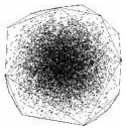
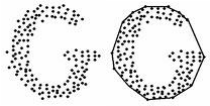


# 5 . Invólucros Convexos no Plano (cont...)

**Antonio Leslie Bajuelos**
Departamento de Matemática
Universidade de Aveiro



Mestrado em Matemática e Aplicações

## Algoritmo Quickhull

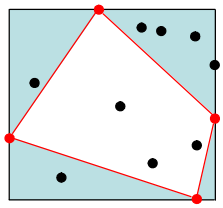
(dividir para conquistar)

- Foi proposto independentemente por várias pessoas quase ao mesmo tempo.
- Devido à semelhança com o algoritmo ***QuickSort***, o algoritmo que estudaremos foi baptizado de ***QuickHull*** por Preparata e Shamos.
- **Ideia básica do algoritmo:**

  ***"para a maioria dos conjuntos de pontos é mais fácil descartar muitos pontos que estão no interior do invólucro convexo e concentrar o trabalho nos pontos que estão próximos da fronteira"***

# Algoritmo Quickhull
(dividir para conquistar)

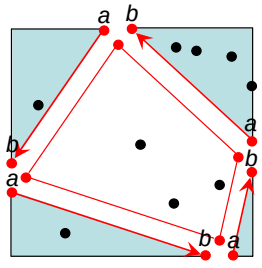
- **Pré-Processamento**
  - Heurística simples e eficiente (S. Aki e G. Toussaint, 1978).
    - **Excluir (rapidamente) pontos que, de certeza, não fazem parte do invólucro convexo**
      - **Heurística Nº1:** Encontrar os pontos extremos nas direcções: mais alta, mais baixa, mais à esquerda e mais à direita (em caso de empate escolha, por exemplo, os pontos com a menor *x*-coordenadas ou *y*-coordenadas).
      - A seguir podemos excluir os pontos interiores ao quadrilátero formado pelos pontos encontrados anteriormente, por não fazerem parte do invólucro convexo.





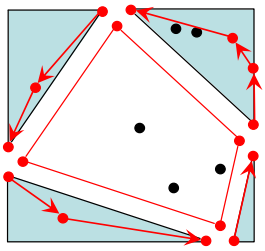
# Algoritmo Quickhull
(dividir para conquistar)

- **Dividir**
  - o nosso problema reduz-se a quatro subproblemas independentes: ***quatro regiões triangulares exteriores ao quadrilátero***



## Algoritmo Quickhull
(dividir para conquistar)

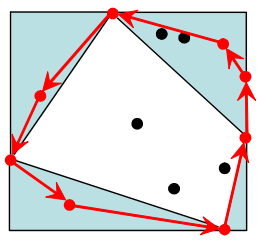
- **Conquistar**
  - Determinar o **invólucro convexo** dos pontos contidos nas ***quatro regiões triangulares exteriores ao quadrilátero***





## Algoritmo Quickhull
(dividir para conquistar)

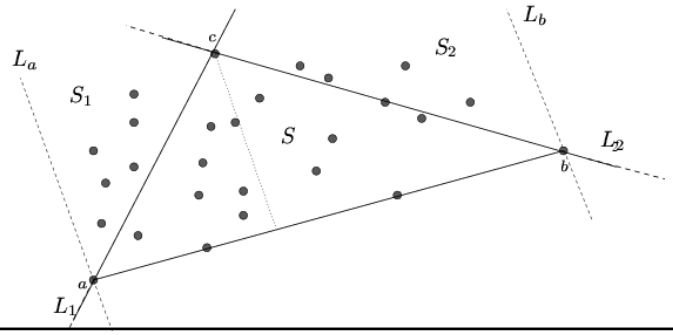
- **Combinar**
  - Concatenar as **quatro cadeias correspondentes às quatro regiões** obtidas para obter o **conv(*S*)**



# Algoritmo Quickhull

(dividir para conquistar)

- **Execução da fase Conquistar**
  - Seja *S* o conjunto de pontos de um desses subconjuntos. Sejam *a* e *b* os pontos extremos de *S* que formam a base do futuro triângulo.
  - Basta procurar o ponto extremo *c* de *S* e descartar os pontos que estão no interior do $\Delta(a,b,c)$. Depois disto o nosso problema fica dividido em dois problemas idênticos menores ($S_1$ e $S_2$)

