# Geometria Computacional
# Fecho Convexo

*Claudio Esperança*

*Paulo Roma Cavalcanti*





# Motivação

- O fecho convexo de um conjunto de pontos é uma aproximação simples
- Necessariamente, não ocupa mais espaço do que o próprio conjunto de pontos
  - No pior caso, polígono tem o mesmo número de vértices do próprio conjunto
- Computar o fecho convexo muitas vezes é um passo que precede outros algoritmos sobre conjuntos de pontos





# Convexidade

- Um conjunto $S$ é convexo se para quaisquer pontos $p,q \in S$ qualquer combinação convexa $r \in S$
  - Isto é, o segmento de reta $pq \in S$
- O fecho convexo (*convex hull*) de um conjunto de pontos $S$ é
  - A interseção de todos os conjuntos convexos que contêm $S$
  - O "menor" de todos os conjuntos convexos que contêm $S$
  - O conjunto de todos os pontos que podem ser expressos como combinações convexas de pontos em $S$





# Conjuntos Convexos

- Podem ter fronteiras retas ou curvas
- Podem ser fechados ou abertos
  - Isto é podem ou não conter suas fronteiras (conceito de topologia)
- Podem ser limitados ou ilimitados
  - Um semi-espaço plano ou um cone infinito são ilimitados
- O fecho convexo de um conjunto finito de pontos é limitado, fechado e cuja fronteira é linear por partes
  - Em 2D, um polígono convexo
  - Em 3D, um poliedro convexo





# Problema do Fecho Convexo

- Dado um conjunto de pontos, computar seu fecho convexo
  - Polígono é definido normalmente por uma circulação de vértices
  - Vértices são pontos *extremos*
    - Ângulos internos estritamente convexos ($< \pi$)
      - Se 3 pontos na fronteira do polígono são colineares, o ponto do meio não é considerado
- Algoritmo "força bruta"
  - Considere todos os pares de pontos $p, q \in S$
  - Se todos os demais pontos estão do mesmo lado do semi-espaço plano correspondente à reta que passa por p e q, então o segmento de reta *pq* pertence ao fecho convexo de *S*
    - (Usar o operador *orientação)*
  - Complexidade: $O(n^3)$





# Varredura de Graham (*Graham Scan*)

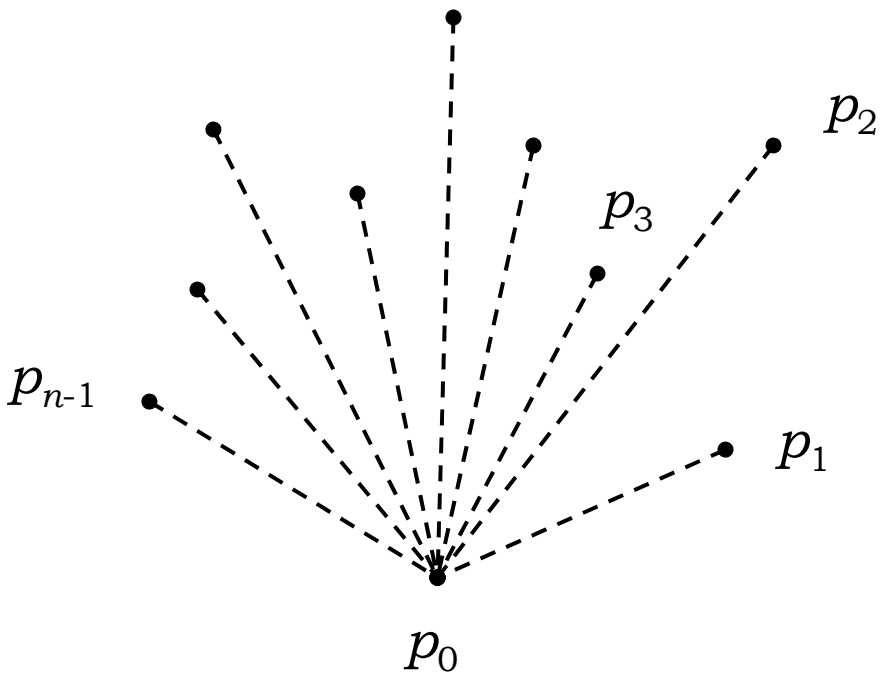
- Considerado o primeiro algoritmo de Geometria Computacional (1972)
- Algoritmo incremental (2D)
  - Pontos são pré-ordenados de forma conveniente
  - Cada ponto é adicionado ao fecho convexo e testado
- Precisamos de um ponto inicial $p_0$ que garantidamente faz parte do fecho convexo
  - Solução: Tomar o ponto com menor coordenada $x$ (ou $y$)
  - Na verdade, um ponto extremo em qualquer direção serve





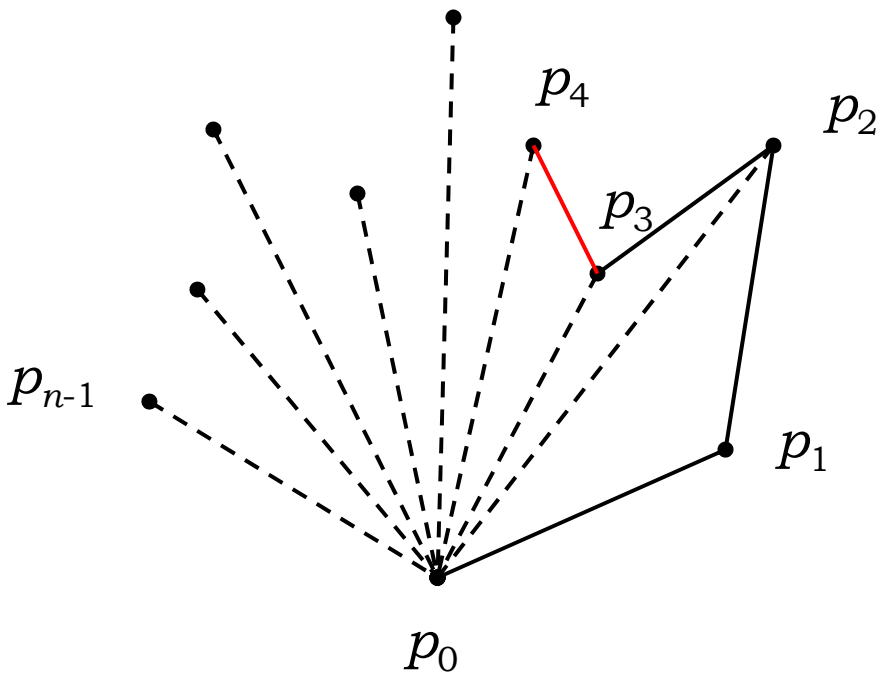
# Varredura de Graham (*Graham Scan*)

- Pontos restantes são ordenados de forma cíclica com respeito aos ângulos formados pelas retas $p_0p_i$
- Pontos colineares são removidos nesse processo



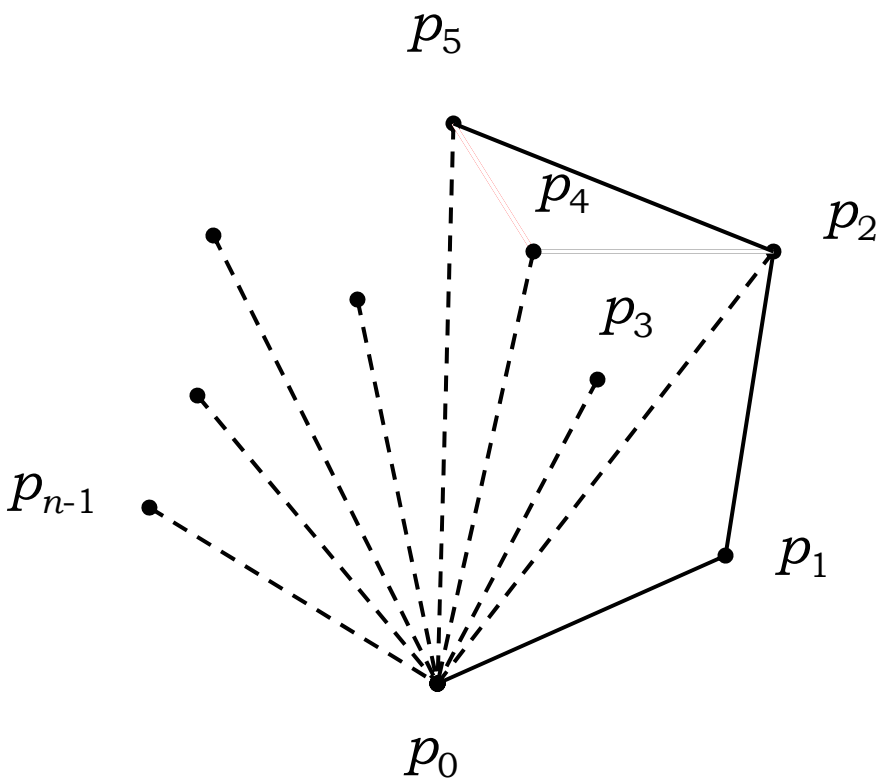
# Varredura de Graham (*Graham Scan*)

