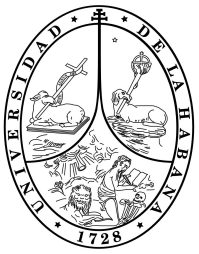


Proyecto de Diseño y Análisis de Algoritmos
"Convex hull"

Est. **Ariel Plasencia Díaz** a.plasencia@estudiantes.matcom.uh.cu

C-312

Índice

1 Introducción

El objetivo de este trabajo es presentar algunos algoritmos para calcular el menor polígono convexo que contiene a todos los puntos en un plano, es decir, la menor envoltura convexa o **convex hull**.

El análisis teórico del algoritmo será acompañado por la resolución de problemas cuya solución utiliza como paso principal el cálculo de **convex hull** y, en muchos casos, será guiado por él. Además de los problemas se verán análisis e implementación donde mostraremos la gran potencia y utilidad de este algoritmo.

2 Conceptos fundamentales

A continuación veremos algunos conceptos fundamentales que nos serán de gran utilidad a lo largo de nuestro trabajo.

2.1 Definición

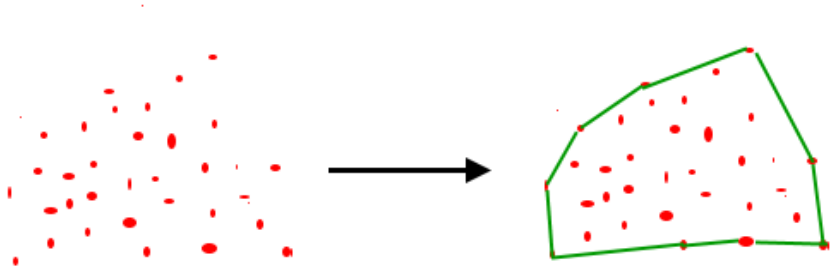
Un polígono es una figura plana compuesta por una secuencia limitada de segmentos rectos consecutivos que cierran una región del plano. Estos segmentos son llamados lados, y los puntos en que se intersecan se llaman vértices.

2.2 Definición

Sea P un polígono y $V(P)$ los vértices de P. Se dice que P es convexo si $\forall\ x, y \in V(P)$ el segmento $\overline{xy}$ está contenido dentro de P.

2.3 Definición

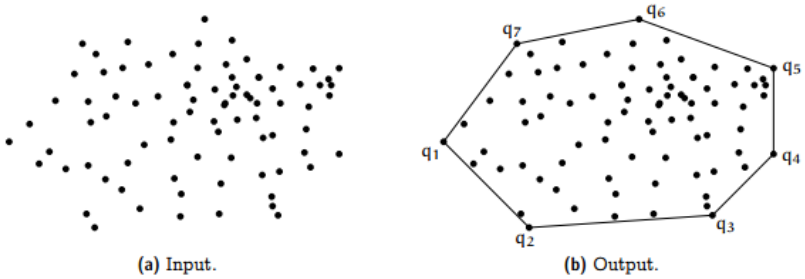
Sea P un polígono. La envoltura convexa o **convex hull** de P, se denota $CH(P)$, se define como la región de menor área en la cual está contenido P.



A partir de este momento nos referiremos a envoltura convexa como **convex hull**.

3 Presentación del algoritmo

En particular, en R^2 el problema de calcular el **convex hull** consiste en encontrar el polígono convexo más pequeño, tal que todos sus puntos estén en su frontera o en su interior. Para su solución propondremos algoritmos conocidos como el recorrido de Jarvis (1973), el algoritmo QuickHull (1977), el de Graham (1972), entre otros.



(a) Input. (b) Output.

A continuación, un simple ejemplo, al cual le daremos solución por distintos algoritmos.

3.1 Primer problema

Dado un entero N y un conjunto S de puntos en el plano, (x_1, y_1), (x_2, y_2), ..., (x_N, y_N). Devuelva el menor polígono convexo que contiene a todos los puntos del conjunto.

Entrada:
$N = 7$
$S = \{(0,3), (2,2), (1,1), (2,1), (3,0), (0,0), (3,3)\}$
Salida:
$CH = \{(0,3), (0,0), (3,0), (3,3)\}$

3.2 Algoritmo de Jarvis

Este algoritmo fue inventado independientemente por Chand y Kapur en 1970 y por R. A. Jarvis en 1973, aunque lleva el nombre de este último.

3.2.1 Explicación

El algoritmo usa una idea simple. Comenzamos por el punto más a la izquierda, es decir, el punto que contenga el mínimo valor de x y vamos guardando a partir de él, el próximo que pertenezca al **convex hull**, en sentido contrario a las agujas del reloj.

Dado un punto p, cómo saber cuál es el próximo punto que pertenece al **convex hull** en sentido contrario a las agujas del reloj?

Aquí la idea es usar el producto cruz, es decir, el próximo punto es q si para cualquier punto r, se cumple que orient(p, q, r) < 0 (ver epígrafe 4.1).

