've taken first year physics you should be familiar with this. That is to say, if the angle from a to b travelling clockwise around the origin is between $0°$ and $180°$ then the cross product is negative, and positive otherwise. Hence the cross product in the example above is positive. On the other hand $b \times a$ would be negative.

Appendix B: Using The Command Line Tool

Running `a1` without any command line options, or by using the `-h` or `--help` options will print the following usage message:

```
usage: ./a1 [options] < <input_file>

options:
 -o | --orientation  Test the orientation() function
 -d | --deque        Test deque functionality
 -i | --inside-hull  Test the inside_hull() function
 -h | --help         Print this help message
```

The three options (`--orientation`, `--deque` and `--inside-hull`) test the three coding tasks respectively.

Note: you do not need to implement the input/output for these command line options, this is done for you in `main.c` you just need to implement the functions described in the Tasks section.

`--orientation` expects three points in the following format:

```
x0 y0
x1 y1
x2 y2
```

And outputs the result of `orientation(p0, p1, p2)`. For example if `input.txt` contains:

```
0.0 0.0
1.0 1.0
1.0 0.0
```

Then the output will be:

```
$ ./a1 --orientation < input.txt
p0: 0.00 0.00
p1: 1.00 1.00
p2: 1.00 0.00
p2 is to the right of the line p0p1
```

Input to `--deque` is some number of commands (`push x y`, `insert x y`, `pop`, `remove`, `size`) to stdin. For example if `input.txt` contains:

```
push 1 1
insert 0 0
size
push 2.5 2.5
push 3 2
size
pop
```

```
pop
remove
pop
```

Then the output will be:

```
$ ./a1 --deque < input.txt
2
4
3.00 2.00
2.50 2.50
0.00 0.00
1.00 1.00
```

Input to `--inside-hull` is in the format:

```
n
x1 y1
...
xn yn
```

These points are read into a polygon and `inside_hull()` is called. For example if `input.txt` contains:

```
5
0 0
2 0
2 2
1.5 0.5
0 2
```

Then the output will be:

```
$ ./a1 --inside_hull < input.txt
0.00 2.00
0.00 0.00
2.00 0.00
2.00 2.00
```