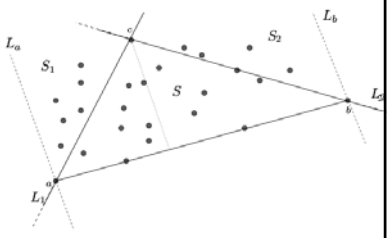



# Algoritmo Quickhull

(dividir para conquistar)

- **Execução da fase Conquistar (cont…)**
  - Como determinar o ponto *c*?
    - **Escolher o ponto mais distante da recta de suporte do segmento *ab*.**
  - Seguidamente classificamos os outros pontos.

    Seja *p* um ponto de *S*
    - Se $A(\Delta(a,c,p)) = 0 \vee A(\Delta(c,b,p)) = 0$
      - *p* sobre a aresta → descartar
    - Se $A(\Delta(a,c,p)) < 0 \wedge A(\Delta(c,b,p)) < 0$
      - *p* dentro do triângulo → descartar
    - Senão,
      - Se $A(\Delta(a,c,p)) > 0$
        - *p* "fora" da aresta *ac*
      - Se $A(\Delta(c,b,p)) > 0$
        - *p* "fora" da aresta *cb*




  - **O algoritmo é aplicado recursivamente aos grupos das novas arestas assim formadas.**

# Algoritmo Quickhull

## resolução de cada sub-problema

- Dados de Entrada: dois pontos distintos a e b e um conjunto finito $S' = \{p_1,\ldots,p_k\}$ de $k$ pontos no plano tal que todos os pontos de $S'$ estão à direita de ab
- Dados de Saída: as arestas que definem conv($S$), orientadas no sentido positivo
- Algoritmo:

  if $S' = \varnothing$
  then return ab
  else c ← ponto com distância máxima
  $S_1$ ← pontos à direita de ca
  $S_2$ ← pontos à direita de bc
  return Quickhull (a,c,$S_1$) concatenado com Quickhull (c,b,$S_2$)

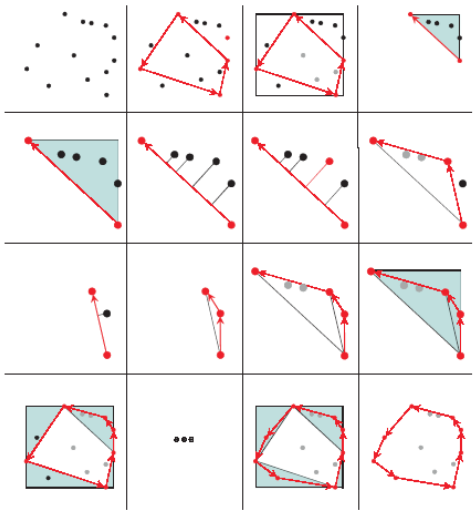
- **Da união dos vértices resultantes dos 4 sub-problemas iniciais resulta o invólucro convexo de S**



# Algoritmo Quickhull

## (dividir para conquistar)

- Exemplo da execução do Algoritmo de **Quickhull**



# Algoritmo Quickhull

(dividir para conquistar)

## Análise:

- Encontrar os quatro pontos extremos que formam o quadrilátero inicial durante o pré-processamento é $O(n)$ operações elementares onde $n$ é o número de pontos de $S$.
- Então a complexidade do algoritmo só depende do número de pontos no conjunto $S_1$ e $S_2$. Seja $|S_1| = \alpha$ e $|S_2| = \beta$.
- Se $T(n)$ - função complexidade de tempo do algoritmo **QuickHull** para encontrar o **conv(*S*)** com $n$ pontos teremos que $T(n) \leq T(\alpha) + T(\beta) + cn$, onde o termo $cn$ da desigualdade corresponde ao número de operações gastas pelo algoritmo para encontrarmos o ponto $c$ e construirmos $S_1$ e $S_2$
- Quando $\alpha + \beta = n$ e cada sub-problema tem aproximadamente o mesmo tamanho teremos que: $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn = O(n \log n)$
  - $T(n) = O(n \log n)$ é o melhor que pode ocorrer se tivermos $\alpha + \beta = n$.
- No pior caso, a sua complexidade de tempo é $O(n^2)$, isto ocorre quando $\alpha + \beta = n$ e $\alpha = 1$ e $\beta = n - 1$
  - Nesse caso a nossa recorrência toma a forma: $T(n) \leq T(n\text{-}1) + cn = T(n\text{-}2) + c(n\text{-}1) + cn$. Repetindo a expansão nos chegaremos a: $T(n) \leq c + c2 + \ldots + c(n\text{-}2) + c(n\text{-}1) + cn = O(n^2)$.