3.2.2 Código

Solución en C++

```
#include <bits/stdc++.h>
#define endl "\n"

using namespace std;

struct Point {
    int x, y;
};

int orientation(Point p, Point q, Point r) {
    int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);

    if(val == 0)
        return 0;
    else if(val > 0)
        return 1;
    else
        return 2;
}

vector<Point> convex_hull(int n, Point points[]) {
    int left = 0;
    vector<Point> hull;

    for(int i = 1; i < n; i++) {
        if(points[i].x < points[left].x)
            left = i;
    }

    int p = left, q;
```

```
    do {

        hull.push_back(points[p]);
        q = (p + 1) % n;

        for(int i = 0; i < n; i++) {
            if (orientation(points[p], points[i], points[q]) == 2)
                q = i;
        }

        p = q;

    } while(p != left);

    return hull;
}

void print(vector<Point> hull) {
    for(int i = 0; i < hull.size(); i++) {
        cout << ( << hull[i].x << , << hull[i].y << ) << endl;
    }
}

int main() {

    int N = 7;
    Point S[] = { {0, 3}, {2, 2}, {1, 1}, {2, 1}, {3, 0}, {0, 0}, {3, 3} };

    // there must be at least 3 points
    if(N < 3)
        return 0;

    vector<Point> hull = convex_hull(N, S);
    print(hull);
}
```

3.2.3 Complejidad temporal

Este algoritmo es $O(N \cdot H)$, donde N es la cantidad de puntos y H ($H \leq N$) la cantidad de puntos del **convex hull**, por tanto es $O(N^2)$ para el caso peor.

3.3 Algoritmo QuickHull

Este algoritmo fue creado independientemente por W. Eddy en 1977 y por A. Bykat en 1978.

3.3.1 Explicación

El algoritmo usa la idea de divide y vencerás muy parecido al QuickSort. Sigue los siguientes pasos:

1. Encuentra el punto cuya coordenada x (min_x) es mínima y donde la coordenada y (max_y) es máxima.
2. Realiza una línea uniendo estos dos puntos, llamémosle a la línea L.
3. Para cada parte, busca el punto P cuya distancia con L es máxima. P forma un triángulo con los puntos min_x y max_y, notar que los puntos dentro de este triángulo no pueden ser parte del **convex hull** (envoltura convexa).
4. El paso anterior divide nuestro problema en dos subproblemas que se resuelven recursivamente, ahora la línea L es la que une los puntos P con min_x y P con max_y.

5. Repita el paso 3 hasta que no quede ningún punto fuera, en tal caso estos últimos puntos pertenecen al **convex hull**.

En la implementación siguiente los puntos se representan por pares (pair<int, int>) que se guardan en un conjunto ordenado (set<pair>).

3.3.2 Código

Solución en C++

```
#include<bits/stdc++.h>
#define pii pair<int, int>
#define endl "\n"

using namespace std;

set<pii> hull;

int find_side(pii p1, pii p2, pii p) {
    int val = (p.second - p1.second) * (p2.first - p1.first)
            - (p2.second - p1.second) * (p.first - p1.first);

    if(val > 0)
        return 1;
    else if(val < 0)
        return -1;
    else
        return 0;
}

int line_dist(pii p1, pii p2, pii p) {
    return abs((p.second - p1.second) * (p2.first - p1.first)
            - (p2.second - p1.second) * (p.first - p1.first));
}

void quick_hull(pii a[], int n, pii p1, pii p2, int side) {
    int ind = -1;
    int max_dist = 0;

    for(int i = 0; i < n; i++) {
        int temp = line_dist(p1, p2, a[i]);
        if(find_side(p1, p2, a[i]) == side && temp > max_dist) {
            ind = i;
            max_dist = temp;
        }
    }

    if(ind == -1) {
        hull.insert(p1);
        hull.insert(p2);
        return;
    }

    quick_hull(a, n, a[ind], p1, -find_side(a[ind], p1, p2));
    quick_hull(a, n, a[ind], p2, -find_side(a[ind], p2, p1));
}

void print_hull(pii a[], int n) {
    if(n < 3) {
        cout << convex hull not possible << endl;
```

```
        return;
    }

    int min_x = 0, max_x = 0;
    for(int i = 1; i < n; i++) {
        if(a[i].first < a[min_x].first)
            min_x = i;
        if(a[i].first > a[max_x].first)
            max_x = i;
    }

    quick_hull(a, n, a[min_x], a[max_x], 1);
    quick_hull(a, n, a[min_x], a[max_x], -1);
}

void print() {
    while(!hull.empty()) {
        cout << ( << (*hull.begin()).first << , << (*hull.begin()).second << ) << endl;
        hull.erase(hull.begin());
    }
}

int main() {

    int N = 7;
    pii points[] = {{0, 3}, {2, 2}, {1, 1}, {2, 1}, {3, 0}, {0, 0}, {3, 3}};

    print_hull(points, N);
    print();
}
```

3.3.3 Complejidad temporal

N_1 y N_2 son el tamaño de los subproblemas.

$$T(N) = \left\{ \begin{array}{cc} 1, & si\ N\ =\ 1 \\ N + T(N_1) + T(N_2), & donde\ N_1\ +\ N_2\ \leq\ N \end{array} \right\}$$

El costo de este algoritmo es similar al QuickSort. En promedio, es $O(N \cdot log(N))$ pero su caso peor es $O(N^2)$, donde N es la cantidad de puntos.

