Geometría computacional

Pedro O. Pérez M., PhD.

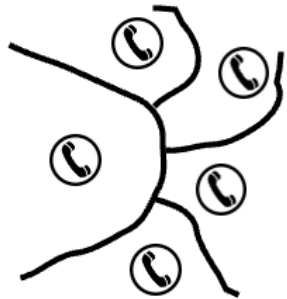
Análisis y diseño de algoritmos avanzados
Tecnológico de Monterrey

pperezm@tec.mx

08-2022

La geometría computacional es el área de las ciencias computacionales que se ocupa de calcular las propiedades geométricas de conjuntos de objetos geométricos en el espacio, como la sencilla operación de determinar si un punto está encima o debajo de una línea determinada. En un sentido más general, la geometría computacional se ocupa del diseño y análisis de algoritmos para resolver problemas geométricos.

Imagina que está caminando por el campus de tu universidad y, de repente, recuerdas que tienes que hacer una llamada telefónica urgente y, sucede algunas veces, tu celular se ha quedado sin batería. Lo bueno, es que todavía hay muchos teléfonos públicos en el campus y, por supuesto, deseas ir al más cercano. ¿Pero cuál es el más cercano? Sería muy útil poder contar con un mapa que nos muestra, por regiones, el teléfono público más cercado, ¿no es cierto? Pero ¿cómo definimos estas regiones?



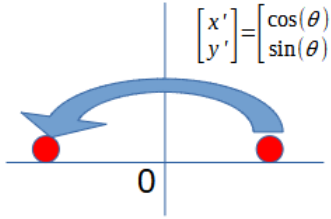
Bueno, supongamos que ya hemos encontrado el ansiado teléfono público más cercano. Ahora, con un mapa del campus en mano, probablemente tendremos pocos problemas para llegar a él por el camino más corto, sin golpear paredes u otros obstáculos. Pero, programar a un robot para que realice la misma tarea es mucho más difícil. Lo anterior, implicaría que tendríamos que calcular el camino más corto entre dos puntos evitando colisionar con una colección de obstáculos geométricos.

Punto

Empecemos con la primitiva geométrica más simple: el punto. El punto es el bloque de construcción básico de los objetos de geometría de dimensiones superiores. En el espacio euclidiano 2D, un punto es, generalmente, representado como estructura o clase.

```
1   class Point {
2   private:
3     double x, y;
4
5   public:
6     Point();
7     Point(double, double);
8     Point(const Point&);
9
10    double getX() const;
11    double getY() const;
12  };
```

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$




```
1   Point rotate(const Point &p, double theta) {
2     double rad = degToRad(theta);
3     return Point(p.getX() * cos(rad) - p.getY() * sin(rad),
4                  p.getX() * sin(rad) + p.getY() * cos(rad));
5   }
```

Las líneas son de longitud infinita en ambas direcciones. Si son infinitas, ¿cómo representamos una línea en un plano? Las líneas pueden ser representadas de dos formas diferentes, ya sea usando puntos o la ecuación de la línea.

- En el primer caso, cada línea l está completamente representada por cualquier par de puntos (x_1, y_1) y (x_2, y_2) que se encuentren en la línea.
- Si usamos la ecuación de la línea, una línea se describe mediante la ecuación $y = m * x + b$, donde m es la pendiente de la línea y b es la intersección con el eje, es decir, el punto único $(0, b)$ donde cruza el eje X. La línea l tiene una pendiente $m = \Delta y / \Delta x = \frac{(y_1 - y_2)}{(x_1 - x_2)}$ e intercepta en $b = y_1 - m * x_1$.

Lo mejor es usar la fórmula general de la línea: $ax + by + c = 0$.

```
1   class Line {
2   private:
3     double a, b, c;
4
5   public:
6     Line();
7     Line(double, double, double);
8     Line(const Line&);
9
10    double getA() const;
11    double getB() const;
12    double getC() const;
13  };
```

```
1   Line pointsToLine(const Point &p1, const Point &p2) {
2     double aux;
3
4     if (fabs(p1.getX() - p2.getX()) < EPSILON) {
5       return Line(1.0, 0.0, -p1.getX());
6     } else {
7       aux = (p1.getY() - p2.getY()) / (p1.getX() - p2.getX());
8       return Line (-aux,
9                    1.0,
10                   -(aux * p1.getX()) - p1.getY());
11    }
12  }
```

```
1   Line pointAndSlopeToLine(const Point &p, double m) {
2     return Line(-m, 1, -(-m * p.getX()) + p.getY());
3   }
```

Decimos que dos líneas siempre tienen un punto de intersección a menos que sean paralelas; en cuyo caso, no lo tienen. Para determinar si dos rectas son paralelas, solo debemos verificar si sus coeficientes a y b son iguales.

```
1   bool areParallel(const Line &l1, const Line &l2) {
2     return ( (fabs(l1.getA() - l2.getA()) <= EPSILON) &&
3              (fabs(l1.getB() - l2.getB()) <= EPSILON) );
4   }
```