- Cada ponto considerado tem que estar à esquerda da aresta anteriormente computada do fecho convexo (teste de orientação)
  - Ou o ponto anterior faz uma curva para esquerda



# Varredura de Graham (*Graham Scan*)

- O fecho convexo é mantido como uma pilha de pontos.
- Enquanto o ponto no topo da pilha não fizer uma curva para à esquerda, quando se considera o novo ponto, ele é desempilhado
- Em seguida estende-se a cadeia empilhando-se o novo ponto



# Complexidade da Varredura de Graham

- Achar o ponto mínimo: $O(n)$
- Ordenar pontos restantes: $O(n \log n)$
- Colocar e remover cada ponto
  - Cada ponto entra no fecho convexo 1 vez
  - Cada ponto pode sair do fecho convexo no máximo 1 vez
  - Teste de orientação é $O(1)$
  - Logo, testar todos os pontos é $O(n)$
- Conclusão: Varredura é $O(n \log n)$
  - Solução de pior caso ótima





# Ordenando Via Fecho Convexo

- Crie um conjunto 2D de pontos $(x_i, x_i^2)$ sobre uma parábola e compute o seu fecho convexo
- Identifique o ponto inferior $a$ do fecho em $O$(n), ou seja, o ponto de menor $x_i$
- A ordem em que os pontos aparecem no fecho convexo no sentido anti-horário, a partir de $a$, é a ordem crescente dos $x_i$
- Logo, o fecho convexo pode ser usado para ordenar um conjunto de valores onde, sabidamente, trata-se de um problema $\Omega$ ($n \log n$)





# Algoritmo "Dividir para Conquistar"

- Técnica tradicional de projeto de algoritmos
- Semelhante ao "*MergeSort*"
- Idéia:
  - Dividir o problema em 2 subproblemas de tamanho aproximadamente igual
  - Achar (recursivamente) a solução dos subproblemas
  - Combinar as soluções dos subproblemas para obter a solução do problema





# Algoritmo "Dividir para Conquistar"

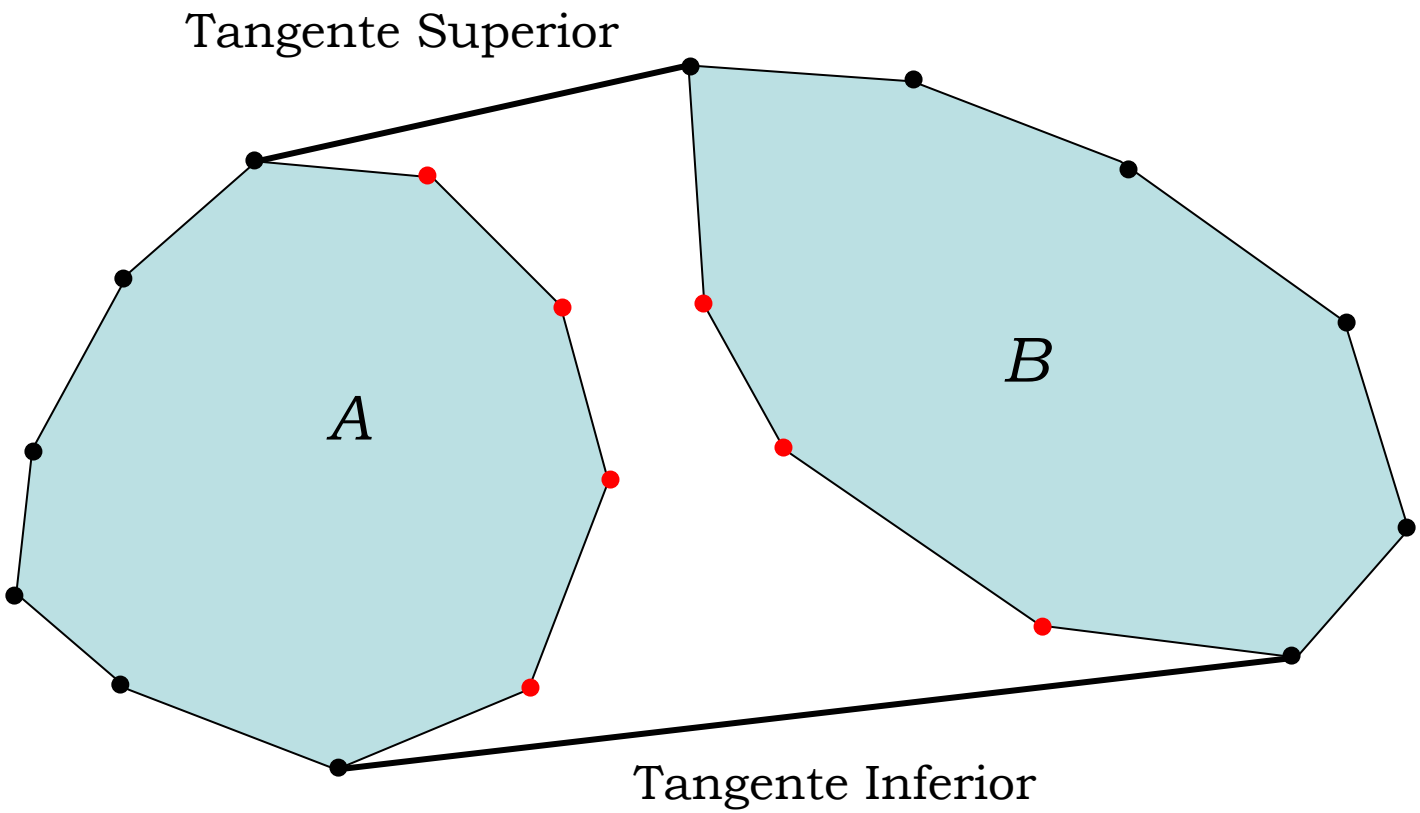
- Caso básico
  - $S$ tem 3 pontos ou menos → resolver trivialmente
- Dividir
  - Ordenar pontos por $x$ e dividir $S$ em dois subconjuntos, cada um com aproximadamente a metade dos pontos de $S$ (usar a mediana em $x$): $A$ tem os pontos com menores valores de $x$ e $B$ os com maiores valores
  - Achar recursivamente $H_A$ = Hull ($A$) e $H_B$ = Hull ($B$)





