TÉCNICAS DE OTIMIZAÇÃO

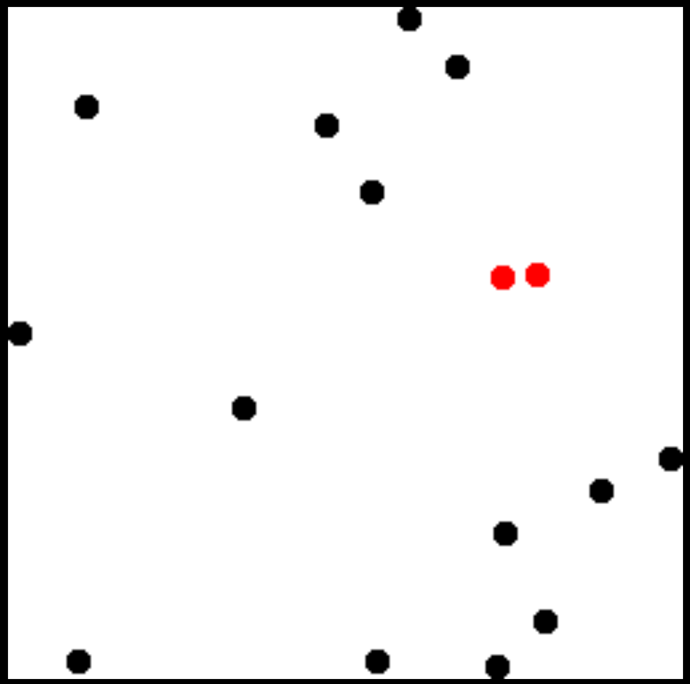
PROBLEMAS DE GEOMETRIA COMPUTACIONAL

Prof. Leandro Tonietto

CLOSEST PAIR OF POINTS

- O problema de geometria computacional do par de pontos mais próximo é um caso de uso de estudo para complexidade de algoritmos.
- O objetivo é encontrar o par de pontos que é mais próximo.

Como poderíamos resolver este problema da forma mais simples possível? (algoritmo de força bruta)



CLOSEST PAIR OF POINTS

FORÇA BRUTA

- **Algoritmo de força bruta:**

```
int* closestPair(int *P, int n){
    float minDist = INFINITY;
    int *closest = new int[2];
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            int p = P[i], q = P[j];
            // distância euclidiana
            float d = dist(p, q);
            if (d < minDist) {
                minDist = d;
                closest[0] = p; // or i
                closest[1] = q; // or j
            }
        }
    }
    return closest;
}
```

Qual é a complexidade do algoritmo?

Complexidade temporal $O(n^2)$

Como poderíamos melhorar?

Algoritmos de dividir-para-conquistar porem reduzir complexidade para $O(n\log n)$

CLOSEST PAIR OF POINTS

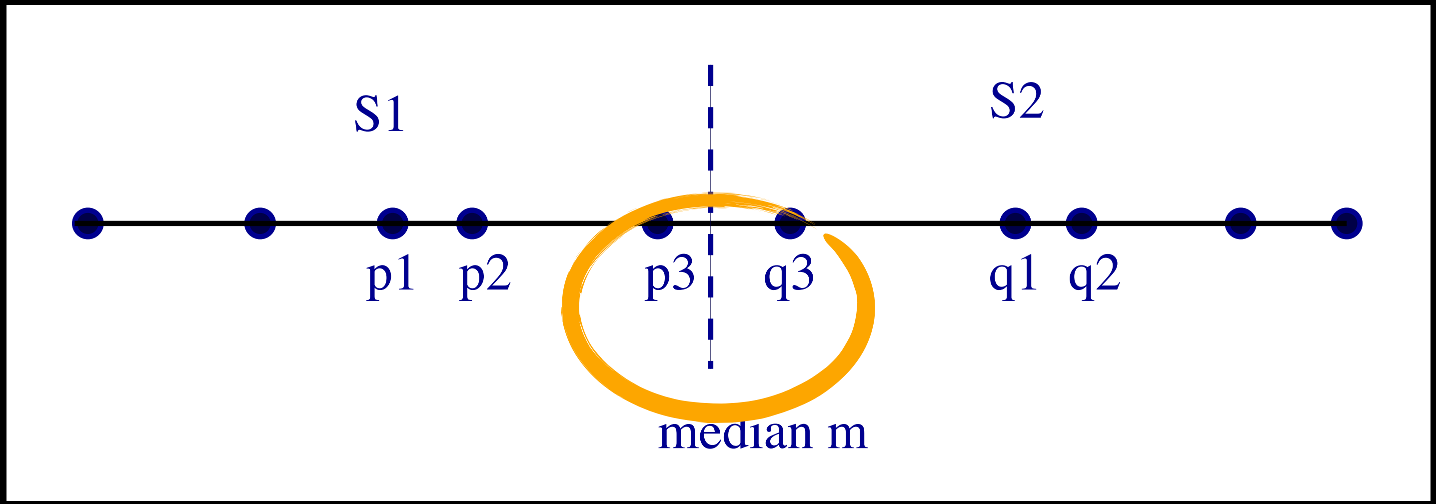
DIVIDIR PARA CONQUISTAR

- Antes de dividir para conquistar:
    - Dados podem ser previamente ordenados, o que pode melhorar a busca em `O(n)` dependo do valores.
        - Contudo, a ordenação deve ser feita pelo menos uma vez.
- Melhor dividir-para-conquistar…

CLOSEST PAIR OF POINTS

DIVIDIR PARA CONQUISTAR

- **Dividir para conquistar:**
  - Uma das técnicas mais comuns da ciência da computação.
  - Basicamente, dividi-se o problema em problemas menores. E assim sucessivamente até que o problema fique “simples" para resolver. Depois compor subproblemas resolvidos até retornar ao ponto original (conquista)
  - Para entendimento, vamos analisar um array de pontos (apenas em relação ao eixo X):

