Geometria Computacional (discreta)

Notas de aula da disciplina
TE: Técnicas de Construção de Algoritmos

Fabiano de Souza Oliveira
(fabiano.oliveira@ime.uerj.br)

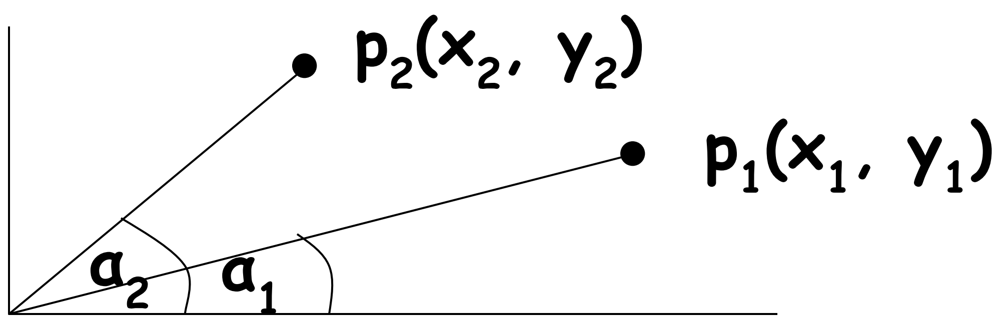
Paulo Eustáquio Duarte Pinto
(pauloedp@ime.uerj.br)

outubro/2020

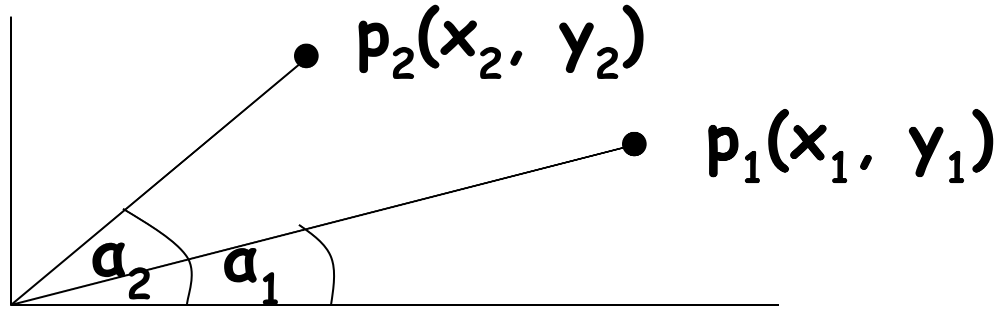
Geometria computacional é um ramo novo, pois não se aplicam resultados da geometria tradicional.

A Geometria computacional é baseada na matemática discreta. Trabalha-se fortemente com inteiros. Usa-se +, -, *, mas procura-se evitar problemas de arredondamento.

Exemplo: qual o maior ângulo com a origem, α_1 ou α_2 ?

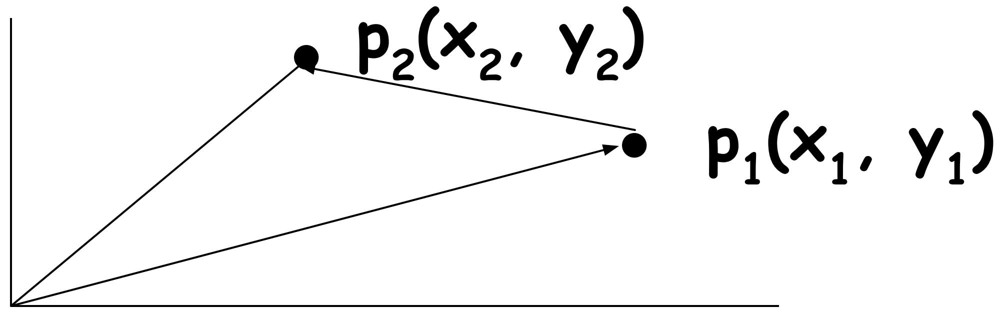


Exemplo: qual o maior ângulo com a origem, α_1 ou α_2 ?



Solução tradicional: calcula-se $\tan^{-1}(y/x)$.

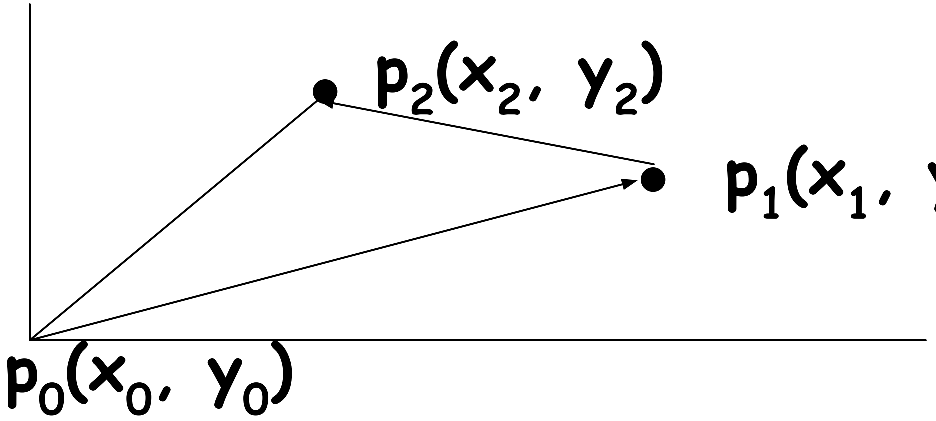
Geometria computacional: calcula-se o sentido do percurso $(0, 0) \rightarrow p_1 \rightarrow p_2$



anti-horário: $\alpha_1 < \alpha_2$
horário: $\alpha_1 > \alpha_2$
colinear: $\alpha_1 = \alpha_2$

Sentido do percurso:

Dados p_0, p_1 e p_2, qual o sentido do percurso
$p_0 \rightarrow p_1 \rightarrow p_2$?

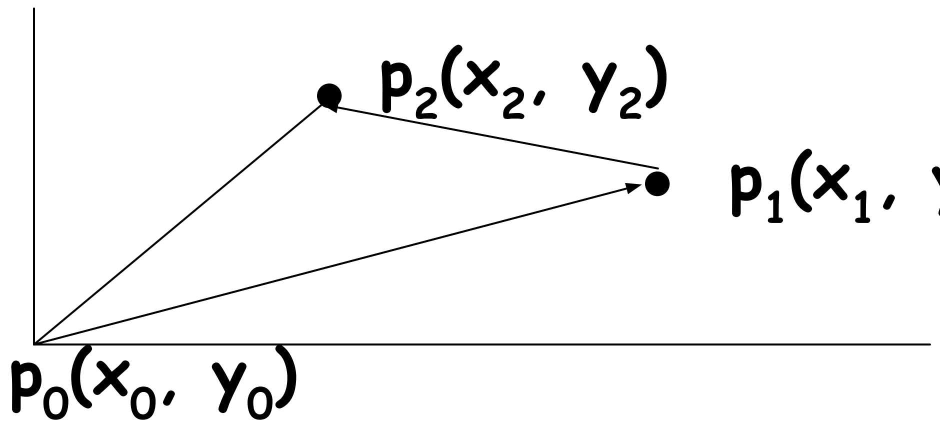


Calcula-se $s = (x_1-x_0)(y_2-y_0) - (x_2-x_0)(y_1-y_0)$

$s > 0 \rightarrow$ anti-horário
$s < 0 \rightarrow$ horário
$s = 0 \rightarrow$ colinear

Sentido do percurso:

Dados p_0, p_1 e p_2, qual o sentido do percurso $p_0 \rightarrow p_1 \rightarrow p_2$?



inteiro SentidoPercurso (ponto p_0, p_1, p_2):

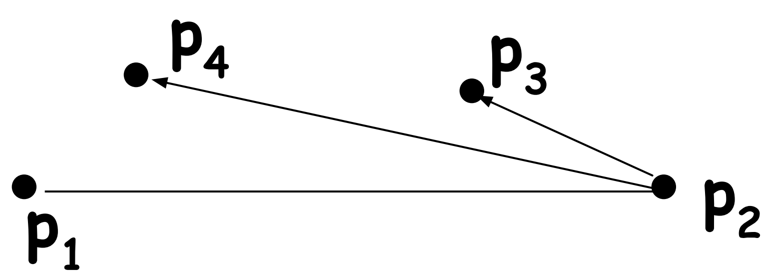
```
s ← (x1-x0)*(y2-y0) - (x2-x0)*(y1-y0)
se s > 0 então
    retornar 1
senão se s < 0 então
    retornar -1
senão
    retornar 0
```

Intersecção de segmentos:

Os segmentos $s_1(p_1,p_2)$ e $s_2(p_3,p_4)$ se interceptam ?

Para verificar se s_2 intercepta s_1, fixamos s_1 e verificamos se o sentido do percurso $p_1 \rightarrow p_2 \rightarrow p_3$ é igual ou diferente do sentido do percurso $p_1 \rightarrow p_2 \rightarrow p_4$.