# Algoritmo Quickhull

(dividir para conquistar)

## Exercício:

Analisar a complexidade da seguinte função que representa um algoritmo recursivo, típicos de problemas resolvidos pelo método dividir-para conquistar. Para o caso geral n > 1, o processo divide o problema em duas metades de aproximadamente o mesmo tamanho. Quando este processo termina necessita um tempo linear para combinar os resultados obtidos nas duas metades

Solução:

$$T(n) = \begin{cases} 2\,T(n/2) + c_2 n & \text{se } n > 1 \\ d & \text{se } n \leq 1 \end{cases}$$

Para $n > 2$ temos que $T(n/2) = 2T(n/4) + c_2 n/2$

Substituindo em T(n): $T(n) = 2(2T(n/2^2) + c_2 n/2) + c_2 n = 2^2 T(n/2^2) + 2\, c_2 n$

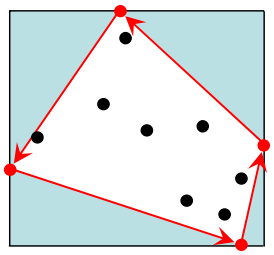
Para o termo i temos que $T(n) = 2^i T(n/2^i) + i\, c_2 n$

Deve existir k tal que $2^k \cong n$; $k = \log_2 n$

Por tanto $T(n) = nT(n/n) + k\, c_2 n = nT(1) + c_2 n \log_2 n = n + n \log_2 n \cong O(n \log n)$

# Algoritmo Quickhull
(dividir para conquistar)

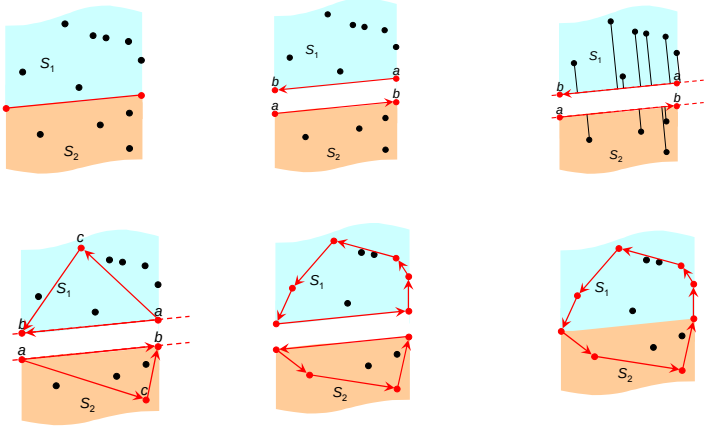
- Exemplo do melhor caso do algoritmo QuickHull





# Algoritmo Quickhull
(dividir para conquistar)

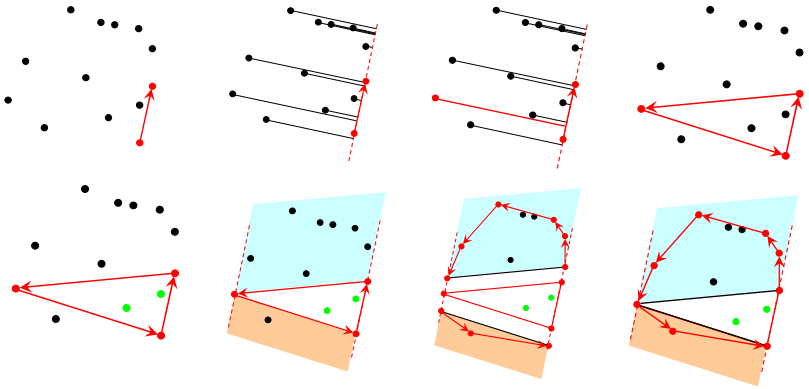
- Outra forma de iniciar o algoritmo:
  - **Heurística Nº 2:** Escolher apenas os pontos de maior e menor abcissa



# Algoritmo Quickhull

(dividir para conquistar)