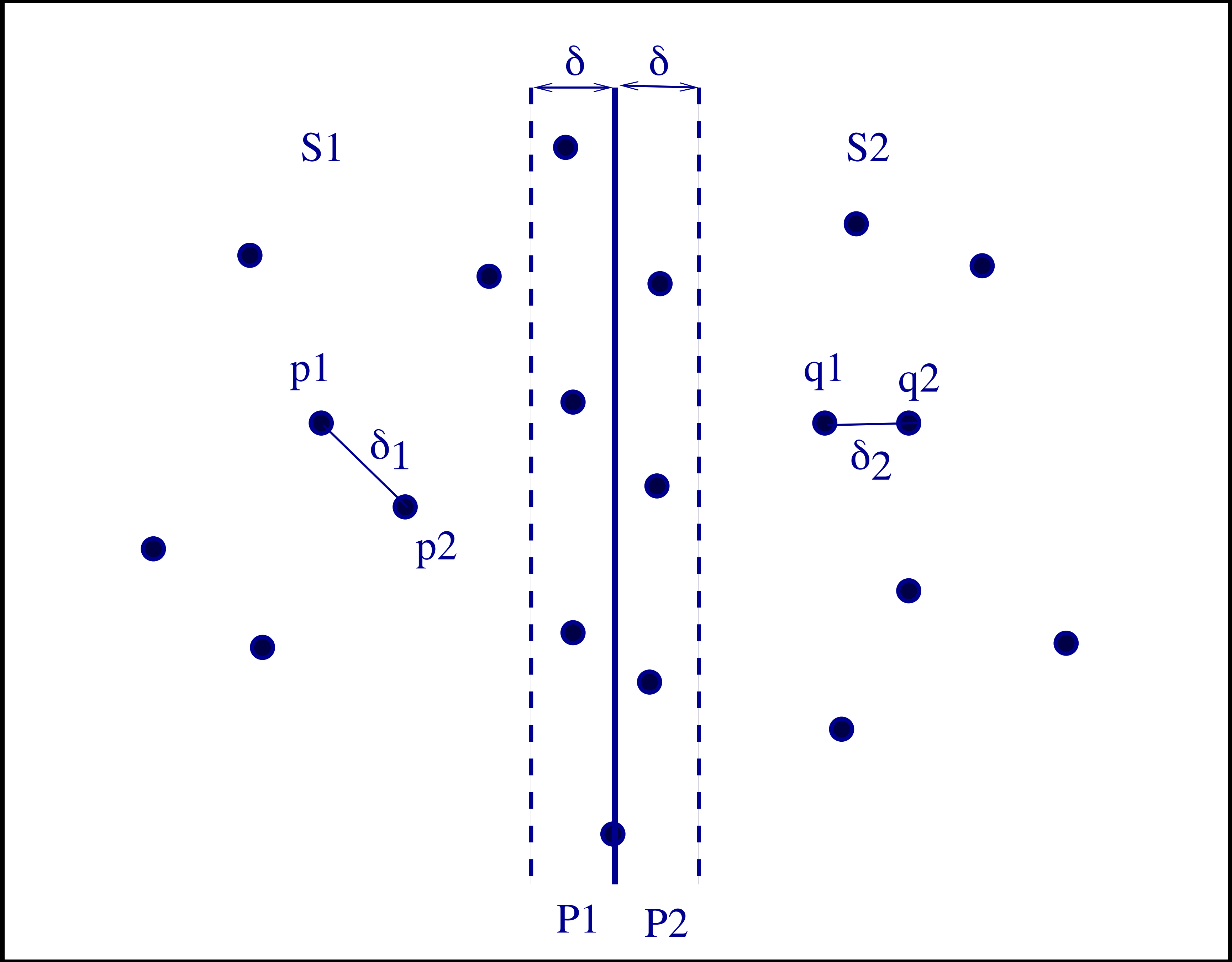


O par mais próximo é {p1,p2}, {q1,q2} ou {p3,q3}, onde $p3 \in S1$ e $q3 \in S2$.

CLOSEST PAIR OF POINTS

DIVIDIR PARA CONQUISTAR

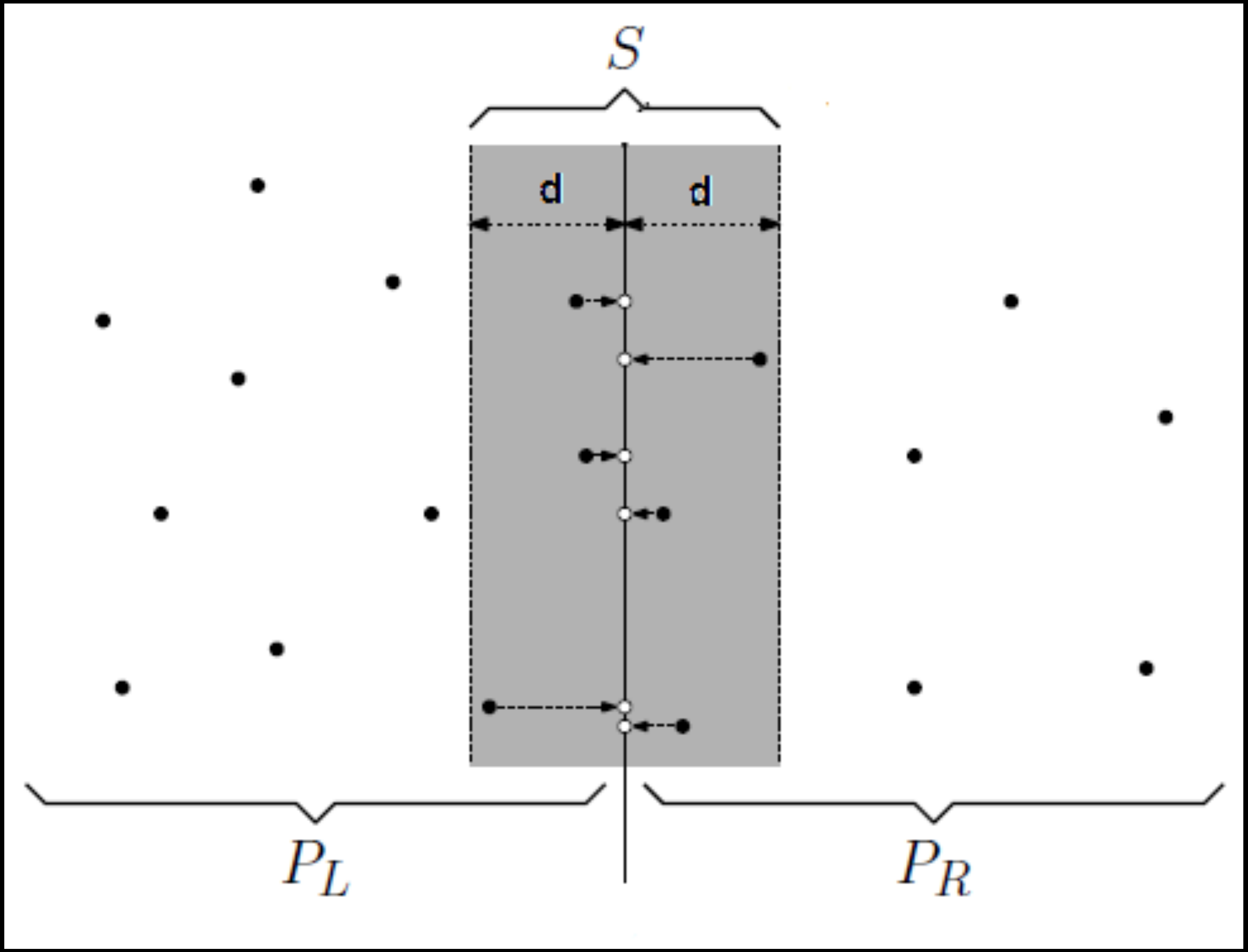
- Para 2D, considerar o eixo Y



CLOSEST PAIR OF POINTS

DIVIDIR PARA CONQUISTAR

- Para 2D, considerar o eixo Y



CLOSEST PAIR OF POINTS

DIVIDIR PARA CONQUISTAR

- Algoritmo:
  - Considere `d` a distância entre dois pontos de um conjunto.
    - E também quais são os dois pontos! Portanto, uma linha!
  - Considere `P` uma partição de array de pontos. E `|P|` a quantidade de elementos.
  - Considere `m` o valor mediano dentre os pontos da partição do do array.