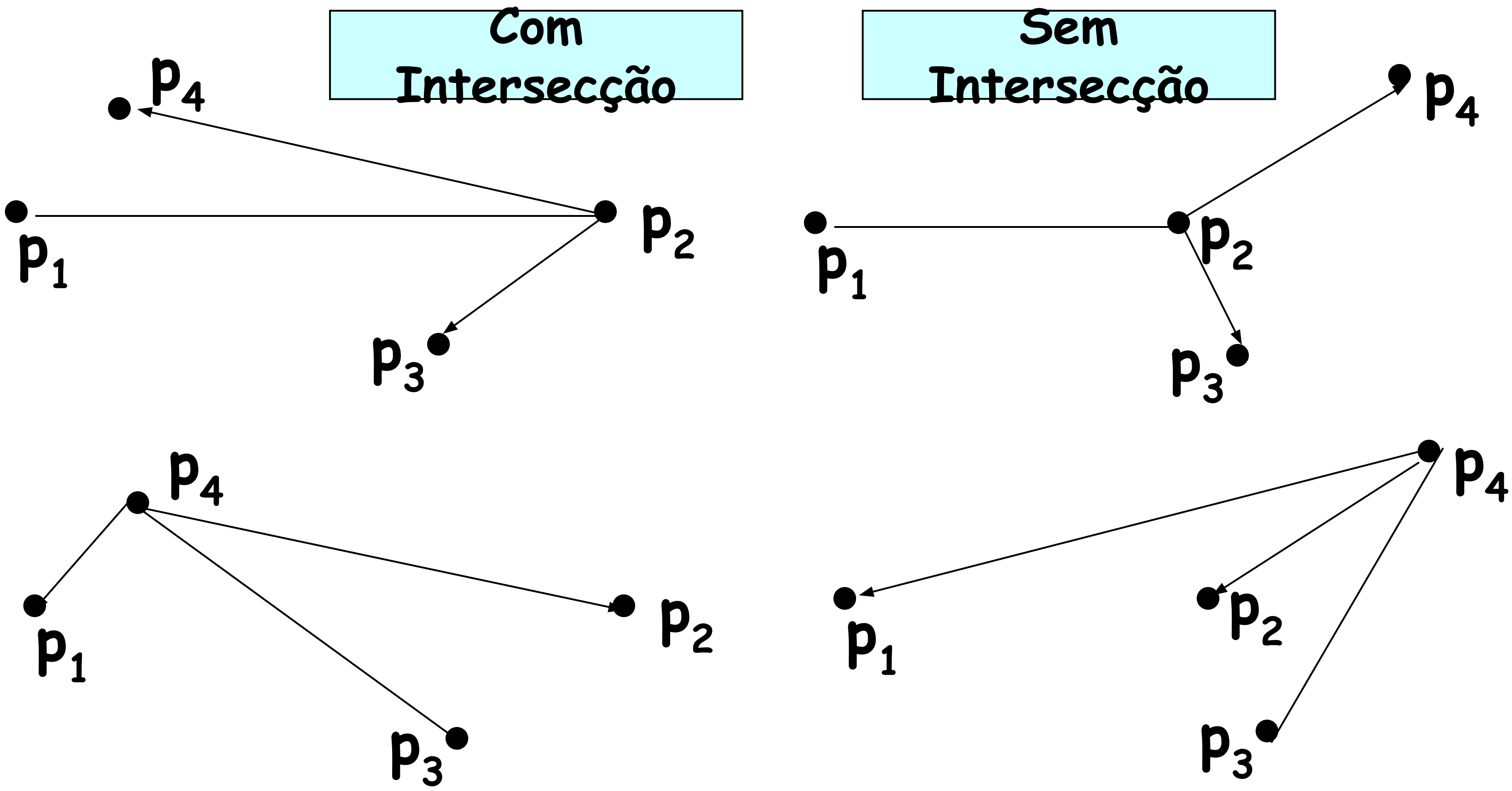
Se mesmo sentido e não colineares ⇒ não têm intersecção



Intersecção de segmentos:

Os segmentos $s_1(p_1,p_2)$ e $s_2(p_3,p_4)$ se interceptam ?

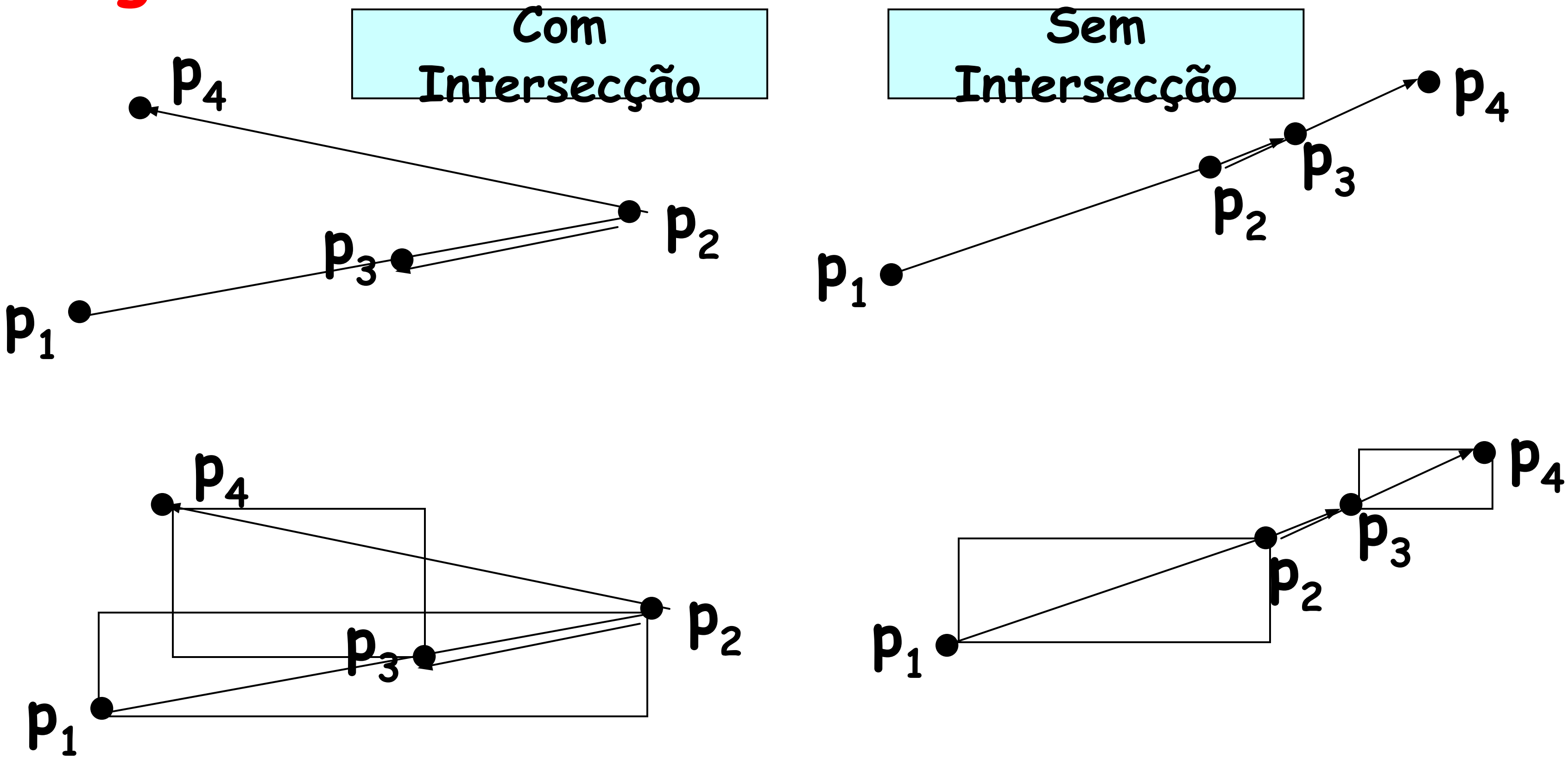
Se sentidos opostos: teste adicional, fixando s_2.



Intersecção de segmentos:

Os segmentos $s_1(p_1,p_2)$ e $s_2(p_3,p_4)$ se interceptam ?

Se alguma colinearidade: teste adicional dos retângulos.



