Solution for Homework 1

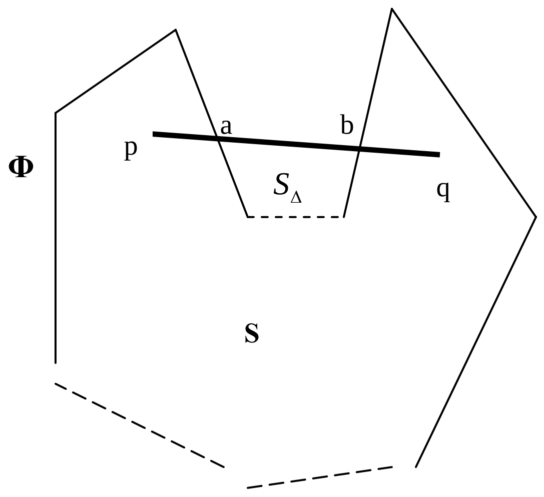
1.1

(a) Given two convex set S1, S2, their intersection is $S_\wedge = S_1 \wedge S_2$;

For any points $p, q \in S_\wedge$, we have: $\overline{pq} \subseteq S_1$ and $\overline{pq} \subseteq S_2$, so $\overline{pq} \subseteq S_1 \wedge S_2$

Thus $S_\wedge$ is convex.

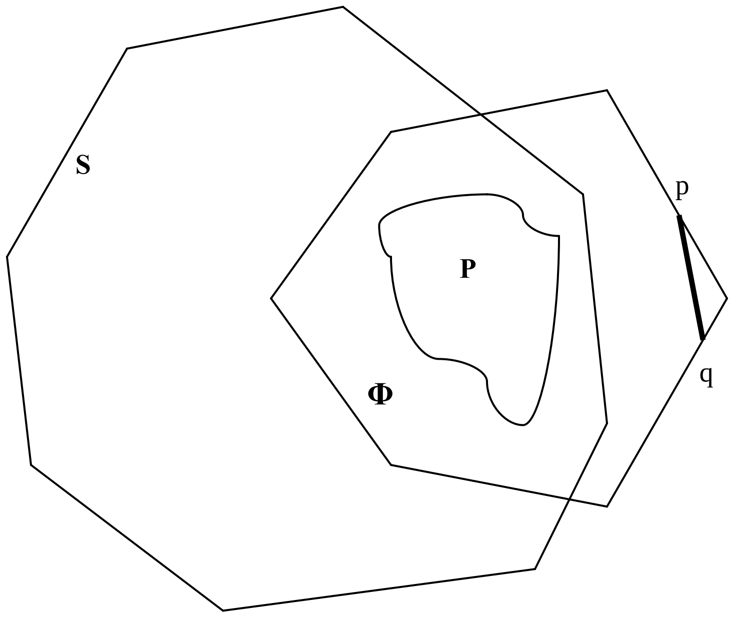
(b) Proof by contradiction. Suppose the smallest perimeter polygon $\mathbf{\Phi}$ containing the point set **P** is not convex. So there exist two points p, q inside $\mathbf{\Phi}$, such that segment $\overline{pq} \not\subset \mathbf{S}$, where **S** is the inner areas bounded by $\mathbf{\Phi}$. Suppose $\overline{pq}$ cross out and in $\mathbf{\Phi}$ at point a, b. The situation is shown below:



If the boundary of $\mathbf{\Phi}$ where a, b is connected is replaced by segment $\overline{ab}$, the area of S is increasing by the region S_Δ . So the new polygon $\mathbf{\Phi}'$ will still contain all points of **P**. As segment $\overline{ab}$ is shortest path between point p, q, $\mathbf{\Phi}'$ will certainly have shorter perimeter than $\mathbf{\Phi}$. This contradicts the fact that $\mathbf{\Phi}$ is the shortest perimeter polygon. So $\mathbf{\Phi}$ must be convex.

(c) Proof by contradiction. Suppose **S** is a convex set containing the point set **P**, and $\mathbf{\Phi}$ is the smallest perimeter polygon containing **P**. However, $\mathbf{\Phi} \not\subset \mathbf{S}$. The situation is shown below:

As $\mathbf{\Phi} \not\subset \mathbf{S}$, we can always find two points p, q on $\mathbf{\Phi}$, which are not inside **S**, and the segment ($\overline{pq}$) between which are not inside **S** as well. As proved in (b), $\mathbf{\Phi}$ is convex. Thus $\overline{pq}$ is inside $\mathbf{\Phi}$. If the boundary of $\mathbf{\Phi}$ connecting p, q is replaced by $\overline{pq}$, forming a new convex polygon $\mathbf{\Phi}'$, $\mathbf{\Phi}'$ will still contain the point set **P**, and its perimeter is certainly shorter than $\mathbf{\Phi}$. This contradicts the claim that $\mathbf{\Phi}$ is the shortest one. So **S** must contain $\mathbf{\Phi}$.