d = SQRT((x₁−x₂)²+(y₁−y₂)²)
na verdade não precisa SQRT

```
closestDivideAndConquer(P){
    se |P| = 1 então não considera d, retorna "infinito"
    se |P| = 2 então retorna a d.
    para os demais casos
        m = mediano(P)
        separa P em P1 e P2, sendo P1 os menores que m e P2 os
        maiores que m.
        d1 = closestDivideAndConquer(P1);
        d2 = closestDivideAndConquer(P2);
        d3 = calcula d considerando P1[n−1] e P2[0]; // pontos
        entre as partições!
    retorna menor entre d1, d2 e d3
```

CLOSEST PAIR OF POINTS

DIVIDIR PARA CONQUISTAR

- Análise do algoritmo:
  - O objetivo é dividir recursivamente o array original em partições menores, ordenando elementos por um dos eixos (x por exemplo).
    - `P1: P[0]` até `P[n/2]`
    - `P2: P[n/2+1]` até `P[n-1]`
    - Considere um algoritmo **`O(nlogn)`**, como *Quick Sort* ou *Merge Sort*.
  - Cada partição possui a metade dos pontos.
  - Quando a partição atinge um número adequado, resolve por força bruta.
  - Não esquecer de analisar os elementos que ficam na borda da partição do array.
    - Testar elementos que fiquem até `2d` distancia da borda.
    - `d` é a distância mínima da partição.

CLOSEST PAIR OF POINTS

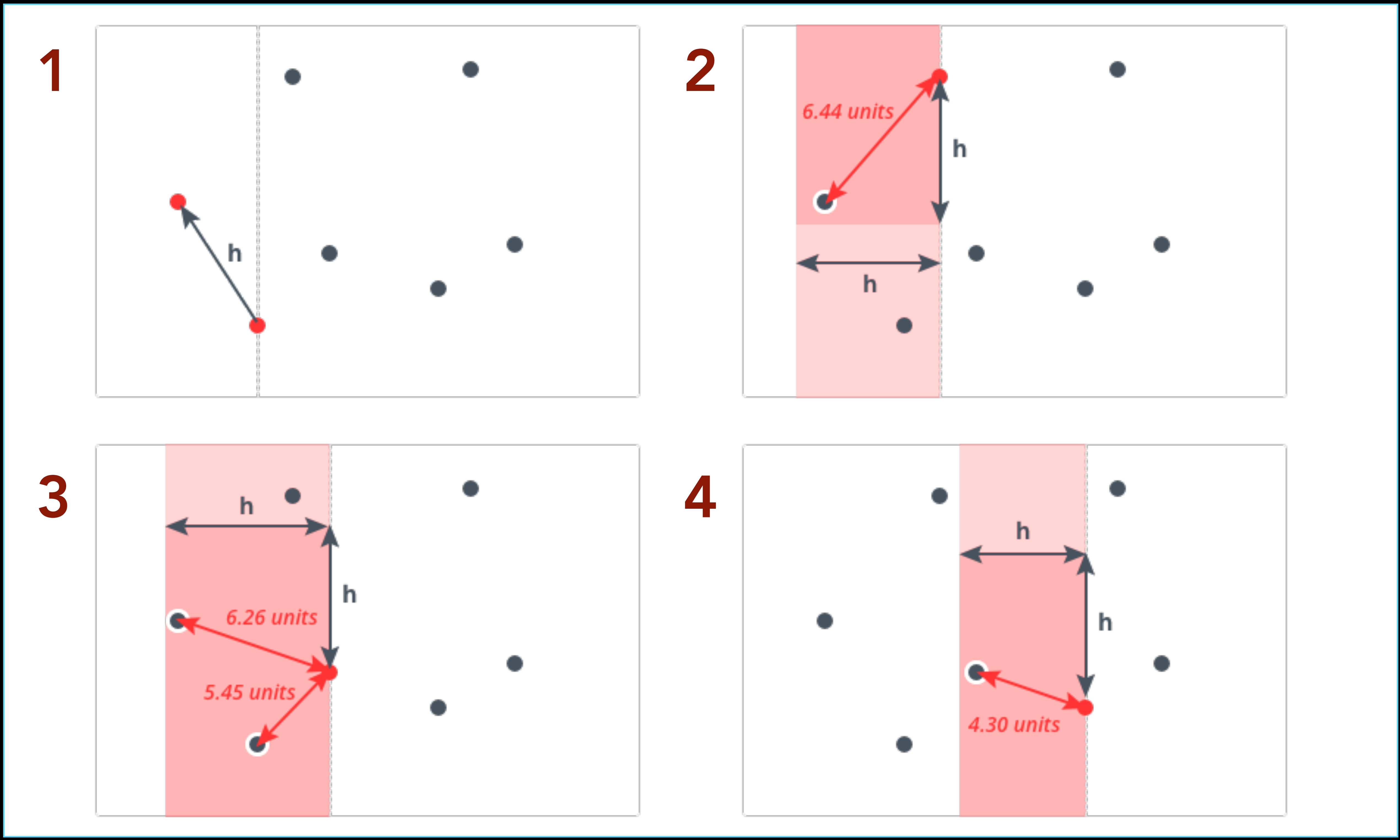
LINE SWEEP

- **Line sweep:**
  - Outro algoritmo eficiente para descobrir os pontos mais próximos é o *line-sweep*.
  - Basicamente, é um algoritmo de varredura de linhas, da esquerda para direta, buscando a menor distância em x e y.
  - Podemos considerar que para o próximo ponto a ser processado, deve ser encontrado o ponto que cuja distância para o *n*-ésimo ponto seja **menor ou igual** que a menor distância encontrada até o momento.
    - Para isto é necessária a ordenação dos pontos pelo eixo x, no sentido da esquerda para a direita.
    - Então pontos seguintes onde x é menor $x_N - h$ e com y entre $y_N - h$ e $y_N + h$ devem ser considerados.