Intersecção de segmentos:

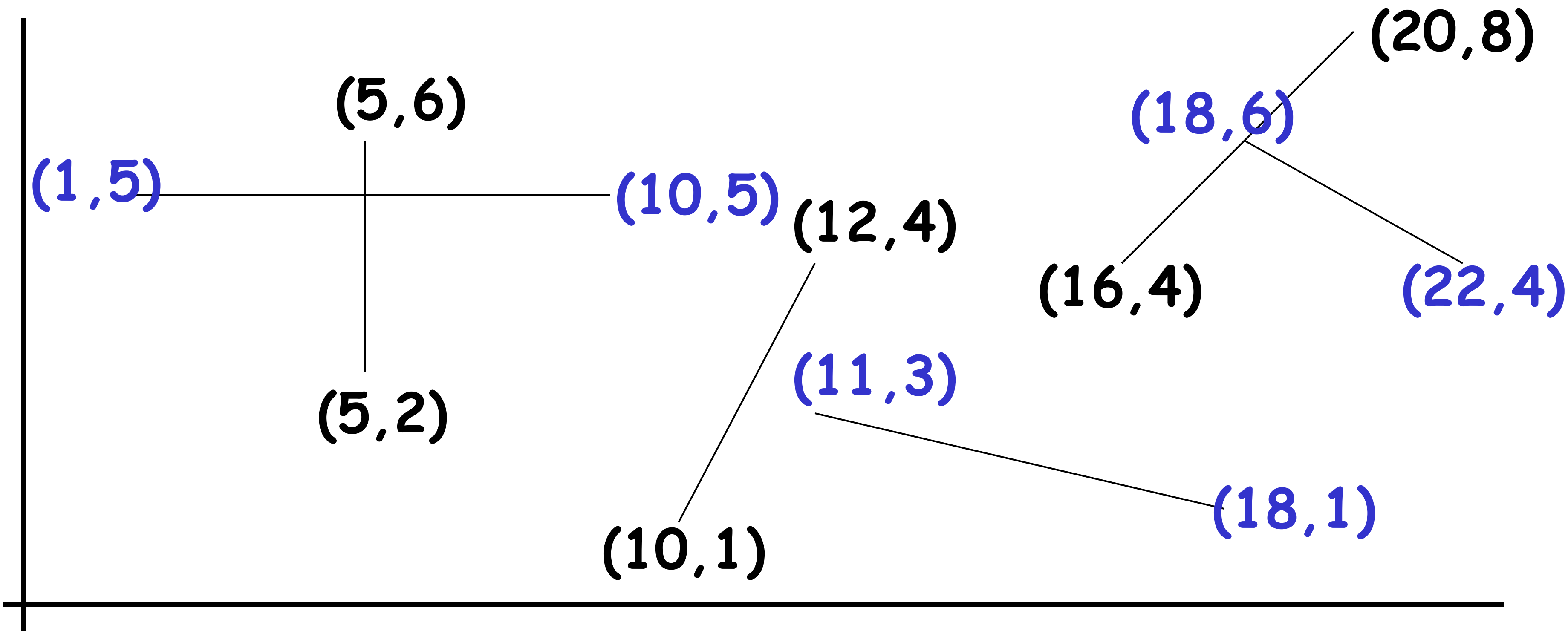
Os segmentos $s_1(p_1,p_2)$ e $s_2(p_3,p_4)$ se interceptam ?

lógico InterceptaSegmento(ponto p_1, p_2, p_3, p_4):
retornar
SentidoPercurso(p_1,p_2,p_3)*SentidoPercurso(p_1,p_2,p_4) $\leq$ 0 e
SentidoPercurso(p_3,p_4,p_1)*SentidoPercurso(p_3,p_4,p_2) $\leq$ 0 e
max(p_1.x,p_2.x) $\geq$ min(p_3.x,p_4.x) e max(p_3.x,p_4.x) $\geq$ min(p_1.x, p_2.x) e
max(p_1.y,p_2.y) $\geq$ min(p_3.y,p_4.y) e max(p_3.y,p_4.y) $\geq$ min(p_1.y_1, p_2.y_2)

Teste dos retângulos

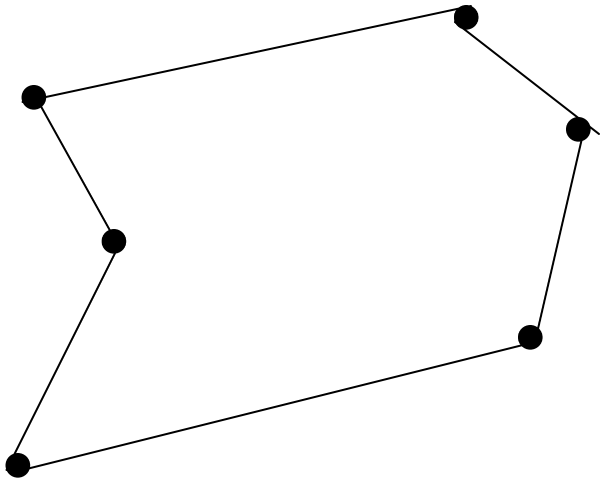
Intersecção de segmentos:

Exercício: fazer um programa que determina quais as intersecções existentes entre 6 segmentos.



Caminho fechado de pontos.

dados n pontos, definir um polígono que passe por todos os pontos, tal que não haja intersecção de lados.



Representação de polígono:

$P \langle p_0, p_1, \ldots p_n \rangle$, onde $p_n = p_0$.

Caminho fechado de pontos

CaminhoFechado(ponto S[], ref ponto P[], inteiro n):

P[1..n] ← S[1..n]

i ← índice do ponto de P mais à esquerda e, dentre aqueles mais à esquerda, daquele mais abaixo

P[0], P[n], P[i] ← S[i], S[i], S[n]

Ordenar(P[1..n-1]), tal que

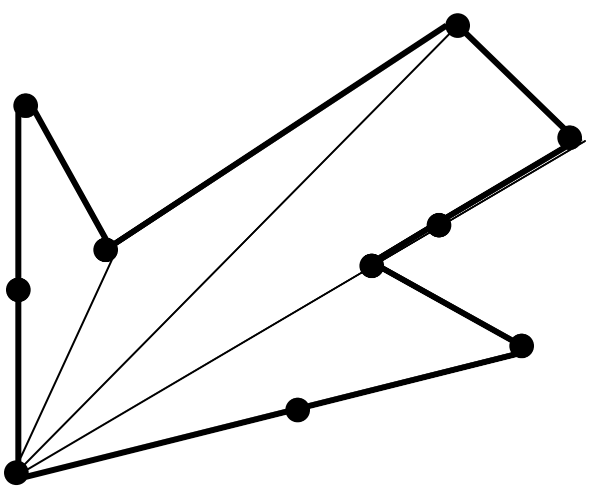
P[i] < P[j] ⇔ ang(i) < ang(j) ou (ang(i) = ang(j) e d(i) < d(j))

onde:

ang(x) = ângulo que o segumento P[x], P[0] faz com o eixo x

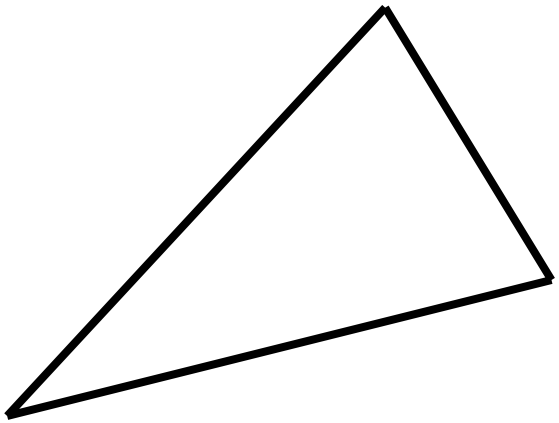
d(x) = distância entre P[x] e P[0]

Inverter a ordem dos pontos finais de P que são colineares com P[n],P[n-1].



Área (sinalizada) de triângulo.

dado um triângulo $T \langle p_1, p_2, p_3 \rangle$ a área (sinalizada) é dada (Álgebra linear) por:



$S = 1/2((x_1.y_2 - x_2.y_1)+(x_2.y_3 - x_3.y_2)+(x_3.y_1 - x_1.y_3))$

$1/2.\Sigma\ (x_i.y_{i+1} - x_{i+1}.y_i),\ 1 \leq i \leq 3$, onde $p_4 = p_1$

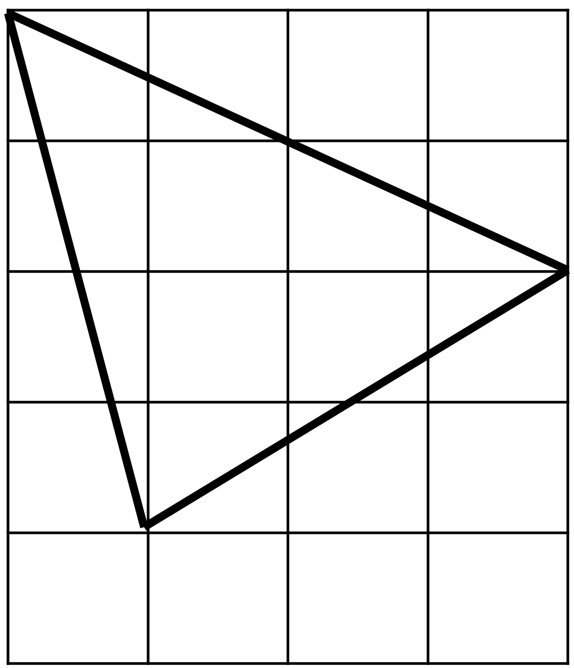
S é + se os pontos foram tomados em sentido anti-horário e

\- caso contrário.