3.4 Algoritmo de Graham Scan

Fue publicado por Ronald Graham en 1972.

3.4.1 Explicación

A grandes reagos, nuestro algoritmo sigue los siguientes pasos:

1. Encontrar el punto con menor y, en el caso que hayan dos con la misma guardar la de menor x. Este punto lo colocamos como primer elemento de nuestro array.
2. Ordenar el array, a partir del segundo elemento, según el ángulo que forman con el primer elemento en el sentido contrario a las agujas del reloj.
3. Chequear si dos puntos tienen el mismo ángulo y en este caso quitar el más alejado.
4. Si, en este momento, tenemos menos de tres puntos no es posible calcular el **convex hull**.
5. Creamos una pila y añadimos los tres primeros puntos del array, a partir de la posición 0.

6. Procesar los restantes puntos de la manera siguiente, para cada punto *isimo*
   (a) Quitar de la pila mientras la orientación de los puntos $pila.next_{top}$, *pila.top* y *point*[*i*] no sea en sentido a las agujas del reloj.
   (b) Añadimos *point*[*i*].
7. Imprimir el contenido de la pila.

3.4.2 Código

Solución en C++

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

struct Point {
    int x, y;
};

Point p0;

Point next_to_top(stack<Point> &S) {
    Point p = S.top();
    S.pop();
    Point res = S.top();
    S.push(p);
    return res;
}

int swap(Point &p1, Point &p2) {
    Point temp = p1;
    p1 = p2;
    p2 = temp;
}

int dist_square(Point p1, Point p2) {
    return (p1.x - p2.x) * (p1.x - p2.x) +
           (p1.y - p2.y) * (p1.y - p2.y);
}

int orientation(Point p, Point q, Point r) {
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);
    if(val == 0)
        return 0;
    else if(val > 0)
        return 1;
    else
        return 2;
}

int compare(const void *vp1, const void *vp2) {
    Point *p1 = (Point *)vp1;
    Point *p2 = (Point *)vp2;

    int o = orientation(p0, *p1, *p2);
    if(o == 0) {
        if((dist_square(p0, *p2) >= dist_square(p0, *p1)))
```

```
            return -1;
        else
            return 1;
    }
    else if(o == 2)
        return -1;
    else
        return 1;
}

void print(stack<Point> S) {
    while(!S.empty()) {
        Point p = S.top();
        cout << ( << p.x << , << p.y << ) << endl;
        S.pop();
    }
}

void convex_hull(Point points[], int n) {

    int ymin = points[0].y, min = 0;
    for (int i = 1; i < n; i++) {
        int y = points[i].y;

        if((y < ymin) || (ymin == y && points[i].x < points[min].x)) {
            ymin = points[i].y;
            min = i;
        }
    }

    swap(points[0], points[min]);

    p0 = points[0];
    qsort(&points[1], n - 1, sizeof(Point), compare);

    int m = 1;
    for(int i = 1; i < n; i++) {

        while(i < n - 1 && orientation(p0, points[i], points[i + 1]) == 0)
            i++;

        points[m] = points[i];
        m++;
    }

    if(m < 3)
        return;

    stack<Point> S;
    S.push(points[0]);
    S.push(points[1]);
    S.push(points[2]);

    for(int i = 3; i < m; i++) {
        while(orientation(next_to_top(S), S.top(), points[i]) != 2)
            S.pop();

        S.push(points[i]);
```

```
    }

    print(S);
}

int main() {

    int N = 7;
    Point points[] = { {0, 3}, {2, 2}, {1, 1}, {2, 1}, {3, 0}, {0, 0}, {3, 3} };