CLOSEST PAIR OF POINTS

LINE SWEEP

https://www.hackerearth.com/pt-br/practice/math/geometry/line-sweep-technique/tutorial/



CLOSEST PAIR OF POINTS

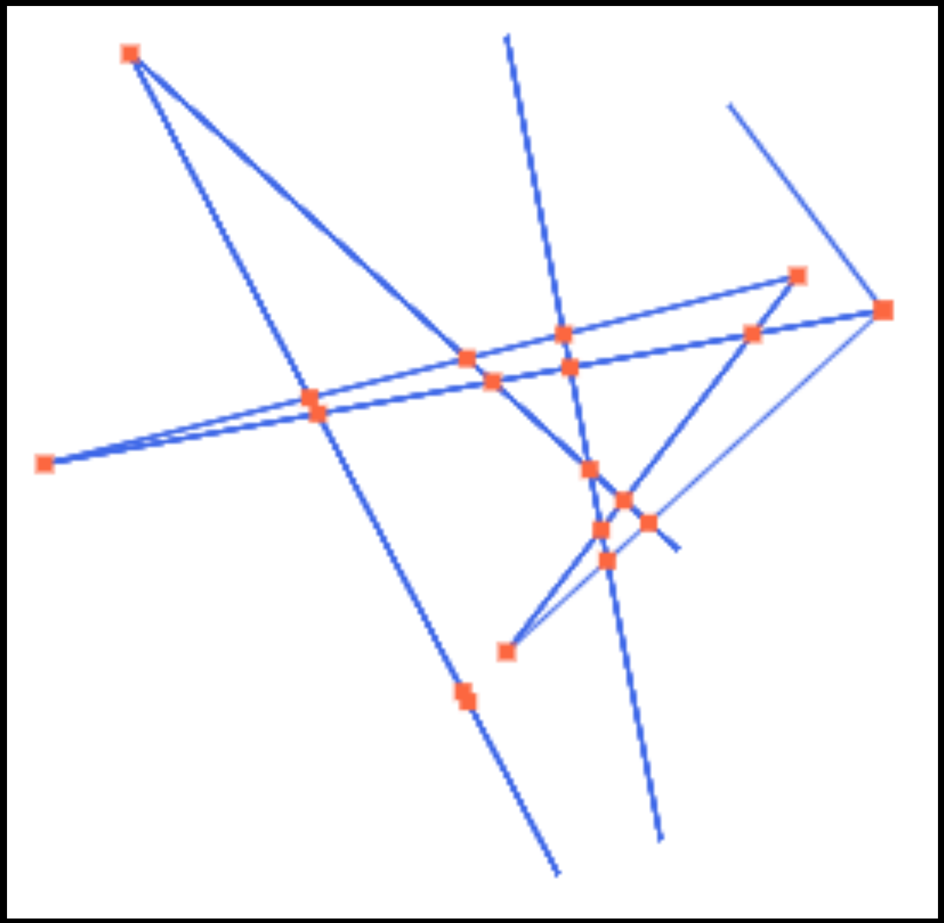
LINE SWEEP

- **Sobre o algortimo:**
  - https://www.hackerearth.com/pt-br/practice/math/geometry/line-sweep-technique/tutorial/
  - Pontos devem ser ordenados em X.
  - No box, ordenados em Y.
  - Complexidade:
    - Para remover do box elementos maiores que a distância: `O(n)`… função para remover elemento num objeto `set` é de `O(log(n))`, no geral: `O(n * log(n))`.
    - Segundo loop (que define o menor) tem complexidade: `O(log(n))` … pois são apenas os elementos dentro do box.
    - Tem ainda a inserção no box, que custa `O(log(n))`.
    - No geral, portanto, `O(n*log(n))`.

INTERSEÇÃO DE LINHAS

- Outro tipo de problema clássico em geometria computacional é determinar a interseção entre linhas.
- Dado um conjunto de segmentos de linha, encontrar as interseções entre as linhas.

Como poderíamos resolver este problema da forma mais simples possível? (algoritmo ingênuo)



INTERSEÇÃO DE LINHAS

ALGORITMO INGÊNUO

- **Algoritmo ingênuo:**

```
function getLinesIntersections(L){
 var allIntersections = [];
  for (var i = 0; i < L.length; i++) {
     for (var j = i + 1; j < L.length; j++) {
        var intersection = linesIntersection(L[i], L[j]);
        if(intersection){
           allIntersections.push(intersection);
        }
     }
  }
  return allIntersections;
}
```

INTERSEÇÃO DE LINHAS

ALGORITMO INGÊNUO

- **Algoritmo ingênuo:**

```
function linesIntersection(L1, L2) {
    var det = (L2.p1.x - L2.p0.x) * (L1.p1.y - L1.p0.y)
            - (L2.p1.y - L2.p0.y) * (L1.p1.x - L1.p0.x);
    if (det === 0) {
        return null; // não há intersecção
    }
    var s = ((L2.p1.x - L2.p0.x) * (L2.p0.y - L1.p0.y)
          - (L2.p1.y - L2.p0.y) * (L2.p0.x - L1.p0.x)) / det;
    if((s > 1) || (s < 0)) {
        return null; // não há interseção
    }
    var x = L1.p0.x + (L1.p1.x-L1.p0.x)*s;
    var y = L1.p0.y + (L1.p1.y-L1.p0.y)*s;
  return {"x": x, "y": y}; // ponto de interseção!
}
```

http://www.inf.pucrs.br/~pinho/CG/Aulas/OpenGL/Interseccao/CalcIntersec.html

INTERSEÇÃO DE LINHAS

ALGORITMO PARA LINHAS VERTICAIS E HORIZONTAIS

$O(N \log N + K)$

- **Caso especial com linhas horizontais e verticais:**
  - São considerados apenas pontos internos das linhas (sem colisões de extremidades).
  - Partindo de uma linha da esquerda para a direita traçar uma linha “sweep"
  - A cada x inicial de linha horizontal que coincide com linha sweep, o ponto inicial da linha é memorizado numa lista.
  - Quando o ponto final da linha é alcançado, o ponto inicial é removido da linha.
  - Para cada linha vertical que colidiu com linha sweep, verifica as linhas horizontais que estão selecionadas se o seu y0 está dentro dos limites de y0 e y1 definidos pela linha vertical; em caso positivo, o x0 da linha vertical e o y0 da linha horizontal definem um ponto de intersecção.