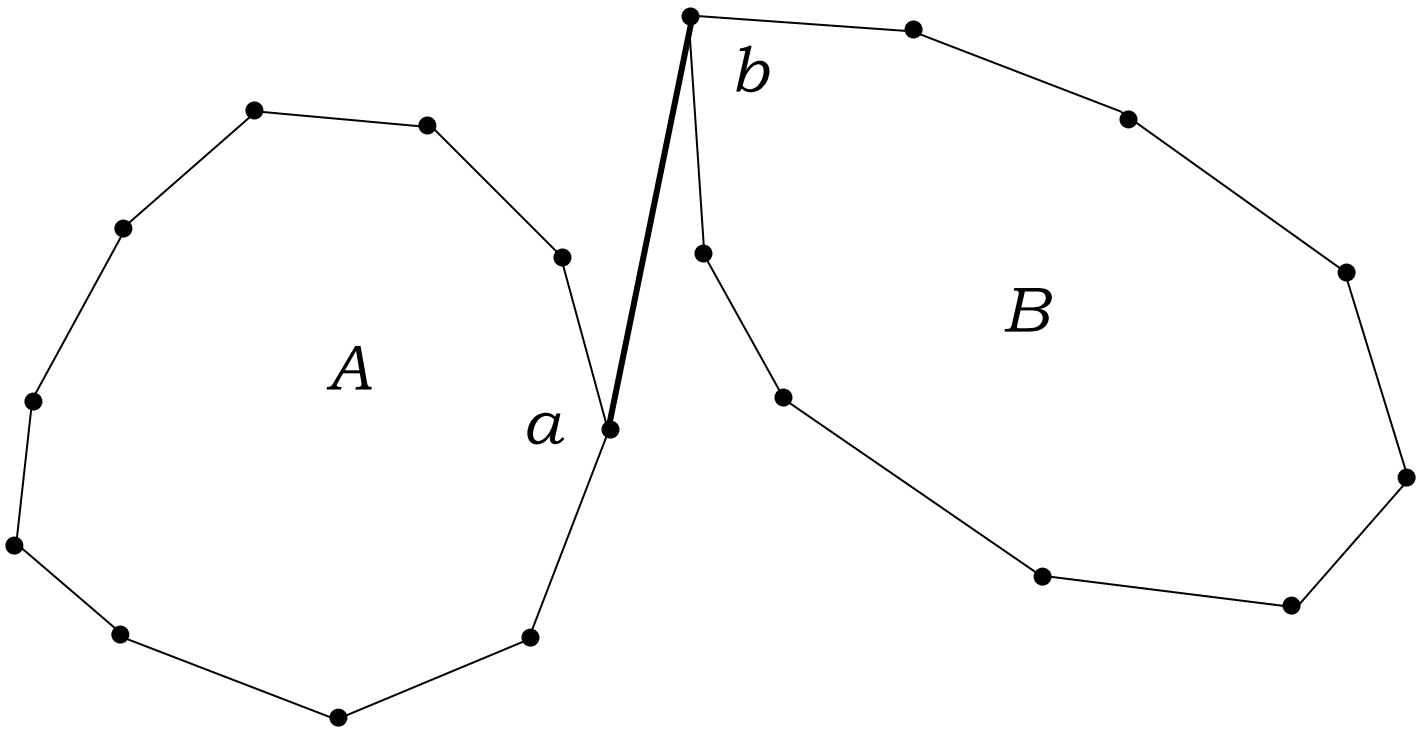
# Algoritmo “Dividir para Conquistar”

- Conquistar:
  - Computar as tangentes inferior e superior e remover os pontos entre elas



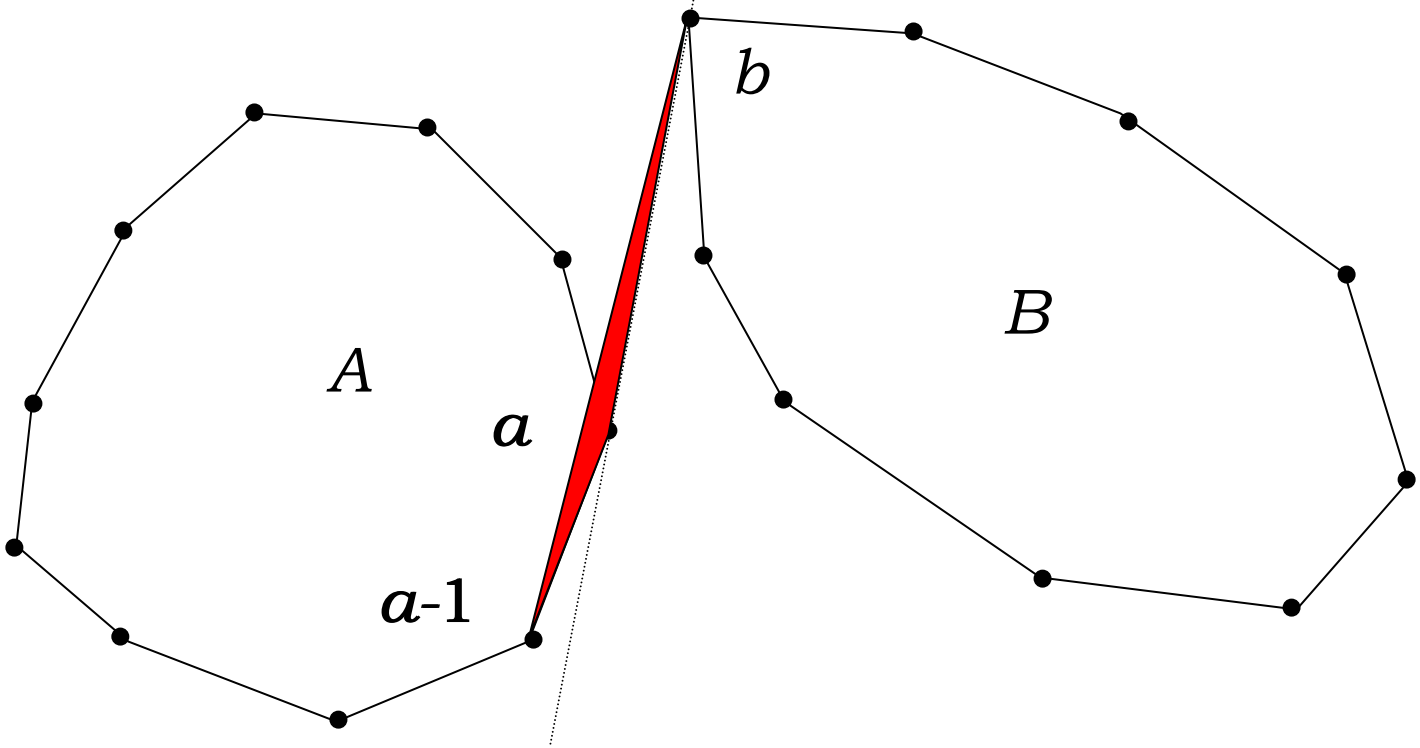
# Algoritmo “Dividir para Conquistar”

- Para computar a tangente inferior:
  - Seja $a$ o ponto mais à direita (maior $x$) de $A$
  - Seja $b$ o ponto mais à esquerda (menor $x$) de $B$



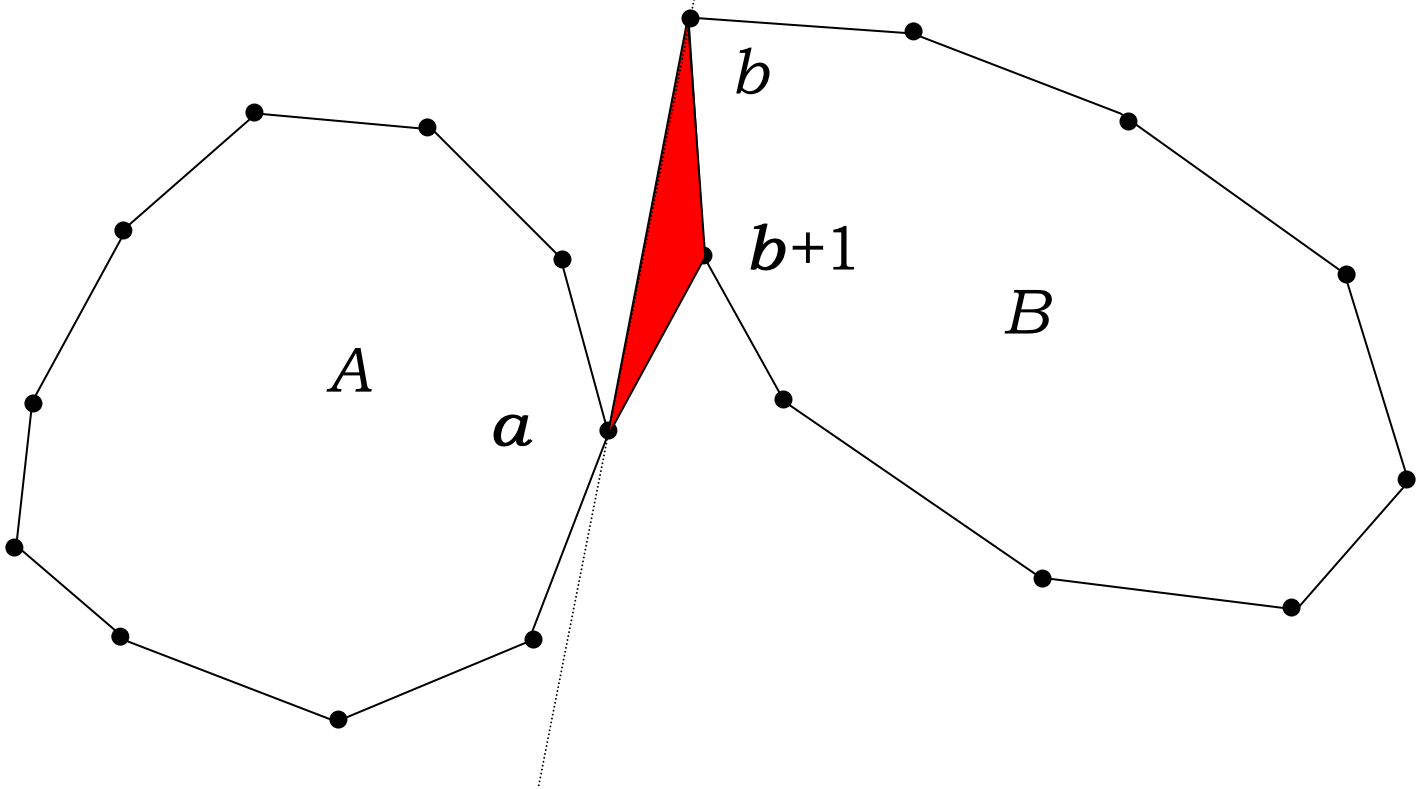
# Algoritmo "Dividir para Conquistar"