    convex_hull(points, N);
}
```

3.4.3 Complejidad temporal

Sea N la cantidad de puntos, entonces la complejidad temporal nos queda:

1. Encontrar el punto tal que su coordenada sea la menor ($O(N)$).
2. Ordenar el array ($O(N \cdot log(N))$).
3. Chequear si dos puntos tienen el mismo ángulo ($O(N)$).
4. Insertar los tres primeros puntos en una pila ($O(1)$).
5. Procesar los puntos restantes ($O(N)$).

Notar que todos los puntos se insertan y se elimina a lo sumo una vez de la pila, las cuales son operaciones constantes.

$\Rightarrow T(N) = O(N) + O(N \cdot log(N)) + O(N) + O(1) + O(N)$
$\Rightarrow T(N) = O(N \cdot log(N))$

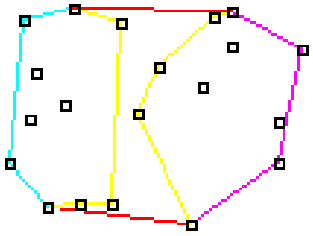
3.5 Algoritmo por Divide y Vencerás

Este algoritmo fue publicado en 1977 por Preparata y Hong.

3.5.1 Explicación

Como su nombre lo indica este algoritmo calcula el **convex hull** de la mitad hacia la izquierda (color azul) y de la mitad hacia la derecha (color rosado). Para combinar ambos subproblemas nos apoyamos en la tangente tanto superior como inferior (color rojo).

El problema está cuando queremos encontrar el **convex hull** del medio (color amarillo). La recursión entra en el caso base, cuando la cantidad de puntos es muy pequeña, digamos 5, que se calcula de forma bruta. La fusión de estas ideas nos proporciona el **convex hull** del conjunto de puntos.



3.5.2 Código

No mostraremos la implementación porque es muy extensa, sin embargo les dejamos un link a la solución.

Solución en C++

3.5.3 Complejidad temporal

Nuestra complejidad temporal quedaría $T(N) = 2 \cdot T(\frac{N}{2}) + N$, que usando es Teorema Maestro:

$\Rightarrow a = 2 \wedge b = 2$

$\Rightarrow N^{\log_b(a)} = N^{\log_2(2)} = N^1 = N$

como $f(N) = N$

$\Rightarrow f(N) \in \Theta(N^{\log_2(2)})$

$\Rightarrow f(N) \in \Theta(N)$

$\Rightarrow T(N) \in \Theta(N \cdot log(N))$

En fin, $T(N) \in \Theta(N \cdot log(N))$ por el Teorema Maestro.

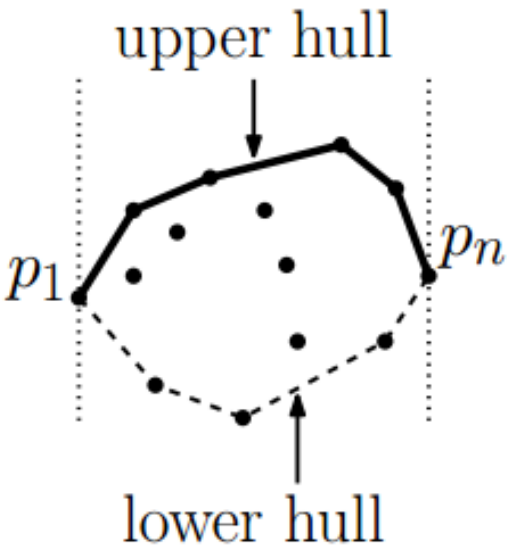
3.6 Algoritmo de Andrew

Este algoritmo fue publicado por A. M. Andrew en 1979. Es considerado como una variación al algoritmo de Graham Scan, visto anteriormente. Se destaca por su corta implementación.

3.6.1 Explicación

El algoritmo propuesto por Andrew sigue los pasos siguientes:

1. Ordena el conjunto de puntos según las abscisas.
2. Calcula el upper hull (envoltura de arriba) desde el punto más a la derecha hasta el más a la derecha en sentido contrario a las agujas del reloj.
3. De manera análoga, calcula el lower hull o envoltura de abajo.
4. El **convex hull** final es la union de las dos envolturas.



3.6.2 Código

Solución en C++

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

struct Point {
    int x, y;

    bool operator <(const Point &p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};

double cross(const Point &O, const Point &A, const Point &B) {
    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

vector<Point> convex_hull(vector<Point> &p) {

    int n = p.size(), k = 0;
    vector<Point> h(2 * n);
    sort(p.begin(), p.end());

    for(int i = 0; i < n; h[k++] = p[i++]) {
        while(k >= 2 && cross(h[k - 2], h[k - 1], p[i]) <= 0)
            --k;
    }

    for(int i = n - 2, t = k + 1; i >= 0; h[k++] = p[i--]) {
        while(k >= t && cross(h[k - 2], h[k - 1], p[i]) <= 0)
            --k;
    }

    return vector<Point>(h.begin(), h.begin() + k - (k > 1));
}

void print(vector<Point> hull) {
    for(int i = 0; i < hull.size(); i++) {
        cout << hull[i].x << << hull[i].y << endl;
    }
}

int main() {

    int N, x, y;
    cin >> N;

    Point point;
    vector<Point> p;

    for(int i = 0; i < N; i++) {
        cin >> x >> y;
        point.x = x;
        point.y = y;
        p.push_back(point);
```

```
    }

    vector<Point> conv_hull = convex_hull(p);
    print(conv_hull);
}
```

3.6.3 Complejidad temporal

1. Ordenar ($O(N \cdot log(N))$).
2. Calcular el upper hull ($O(N)$).
3. Calcular el lower hull ($O(N)$).

$\Rightarrow T(N) \in (O(N \cdot log(N)) + O(N) + O(N))$

$\Rightarrow T(N) \in O(N \cdot log(N))$

4 Demostraciones

4.1 Correctitud del método orient

4.1.1 Teorema

$\overrightarrow{qr}\ X\ \overrightarrow{pq} = (q.y - p.y) \cdot (r.x - q.x) - (q.x - p.x) \cdot (r.y - q.y)$.