INTERSEÇÃO DE LINHAS

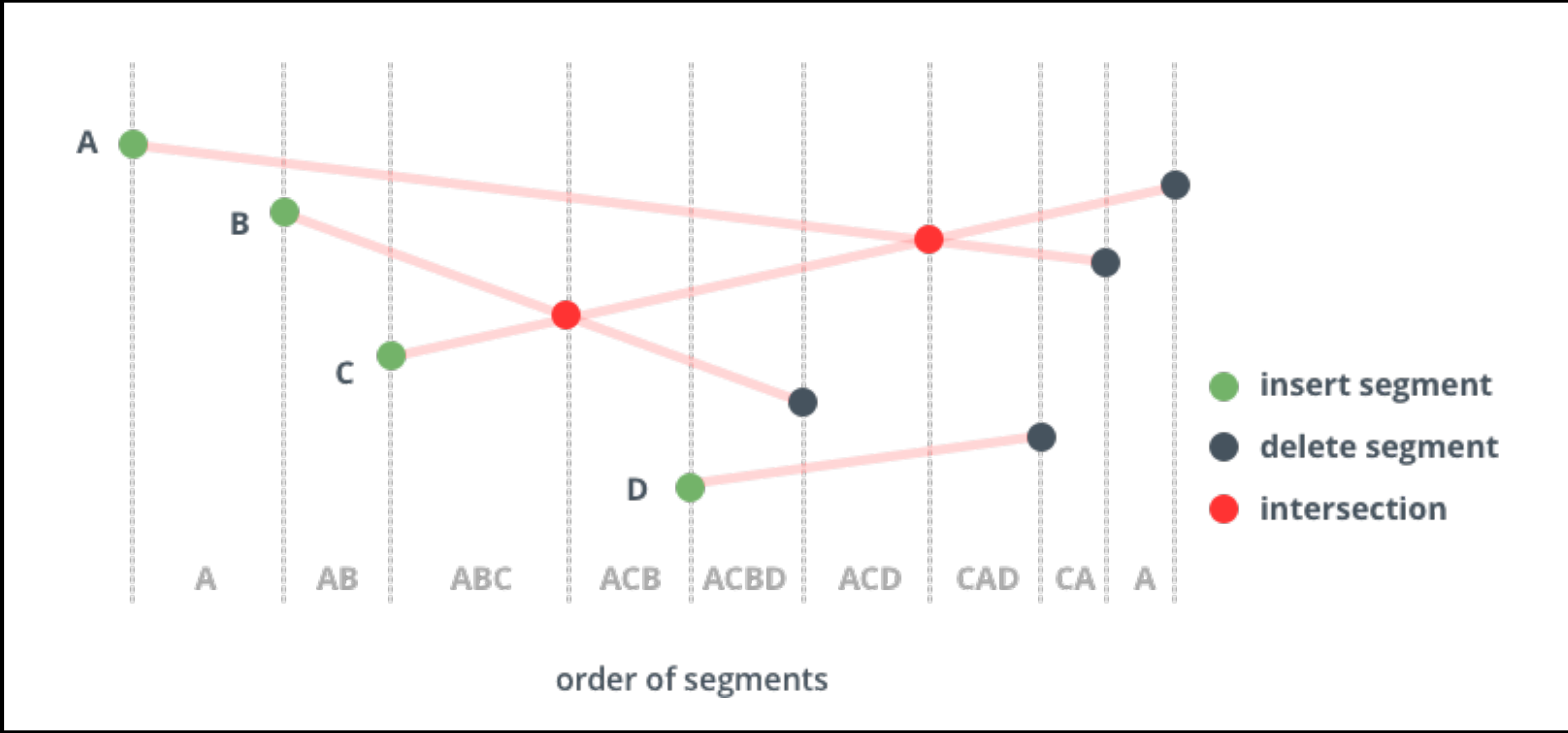
ALGORITMO BENTLEY-OTTMANN

- **Algortimo Bentley-Ottmann (sweep-line)**
  - Problemas do algoritmo ingênuo: comparações desnecessárias e ordenação dos elementos poderia melhor tempo de computação.
  - Considerações para o algoritmo:
    1. Não há segmentos verticais.
    2. Dois segmentos não se cruzam em seus pontos finais.
    3. Não considera que 3 ou mais segmentos têm uma interseção comum.
    4. Todos os pontos finais dos segmentos e todos os pontos de interseção têm diferentes coordenadas x.
    5. Não se sobrepõem dois segmentos.

INTERSEÇÃO DE LINHAS

ALGORITMO BENTLEY-OTTMANN

- **Algortimo Bentley-Ottmann (sweep-line)**
  - Basicamente, deveríamos ordenar as linhas e testar as que são adjacentes.
  - Ver algoritmo em: https://www.hackerearth.com/pt-br/practice/math/geometry/line-intersection-using-bentley-ottmann-algorithm/tutorial/

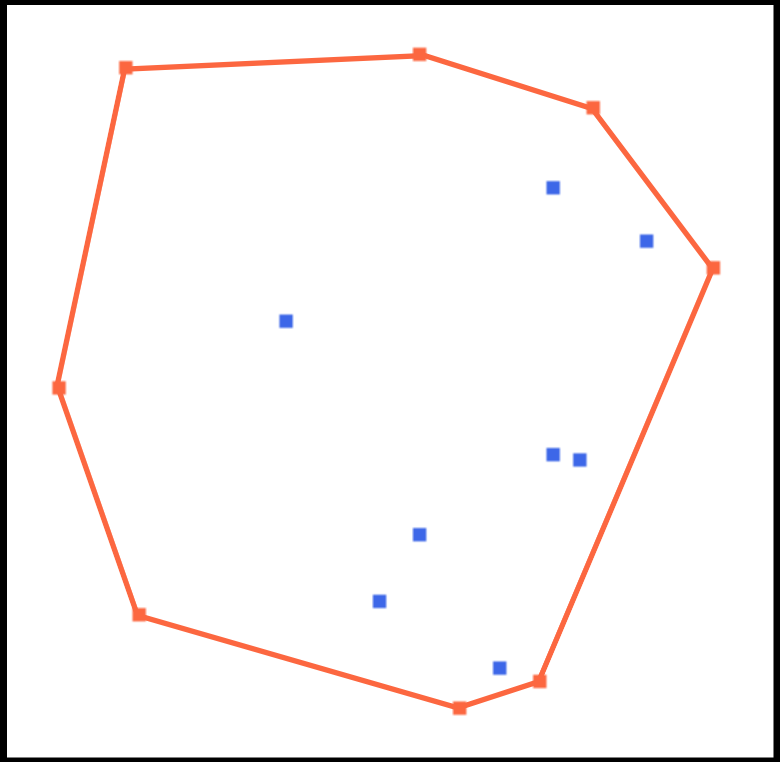


O(NLOGN+KLOGN), PORÉM SE K É O(N²), ENTÃO O ALGORITMO É PIOR QUE O INGÊNUO.

CONVEX HULL - FECHO CONVEXO

- Definir o fecho convexo que envolve uma lista de pontos
- Interessa é o polígono que envolve os pontos, ou seja, pontos internos não são necessários.