1.3

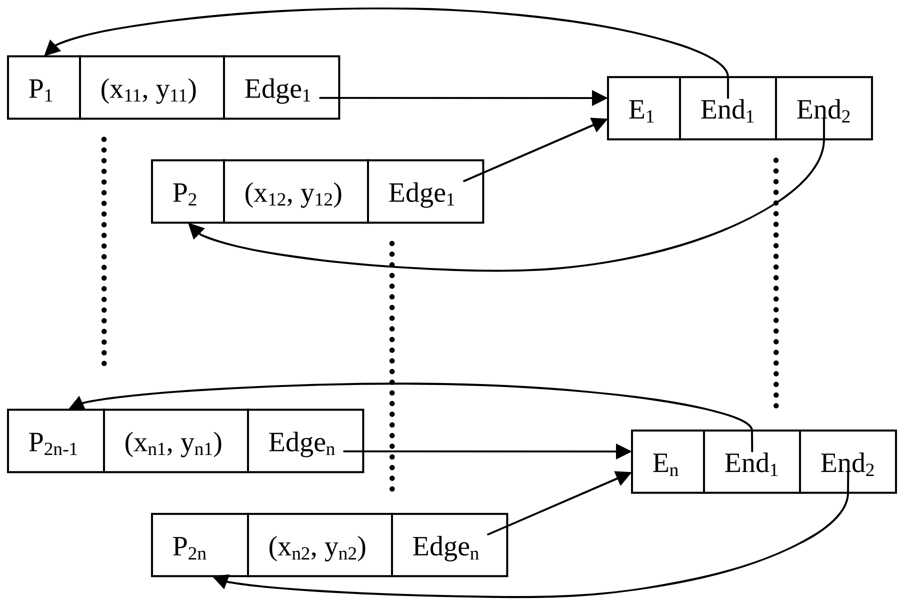
Suppose we are given n unsorted edges:

$$\{(x_{11}, y_{11}), (x_{12}, y_{12}) : e_1\}, \{(x_{21}, y_{21}), (x_{22}, y_{22}) : e_2\}, \ldots, \{(x_{n1}, y_{n1}), (x_{n2}, y_{n2}) : e_n\}$$

First, we construct a new data structure to represent the input:

For each edge, say, $\{(x_{i1}, y_{i1}), (x_{i2}, y_{i2}) : e_i\}$, we build 3 elemental units as: $\{x_{i1}, y_{i1} : \ edge_i\}$, $\{x_{i2}, y_{i2} : \ edge_i\}$, $\{e_i : \ end_{i1}, end_{i2}\}$, in which $edge_i$ is a pointer from the endpoint coordinate to its edge, while end_{i1}, end_{i2} is a pointer from the edge to its two endpoints separately.

This operation will surely take O(n) time, and is shown as:



Second, we sort the 2n endpoints structure $\left\{x_{i,\{1,2\}}, y_{i\{1,2\}} : edge_i\right\}$ from left to right with respect to x coordinates. This takes $2n\log(2n)$ time, or more essentially, $(2\log 2)n + 2n\log n = n\log n$ time. Apparently, each two identical endpoints shared by two edges must be next to each other.

Third, we start from the left most endpoint p_1, query and record its associated edge e_1 by **endpoint pointer operation** $P_1 \rightarrow edge$, and query the other endpoint p_2 of e_1 by **edge pointer operation** $P_1 \rightarrow edge \rightarrow end$. Then from p_2, we find its left or right neighbor $p_2{}'$ which has identical coordinates (remember we have sorted all the 2n endpoint), and record its associated edge and the other endpoint with another set of pointer operation.

We perform these endpoint-edge-endpoint operations iteratively, until we finally return to the start point. As each pointer operation takes O(1) time, the total time is in O(n).