- Enquanto *ab* não for a tangente inferior
  - Se *orientação* (*a*, *a*-1, *b*) = anti-horária, então $a \leftarrow a\text{-}1$
    - *a* -1 ou *a* +1 à direita de *ab*
  - Se *orientação* (*a*, *b*, *b*+1) = horária, então $b \leftarrow b+1$
    - *b* - 1 ou *b* + 1 à direita de *ab*



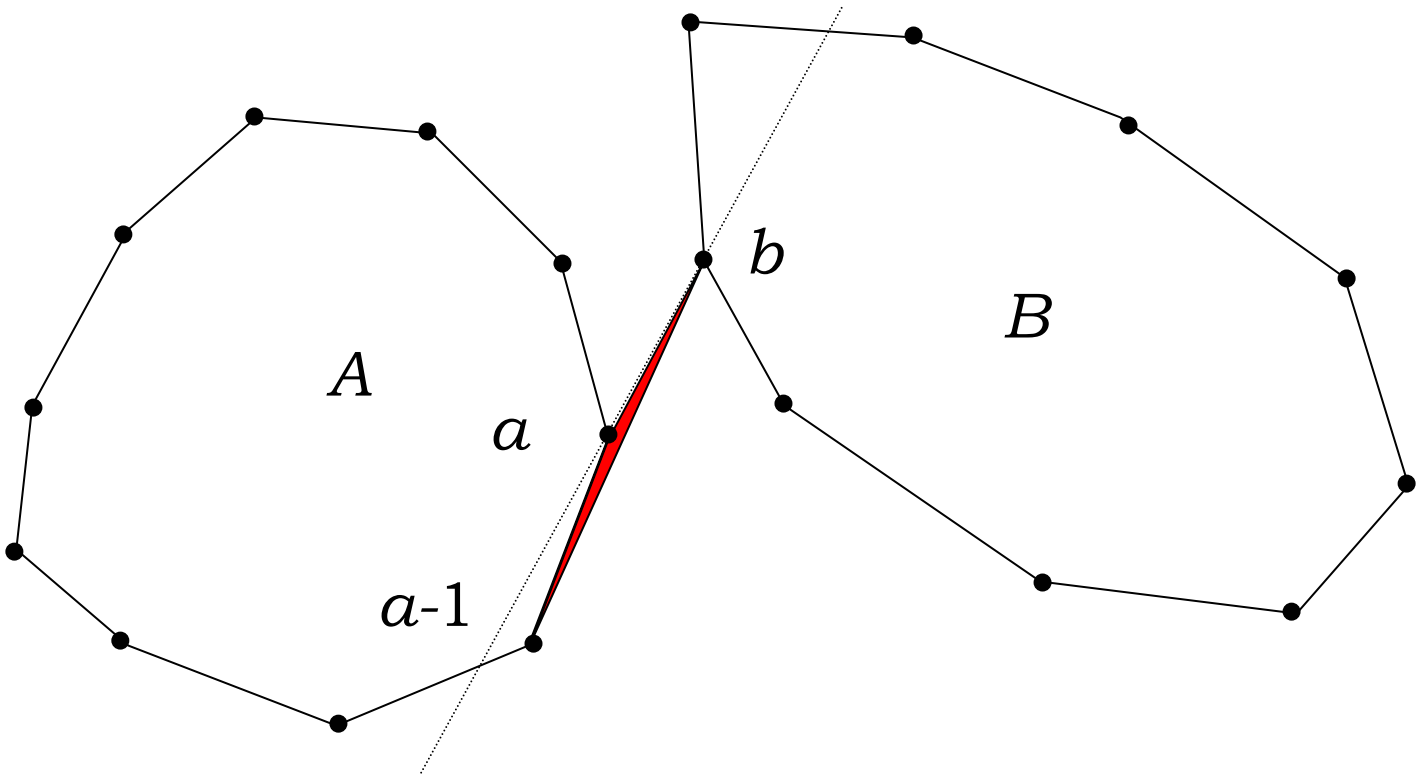
# Algoritmo "Dividir para Conquistar"

- Enquanto *ab* não for a tangente inferior
  - Se *orientação* (*a*, *a*-1, *b*) = anti-horária, então $a \leftarrow a\text{-}1$
    - *a* -1 ou *a* +1 à direita de *ab*
  - Se *orientação* (*a*, *b*, *b*+1) = horária, então $b \leftarrow b+1$
    - *b* - 1 ou *b* + 1 à direita de *ab*



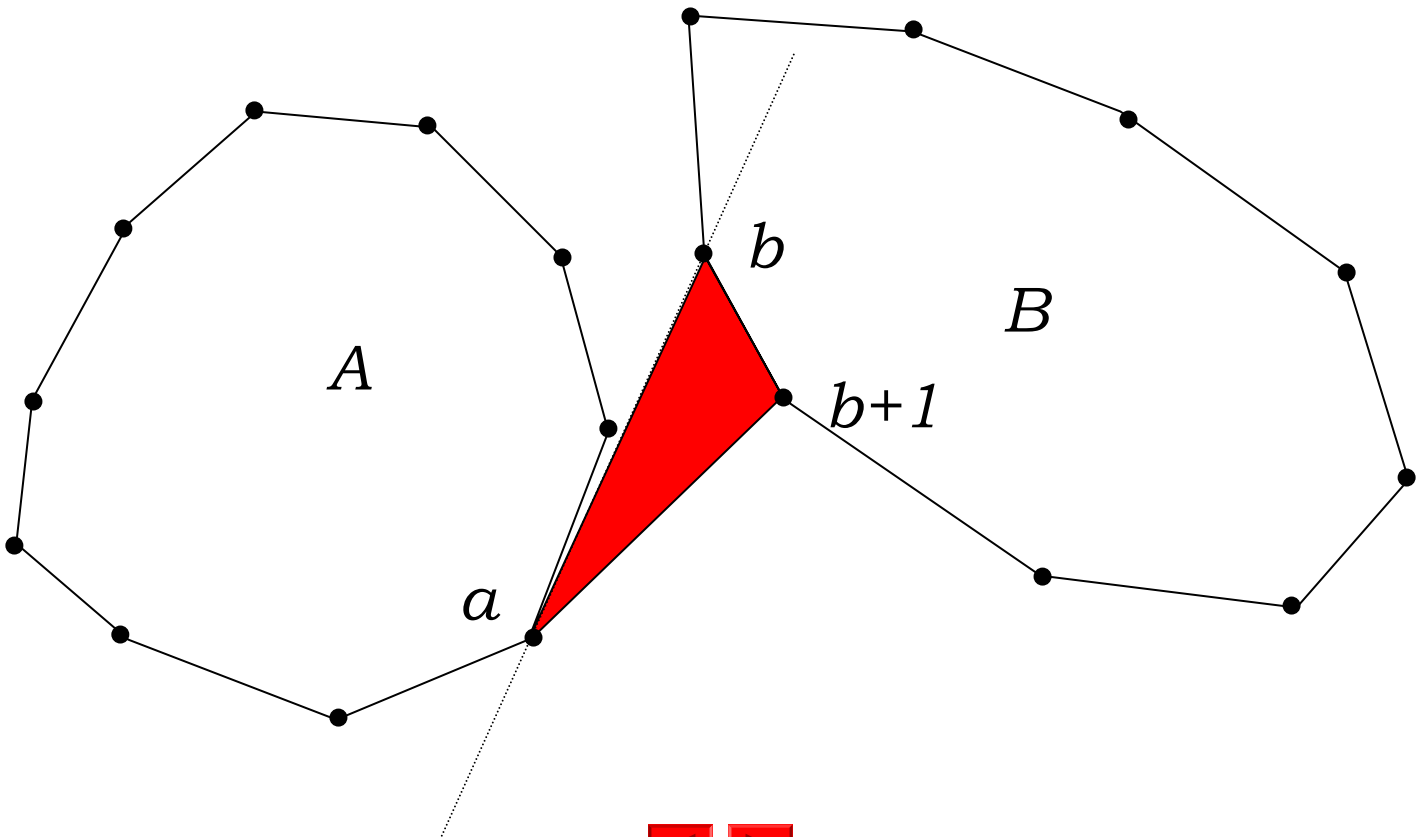
# Algoritmo "Dividir para Conquistar"