Para obter a área não sinalizada toma-se A = fabs(S)

Área (sinalizada) de triângulo. Exemplo:

dado um triângulo $T \langle p_1, p_2, p_3 \rangle$ a área (sinalizada) é dada (Álgebra linear) por:



Se $p_1(1,1)$, $p_2(4,3)$ e $p_3(0,5)$,

$S = (1.3-4.1+4.5-0.3+0.1-1.5)/2$

$= 14/2 = 7$

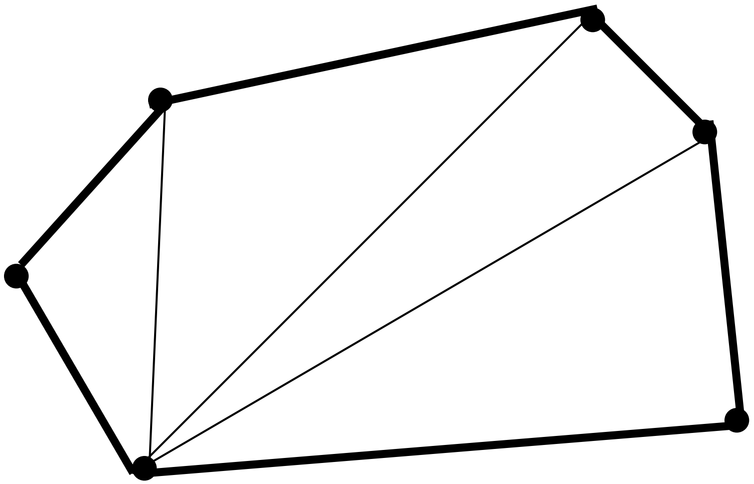
Se $p_1(1,1)$, $p_2(0,5)$ e $p_3(4,3)$,

$S = (1.5-0.1+0.3-4.5+4.1-3.1)/2$

$= -14/2 = -7$

Área de polígono.

dado um polígono, P $\langle p_0, p_1, \ldots p_n \rangle$, onde $p_n = p_0$, determinar sua área.

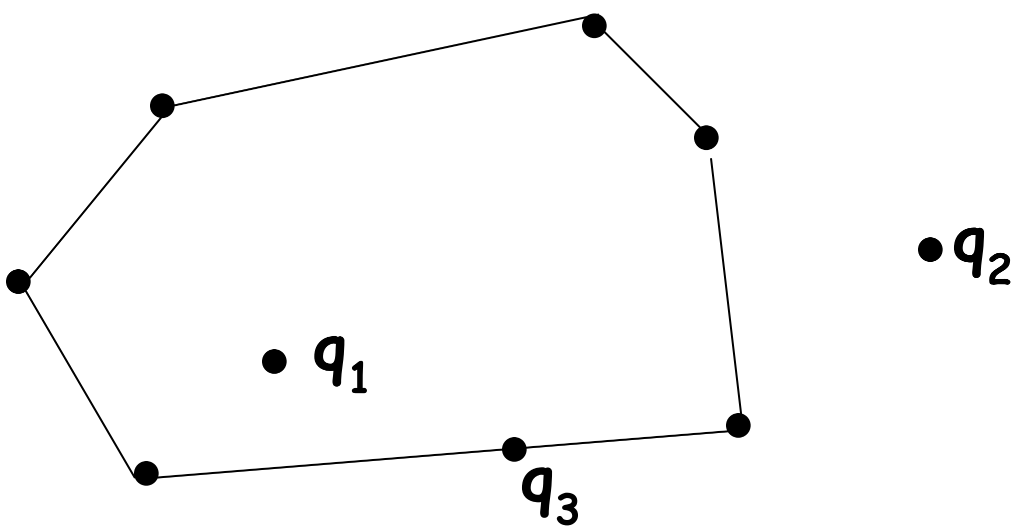


Área = $1/2.\Sigma\,(x_i.y_{i+1} - x_{i+1}.y_i)$, $0 \leq i \leq n-1$

(Generalização do cálculo da área de triângulo)

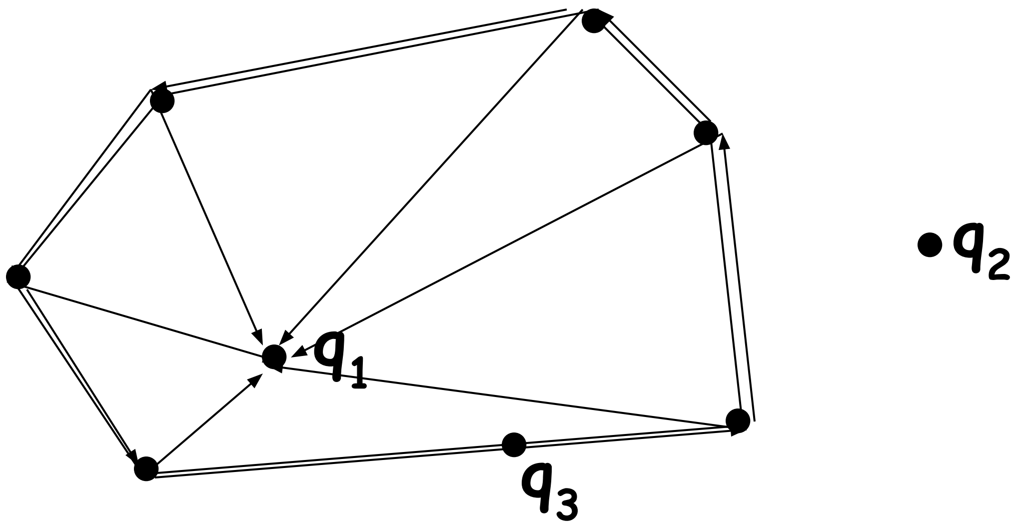
Ponto Interior a Polígono convexo.

dado um polígono convexo, $P <p_0, p_1, \ldots p_n>$, onde $p_n = p_0$, determinar se o ponto q é interior a P.



Ponto Interior a Polígono convexo.

dado um polígono convexo, $P \langle p_0, p_1, \ldots p_n \rangle$, onde $p_n = p_0$, determinar se o ponto q é interior a P.



Ponto Interior (q) a Polígono convexo

P $\langle p_0, p_1, \ldots p_n \rangle$, onde $p_n = p_0$.

lógico PontoNoSegmento(ponto q, p1, p2):
    retornar SentidoPercurso(q, p1, p2) = 0 e
        q.x ≥ min(p1.x, p2.x) e q.x ≤ max(p1.x, p2.x) e
        q.y ≥ min(p1.y, p2.y) e q.y ≤ max(p1.y, p2.y)

lógico PontoInteriorPoligonoConvexo(polígono P, ponto q):
    t ← SentidoPercurso(q, p_{n-1}, p_n)
    se t=0 então
        retornar PontoNoSegmento(q, p_{n-1}, p_n)
    para i ← 1 até n-1 faça
        t' = SentidoPercurso(q, p_{i-1}, p_i)
        se t' = 0 então
            retornar PontoNoSegmento(q, p_{i-1}, p_i)
        senão se t' ≠ t então
            retornar falso
    retornar verdadeiro

Reconhecimento de polígono convexo.

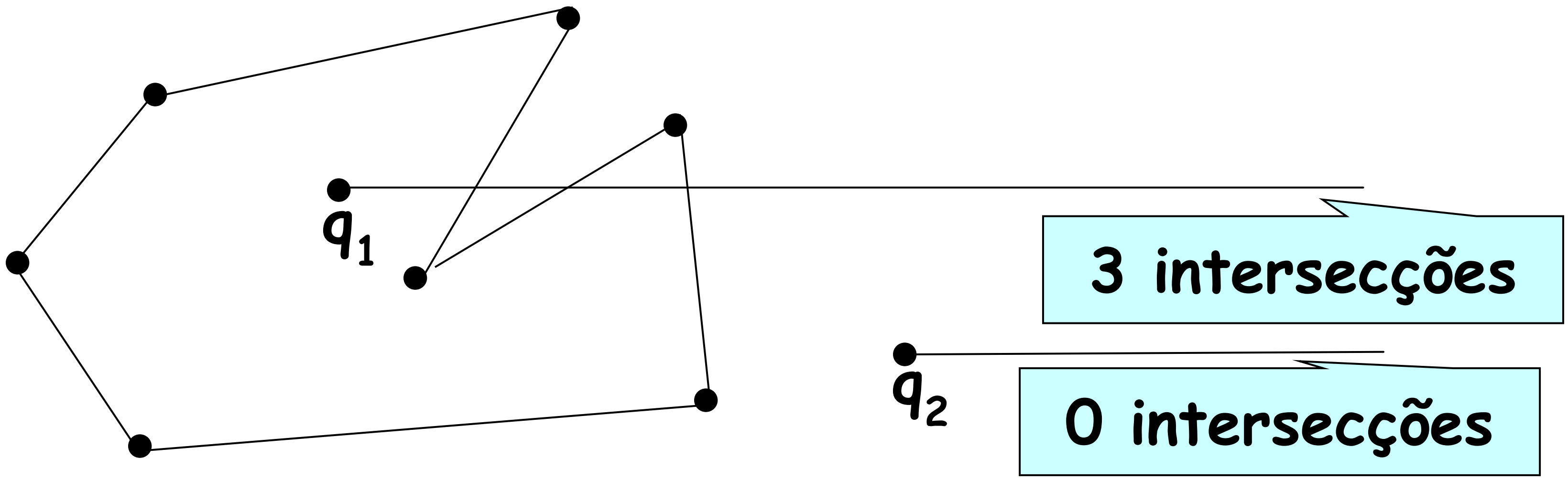
Análogo ao reconhecimento de ponto anterior. O sentido do percurso de cada três vértices consecutivos tem que ser sempre o mesmo.