- Outra forma de iniciar o algoritmo:
  - **Heurística Nº 3:** Escolher uma qualquer aresta de conv(*S*) – pode ser obtida, por exemplo, através de uma iteração do algoritmo de **Jarvis**





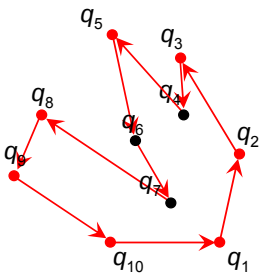
# Algoritmo de Graham

(o primeiro algoritmo óptimo)

- Segundo O'Rourke talvez o primeiro artigo na área de *Geometria Computacional* tenha sido o de Graham
- **Ideia geral do algoritmo:**
  - **1ª Fase:** consiste num pré-processamento onde primeiramente, um ponto $p_0$ de *S* é seleccionado e depois os pontos restantes são ordenados (segundo os seus ângulos) ao redor de $p_0$
  - **2ª Fase:** o algoritmo processa iterativamente os pontos produzindo uma sequência de invólucros convexos que converge para conv(*S*)

# Algoritmo de Graham
(o primeiro algoritmo óptimo)

- A etapa de **pré-processamento** (ordenação) reduz o problema de encontrar o invólucro convexo de um conjunto de pontos ao problema de encontrar o invólucro convexo de um polígono estrelado

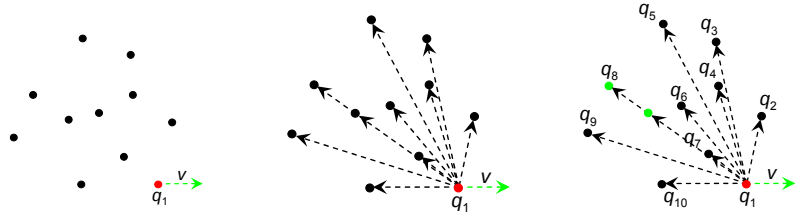





# Algoritmo de Graham
(o primeiro algoritmo óptimo)

- Dados de Entrada: um conjunto finito $S'=\{p_1,\ldots,p_n\}$ de $n$ pontos no plano
- Dados de Saída: um conjunto finito $S=\{q_1,\ldots,q_n\}$ de $n$ pontos no plano
- Algoritmo **Graham (1ª Fase):**

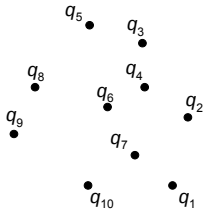
  $q_1$ ← ponto de $S$ com menor y-coordenada
  v ← vector horizontal que passa por $q_1$ e é orientado para a direita
  Ordenar os pontos $p_i$ em relação ao ângulo polar que v forma com $q_1p_i$
  Output: $\{q_1,\ldots,q_n\}$ pela ordem definida



# Algoritmo de Graham
## (o primeiro algoritmo óptimo)

- Dados de Entrada: um conjunto finito $S=\{q_1,\ldots,q_n\}$ de $n$ pontos no plano previamente ordenados
- Dados de Saída: as arestas que definem conv(S), orientadas no sentido positivo
- Algoritmo **Graham (2ª Fase):** :

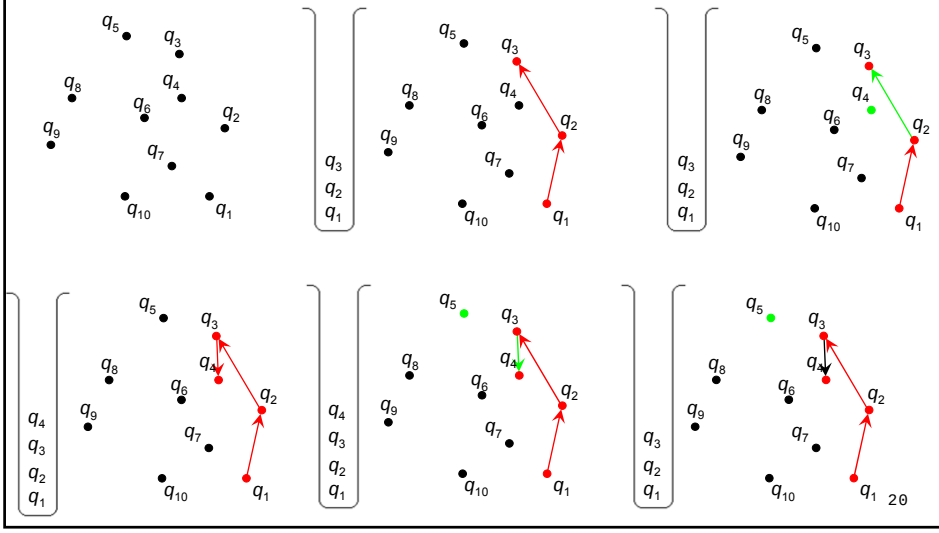
T ← ∅
Empilhar $q_1$, $q_2$ e $q_3$ (nesta ordem) na pilha T
$i \leftarrow 4$
While $i < n$ do
  $q_t$ ← ponto do topo de T
  $q_{t-1}$ ← ponto abaixo de $q_t$ em T
  If $q_i$ está à esquerda do segmento $q_{t-1}q_t$
  then empilhar $q_i$ em T
    $i \leftarrow i+1$
  else desempilhar $q_t$ de T
return T