Como poderíamos resolver este problema da forma mais simples possível? (algoritmo ingênuo)



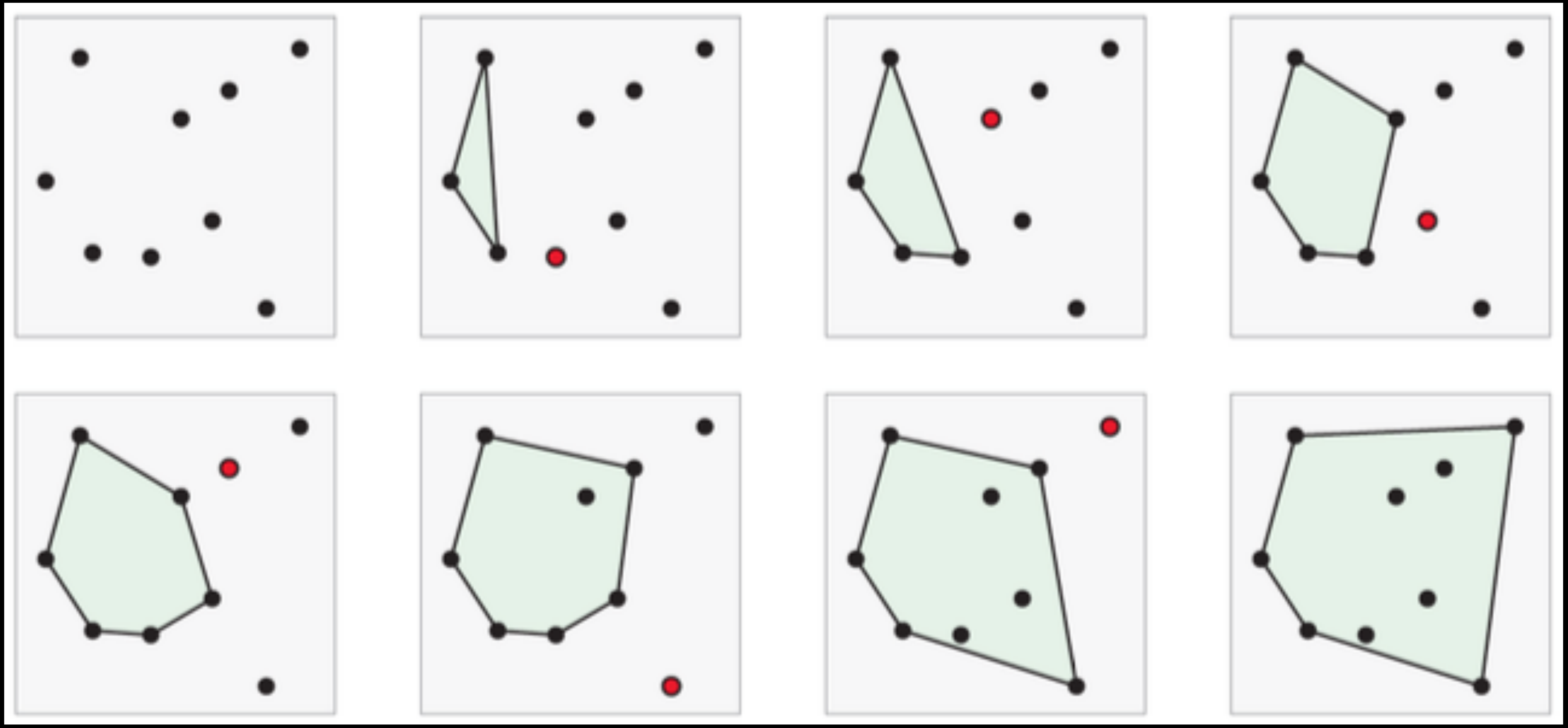
CONVEX HULL - FECHO CONVEXO

- Algoritmo ingênuo:
  - Testar cada ponto com uma linha formada pelos pontos da lista.
  - Determinar se o ponto está ha esquerda ou a direita da linha.

CONVEX HULL - FECHO CONVEXO

CONVEX HULL - INCREMENTAL

- Algoritmo incremental

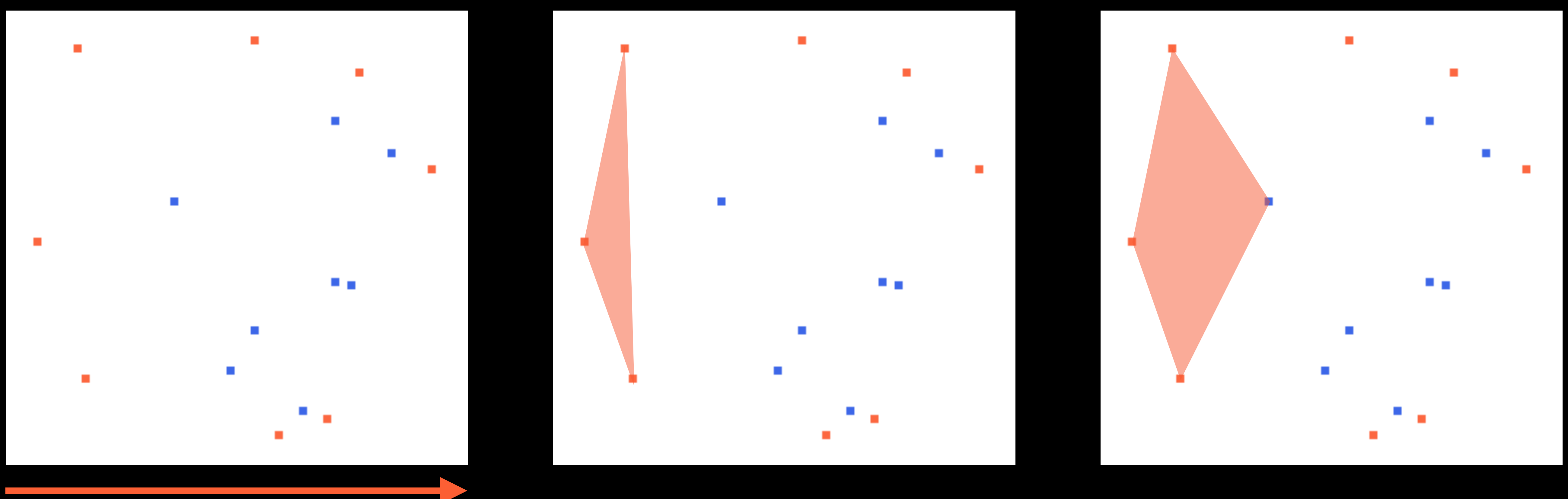


CONVEX HULL - FECHO CONVEXO

CONVEX HULL - INCREMENTAL

- Algoritmo incremental
    1. Ordenação dos pontos em x, da esquerda para direita
    2. Os três primeiros pontos formam um triângulo inicial
    3. Cada novo ponto deve ser verificado se está dentro ou fora e pontos internos devem ser removidos