lógico PoligonoConvexo (polígono P):

```
t ← SentidoPercurso(p_{n-2}, p_{n-1}, p_0)
se  t ≠ SentidoPercurso(p_{n-1}, p_0, p_1) então
    retornar falso
para i ← 0 até n-3 faça
    se  t ≠ SentidoPercurso(p_i, p_{i+1}, p_{i+2}) então
        retornar falso

retornar verdadeiro
```

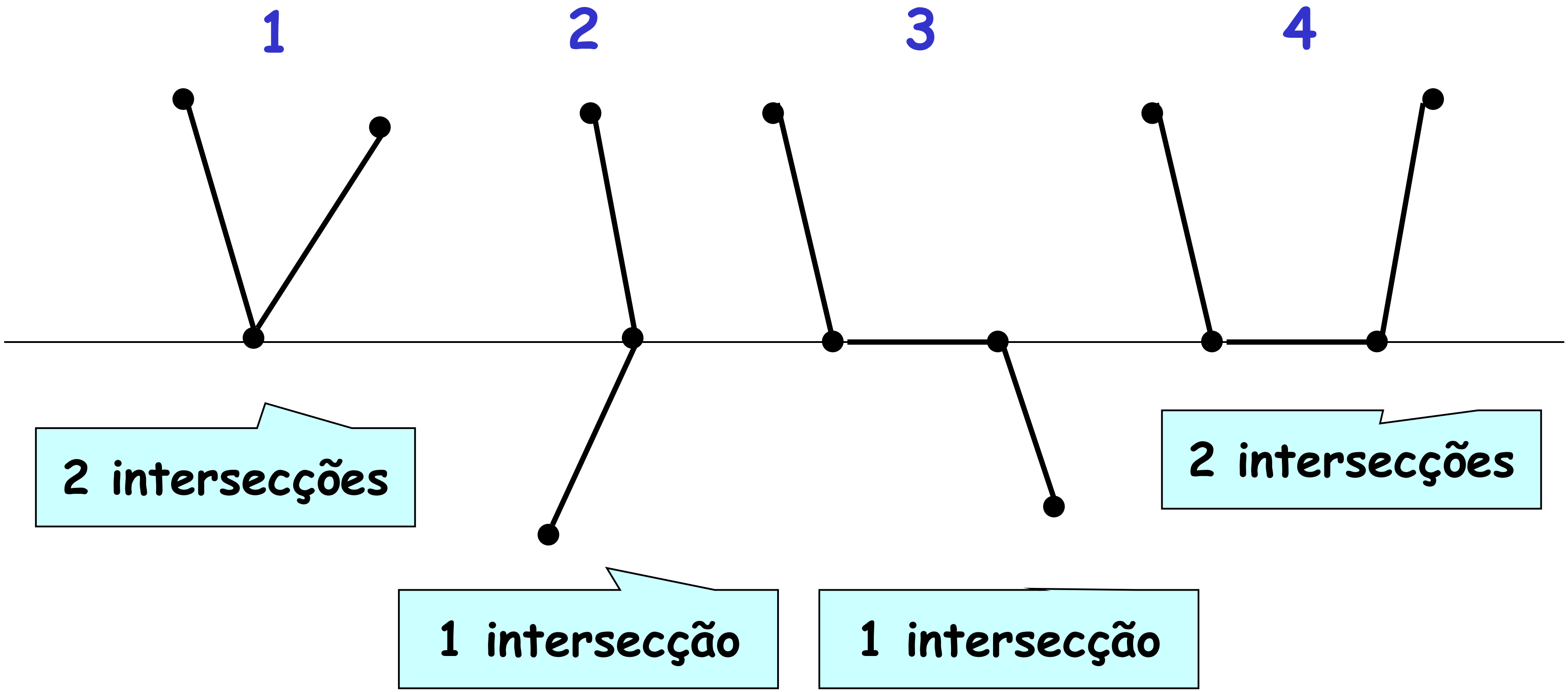
Ponto Interior a Polígono.

dado um polígono, $P \langle p_0, p_1, \ldots p_n \rangle$, onde $p_n = p_0$, determinar se o ponto q é interior a P.

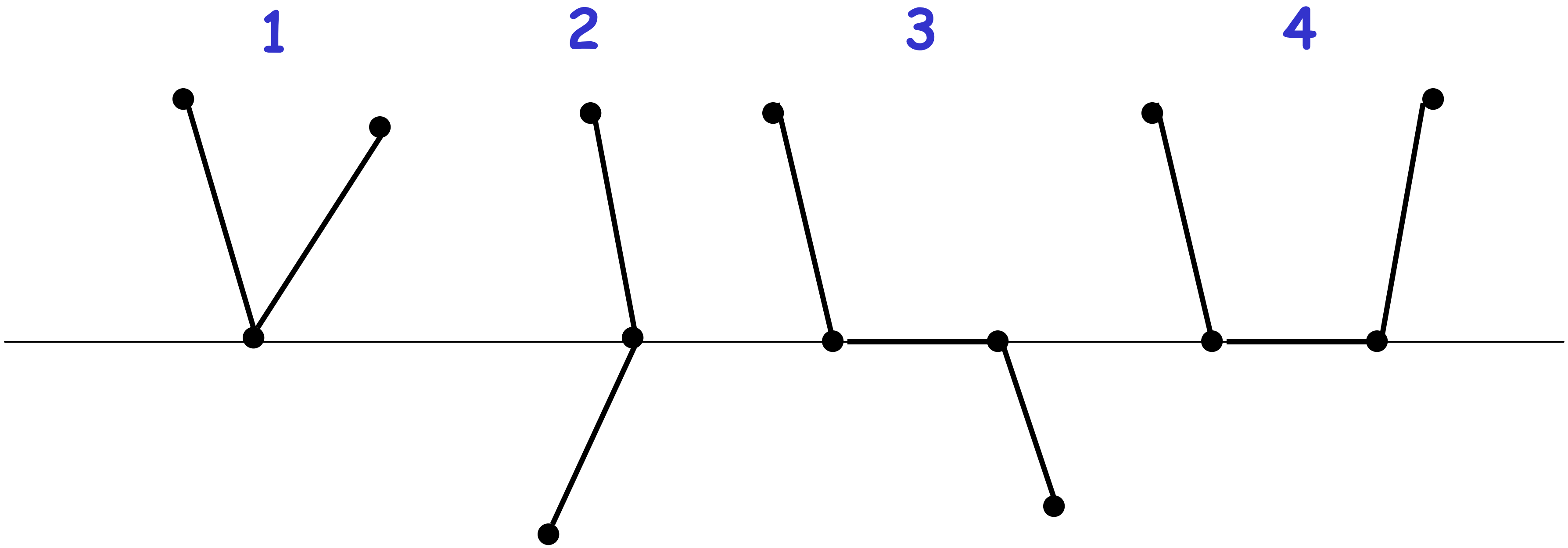


Ideia: checar a paridade do número de intersecções da semirreta paralela ao eixo x que inicia em q e vai para +∞. Se par, está fora, cc. dentro.

Ponto Interior a Polígono - casos especiais.



Ponto Interior a Polígono - casos especiais.



Regra geral: só contar intersecção quando um dos pontos estiver acima da reta de referência e o outro sobre essa reta ou abaixo.