Basándonos en la función previa, también podemos comprobar si esas dos líneas son la misma. Decimos que las líneas a y b son la misma, si son paralelas y, además, sus coeficientes c son los mismos. En otras palabras, los tres coeficientes, a, b y c, son los mismos.

```
1   bool areSameLine(const Line &l1, const Line &l2) {
2     return ( areParallel(l1, l2) &&
3              (fabs(l1.getC() - l2.getC()) <= EPSILON) );
4   }
```

Ahora, si dos líneas no son paralelas, y por lo tanto no pueden ser la misma, ambas líneas se intersecarán en un cierto punto. El punto de intersección, (x, y), se puede encontrar resolviendo el sistema de ecuaciones que se genera de las fórmulas: $a_1x + b_1y + c_1 = 0$ y $a_2x + b_2y + c_2 = 0$.

```
1   Point* intersectsAt(const Line &l1, const Line &l2) {
2     double x, y;
3
4     if (areParallel(l1, l2)) {
5       return NULL;
6     }
7     x = ((l1.getB() * l1.getC()) - (l1.getB() * l2.getC())) /
8         ((l2.getA() * l1.getB()) - (l1.getA() * l2.getB()));
9     if (fabs(l1.getB()) < EPSILON) {
10      y = -((l1.getA() * x) + l1.getC());
11    } else {
12      y = -((l1.getA() * x) + l2.getC());
13    }
14    return new Point(x, y);
15  }
```

Además, podemos determinar el ángulo en que se intersecan. Dadas dos línea, $l_1 : a_1x + b_1y + c_1 = 0$ y $l_2 : a_2x + b_2y + c_2 = 0$.

$$\tan(\theta) = \frac{a_1b_2 - a_2b_1}{a_1a_2 + b_1b_2}$$

```
1   double intersectionAngle(const Line &l1, const Line &l2) {
2     return atan ( (l1.getA() * l2.getB()) - (l2.getA() * l1.getB())
3                 / (l1.getA() * l2.getA()) - (l1.getB() * l2.getB()));
4   }
```

Por último, un problema muy útil cuando manejamos líneas y puntos es identificar el punto de una línea que está más cerca de un punto p determinado. Este punto más cercano se encuentra en la línea que pasa por p, que es perpendicular a l y, por lo tanto, se puede encontrar usando las funciones que desarrollamos anteriormente.

```
1   Point* closestPoint(const Point &p, const Line &l) {
2     if (fabs(l.getB()) <= EPSILON) {
3       return new Point(-l.getC(), p.getY());
4     }
5
6     if (fabs(l.getA()) <= EPSILON) {
7       return new Point(p.getX(), -l.getC());
8     }
9
10    Line aux = pointAndSlopeToLine(p, 1 / l.getA());
11    return intersectsAt(l, aux);
12  }
```

Un segmento de línea, s, es la porción de una línea l que se encuentra entre dos puntos datos inclusive. Por lo tanto, los segmentos de línea se representan fácilmente por un par de puntos:

```
1   class Segment {
2   private:
3     Point p1, p2;
4
5   public:
6       Segment();
7       Segment(const Point&, const Point&);
8       Segment(const Segment&);
9
10      Point getP1() const;
11      Point getP2() const;
12  };
```

La primitiva geométrica más importante de los segmentos, que determina si un par determinado de ellos se intersecan, resulta ser sorprendentemente complicada debido a los casos especiales que surgen. Dos segmentos pueden estar en línea paralelas, lo que significa que no se cruzan en absoluto. Un segmento puede intersecarse en el punto final de otro, o los dos segmentos pueden estar uno encima del otro así que se intersecan en todo el segmento y no solo en un punto.

```
1   bool pointInBox(const Point &p, const Point &b1, const Point &b2) {
2     return ( (p.getX() >= std::min(b1.getX(), b2.getX())) &&
3              (p.getX() <= std::max(b1.getX(), b2.getX())) &&
4              (p.getY() >= std::min(b1.getY(), b2.getY())) &&
5              (p.getY() <= std::max(b1.getY(), b2.getY())) );
6   }
7
8   bool intersectAt(const Segment &s1, const Segment &s2) {
9     Line l1 = pointsToLine(s1.getP1(), s1.getP2());
10    Line l2 = pointsToLine(s2.getP2(), s2.getP2());
11    bool result;
12
13    if (areParallel(l1, l2)) {
14      return false;
15    }
16
17    if (areSameLine(l1, l2)) {
18      return ( pointInBox(s1.getP1(), s2.getP1(), s2.getP2()) ||
19               pointInBox(s1.getP2(), s2.getP1(), s2.getP2()) ||
20               pointInBox(s2.getP1(), s1.getP1(), s1.getP2()) ||
21               pointInBox(s2.getP2(), s1.getP1(), s1.getP2()) );
22    }
23
24    Point *p = intersectAt(l1, l2);
25    result = (pointInBox(*p, s1.getP1(), s1.getP2()) &&
26              pointInBox(*p, s2.getP1(), s2.getP2()));
27    delete p;
28    return result;
29  }
```