4.1.2 Demostración

$$\overrightarrow{qr}\ X\ \overrightarrow{pq} = \det \begin{pmatrix} r.x - q.x & q.x - p.x \\ r.y - q.y & q.y - p.y \end{pmatrix}$$

$$\overrightarrow{qr}\ X\ \overrightarrow{pq} = \begin{vmatrix} r.x - q.x & q.x - p.x \\ r.y - q.y & q.y - p.y \end{vmatrix}$$

$$\overrightarrow{qr}\ X\ \overrightarrow{pq} = (r.x - q.x) \cdot (q.y - p.y) - (r.y - q.y) \cdot (q.x - p.x)$$

$$\overrightarrow{qr}\ X\ \overrightarrow{pq} = (q.y - p.y) \cdot (r.x - q.x) - (q.x - p.x) \cdot (r.y - q.y)$$

4.1.3 Teorema

Si $\overrightarrow{qr}\ X\ \overrightarrow{pq} > 0$ entonces el punto r está al lado izquierdo del vector $\overrightarrow{pq}$.

4.1.4 Demostración

$\overrightarrow{qr}\ X\ \overrightarrow{pq} = \| \overrightarrow{qr} \| \cdot \| \overrightarrow{pq} \| \cdot \sin\theta$, donde θ es el ángulo entre $\overrightarrow{qr}$ y $\overrightarrow{pq}$

como $\overrightarrow{qr}\ X\ \overrightarrow{pq} > 0 \Rightarrow 0 < \theta < \pi$, ya que sin es positivo en el primer y segundo cuadrante

$\Rightarrow \overrightarrow{qr}$ apunta hacia el lado izquierdo

$\Rightarrow$ el punto r se encuentra a la izquierda del vector $\overrightarrow{pq}$

4.1.5 Teorema

Si $\overrightarrow{qr}\ X\ \overrightarrow{pq} < 0$ entonces el punto r está al lado derecho del vector $\overrightarrow{pq}$.

4.1.6 Demostración

$\overrightarrow{qr}\ X\ \overrightarrow{pq} = \| \overrightarrow{qr} \| \cdot \| \overrightarrow{pq} \| \cdot \sin\theta$, donde θ es el ángulo entre $\overrightarrow{qr}$ y $\overrightarrow{pq}$

como $\overrightarrow{qr}\ X\ \overrightarrow{pq} < 0 \Rightarrow -\pi < \theta < 0$, ya que sin es negativo en el tercer y cuarto cuadrante

$\Rightarrow \overrightarrow{qr}$ apunta hacia el lado derecho

$\Rightarrow$ el punto r se encuentra a la derecha del vector $\overrightarrow{pq}$

4.1.7 Teorema

Si $\overrightarrow{qr}\ X\ \overrightarrow{pq} = 0$ entonces los puntos p, q y r son colineales, es decir, están en la misma recta.

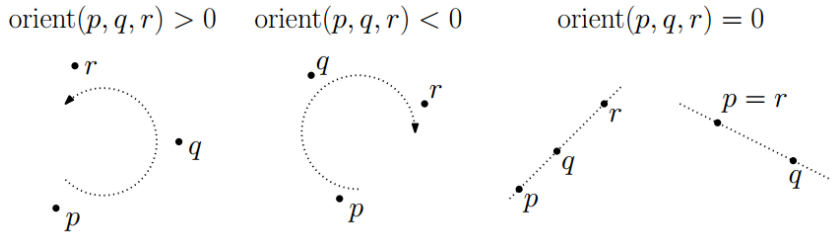
4.1.8 Demostración

$\overrightarrow{qr}\ X\ \overrightarrow{pq} = \| \overrightarrow{qr} \| \cdot \| \overrightarrow{pq} \| \cdot \sin\theta$, donde θ es el ángulo entre $\overrightarrow{qr}$ y $\overrightarrow{pq}$

como $\overrightarrow{qr}\ X\ \overrightarrow{pq} = 0 \Rightarrow \theta = 0 \vee \theta = \pi$

$\Rightarrow \overrightarrow{qr}$ y $\overrightarrow{pq}$ están en la misma dirección y sentido

$\Rightarrow$ los puntos p, q y r son colineales



4.2 Colorario de la definición de polígono convexo

4.2.1 Teorema

Un polígono convexo no tiene ángulos sobreobtusos.

4.2.2 Demostración

sea P un polígono convexo

supongamos que $\exists\ m \in V(P)$ tal que m forma un ángulo sobreobtuso

sean $a, b \in V(P)$ tal que a y b son adyacentes a m

$\Rightarrow$ contradicción porque el segmento $\overline{ab}$ no está dentro del polígono convexo P

5 Ejercicios

En esta sección se propondrán y se resolverán ejercicios un poco más interesantes, los cuales tendrán como peso esencial el cálculo del polígono convexo de menor área que con tiene a todos los puntos. En los ejercicios utilizaremos, por lo general, el Algoritmo de Andrew debido a su corta implementación, pero cabe destacar que casi todos resuelven el mismo problema, incluso con tiempos muy parecidos.

5.1 Build the Fence

5.1.1 Problema

Ver problema

At the beginning of spring all the sheep move to the higher pastures in the mountains. If there are thousands of them, it is well worthwhile gathering them together in one place. But sheep don't like to leave their grass-lands. Help the shepherd and build him a fence which would surround all the sheep. The fence should have the smallest possible length! Assume that sheep are negligibly small and that they are not moving. Sometimes a few sheep are standing in the same place. If there is only one sheep, it is probably dying, so no fence is needed at all...