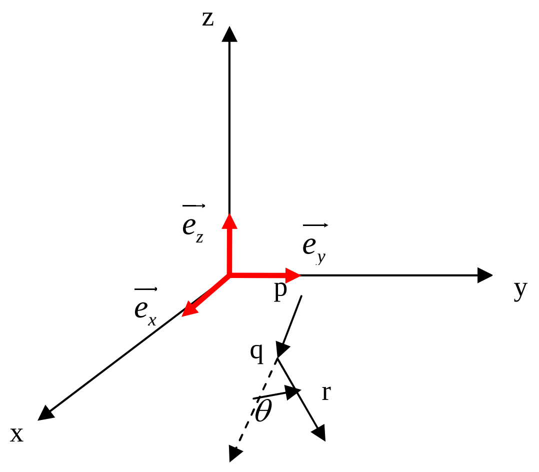
Finally, we get a sorted edge list, either clockwise or counterclockwise. Then, we pick just out two neighboring edges, and check whether the next edge is right to the previous one. If so, we are done. Otherwise, it's counterclockwise. We just reverse the edge list, and get a clockwise edition. So the step requires at most O(n) time.

In general, this algorithm can give us a clockwise sorted edge list in:

$O(n) + O(n\log n) + O(n) + O(n) = O(n\log n)$ time complexity.

1.4

(a) In a right-hand coordinate system shown below, point p, q, r are lying in the xOy plane.



Whether r lies to left or right of segment pq can be determined by sign of the directed angle θ. Further, according to the definition of cross product, we have:

$$\overrightarrow{pq} \times \overrightarrow{qr} = \left\|\overrightarrow{pq}\right\| \cdot \left\|\overrightarrow{qr}\right\| \sin\theta \cdot \overrightarrow{e_z}\ ; \qquad (1)$$

Also,

$$\overrightarrow{pq} \times \overrightarrow{qr} = \begin{vmatrix} \overrightarrow{e_x} & \overrightarrow{e_y} & \overrightarrow{e_z} \\ q_x - p_x & q_y - p_y & 0 \\ r_x - q_x & r_y - q_y & 0 \end{vmatrix} = \overrightarrow{e_z} \begin{vmatrix} q_x - p_x & q_y - p_y \\ r_x - q_x & r_y - q_y \end{vmatrix}. \qquad (2)$$

Therefore, by determining the sign of the scale value of the cross product $\overrightarrow{pq} \times \overrightarrow{qr}$, we can determine the sign of $\sin\theta$, and in this way determine the direction of θ. Let:

$$\overrightarrow{pq} = \begin{pmatrix} q_x - p_x \\ q_y - p_y \\ 0 \end{pmatrix}, \quad \overrightarrow{qr} = \begin{pmatrix} r_x - q_x \\ r_y - q_y \\ 0 \end{pmatrix}$$

The scale value of the cross product is: $\begin{vmatrix} q_x - p_x & q_y - p_y \\ r_x - q_x & r_y - q_y \end{vmatrix}$

On the other hand,

$$D = \begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix} = \begin{vmatrix} 0 & p_x - q_x & p_y - q_y \\ 1 & q_x & q_y \\ 0 & r_x - q_x & r_y - q_y \end{vmatrix} = -\begin{vmatrix} p_x - q_x & p_y - q_y \\ r_x - q_x & r_y - q_y \end{vmatrix} = \begin{vmatrix} q_x - p_x & q_y - p_y \\ r_x - q_x & r_y - q_y \end{vmatrix}$$

So by determining the sign of D, we find the sign of θ, and find whether r lies to the left or right side of pq.

(b) Equations (1) and (3) shows both the geometric and algebra definitions of cross production. So,

$$\left\|\overrightarrow{pq}\right\| \cdot \left\|\overrightarrow{qr}\right\| \sin\theta = \begin{vmatrix} q_x - p_x & q_y - p_y \\ r_x - q_x & r_y - q_y \end{vmatrix} = D;$$

Taking absolute value on both sides, we have:

$$\left| \left\|\overrightarrow{pq}\right\| \cdot \left\|\overrightarrow{qr}\right\| \sin\theta \right| = |D|, \tag{3}$$

The left side of Equation (3) is twice the area of triangles determined by p, q and r. So |D| is also this value.

(c) This method to determine the direction is good, because it avoid float-point error smartly.
On the one hand, it only includes plus, subtract and multiplication operations. For float point coordinate, multiplication needs double float accuracy, which can be satisfied by the computer, so we don't have to do "rounding" in calculation. For integer coordinate, multiplications are accommodated well by a long integer variable.
On the other hand, it doesn't have other complicate operations such as division, trigonometric or square root. All these operations are potential source for accumulation of float error.