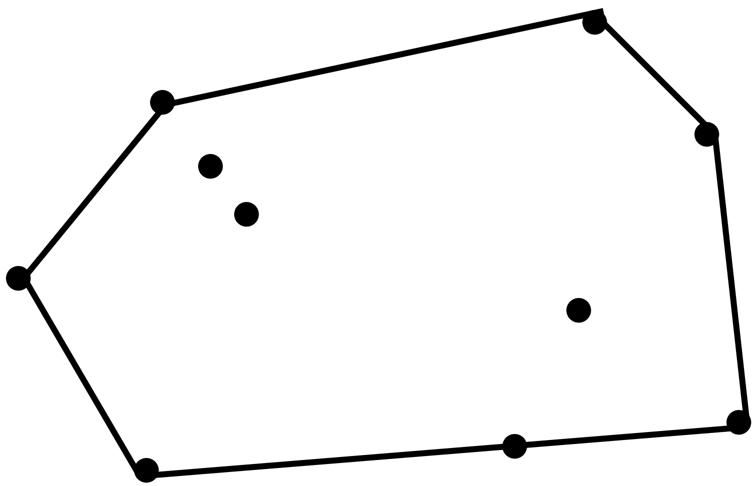
Ponto interior q a polígono

P $\langle p_0, p_1, \ldots p_n \rangle$, onde $p_n = p_0$

```
lógico PontoInterior(polígono P, ponto q):
    para i ← 1 até n faça
        se PontoNoSegmento(q, p_{i-1}, p_i):
            retornar verdadeiro

    ponto q_f;   q_f.x← +∞;   q_f.y←q.y

    c ← 0
    para i ← 1 até n faça
        se (InterceptaSegmento(q, q_f, p_{i-1}, p_i) e
            ((p_i.y > q.y) e (p_{i-1}.y ≤ q.y) ou (p_{i-1}.y > q.y) e (p_i.y ≤
        q.y)) então
            c ← c + 1

    retornar (c mod 2) ≠ 0
```

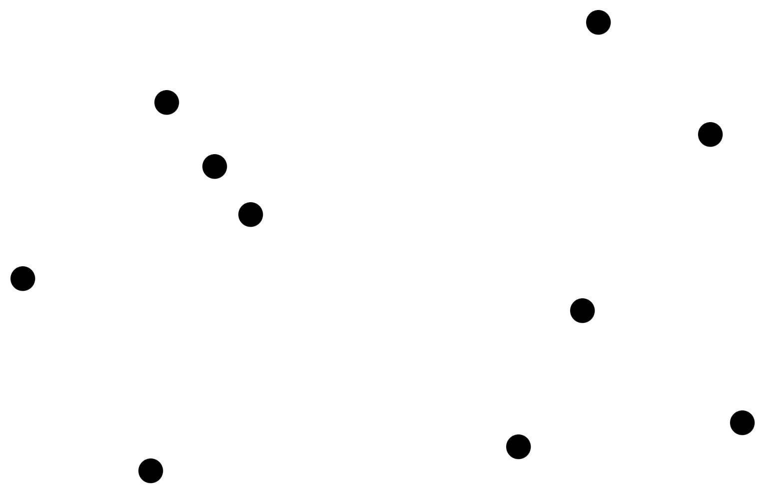
Determinação de contorno convexo.

dado um conjunto de n pontos, determinar o "menor" polígono convexo que contenha todos os pontos.



Determinação de contorno convexo.

Algoritmo GRAHAM SCAN.



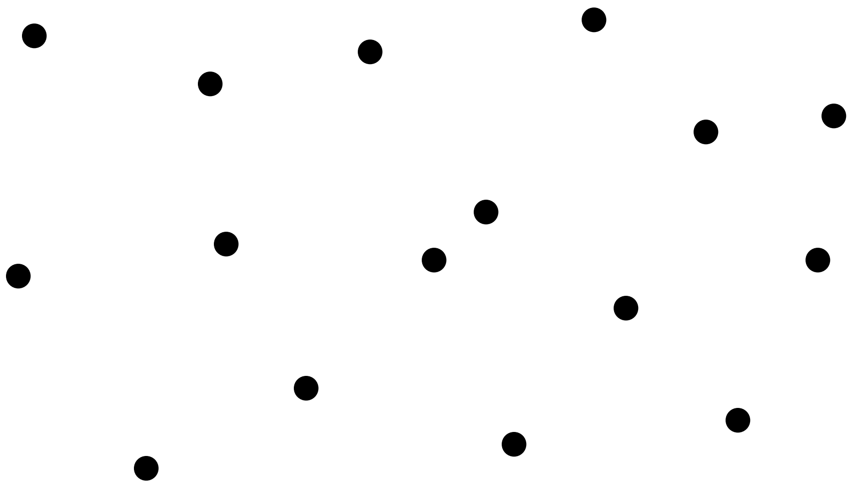
Determinação de contorno convexo.

```
GrahamScan(ponto P[], inteiro m, ref ponto S[], ref n):
    i ← índice do ponto de P mais abaixo e, dentre
         aqueles mais abaixo, daquele mais à esquerda
    P[1],P[i] ← P[i], P[1]
    Ordenar(P[2..m]), tal que
        P[i] < P[j] ⇔ SentidoPercurso(P[1],P[i],P[j]) = 1 ou
                  (SentidoPercurso(P[1],P[i],P[j]) = 0 e d(i) < d(j))
        onde d(x) = distância entre P[x] e P[0]
    definir pilha S[1..m] e manter n = número de elementos inseridos
    em S
    para i ← 1 até m faça
        se n < 2 então
            Empilha(S, P[i])
        senão
            enquanto (n > 1) e (SentidoPercurso(S[n-1], S[n], P[i]) ≠ 1)
        faça
                Desempilha(S)
            Empilha (S, P[i])
```

Tempo:
$\Theta(m \log m)$

Par de pontos mais próximos.

dado um conjunto de n pontos, determinar o par de pontos “mais próximo”.



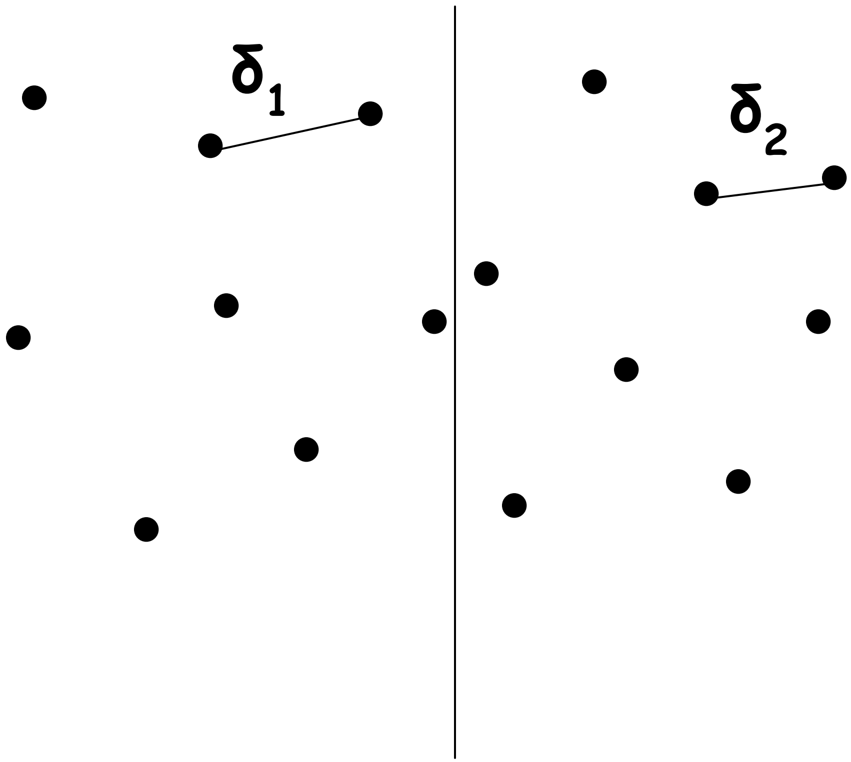
Solução de força bruta: calcular distâncias de cada ponto para os demais. Complexidade: $\Theta(n^2)$.

Par de pontos mais próximos. Força bruta melhorada.

```
real ParMaisProximo(ponto P[], inteiro n):
    Ordenar(P[1..n]), tal que P[i] < P[j] ⇔ P[i].x < P[j].x
    d ← Dist(P[1], P[2])
    para i ← 2 até n-1 faça
        d ← min(Dist(P[i], P[i+1]), d)
    para i ← 1 até n-1 faça
        para j ← i+1 até n faça
            se P[j].x-P[i].x > d então
                sair-para
            d ← min (Dist(P[i], P[j]), d)
    retornar d
```

Par de pontos mais próximos.

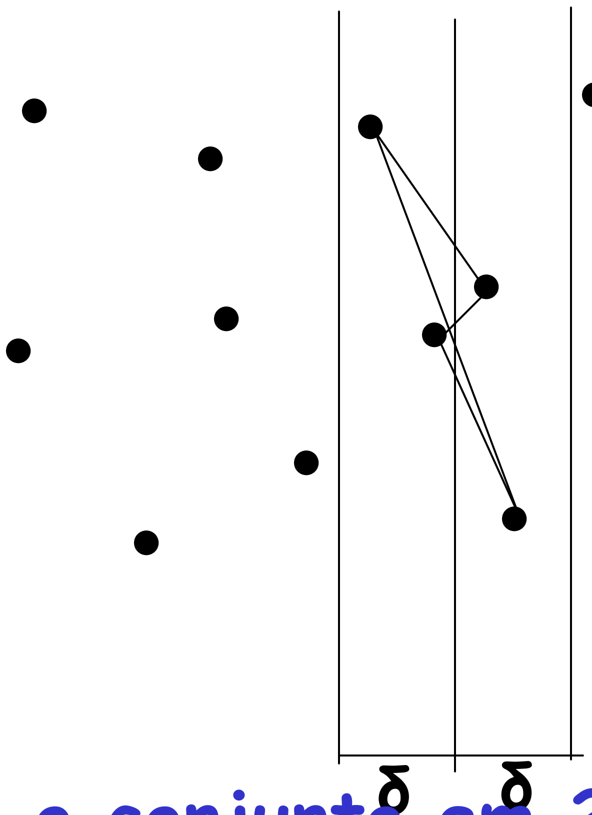
Solução recursiva.