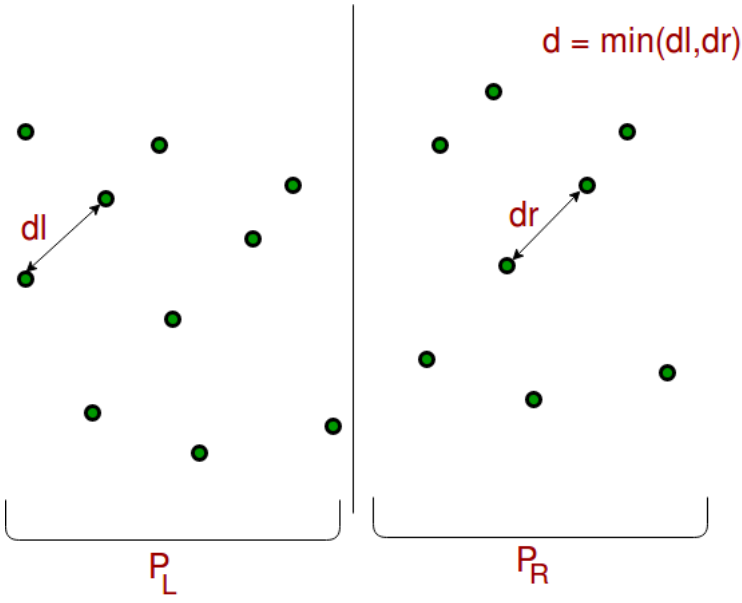
Dado un arreglo de N puntos en un plano, el problema es encontrar el par de puntos más cercano del arreglo. Este problema lo podemos encontrar en una serie de aplicaciones. Por ejemplo, en el control de tráfico áreo, se requiere monitorear los aviones que se acercan demasiado, ya que esto puede indicar una posible colisión.
La solución a fuerza bruta, de $O(n^2)$, calcula la distancia existente ente cada par de puntos y devuelve el más cercano. Podemos calcuolar la distancia más cercana en $O(n \log n)$ utilizando la estrategia de Dividir y Conquistar. [1]

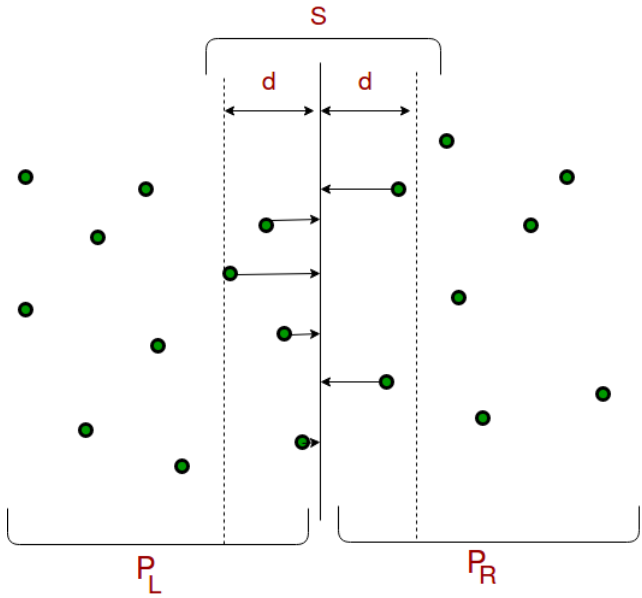
[1] https://goo.gl/36zwaA

Procedure 1 CLOSEST_PAIR

Input: $A : Array < Point >$

1. Ordenamos todos los puntos según sus coordenadas x.
2. Divide todos los puntos en dos mitades.
3. Encuentra, recursivamente, las distancias más pequeñas en ambas mitades.
4. Tomamos el mínimo de ambas distancias, d.
5. Crea un arreglo que almacene todos los puntos que se encuentran a una distancia máxima de la línea media que divide ambas mitades.
6. Encuentramos la distancia más corta de este subconjunto.
7. Devolvemos el mínimo, d, y la distancia más pequeña calculada en 4.