Input:
T [the number of tests ≤ 100]
[empty line]
N [the number of sheep $\leq 10^5$]
x_1 y_1 [coordinates of the first sheep]
...
x_N y_N
[integer coordinates from -10000 to 10000]
[empty line]
[other lists of sheep]
Text grouped in [ ] does not appear in the input file. Assume that sheep are numbered in the input order.

Output:
O [length of circumference, rounded to 2 decimal places]
p_1 p_2 ... p_K
[the sheep that are standing in the corners of the fence; the first one should be positioned bottommost and as far to the left as possible, the others ought to be written in anticlockwise order; ignore all sheep standing in the same place but the first to appear in the input file; the number of sheep should be the smallest possible]
[empty line]
[next solutions]

5.1.2 Análisis

En este problema nos mandan a calcular el perímetro del menor polígono convexo tal que contenga a todos los puntos de la entrada, así como las posiciones de los puntos que participan en dicho polígono. Notar que la definición anterior coincide con la **convex hull**.

Para dar solución a este problema creamos una estructura *Point* donde guardamos las coordenadas del punto y su posición. Calculamos nuestro **convex hull**, así como su perímetro, que no es más que la suma de todos los lados del polígono. Para terminar imprimimos tanto el perímetro calculado como las posiciones guardadas estructura que pertenecen al **convex hull**.

5.1.3 Implementación

Solución en C++

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

struct Point {
    int x, y, id;

    bool operator <(const Point &p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};

double cross(const Point &O, const Point &A, const Point &B) {
    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

void print_id(vector<Point> p) {
    for(int i = 0; i < p.size(); i++) {
        cout << p[i].id << ;
    }
    cout << endl;
}
```

```
double get_distance(Point a, Point b) {
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

double get_perimeter(vector<Point> conv_hull) {
    double per = 0;
    for(int i = 0; i < conv_hull.size() - 1; i++) {
        per += get_distance(conv_hull[i], conv_hull[i + 1]);
    }
    per += get_distance(conv_hull[0], conv_hull[conv_hull.size() - 1]);
    return per;
}

vector<Point> convex_hull(vector<Point> &p) {

    int n = p.size(), k = 0;
    vector<Point> h(2 * n);
    sort(p.begin(), p.end());

    for (int i = 0; i < n; h[k++] = p[i++]) {
        while (k >= 2 && cross(h[k - 2], h[k - 1], p[i]) <= 0)
            --k;
    }

    for (int i = n - 2, t = k + 1; i >= 0; h[k++] = p[i--]) {
        while (k >= t && cross(h[k - 2], h[k - 1], p[i]) <= 0)
            --k;
    }

    return vector<Point>(h.begin(), h.begin() + k - (k > 1));
}

int main() {

    ios_base :: sync_with_stdio(0);
    cin.tie(0);
    cout << setprecision(2) << fixed;

    int T, N, x, y, z;
    cin >> T;

    while(T--) {

        cin >> N;

        Point point;
        vector<Point> p;

        for(int i = 0; i < N; i++) {
            cin >> x >> y;
            point.x = x;
            point.y = y;
            point.id = i + 1;
            p.push_back(point);
        }

        vector<Point> conv_hull = convex_hull(p);
```

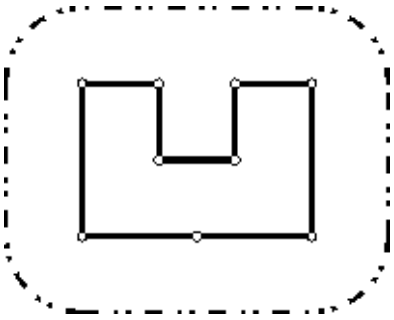
```
        double perimeter = get_perimeter(conv_hull);
        cout << perimeter << endl;
        print_id(conv_hull);

        if(T)
            cout << endl;
    }
}
```

5.2 Wall

5.2.1 Problema

Ver problema



Once upon a time there was a greedy King who ordered his chief Architect to build a wall around the King's castle. The King was so greedy, that he would not listen to his Architect's proposals to build a beautiful brick wall with a perfect shape and nice tall towers. Instead, he ordered to build the wall around the whole castle using the least amount of stone and labor, but demanded that the wall should not come closer to the castle than a certain distance. If the King finds that the Architect has used more resources to build the wall than it was absolutely necessary to satisfy those requirements, then the Architect will loose his head. Moreover, he demanded Architect to introduce at once a plan of the wall listing the exact amount of resources that are needed to build the wall.

Your task is to help poor Architect to save his head, by writing a program that will find the minimum possible length of the wall that he could build around the castle to satisfy King's requirements.

The task is somewhat simplified by the fact, that the King's castle has a polygonal shape and is situated on a flat ground. The Architect has already established a Cartesian coordinate system and has precisely measured the coordinates of all castle's vertices in feet.

Input: The first line contains two integers N and L separated by a space. N ($3 \leq N \leq 1000$) is the number of vertices in the King's castle, and L ($1 \leq L \leq 1000$) is the minimal number of feet that King allows for the wall to come close to the castle. Next N lines describe coordinates of castle's vertices in a clockwise order. Each line contains two integers x_i and y_i separated by a space ($-10000 \leq Xi, Yi \leq 10000$) that represent the coordinates of *ith* vertex. All vertices are different and the sides of the castle do not intersect anywhere except for vertices.

Output: Write the single number that represents the minimal possible length of the wall in feet that could be built around the castle to satisfy King's requirements. You must present the integer number of feet to the King, because the floating numbers are not invented yet. However, you must round the result in such a way, that it is accurate to 8 inches (1 foot is equal to 12 inches), since the King will not tolerate larger error in the estimates.

5.2.2 Análisis

Este problema es muy similar al anterior, con la simple peculiaridad de convertir la respuesta final a otra unidad de medida.

5.2.3 Implementación

Solución en C++