- Enquanto *ab* não for a tangente inferior
  - Se *orientação* (*a*, *a*-1, *b*) = anti-horária, então $a \leftarrow a\text{-}1$
    - *a* -1 ou *a* +1 à direita de *ab*
  - Se *orientação* (*a*, *b*, *b*+1) = horária, então $b \leftarrow b+1$
    - *b* - 1 ou *b* + 1 à direita de *ab*



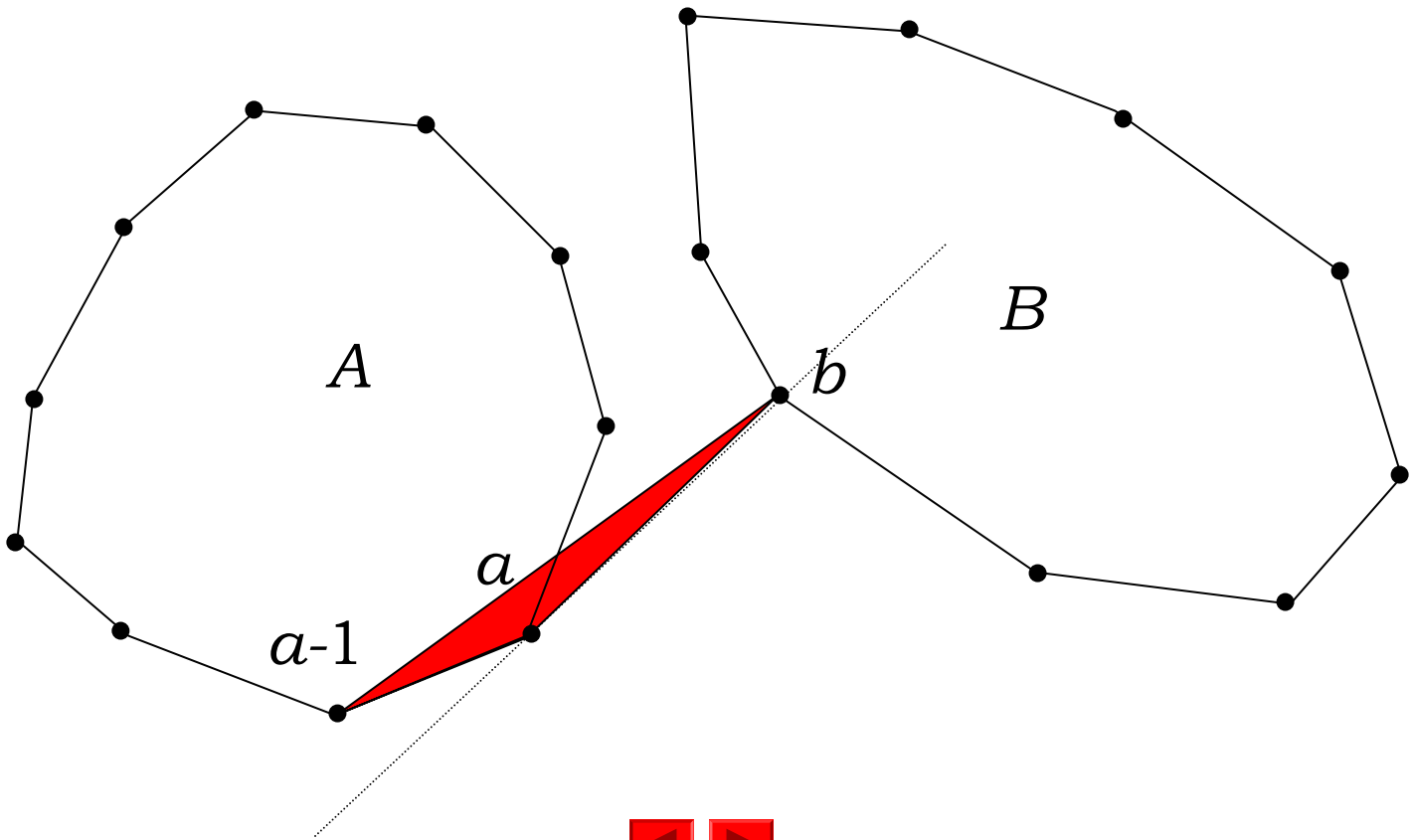
# Algoritmo "Dividir para Conquistar"

- Enquanto *ab* não for a tangente inferior
  - Se *orientação* (*a*, *a*-1, *b*) = anti-horária, então $a \leftarrow a\text{-}1$
    - *a* -1 ou *a* +1 à direita de *ab*
  - Se *orientação* (*a*, *b*, *b*+1) = horária, então $b \leftarrow b+1$
    - *b* - 1 ou *b* + 1 à direita de *ab*



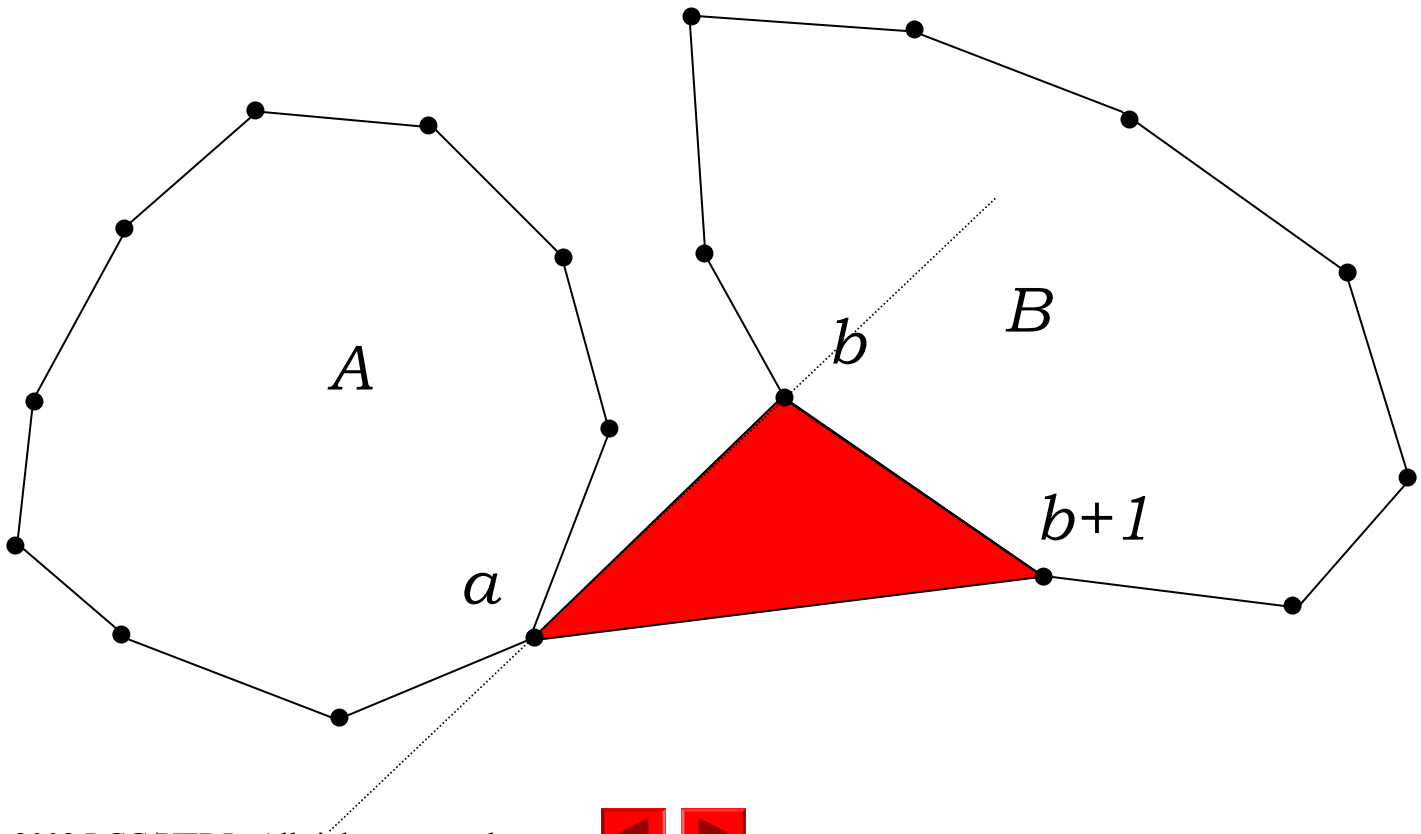
# Algoritmo "Dividir para Conquistar"