# Algoritmo de Graham
## (o primeiro algoritmo óptimo)

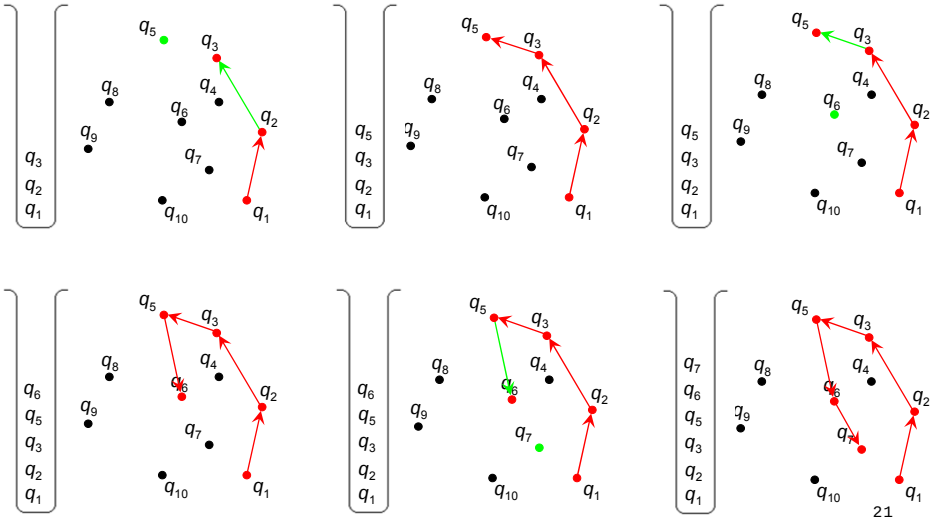
- Exemplo da execução **(2ª Fase)**



# Algoritmo de Graham

(o primeiro algoritmo óptimo)

- Exemplo da execução **(2ª Fase)**

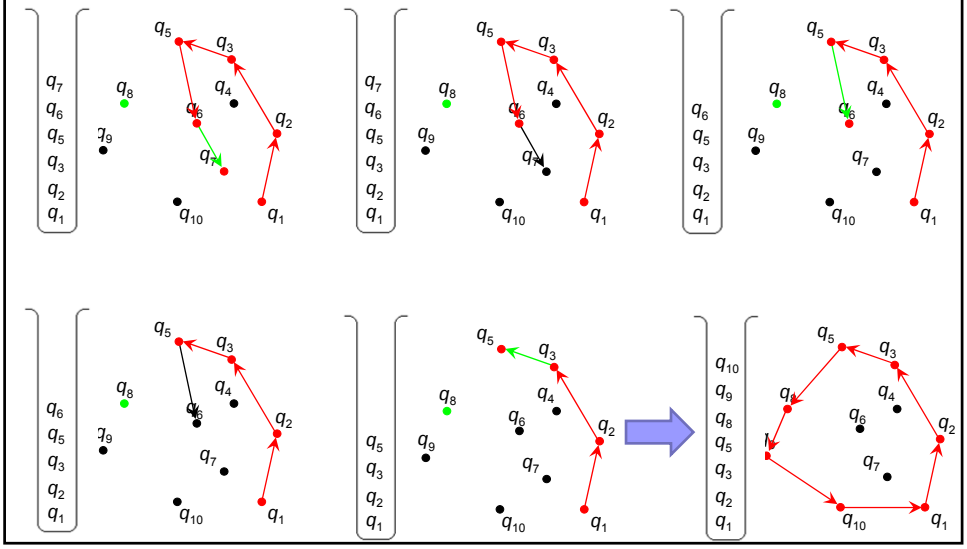





# Algoritmo de Graham

(o primeiro algoritmo óptimo)