Dividir o conjunto em 2, calcular δ_1 e δ_2 e verificar se, na interface, existe algum par com distância < $\delta = \min(\delta_1, \delta_2)$.

Par de pontos mais próximos.

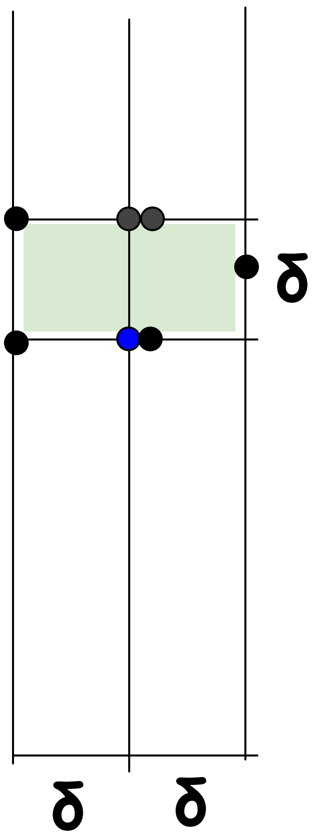
Solução recursiva.



Dividir o conjunto em 2, calcular δ_1 e δ_2 e verificar se, na interface, existe algum par com distância < $\delta = \min(\delta_1, \delta_2)$.

Par de pontos mais próximos.

Solução recursiva.



Tratamento da interface: para cada ponto, há, no máximo, 6 pontos acima que podem estar no retângulo mostrado.

Par de pontos mais próximos.

Complexidade: $O(n \log^2 n)$

```
real ParMaisProximo (ponto P[], inteiro n):
    Ordenar(P[1..n]), tal que P[i] < P[j] ⇔ P[i].x < P[j].x
    retornar ParMaisProximoRec(P, 1, n)

real ParMaisProximoRec (ponto P[], inteiro ini, fim);
    se (fim-ini ≤ 2) então
        retornar "menor distância entre pontos {P[ini], P[ini+1], P[fim]}"
    senão
        m  ← (ini+fim) div 2
        d1 ← ParMaisProximo (P, ini, m); d2 ← ParMaisProximo (P, m+1, fim)
        d  ← min(d1, d2);   S ← { p ∈ P[ini..fim] | P[m].x-d ≤ p.x ≤ P[m].x+d }
        Ordenar(S) tal que P[i] < P[j] ⇔ P[i].y < P[j].y
        para cada p ∈ S faça
            d' ← menor distância de p aos 6 pontos a sua frente em S
            d  ← min(d', d)
        retornar d
```

Par de pontos mais próximos.

Solução recursiva.

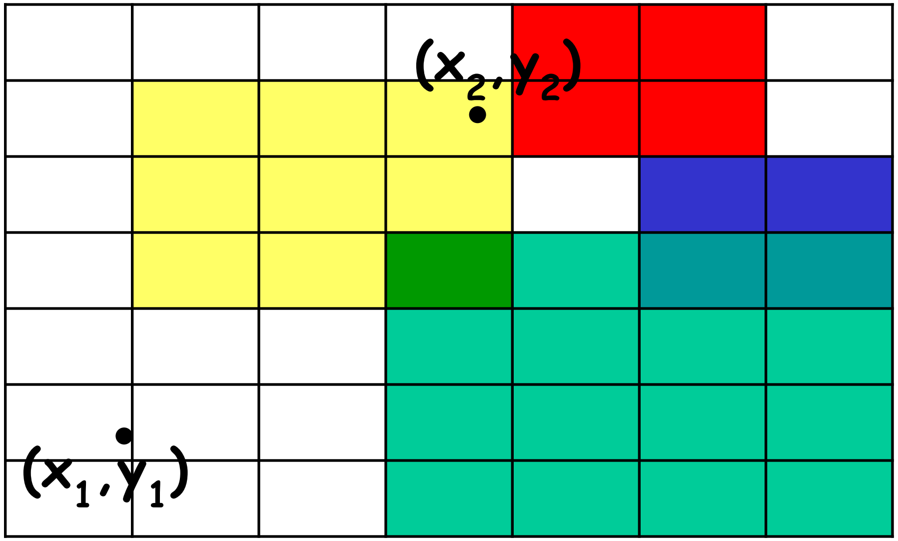
O melhor algoritmo:

Ordenando os pontos por y e obtendo simultâneamente os pontos mais próximos, obtem-se complexidade O(n log n).

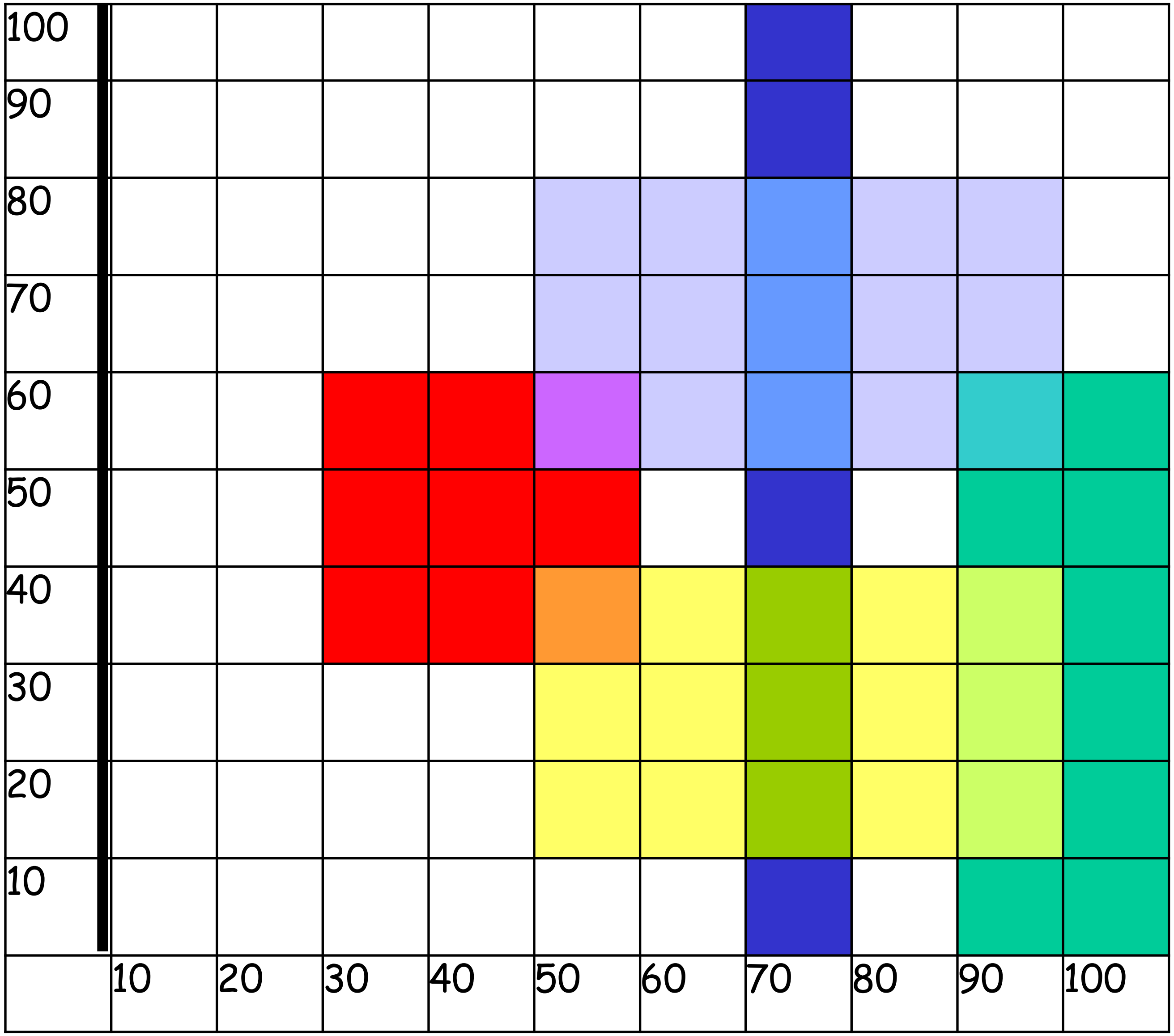
Sweep Line:

muitos algoritmos geométricos consistem na ordenação de pontos do plano segundo uma das coordenadas e varredura ("sweep line") dos dados ordenados, normalmente de forma recursiva.