d = min(dl,dr)
dl
dr
P_L
P_R


S
d
d
P_L
P_R

Los polígonos son la estructura básica de para describir formas en un plano. Un polígono es solo una colección de segmentos de línea que forman un ciclo y no se cruzan entre sí. Cuando decimos que forman un ciclo, nos referimos a que el primer vértice de la secuencia es el mismo que el último. En lugar de representar un polígono como una colección de segmentos, la forma estándar es enumerar los vértices de un polígono en sentido de las agujas del reloj o en el sentido contrario a las agujas del reloj, siendo el primer vértice igual al último. De esta manera, existe un segmento entre el i-ésimo y $(i + 1)$-ésimo punto de la cadena para $0 \leq i \leq n$.

```
1   class Polygon {
2   private:
3     std::vector<Point> points;
4
5   public:
6     Polygon(int n, const Point &startPoint);
7     Polygon(int n, int xs[], int ys[]);
8     Polygon(const Polygon&);
9
10    std::vector<Point> getPoints() const;
11  };
```

Los polígonos pueden ser convexos o cóncavos. Decimos que es un polígono es convexo, si cualquier segmento de línea definido por dos puntos dentro P se encuentra complemente dentro de P; es decir, no hay muescas ni protuberancias de modo que si el segmento, al alargarlo, no interseca un borde de este. Esto implica que todos los ángulos internos de un polígono convexo deben ser agudos; es decir, miden cómo máximo 180 grados o π radianes (Figura 1). Todo polígono que no es convexo se denomina cóncavo (Figura 2).

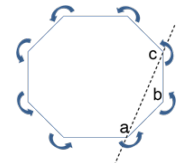


Figura 1: Polígono convexo

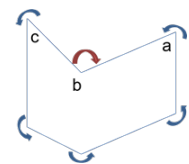


Figura 2: Polígono cóncavo

```cpp
double perimeter(const Polygon &p) {
  std::vector<Point> points = p.getPoints();
  double result = 0.0;

  for (int i = 0; i < points.size() - 1; i++) {
    result += dist(points[i], points[i + 1]);
  }
  return result;
}
```

El área con signo de un polígono (convexo o cóncavo) con n vértices dados en algún orden (en sentido horario o antihorario) se puede determinar calculando el determinando de la matriz que se muestra a continuación.

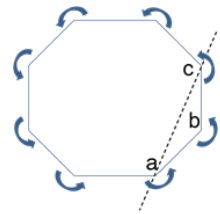
$$A = \frac{1}{2} \times \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \cdots & \cdots \\ x_{n-1} & y_{n-1} \end{bmatrix}$$

Esta fórmula puede ser fácilmente escrita así:

$$A = \frac{1}{2} \times$$
$$(x_0 \times y_1 + x_1 \times y_2 + \cdots + x_{n-1} \times y_0 - x_1 \times y_0 - x_2 \times y_1 - \cdots - x_0 \times y_{n-1})$$

```
 1   double area(const Polygon &p) {
 2     std::vector<Point> points = p.getPoints();
 3     double x1, x2, y1, y2, result = 0;
 4
 5     for (int i = 0 ; i < points.size() - 1; i++) {
 6       x1 = points[i].getX();
 7       x2 = points[i + 1].getX();
 8       y1 = points[i].getY();
 9       y2 = points[i + 1].getY();
10       result += (x1 * y2 - x2 * y1);
11     }
12     return (fabs(result) / 2.0);
13   }
```

- Como mencionamos anteriormente, un polígono es convexo si todos los ángulos internos son agudos. Es decir, cualesquiera tres vértices del polígono, a, b, y c, deben formar un ángulo agudo. Para esto, debemos conocer si, al viajar del primer punto, a, al segundo, b, y al tercero, c, giramos en sentido antihorario. Si es así, el ángulo formado entre a y c gira en sentido antihorario sobre b y por lo tanto es agudo. De lo contrario, el punto se encuentra a la izquierda de esa línea o los tres puntos son colineales.
- Lo anterior se puede resolver calculando el área con signo del triángulo formado por los tres vértices. Un área negativa resulta si el punto c está a la izquierda de la línea que pasa por los puntos a y b. Cero indica que los tres puntos son colineales. Y un valor positivo muestra que c está a la derecha.



```
1   double signedTriangleArea(const Point &a, const Point &b,
2                             const Point &c) {
3     return ((a.getX() * b.getY()) - (a.getY() * b.getX()) +
4             (a.getY() * c.getX()) - (a.getX() * c.getY()) +
5             (b.getX() * c.getY()) - (c.getX() * b.getY())) / 2.0;
6   }
7
8   bool ccw(const Point &a, const Point &b, const Point &c) {
9     return (signedTriangleArea(a, b, c) > EPSILON);
10  }
11
12  bool cw(const Point &a, const Point &b, const Point &c) {
13    return (signedTriangleArea(a, b, c) < EPSILON);
14  }
15
16  bool collinear(const Point &a, const Point &b, const Point &c) {
17    return (fabs(signedTriangleArea(a, b, c)) <= EPSILON);
18  }
```

```
1   bool isConvex(const Polygon &p) {
2     std::vector<Point> points = p.getPoints();
3     bool isLeft;
4     int p3;
5
6     if (points.size() <= 3) return false;
7
8     isLeft = ccw(points[0], points[1], points[2]);
9     for (int i = 1; i < points.size() - 1; i++) {
10      p3 = ((i + 2) == points.size())? 1 : (i + 2);
11      if (ccw(points[i], points[i + 1], points[p3]) != isLeft) {
12        return false;
13      }
14    }
15    return true;
16  }
```