1.6

(a) Suppose the convex hull of S is Φ_1, and the convex hull of its 2n endpoints is Φ_2.

First, as Φ_1 is the convex hull of n line segments S, select any two end-points p and q from S, we have

$\overline{pq} \subseteq \Phi_1$. So $\Phi_1 \supseteq \Phi_2$

Second, as Φ_2 is the convex hull of 2n end-points, select any line segments l from S, we have $l \subseteq \Phi_2$.

So $\Phi_2 \supseteq \Phi_1$

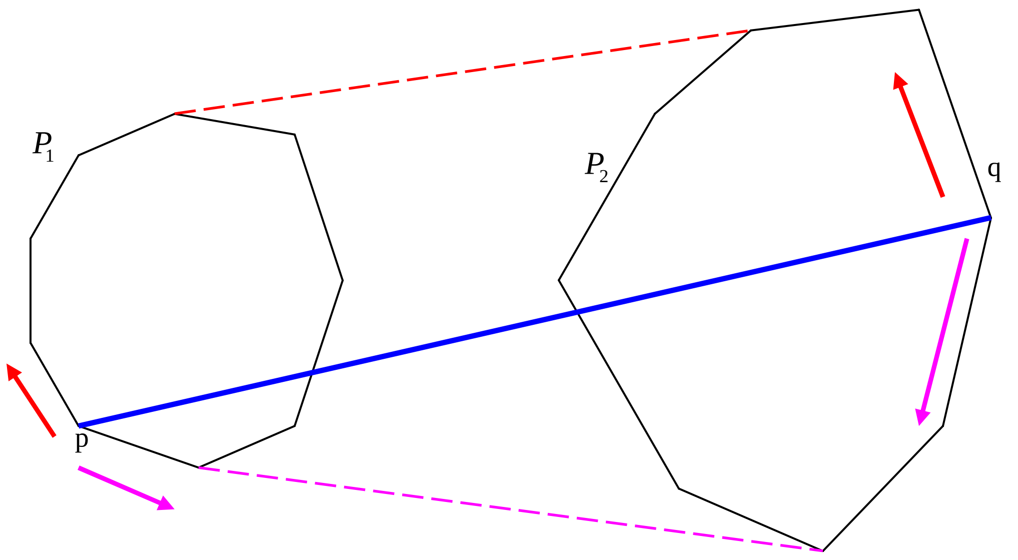
Therefore, $\Phi_1 = \Phi_2$, which means the convex hull of n line segments is exactly the convex hull of its 2n end-points.

(b) Suppose Φ isn't a convex polygon, and is given in a list table of edges $\{l_1, l_2, \ldots, l_n\}$, either counterclockwise or clockwise. Now assume it's clockwise. In addition, we initialize an empty convex hull list CH, and add the first edge l_1 of the polygon to CH.

Then we just perform a similar procedure like Graham-Scan (Textbook Page 6). Each time we visit a new edge l_{t+1}, we just check whether it is to the left or right of the previous edge l_t. If it's right, we add l_{t+1} to CH; Otherwise, we pop out l_t, and establish a new edge between the start point of l_t and the endpoint of l_{t+1}, and check again whether its to the right of l_{t-1} … We do this until we find a right turn or until CH is empty, then we add the new edge to CH and proceed to the next edge.
We do this edge by edge, and it will finally guarantee to find the convex hull for the polygon. As each edge can be added and popped at most once. The algorithm is bound to O(n) time.

1.8

(a) Given two disjoint convex polygons shown below:



Initially, we arbitrarily select one vertex from P_1 and one vertex from P_2, and draw a line segment between them.
We then iteratively update the endpoints of the segment one by one:

```
LOOP DO
    Update endpoint p and q to the right neighboring vertex with respect to line pq alternatively;
Until
    p's 2 neighboring vertices are to the left of pq    AND
        q's 2 neighboring vertices are to the left of pq
```

In this way, we find the upper bound edge for the convex hull of P1 and P2 (red dash line); Similarly, we can update p, q to the left neighboring vertices thus find the lower bound edge for the convex hull of P1 and P2 (pink dash line). As the total vertices are n, this iterative update will be finished in O(n) time.

