La magia de la programación competitiva

Comunidad new liberty

4 de julio de 2017

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

matemáticas

1.1. sucesiones y series

1.1.1. sucesión aritmética

las sucesiones aritméticas son aquellas que restando un elemento con su antecesor siempre da una constante se representan de la siguiente manera.
$an + b$
donde a es la resta entre dos elementos consecutivos y b es el primer elemento

1.1.2. sucesión geométrica

las sucesiones geométricas son aquellas que el cociente de un elemento con su antecesor siempre da una constante se representan de la siguiente manera.
ar^{n-1}
donde a es el primer termino y r es el cociente entre un numero y su anterior

1.1.3. serie aritmética

una serie aritmética es una sucesión creada con la suma de los términos de una sucesión aritmética, su formula es:
$a\frac{n(n+1)}{2} + nb$

1.1.4. serie geométrica

una serie geométrica es una sucesión creada con la suma del los términos de una sucesión geométrica, su formula es:
$a\frac{1-r^n}{1-r}$

Capítulo 2

geometricos

2.1. formulas de geometría

- $\frac{a}{sin(A)} = \frac{b}{sin(B)} = \frac{c}{sin(C)}$

2.2. estructuras geométricas

2.2.1. puntos

punto de enteros

```
// struct point_i { int x, y; };
// basic raw form, minimalist mode
struct point_i { int x, y;
// whenever possible, work with point_i
point_i() { x = y = 0; }
// default constructor
point_i(int _x, int _y) : x(_x), y(_y) {} };
// user-defined
```

punto de reales

```
struct point { double x, y;
// only used if more precision is needed
point() { x = y = 0.0; }
// default constructor
point(double _x, double _y) : x(_x), y(_y) {} };
// user-defined
```

ordenamiento de puntos

```
struct point { double x, y;
point() { x = y = 0.0; }
point(double _x, double _y) : x(_x), y(_y) {}
bool operator < (point other) const { // override less than operator
if (fabs(x - other.x) > EPS)
// useful for sorting
        return x < other.x;
// first criteria , by x-coordinate
return y < other.y; } };
```

```
// second criteria, by y-coordinate
// in int main(), assuming we already have a populated vector<point> P
sort(P.begin(), P.end());
// comparison operator is defined above
```

saber si dos puntos son iguales

```
struct point { double x, y;
point() { x = y = 0.0; }
point(double _x, double _y) : x(_x), y(_y) {}
// use EPS (1e-9) when testing equality of two floating points
bool operator == (point other) const {
return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };
```

distancia euclídea entre 2 puntos

```
double dist(point p1, point p2) {
// Euclidean distance
// hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
return hypot(p1.x - p2.x, p1.y - p2.y); }
// return double
```

rotar un punto con respecto al origen

```
// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
double rad = DEG_to_RAD(theta);
// multiply theta with PI / 180.0
return point(p.x * cos(rad) - p.y * sin(rad),
p.x * sin(rad) + p.y * cos(rad)); }
```

2.2.2. lineas

```
struct line { double a, b, c; };
// a way to represent a line
```

hallar una recta con 2 puntos

```
// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
if (fabs(p1.x - p2.x) < EPS) {
// vertical line is fine
l.a = 1.0;
l.b = 0.0;
l.c = -p1.x;
// default values
} else {
l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
l.b = 1.0;
```

```
12 // IMPORTANT: we fix the value of b to 1.0
13 l.c = -(double)(l.a * p1.x) - p1.y;
14 } }
```

saber si dos lineas son paralelas

```
1 bool areParallel(line l1, line l2) {
2 // check coefficients a & b
3 return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }
```

saber si 2 lineas son iguales

```
1 bool areSame(line l1, line l2) {
2 // also check coefficient c
3 return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }
```

intersección entre 2 lineas

2.2.3. vectores

2.2.4. círculos

ángulos en una circunferencia

2.2.5. triángulos

2.2.6. vectores

2.3. representación de un polígono

2.4. perímetro de un polígono

2.5. area de un polígono

2.6. comprobar si un punto esta dentro de un polígono

2.7. comprobar que un polígono es convexo

2.8. cortar un polígono con una linea recta

2.9. cubierta convexa