Otra prueba común que se realiza en un polígono es verificar si un punto p está dentro o fuera del polígono. El algoritmo calcula la suma de ángulos entres tres puntos, $P[i]$, p, $P[i+1]$, donde $P[i]$ y $P[i+1]$ son lados consecutivos del polígono P.

```
 1   bool inPolygon(const Point &p, const Polygon &P) {
 2     std::vector<Point> points = P.getPoints();
 3     double sum;
 4
 5     if (points.size() == 0) return false;
 6
 7     sum = 0;
 8     for (int i = 0; i < points.size() - 1; i++) {
 9       if (ccw(p, points[i], points[i + 1])) {
10         sum += angle(points[i], p, points[i + 1]);
11       } else {
12         sum -= angle(points[i], p, points[i + 1]);
13       }
14
15       return (fabs(fabs(sum) - 2*PI) < EPSILON);
16     }
```

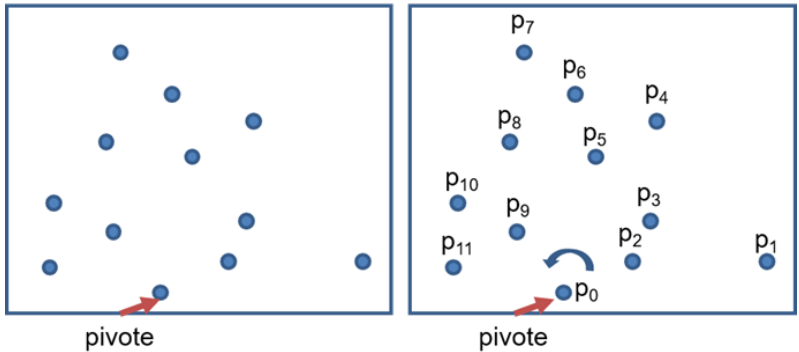
Cascos convexos (Convex hull)

Un casco convexo es una estructura muy importante en geometría, ya que puede ser utilizada en la construcción de muchas otras estructuras geométricas. El casco convexo de un conjunto S de puntos en el plano se define como el polígono convexo más pequeño que contiene todos los puntos de S. Los vértices del casco convexo de un conjunto S de puntos forman un subconjunto (no necesariamente propio) de S.

Un algoritmo muy simple consistiría en obtener los puntos extremos de un conjunto S de puntos en el plano. Para comprobar lo anterior, habría que observar cada posible tripleta de puntos y ver si un p se encuentra en el triángulo formados por estos tres puntos. Si p se encuentra en cualquiera de estos triángulos, no es un punto extremo; de lo contrario lo es. La prueba para determinar si p se encuentra es un triángulo dado se puede hacer en un tiempo $O(1)$. Dado que hay n3 posibles triángulos, se necesitaría un tiempo $O(n^3)$ para determinar si un punto p dado es extremo o no. Y, puesto que hay n puntos, este algoritmo se ejecutaría en un tiempo total de $O(n^4)$. Sin embargo, usando la técnica de Divide y Conquista, podemos resolver el problema del casco convexo en un tiempo $O(nlogn)$.

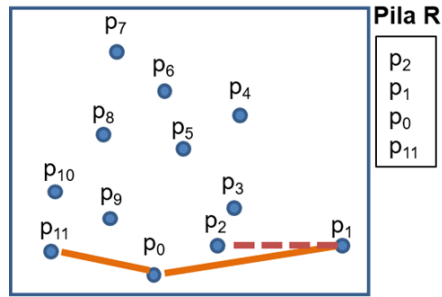
Existen dos algoritmos para que calcular el caso convexo de un conjunto de n puntos. Ambos algoritmos generan los vértices del casco convexo en orden antihorario. El primero, conocido como exploración de Graham, se ejecuta en tiempo $O(nlogn)$. El segundo, llamado marcha de Jarvis, se ejecuta en tiempo $O(nh)$, donde h es el número de vértices del casco convexo.

El algoritmo de exploración de Graham primero ordena todos los n puntos del S basándose en los ángulos que forman con respecto a un punto llamado pivote. Un posible pivote, que es el que usaremos en este ejemplo, es el punto más inferior y más a la derecha posible.