- Enquanto *ab* não for a tangente inferior
  - Se *orientação* (*a*, *a*-1, *b*) = anti-horária, então $a \leftarrow a\text{-}1$
    - *a* -1 ou *a* +1 à direita de *ab*
  - Se *orientação* (*a*, *b*, *b*+1) = horária, então $b \leftarrow b+1$
    - *b* - 1 ou *b* + 1 à direita de *ab*



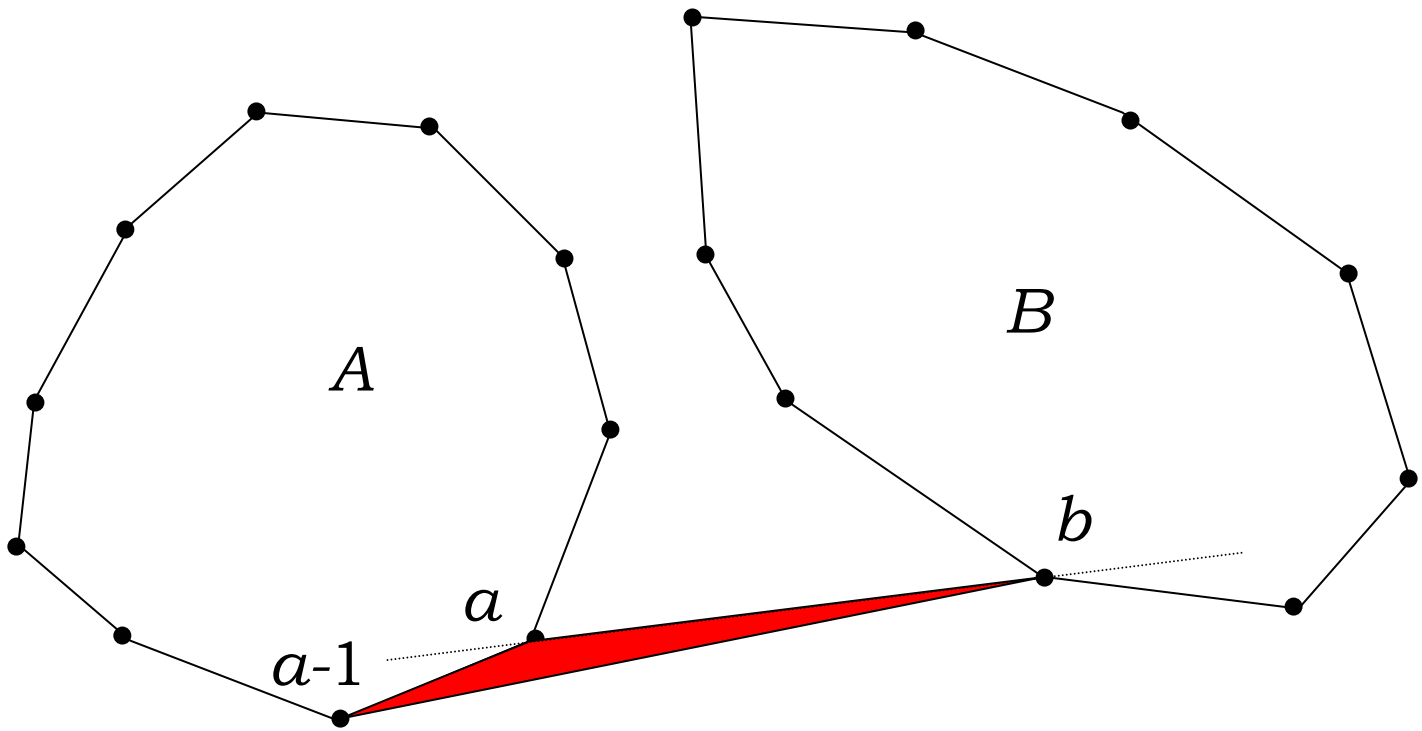
# Algoritmo "Dividir para Conquistar"

- Enquanto *ab* não for a tangente inferior
  - Se *orientação* (*a*, *a*-1, *b*) = anti-horária, então $a \leftarrow a\text{-}1$
    - *a* -1 ou *a* +1 à direita de *ab*
  - Se *orientação* (*a*, *b*, *b*+1) = horária, então $b \leftarrow b+1$
    - *b* - 1 ou *b* + 1 à direita de *ab*



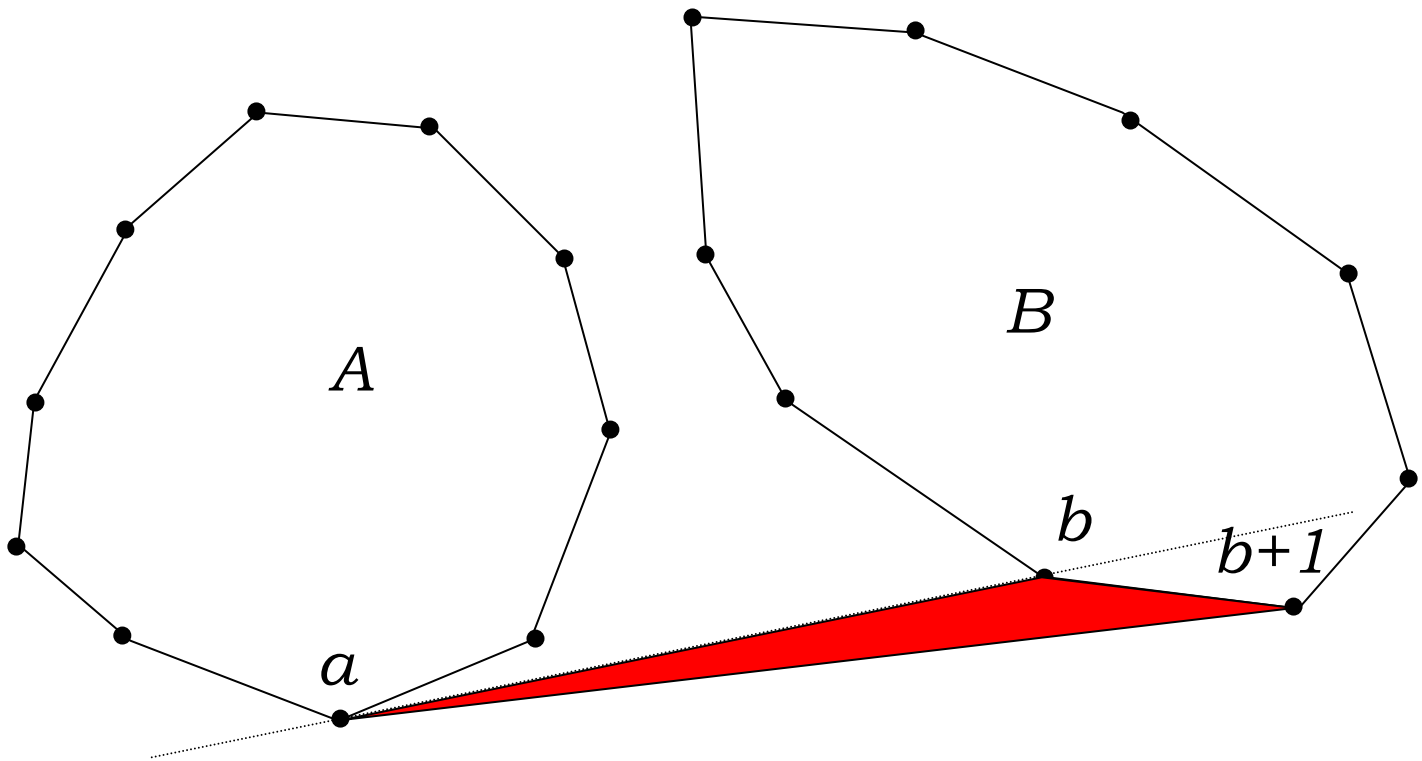
# Algoritmo "Dividir para Conquistar"