```cpp
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

struct Point {
    int x, y;

    bool operator <(const Point &p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};

double cross(const Point &O, const Point &A, const Point &B) {
    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

vector<Point> convex_hull(vector<Point> &p) {

    int n = p.size(), k = 0;
    vector<Point> h(2 * n);
    sort(p.begin(), p.end());

    for (int i = 0; i < n; h[k++] = p[i++]) {
        while (k >= 2 && cross(h[k - 2], h[k - 1], p[i]) <= 0)
            --k;
    }

    for (int i = n - 2, t = k + 1; i >= 0; h[k++] = p[i--]) {
        while (k >= t && cross(h[k - 2], h[k - 1], p[i]) <= 0)
            --k;
    }

    return vector<Point>(h.begin(), h.begin() + k - (k > 1));
}

double get_distance(Point a, Point b) {
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

double get_perimeter(vector<Point> conv_hull) {
    double per = 0;
    for(int i = 0; i < conv_hull.size() - 1; i++) {
        per += get_distance(conv_hull[i], conv_hull[i + 1]);
    }
    per += get_distance(conv_hull[0], conv_hull[conv_hull.size() - 1]);
    return per;
}

int main() {

    int N, L, x, y;
    cin >> N >> L;

    Point point;
    vector<Point> p;

    for(int i = 0; i < N; i++) {
```

```
        cin >> x >> y;
        point.x = x;
        point.y = y;
        p.push_back(point);
    }

    vector<Point> conv_hull = convex_hull(p);

    double perimeter = get_perimeter(conv_hull);

    // if we round to the nearest foot, it is within 6 inches (hence within 8 inches)
    int inches = 4 * acos(0.0) * L;

    cout << perimeter + inches << endl;
}
```

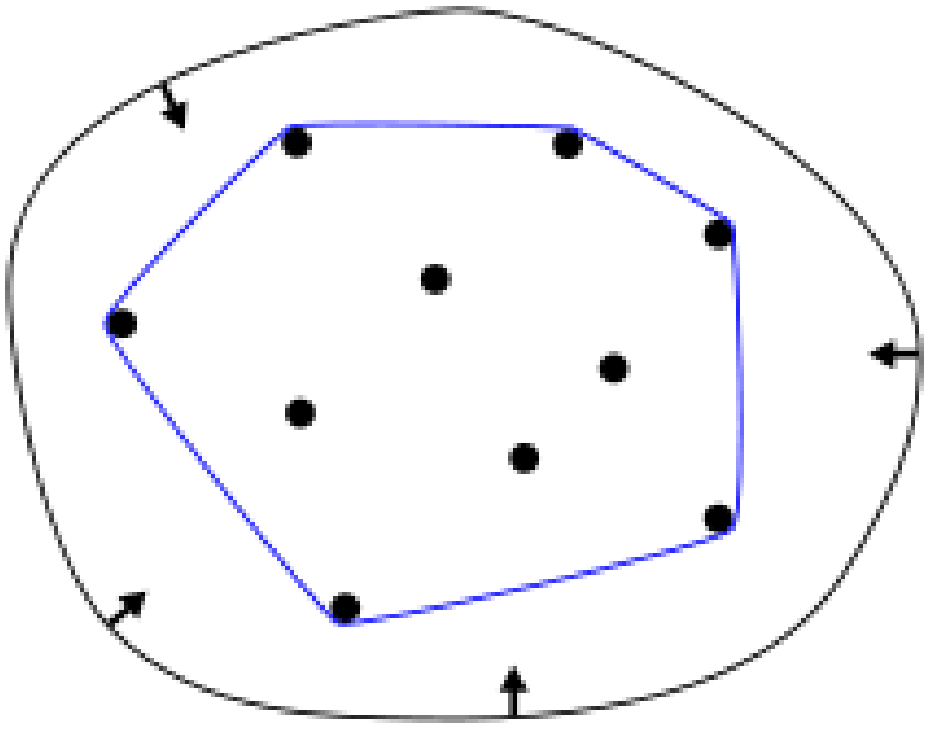
5.3 Polygon

5.3.1 Problema

Ver problema

Convex Hull of a set of points, in 2D plane, is a convex polygon with minimum area such that each point lies either on the boundary of polygon or inside it.

Let's consider a 2D plane, where we plug pegs at the points mentioned. We enclose all the pegs with a elastic band and then release it to take its shape. The closed structure formed by elastic band is similar to that of convex hull.



In the above figure, convex hull of the points, represented as dots, is the polygon formed by blue line.

Given a set of N points, Find the perimeter of the convex hull for the points.

Input: First line of input will contain a integer, N, number of points. Then follow N lines where each line contains the coordinate, $(x_i y_i)$, of ith point...

Output: Print the perimeter of convex hull for the given set of points. Print the perimeter till 1 decimal value.

5.3.2 Implementación

Solución en C++