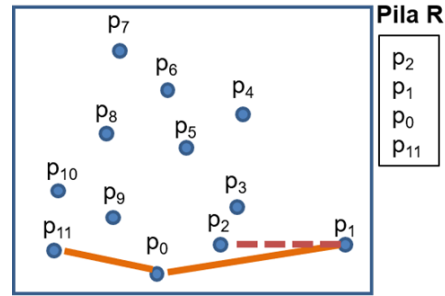


Primero, ordenaremos con respecto al ángulo que forman con el pivote. Una vez ordenados, el algoritmo usa una pila, R, para mantener los puntos candidatos. Cada punto de S se colocar en el tope de la pila R y los puntos que no van a ser parte de $CH(S)$ eventualmente son extraídos de R. El algoritmo de exploración de Graham mantiene esta invariante: los tres elementos superiores de la pila R siempre deben girar a la izquierda (que, cómo recordarás, es una propiedad básica de un polígono convexo).

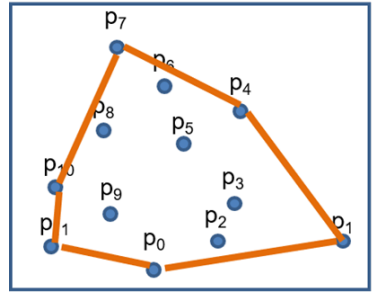
Inicialmente, insertaremos estos tres puntos: p_{n-1}, p_0 y p_1. Esto debido a que el ordenamiento nos asegura que estos tres puntos siempre forman un giro a la izquierda. A continuación, intentaremos insertar el punto p_2 y, dado que p_0-p_1-p_2 forman un giro a la izquierda, lo colocamos en la pila R. Ahora, R contiene, de arriba abajo, los puntos p_{11}-p_0-p_1-p_2.



A continuación, intentamos insertar el punto 3. Sin embargo, p_1-p_2-p_3 (recorremos en ese orden) forman un giro a la derecha. Esto significa que, si aceptamos el punto antes de p_3, que es p_2, no tendremos un polígono convexo. Así que tenemos que sacar p_2 de la pila. La pila R ahora contiene, de arriba abajo, p_{11}-p_0-p_1. Una vez hecho lo anterior, volveremos a intentar colocar p_3. Ahora p_0-p_1-p_3 forman un giro a la izquierda, por lo que colocamos p_3 en la pila. La pila R es ahora, de arriba abajo, p_{11}-p_0-p_1-p_3.



Repetiremos este proceso hasta que hayamos procesado todos los vértices. Cuando terminado el algoritmo de exploración de Graham, lo que queda en la pila R son los puntos de $CH(S)$. El algoritmo elimina todos los giros a la derecha, por lo que tenemos un polígono convexo.



Procedure 2 FIND_PIVOT

Input: P : *Array*

```
P0 ← 1
for i ← 2 to P.size() do
  if P[i].y < P[P0].y||(P[i].y == P[P0].y and P[i].x > P[P0].x) then
    P0 ← i
  end if
end for
return P0
```

Procedure 3 GRAHAM

Input: $P : Array$
  $S : Stack$
  if $P.size() <= 3$ **then**
    return P
  end if
  $pivot = FIND\_PIVOT(P)$
  $SORT(P, byAngleWithPivot)$
  $S.push(P[P.size()])$, $S.push(P[1])$, $S.push(P[2])$
  $i \leftarrow 2$
  while $i < P.size()$ **do**
    if $ccw(S[top-1], S[top], P[i])$ **then**
      $S.push(P[i])$, $i \leftarrow i + 1$
    else
      $S.pop()$
    end if
  end while
  return S

Cuando hablamos de puntos en un plano, es natural preguntarse cuál de esos puntos se encuentra dentro de un área específica. Por ejemplo, ¿cuáles puntos de interés hay dentro un área de un kilómetro del punto en que estoy? Si bien, el problema hace referencia a una figura geométrica (en este caso un círculo de 1 km de radio), esto se puede extender a problemas no geométricos. Por ejemplo, el problema "mostrar a todas aquellas personas entre 21 y 35 años con ingresos entre $25,000 a $100,00" hace referencia a qué "punto" de una base de datos de personas se encuentra dentro de un área determinada de edad-ingresos.

El problema fácilmente se puede ampliar a más de dos dimensiones. Por ejemplo, enumerar todas las estrellas a 50 años luz del sol. En este caso, tenemos un problema tridimensional. Y, aplicado al ejemplo de la base de datos, si no solo queremos a los jóvenes con buenos ingresos, sino que también queremos sean altos y del sexo femenino, estamos hablando de un problema de cuatro dimensiones.

En general, si asumimos que tenemos un conjunto de registros con ciertos atributos que toman valores de algún conjunto ordenado, es decir, una base de datos. Encontrar todos los registros de esta base de datos que satisfagan las restricciones de rango especificadas en un conjunto específico de atributos hablamos de un problema llamado búsqueda de rango. Es una tarea difícil y un problema que, como podrás observar, tiene muchas aplicaciones prácticas.