- Enquanto *ab* não for a tangente inferior
  - Se *orientação* (*a*, *a*-1, *b*) = anti-horária, então $a \leftarrow a\text{-}1$
    - *a* -1 ou *a* +1 à direita de *ab*
  - Se *orientação* (*a*, *b*, *b*+1) = horária, então $b \leftarrow b+1$
    - *b* - 1 ou *b* + 1 à direita de *ab*



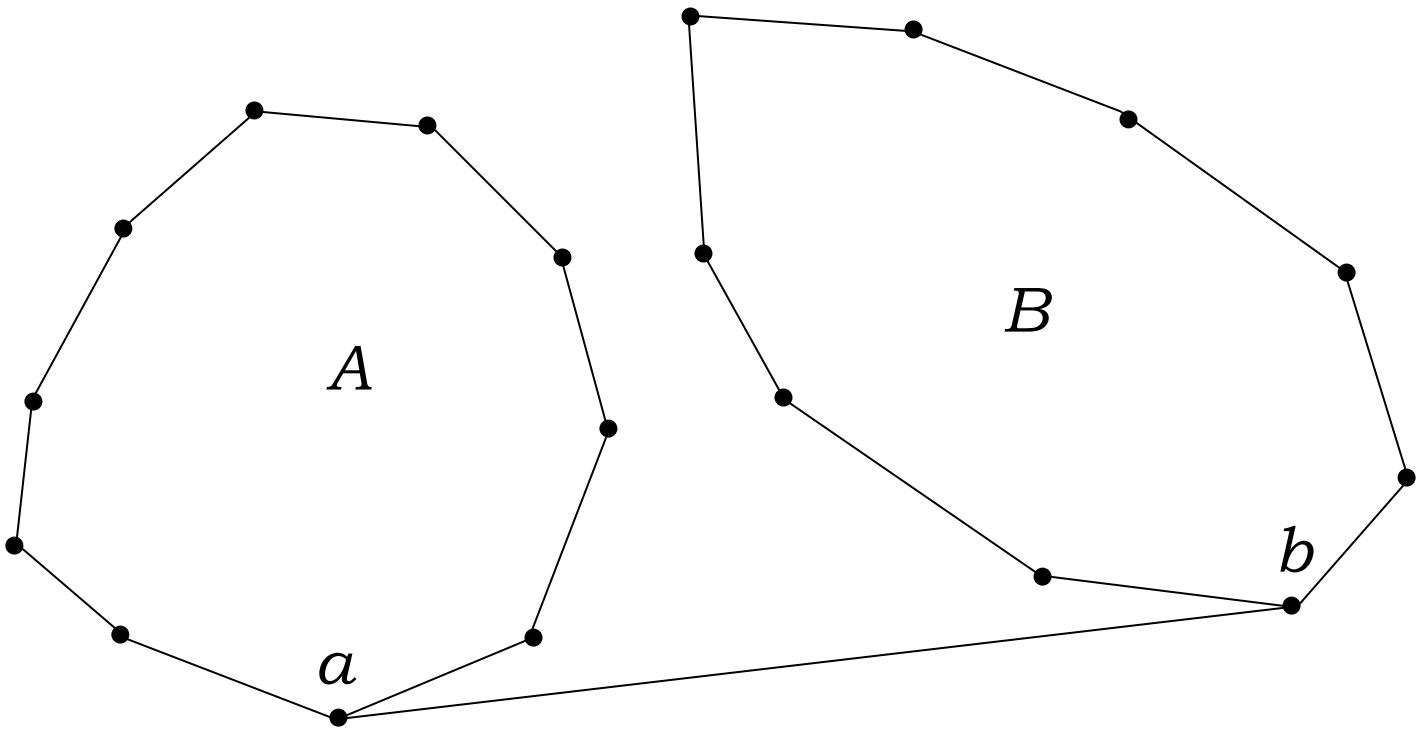
# Algoritmo "Dividir para Conquistar"

- Enquanto *ab* não for a tangente inferior
  - Se *orientação* (*a*, *a*-1, *b*) = anti-horária, então $a \leftarrow a\text{-}1$
    - *a* -1 ou *a* +1 à direita de *ab*
  - Se *orientação* (*a*, *b*, *b*+1) = horária, então $b \leftarrow b+1$
    - *b* - 1 ou *b* + 1 à direita de *ab*



# Algoritmo "Dividir para Conquistar"

- Enquanto *ab* não for a tangente inferior
  - Se *orientação* (*a*, *a*-1, *b*) = anti-horária, então $a \leftarrow a\text{-}1$
    - *a* -1 ou *a* +1 à direita de *ab*
  - Se *orientação* (*a*, *b*, *b*+1) = horária, então $b \leftarrow b+1$
    - *b* - 1 ou *b* + 1 à direita de *ab*



# Algoritmo "Dividir para Conquistar"

- Observações
  - Polígono representado por lista circular de vértices com circulação anti-horária
    - "$a \leftarrow a - 1$" deve ser interpretado como "vértice seguinte a $a$ no sentido horário"
  - Cálculo da tangente superior é feito de forma análoga ao da tangente inferior
  - A remoção dos pontos entre as tangentes é feita de forma trivial uma vez calculadas as tangentes





# "Dividir para Conquistar" - Complexidade

- Algoritmo consiste de uma etapa de ordenação mais uma chamada a uma rotina recursiva
- Etapa de ordenação = $O(n \log n)$
- Rotina recursiva:
  - O trabalho feito localmente (sem contar as chamadas recursivas) consiste de
    - Dividir $S$ em 2 subconjuntos: $O(n)$
    - Achar as 2 tangentes: $O(n)$
      - Cada vértice é analisado no máximo 2 vezes
    - Remover vértices entre as tangentes: $O(n)$
  - Complexidade da rotina é dada então pela recorrência

$$T(n) = \begin{cases} 1 & \text{para } n \leq 3 \\ n + 2T(n/2) & \text{para } n > 3 \end{cases}$$

  - A solução desta recorrência (mesma que a do *MergeSort*) resulta em $T(n) \in O(n \log n)$