Ex: Determinação da área de intersecção de retângulos em um grid:



Sweep Line: área de Intersecção



Sweep Line: Área de Intersecção.

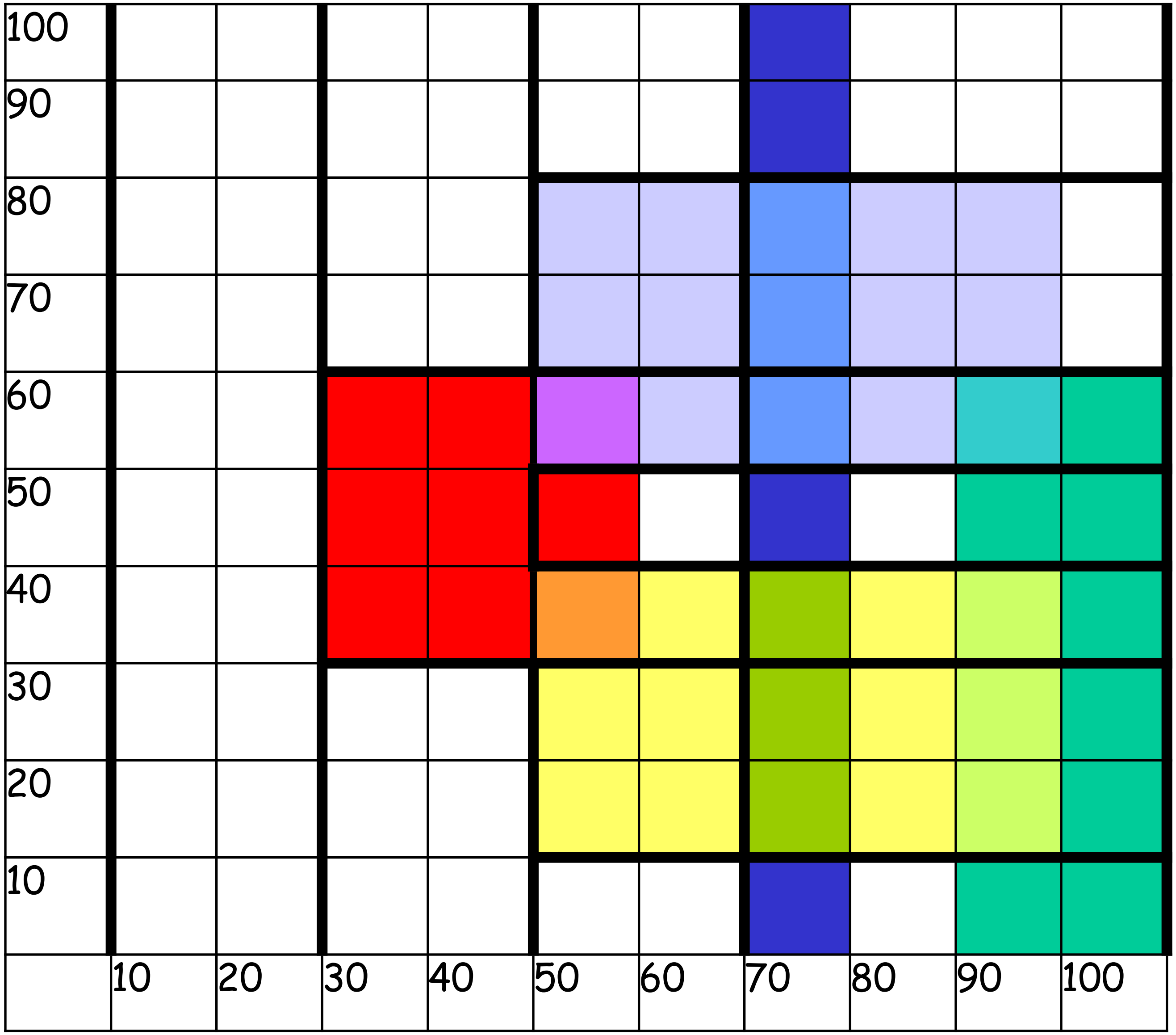
Algoritmo:

Sweep(h_1, h_2, ux, d, p);
    se (p $\leq$ nr):
        se (R[p].y_2 > h_1) E (R[p].y_1 < h_2):
            {Contabiliza área anterior}
            se (R[p].x_1 $\geq$ (ux+ d)): {ar $\leftarrow$ ar +(h_2-h_1)*d;   d $\leftarrow$0;}
            senão: {ar $\leftarrow$ ar+(h_2-h_1)*(R[p].x_1 - ux); d$\leftarrow$d-(R[p].x_1-ux);}
            {Subdivide a região e continua a varredura}
            se (h_2 > R[p].y_2): Sweep(R[p].y_2, h_2, R[p].x_1, ud, p+1);
            se (h_1 < R[p].y_1): Sweep(h_1, R[p].y_1, R[p].x_1, ud, p+1);
            Sweep(max(h_1, R[p].y_1), min(h_2, R[p].y_2), R[p].x_1,
                    max(d, R[p].x_2-R[p].x_1), p+1);

        senão: Sweep(h_1, h_2, ux, d, p+1);

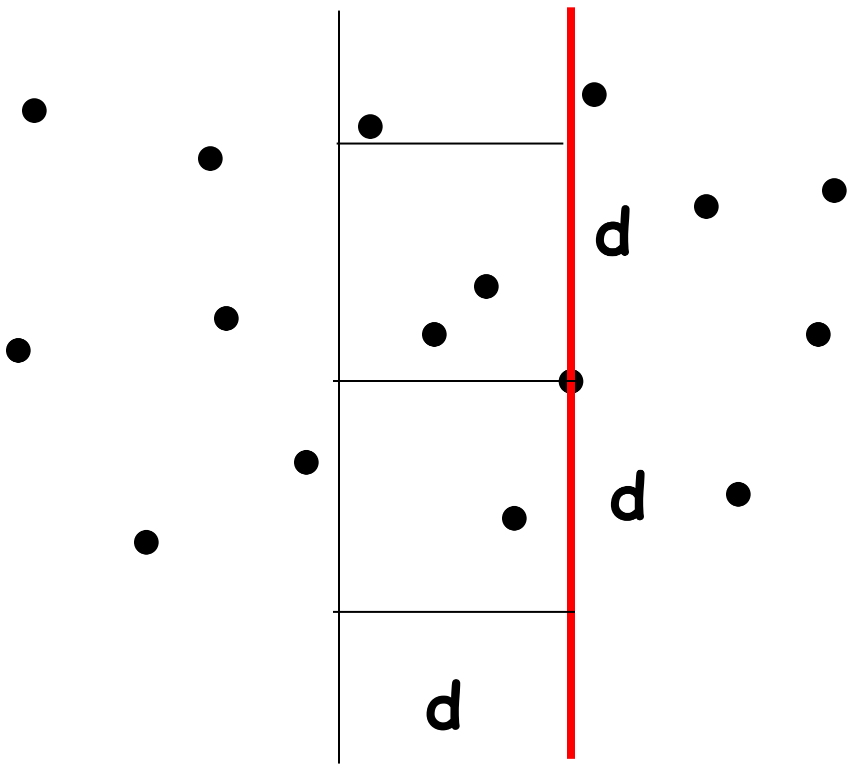
OrdenaPontosx;   nr++; {sentinela}
R[nr].x_1 $\leftarrow$ L; R[nr].y_1 $\leftarrow$ 0; R[nr].x_2 $\leftarrow$ L; R[nr].y_2 $\leftarrow$ H;
ar $\leftarrow$ 0;   Sweep(0, H, 0, 0, 1);

Sweep Line: área de Intersecção



Par de pontos mais próximos.

Solução com Sweep line.



Usa-se uma sweep line no eixo x. Ao se examinar o ponto p_k, calcula-se a distância somente aos pontos do retângulo, atualizando d. Os pontos à distância d são mantidos em uma estrutura do tipo set(árvore balanceada, ordenados pela coordenada y). Antes de examinar o ponto p_k, eliminam-se do set todos os pontos com coordenada x menor que $p_k.x-d$. Ao final, p_k entra no set.

Sweep Line: Par de pontos mais próximo.

Algoritmo:

```
Sweeppp();
    Ordenar pontos por x
    d ← Dist(p_1, p_2)
    para i ← 2..n-1 incl.:
        d ← min(d, Dist(p_i, p_{i+1}))
    Insere(S,p_1)
    Insere(S, p_2);
    para i ← 3..n incl.:
        S ← S - {pontos p_j com (p_j.x < p_i.x-d)}
        para pontos p_j em S com (|p_j.y - p_i.y| ≤ d):
            d ← min(d, Dist(p_i, p_{i+1}))
        Insere(S, p_i)
    retornar d
```

FIM