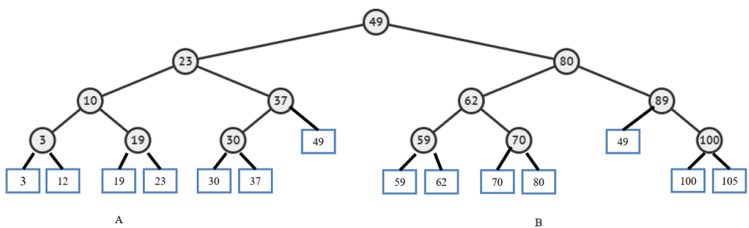
Revisaremos los árboles de rango (range trees), que pueden almacenar puntos multidimensionales para admitir un tipo de consulta llamada consulta de búsqueda de rango. Finalmente, analizaremos una estructura de datos llamada árboles de partición, que dividen el espacio en celdas, en particular nos enfocaremos en los árboles kd (kd-tree). Estas estructuras se utilizan, por ejemplo, en gráficos de computadora, donde necesitamos encontrar vecinos más cercanos a un punto de consulta o trazar la trayectoria de un rayo a través de un entorno digital.

Como comentamos antes, una operación de consulta natural para realizar en un conjunto de puntos multidimensionales es una consulta de búsqueda de rango: "enumerar a todas aquellas personas de sexo femenino entre 21 y 35 años con ingresos entre $25,000 a $100,000 que midan más de 1.70". Pues bien, ahora revisaremos la estructura de datos de árbol de rangos que se puede utilizar para responder tales consultas.

Empecemos con lo básico, consultas de búsqueda en rangos unidimensionales. Tenemos un conjunto de puntos en una sola dimensión, en decir, un conjunto de números reales. Una consulta podría ser: ¿cuáles son los puntos (o valores) que se encuentran en el intervalo $[x_1, x_2]$.

- Entonces, sea $P = p_1, p_2, \cdots, p_n$ el conjunto de puntos, podemos resolver el problema de búsqueda de rango unidimensional de una manera eficiente utilizando una estructura de datos conocida como árbol binario de búsqueda equilibrado T. Las hojas del árbol T almacenan los puntos de P y los nodos internos almacenan valores de división, X_{div}, para guiar la búsqueda. Consideraremos que el subárbol izquierdo de un nodo *div* contiene todos los puntos menores o iguales que X_{div} y que el subárbol derecho contiene todos los puntos estrictamente mayores que X_{div}.

Supongamos que queremos encontrar los puntos que se encuentran en el rango $[X_{low}, X_{high}]$. Sea A y B, las dos hojas donde se encuentran los límites de la búsqueda. Entonces los puntos que se encuentran en el intervalo $[X_{low}, X_{high}]$ son los que se almacenan entre las hojas A y B más, posiblemente, el punto almacenado en A y el punto almacenado en B. Por ejemplo, vamos la búsqueda $[18, 78]$. ¿Cómo podemos encontrar las hojas entre A y B?



Procedure 4 FIND_SPLIT_NODE

Input: T : *RangeTree*, *low*, *high* : *Index*

```
V ← root(T)
while V is not a leaf and high ≥ X_V or low ≤ X_V do
  if high ≥ X_V then
    V ← LEFT(V)
  else
    V ← RIGHT(V)
  end if
end while
```

Procedure 5 REPORT_SUB_TREE

Input: T : *RangeTree*, *low*, *high* : *Index*

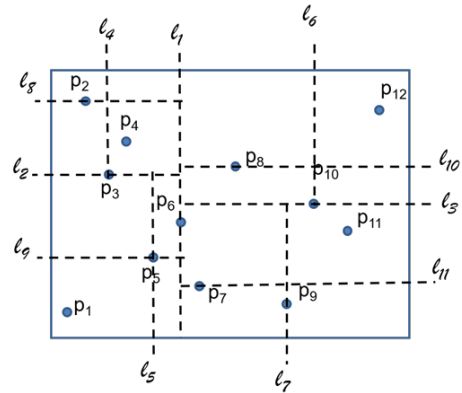
```
V_split ← FIND_SPLIT_NODE(T, low, high)
if V_split is a leaf then
    Check if the point stored in Vsplit should be reported.
else
    V ← LEFT(V_split)
    while V is not a leaf do
        if low ≤ X_V then
            REPORT_SUB_TREE(RIGHT(V))
            V ← LEFT(V)
        else
            V ← RIGHT(V)
        end if
    end while
    Check if the point stored in V should be reported.
end if
```