```
#include<bits/stdc++.h>
#define endl '\n'

using namespace std;

struct Point {
    int x, y;

    bool operator <(const Point &p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};

double cross(const Point &O, const Point &A, const Point &B) {
    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

vector<Point> convex_hull(vector<Point> &p) {
    int n = p.size(), k = 0;
    vector<Point> h(2 * n);
    sort(p.begin(), p.end());

    for (int i = 0; i < n; h[k++] = p[i++]) {
        while (k >= 2 && cross(h[k - 2], h[k - 1], p[i]) <= 0)
            --k;
    }

    for (int i = n - 2, t = k + 1; i >= 0; h[k++] = p[i--]) {
        while (k >= t && cross(h[k - 2], h[k - 1], p[i]) <= 0)
            --k;
    }

    return vector<Point>(h.begin(), h.begin() + k - (k > 1));
}

double get_distance(Point a, Point b) {
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

double get_perimeter(vector<Point> conv_hull) {
    double per = 0;
    for(int i = 0; i < conv_hull.size() - 1; i++) {
        per += get_distance(conv_hull[i], conv_hull[i + 1]);
    }
    per += get_distance(conv_hull[0], conv_hull[conv_hull.size() - 1]);
    return per;
}

int main() {

    cout << setprecision(1) << fixed;

    int N, x, y;
    cin >> N;
```

```
    Point point;
    vector<Point> p;

    for(int i = 0; i < N; i++) {
        cin >> x >> y;
        point.x = x;
        point.y = y;
        p.push_back(point);
    }

    vector<Point> conv_hull = convex_hull(p);
    double perimeter = get_perimeter(conv_hull);
    cout << perimeter << endl;
}
```

5.4 SCUD Busters

5.4.1 Problema

Ver problema

5.4.2 Análisis

Este problema, a grandes rasgos, nos piden calcular varios **convexs hulls**. Luego, después de una sucesión de puntos en forma de queries, devolver la suma de las área de los polígonos que contienen a estos puntos.

Para la resolución de este problema marcamos los polígonos que contienen los puntos dados en forma de queries y calculamos la suma de las áreas de estos polígonos. Aquí la nueva peculiaridad es cómo calcular el área de un polígono convexo.

Dado un polígono P con los vértices $v_0, v_1, ..., v_N$, donde $v_0 = v_N$. El área del polígono es

$$A(P) = \frac{1}{2} \cdot \sum_{i=1}^{N} (x_{i-1}y_i) - (x_i y_{i-1})$$

donde x, y son las coordenadas de $v_i = (x_i y_i)$ y los lados del polígono van desde v_i hasta v_{i+1}, $\forall i = 0, ..., N-1$. Si los puntos del polígono están orientados en contra de las agujas del reloj, entonces el valor de $A(P)$ será positiva, en cambio si los puntos están orientados a favor de las agujas del reloj entonces el área sera negativa.

5.4.3 Implementación

Solución en C++

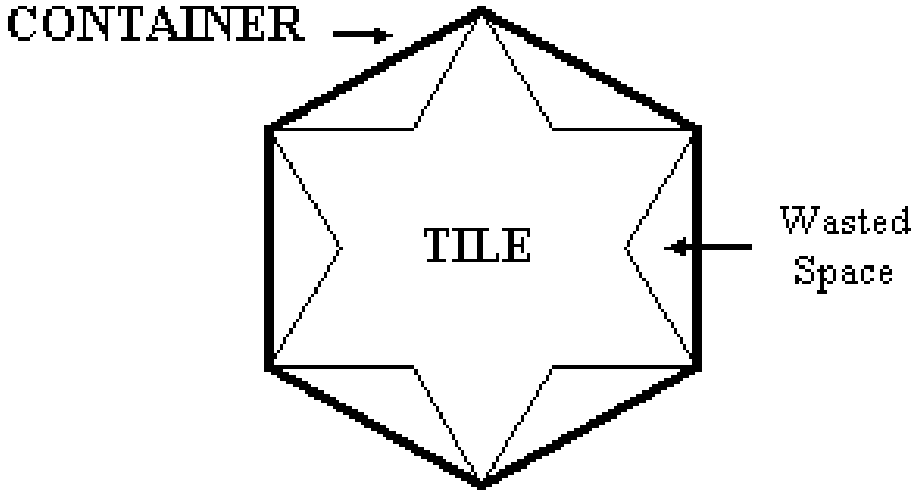
5.5 Useless Tile Packers

5.5.1 Problema

Ver problema

5.5.2 Análisis

Este ejercicio nos pide calcular, dado N puntos en el plano, el tanto por ciento del desaprovechamiento del polígono formado por los N puntos con respecto al menor polígono convexo que contiene a dichos puntos. Se define como desaprovechamiento a la resta modular del área de estos dos polígonos. Notar que el área se puede calcular con la fórmula mostrada en el ejercicio anterior.



5.5.3 Implementación

Solución en C++