COMPLEXIDADE $O(N^2)$



pontos do polígono tomados no sentido anti-horário

CONVEX HULL - FECHO CONVEXO

CONVEX HULL - FUNÇÕES

- Algumas funções interessantes para teste de orientação ou sentido do ponto em relação aos seus vizinhos:

```
function ccw(p1, p2, p3) {
    // ccw > 0: counter-clockwise; ccw < 0: clockwise; ccw = 0: collinear
    return   (p2.x - p1.x) * (p3.y - p1.y)
           - (p2.y - p1.y) * (p3.x - p1.x);
}
function polarAngle(p) {
    return Math.atan(p.y / p.x);
}
function dotProduct(vec1, vec2) {
   return (vec1.x * vec2.x + vec1.y * vec2.y);
}
function norm(vec) {
   return Math.sqrt(vec.x * vec.x + vec.y * vec.y);
}
function computeAngle(v1, v2) {
   var ac = dotProduct(v1, v2);
   return Math.acos(ac / (norm(v1) * norm(v2))) * ONE_RADIAN;
}
```

CONVEX HULL - FECHO CONVEXO

CONVEX HULL - QUICK HULL

- Algoritmo quick hull descobre o fecho convexo baseado na idea do algoritmo de ordenação quicksort.
- A ideia principal é subdividir os pontos em partições menores e tratar os casos com menos pontos possíveis.
- Algoritmo:
  1. Descobrir os pontos (A e B) com menor e maior valor na coordena x. Estes pontos pertencem ao fecho convexo.
  2. Para cada lado do segmento selecionado (AB), obter o ponto mais distante (C_1 e C_2).
  3. Os pontos internos ao triângulo formado ABC_i são descartados.
  4. Para cada nova aresta formada pelo triângulo ABC_i repetir o processo a partir do passo 2, informando os pontos da aresta como segmento selecionado.

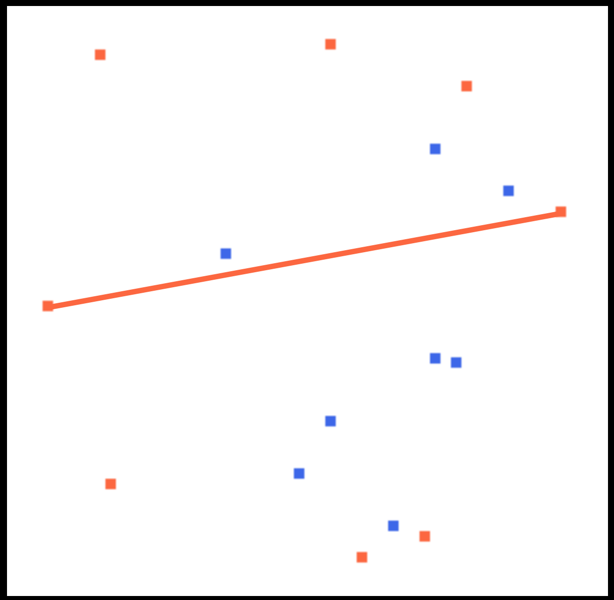
Qual é o pior caso?