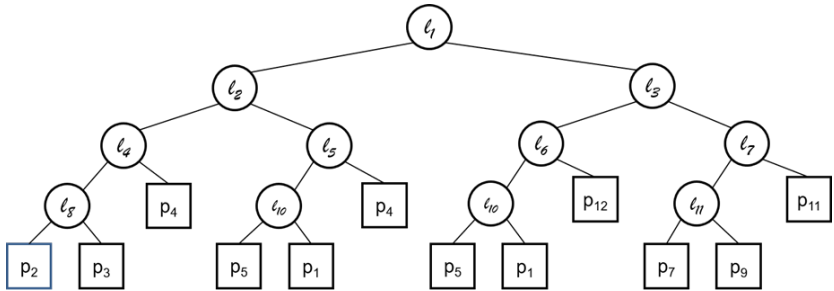
- Ahora, revisaremos el desempeño de la estructura. Debido a que hablamos de un árbol binario de búsqueda balanceado, agregar un elemento es $O(n \log n)$. En lo que respecta a realizar una consulta, el peor caso sería una consulta que incluyera todos los puntos almacenados. En este caso, el tiempo sería $O(n)$, lo que no es realmente malo, sobre todo si tomamos en cuenta que comparamos que comparar todos los elementos nos daría un resultado similar. Por otro lado, no se puede evitar un tiempo de consulta $O(n)$ cuando tenemos que revisar todo el conjunto de puntos.
- Al punto anterior, habría que agregar la llamada REPORT_SUB_TREE que es línea los k puntos reportados. Por tanto, el tiempo total empleado en todas esas llamadas es $O(k)$. Dado a que el árbol está balanceado, el recorrido tiene una longitud $O(\log n)$. El tiempo que tomamos en revisar un nodo es $O(1)$, por lo que el tiempo total invertido en el recorrido de la consulta sería $O(\log n + k)$.

Sea P un conjunto de n puntos en el plano. Para simplificar un poco el análisis, consideráremos que no hay dos puntos en P que tenga la misma coordenada en X y en Y.

Una consulta de rango, en este caso, bidimensional en P solicita los puntos de P que se encuentran dentro de un rectángulo de consulta $[x_{low}, x_{high}]$ x $[y_{low}, y_{high}]$. Un punto (p_x, p_y) se encuentra dentro de este rectángulo si y solo si $p_x \in [x_{low}, x_{high}]$ y $p_y \in [y_{low}, y_{high}]$. Siendo así, podríamos decir que una consulta de rango bidimensional se compone de dos subconsultas unidimensionales, una para la coordenada X y otra para la coordenada Y.
¿Podemos generalizar la estructura vista en la sección anterior, el árbol de búsqueda binaria unidimensional, para realizar consultas bidimensionales? Bueno, revisemos, nuevamente, la definición del árbol de búsqueda binaria de la sección anterior: el conjunto de puntos unidimensionales se divide en dos subconjuntos de aproximadamente el mismo tamaño; un subconjunto contiene los puntos menores o iguales al valor de división, en el otro subconjunto están los puntos mayores al valor de división. El valor de división se almacena de forma recursiva en los dos subárboles.

Ahora, en el caso bidimensional, cada punto tiene dos valores que son su coordenada en X y su coordenada en Y. Por lo tanto, lo que haremos será dividir primero por la coordenada X, luego por la coordenada Y, nuevamente por la coordenada X, y así sucesivamente.




ℓ_1
ℓ_2
ℓ_3
ℓ_4
ℓ_5
ℓ_6
ℓ_7
ℓ_8
p_4
ℓ_{10}
p_4
ℓ_{10}
p_{12}
ℓ_{11}
p_{11}
p_2
p_3
p_5
p_1
p_5
p_1
p_7
p_9

Procedure 6 BUILD_KD_TREE

Input: *points* : *Array*, *currentDepth* : *Integer*

if *points* contains a simple point **then**

return a leaf with the point

else

if *currentDepth* is odd **then**

Separate *point* into two subsets with a vertical line l through the median of the x-coordinate of the points in *points*. Let P_1 be the set to the left of or above it, and let P_2 be the subset to the right of it.

else

Separate *points* into two subsets with a vertical line through the median of the y-coordinate of the points at *points*. Let P_1 be the set below or above it, and let P_2 be the subset above it.

end if

$V_{left} \leftarrow BUILD\_KD\_TREE(P_1, currentDepth + 1)$

$V_{right} \leftarrow BUILD\_KD\_TREE(P_2, currentDepth + 1)$

return a node V containing l, make V_{left} the left of V and V_{right} the right child.

end if

Procedure 7 SEARCH_KD_TREE

Input: $V : KdTree, R : Range$

```
if V is a leaf then
    Reports the point stored in V if it is in region R.
else
    if region(LEFT(V)) is in region R then
        REPORT_SUB_TREE(LEFT(V))
    else
        if region(LEFT(V)) intersects at R then
            SEARCH_KD_TREE(LEFT(V))
        end if
    end if
    if region(RIGHT(V)) is in region R then
        REPORT_SUB_TREE(RIGHT(V))
    else
        if region(RIGHT(V)) intersects at R then
            SEARCH_KD_TREE(RIGHT(V))
        end if
    end if
end if
```