(b) Suppose we have a set of n points. We first sort it with respect to x coordinates from left to right. We divide the point set recursively into two sets until each set has 2 or 3 points. We then just connect them together and perform a merge action as shown in Part (a). The sort and divide-and-conquer part both takes $n \log n$ time (for divide-and-conquer, we have T(n)=2T(n/2)+O(n)), so the total algorithm is in $n \log n$ time. Pseudo-code is shown below:

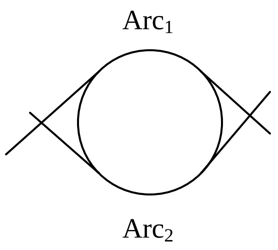
```
Input: n unsorted points {$p_1$, $p_2$, …, $p_n$}.
PointList[n] = Sort({$p_1$, $p_2$, …, $p_n$});
ConvexHull_DivConq(PointList)
    IF Num(PointList)==2
        RETURN the segment between PointList[0] and PointList[1];
    ELSE IF Num(PointList)==3
        RETURN the triangle determined by PointList[0], PointList[1] and PointList[2]
    ELSE
        LeftPointLists = First half of PointList;
        RightPointList = Last half of pointList;
        CH_L=ConvexHull_DivConq(LeftPointList);
        CH_R=ConvexHull_DivConq(RightPointList);
        CH_Merge(CH_L, CH_R);
    END
END
```

Note: in this algorithm, the subroutine CH_Merge(CH_L, CH_R) is just the algorithm to calculate the convex hull of two disjoint convex polygons shown in Part (a).

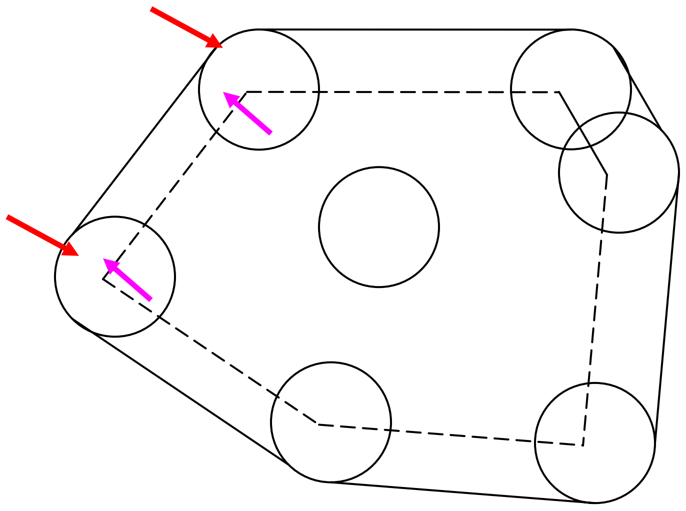
1.10

(a) Any single circle is convex, because the segment between any two points of the circle is always inside it; Moreover, its convex hull is the circle itself, as the circle has the shortest perimeter among all its convex sets. Therefore, for a series of circles, as convex hull is the convex set containing all the circles and has the smallest perimeter, it can only consist of straight lines and the arcs of circles, as shown by the following figure.

(b) Suppose one circle appears twice (Arc_1, Arc_2) on the boundary of the convex hull. Then the arcs cannot be connected by arcs from other circles as the convex hull boundary. Otherwise, it will not be convex. So they are connected by straight lines which are tangent to the arc. However, due to the fact that each circle has identical radius, if the straight lines are on the convex polygon, it will make the circle itself into a single convex hull, isolated from the other circles (shown below). So it's impossible.



(c) For the set of circles that appears on the convex hull, they are interconnected by tangent lines between each other. If we move these tangent lines inward exactly with respect to the direction from the tangent point to the center, we will get a convex hull for all the center points. Inversely, if we have the centers at the convex polygon, we can move the edges outward with respect to its orthogonal direction, and make up the disjoint areas with arcs, thus get a convex hull for the circles.



(d) Just as described in (c), we can first calculate the convex hull of the centers with Graham-Scan. This takes $O(n\log n)$ time. Then we march edge out in the orthogonal direction for a radius distance, and make up the arcs between them. This will takes O(n) time. In general, we need $O(n\log n)$ time to compute the convex hull of S.

(e) We can use Rappaport's divide-and-conquer approach to solve this problem. For rigorous and detailed description of this algorithm, please refer to:

David Rappaport. A convex hull algorithm for discs, and applications, Computational Geometry: Theory

and Applications, vol 1(3), 1992.