COMPLEXIDADE:
CASO MÉDIO: O(NLOGN)
PIOR CASO: $O(N^2)$

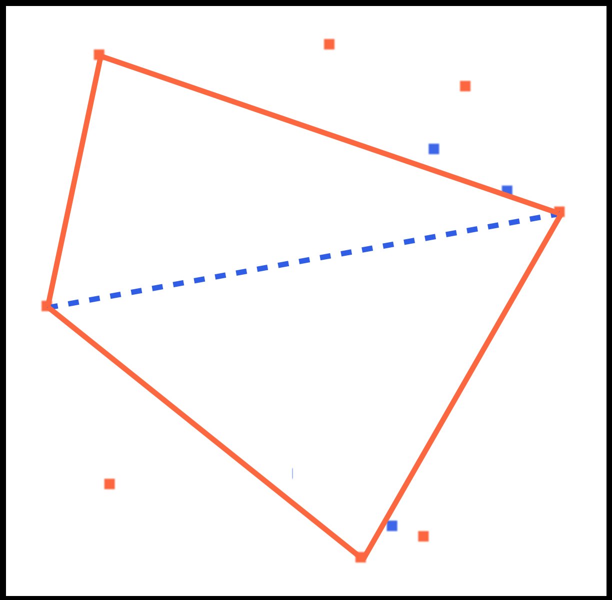
CONVEX HULL - FECHO CONVEXO

CONVEX HULL - QUICK HULL

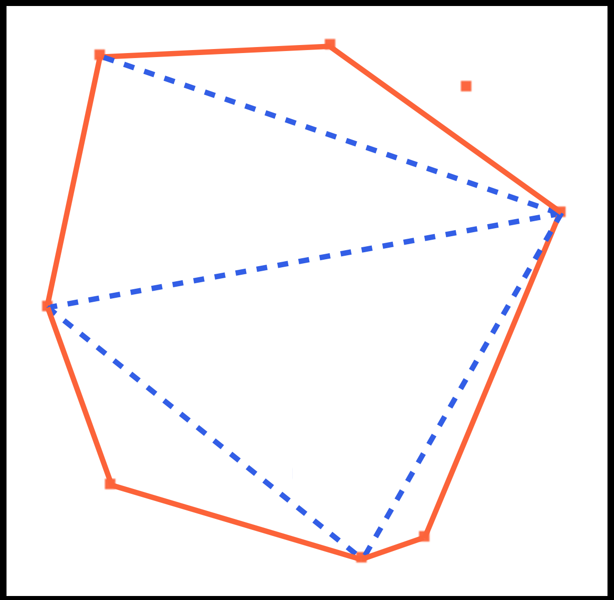
1



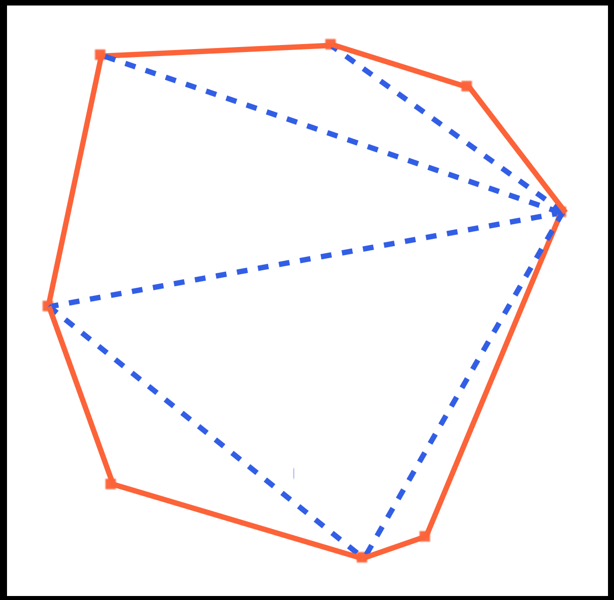
2



3



4



CONVEX HULL - FECHO CONVEXO

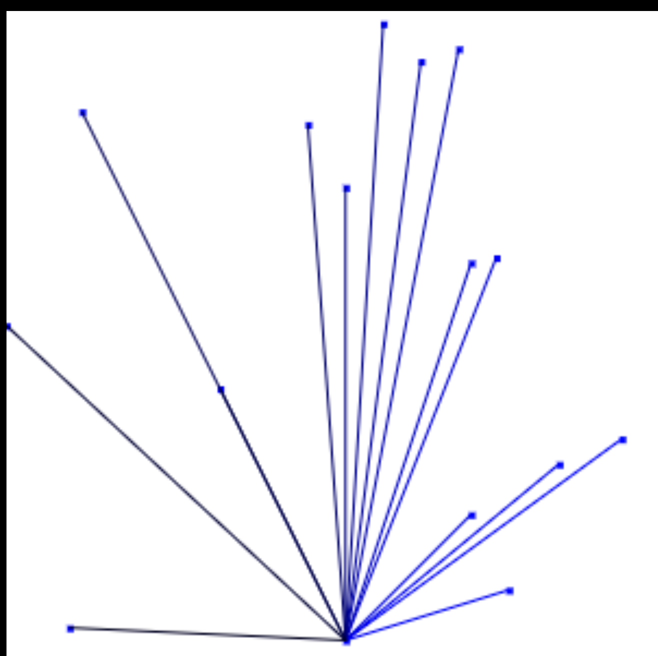
CONVEX HULL - GRAHAM SCAN

- Algoritmo Grahan Scan calcula o fecho convexo a partir do ponto com menor y e depois incorporando o próximo *ponto convexo* ao fecho.
- É importante, entretanto, que os pontos estejam ordenados por ângulo em relação ao ponto inicial.
- Algoritmo:
  1. Obtém ponto com menor y.
  2. Ordena demais pontos pelo ângulo formado com o ponto selecionado.
  3. Para o próximo ponto:
     1. se ele forma um ângulo convexo (sentido anti-horário) com o próximo e o ponto anterior, mantém o ponto no fecho convexo
     2. se o ângulo for "côncavo" (sentido horário), então deve ser removido.
        1. Recalcular o item 3 retroativamente para remover pontos que, com a nova configuração ficaram “côncavos".

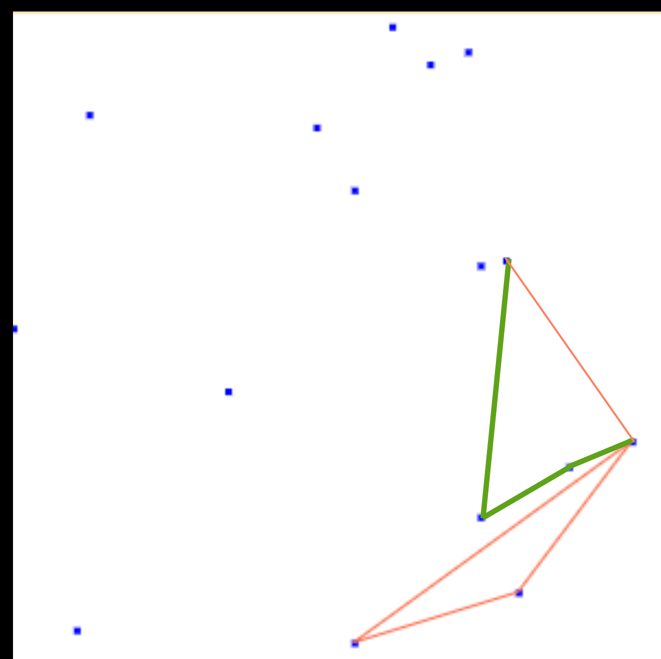
COMPLEXIDADE O(NLOGN)

CONVEX HULL - FECHO CONVEXO

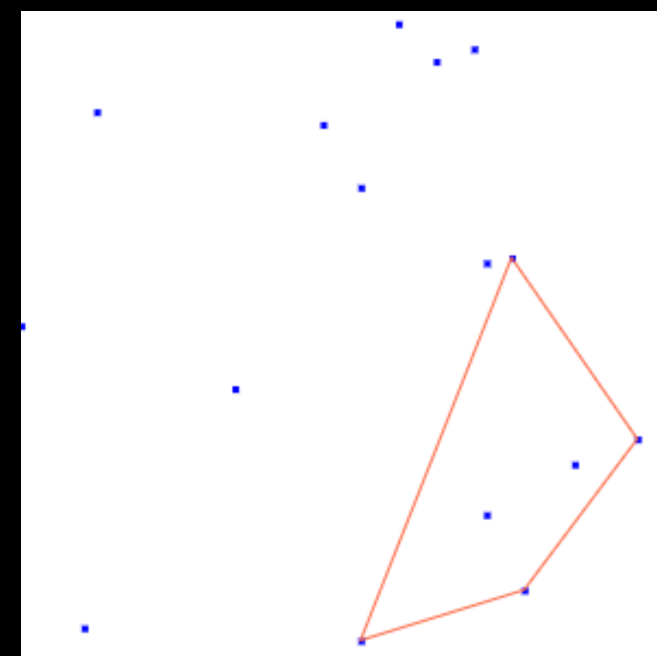
CONVEX HULL - GRAHAM SCAN



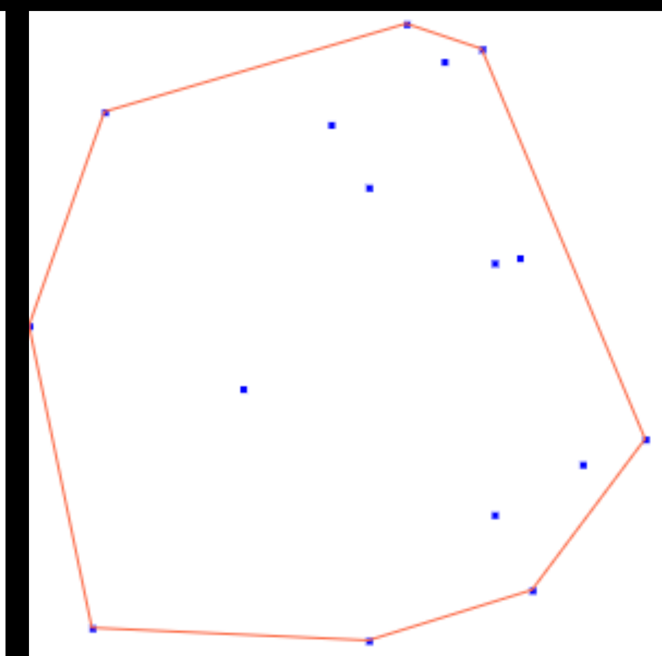
Define ponto com menor y e ordena demais pelo ângulo



Adiciona Pino convex hull. verifica ângulo de Pi com Pi-1 e Pi+1. Se ele formar um ângulo “côncavo" elimina o ponto.



Refaz o processo de verificação até que o ângulo satisfaça o convexo full



Ao final do processo retorna ao ponto inicial e todos pontos que ficaram fazem parte do convex hull, já na ordem certa.