# QuickHull

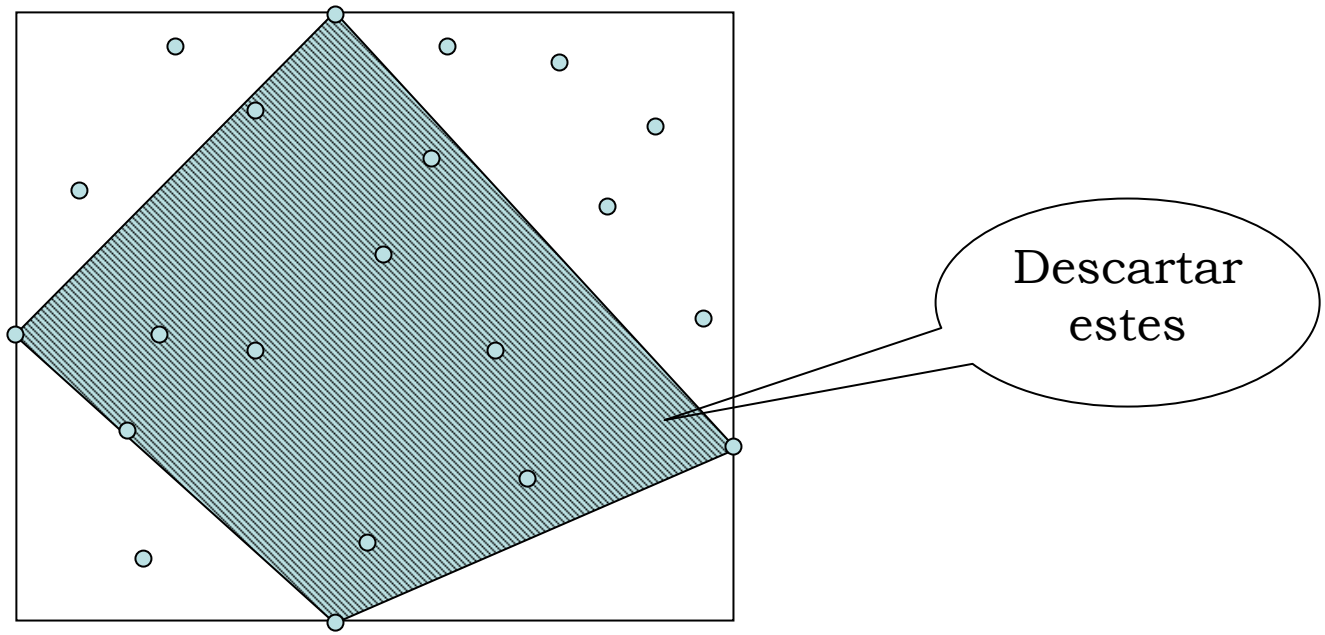
- Está para o *QuickSort* assim como o método "dividir para conquistar" está para o *MergeSort*
- Como o *QuickSort*, tem complexidade $O(n \log n)$ para entradas favoráveis. Porém, no pior caso, tem complexidade $O(n^2)$
  - Diferentemente do *QuickSort*, não se conhece um algoritmo randomizado que tenha complexidade esperada $O(n \log n)$
  - Na prática, entretanto, o desempenho é muito bom na maioria dos casos
- A idéia principal do QuickHull é descartar rapidamente pontos que obviamente estão no interior do fecho
  - Por exemplo, se os pontos são distribuídos uniformemente num quadrado, prova-se que o número de vértices do fecho é $O(\log n)$





# QuickHull

- Inicialmente, o algoritmo acha 4 pontos extremos (máximo e mínimo em $x$ e $y$) que garantidamente fazem parte do fecho convexo e descarta os pontos no interior do quadrilátero



# QuickHull

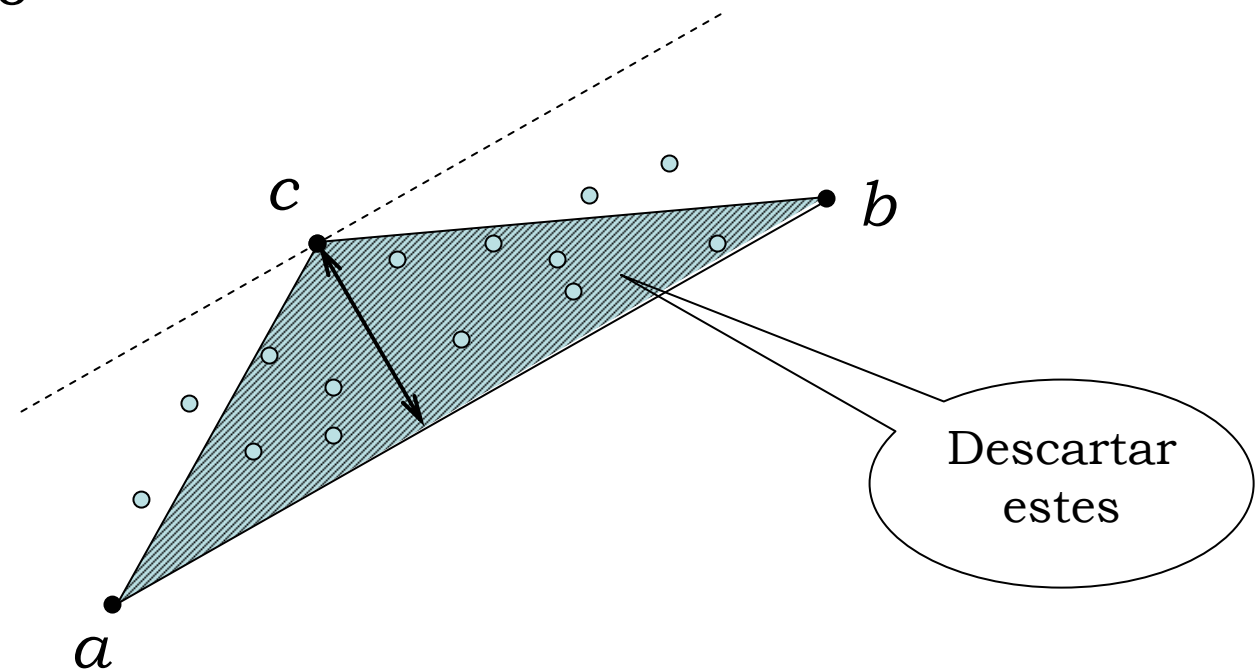
- Vemos que os pontos não descartados podem ser divididos em 4 grupos, cada um associado a uma aresta
  - Diz-se que esses pontos são "testemunhas" de que o segmento não é uma aresta do fecho convexo
  - Se não há "testemunhas", então o segmento é aresta do fecho
  - Em geral, sempre temos os pontos não descartados em grupos associados a segmentos de reta





# QuickHull

- Para cada segmento *ab*, o algoritmo prossegue elegendo um ponto *c* do grupo que se sabe ser um vértice do fecho convexo
  - O método mais usado consiste em escolher o ponto mais distante da reta de suporte do segmento



# QuickHull

- Uma vez escolhido o ponto *c*, os demais pontos precisam ser classificados
  - Se *orient* (*a*,*c*,*p*) ou *orient* (*c*,*b*,*p*) = *colinear*
    - *p* sobre aresta → descartar
  - Se *orient* (*a*,*c*,*p*) = *orient* (*c*,*b*,*p*)
    - *p* dentro do triângulo → descartar
  - Senão,
    - Se *orient* (*a*,*c*,*p*) = anti-horário
      - *p* "fora" da aresta *ac*
    - Se *orient* (*c*,*b*,*p*) = anti-horário
      - *p* "fora" da aresta *cb*
- O algoritmo é aplicado recursivamente aos grupos das novas arestas assim formadas





# Complexidade do QuickHull

- Operações feitas localmente em cada chamada da rotina recursiva
  - Determinar um ponto *c* no fecho convexo: $O(n)$
  - Classificar os demais pontos: $O(n)$
- Tempo total do algoritmo é dado então pela recorrência

$$T(n) = \begin{cases} 1 & \text{para } n = 1, \text{ ou} \\ n + T(n_1) + T(n_2) & \text{onde } n_1 + n_2 \leq n \end{cases}$$

  - $n_1$ e $n_2$ correspondem ao número de pontos "fora" em cada grupo





# Complexidade do QuickHull

- Vemos portanto que a complexidade depende da distribuição de $n_1$ e $n_2$
  - Se a partição é bem balanceada
    - max ($n_1$ , $n_2$) ≤ $\alpha n$ para algum $\alpha < 1$
  - Ou se uma fração fixa de pontos é descartada em cada passo
    - $n_1 + n_2 \leq \alpha n$ para algum $\alpha < 1$,
  - Então a solução da recorrência é $O(n \log n)$
  - Caso contrário, temos um algoritmo que tem complexidade $O(n^2)$
- O algoritmo é bastante rápido quando os pontos seguem uma distribuição aproximadamente uniforme
  - Nesses casos, a convergência é rápida e o algoritmo bate a varredura de Graham