- Exemplo da execução **(2ª Fase)**



# Algoritmo de Graham
(o primeiro algoritmo óptimo)

## Análise do algoritmo (algumas observações)

- O número de operações DESEMPILHA é menor que o número de operações EMPILHA (já que $q_1$, $q_2$ e $q_n$ nunca serão desempilhados de T).
- Como o número total de operações EMPILHA executadas durante todo o algoritmo será *n*, então o número de operações DESEMPILHA será menor que *n*.

⇨ Logo o tempo total gasto com as operações EMPILHA e DESEMPILHA ao longo da execução do algoritmo é ***O(n)***

⇨ O número total de execuções do comando WHILE será ***O(n)***



# Algoritmo de Graham
(o primeiro algoritmo óptimo)

- Complexidade: **O(*n* log *n*)**

O(*n*) → $q_1$ ← ponto de *S* com menor ordenada

O(1) → v ← vector horizontal que passa por $q_1$ e é orientado para a direita

O(*n* log *n*) → Ordenar os pontos $p_i$ em relação ao ângulo polar que v forma com $q_1p_i$

Output: $\{q_1,\ldots,q_n\}$ pela ordem definida

O(1) →
- T ← ∅
- Empilhar $q_1$, $q_2$ e $q_3$ (nesta ordem) na pilha T
- $i \leftarrow 4$

O(*n*) →
- While $i < n$ do
  - $p_t$ ← ponto do topo de T
  - $p_{t-1}$ ← ponto abaixo de $p_t$ em T
  - If $p_i$ está à esquerda do segmento $p_{t-1}p_t$
  - then empilhar $p_i$ em T
    - $i \leftarrow i + 1$
  - else desempilhar $p_t$ de T
- Return T

# Algoritmo de Graham

(o primeiro algoritmo óptimo)

**Observações relevantes:**

1. **O algoritmo de Graham é óptimo – O($n$ log $n$)**
2. Salienta-se ainda a eficiência da estrutura de dados utilizada (pilha)
3. O algoritmo de Graham não se generaliza a dimensões superiores.
4. Não é um algoritmo on-line visto que todos os pontos devem estar disponíveis no início da execução do algoritmo.
5. Quando os pontos estão em posição convexa, nenhum ponto é eliminado através do algoritmo.



# Algoritmo Incremental

- **Ideia geral:**

**Conhecendo conv($S$'), onde $S$' tem $n$-1 pontos, construir o conv($S$' ∪ { $p_n$ })**

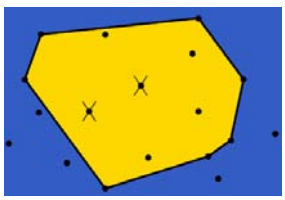
A cada novo ponto $p_n$ a considerar, pode acontecer um dos seguintes casos:

**(i)** $p_n \in \text{conv}(S')$ ⇨ $\text{conv}(S') = \text{conv}(S' \cup \{p_n\})$

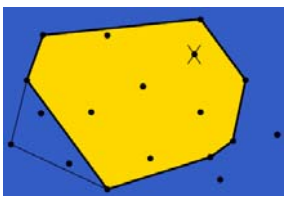
**(ii)** $p_n \notin \text{conv}(S')$ ⇨ devemos construir um novo invólucro convexo

⇨ encontrar duas rectas que passam pelo ponto $p_n$ e que são tangentes ao conv($S'$), e então modificar o invólucro convexo de maneira conveniente.

**(i)**



**(ii)**



# Algoritmo Incremental

- **Observação:**

  **Como determinar se $p \in Q$?**

  - Para decidir se $p \in Q$, podemos utilizar a função primitiva **Lefton**. Este teste pode ser feito em tempo ***O(n)***.
    - ***p* ∈ Q sse está à esquerda ou sobre cada aresta (orientada positivamente) do polígono Q**

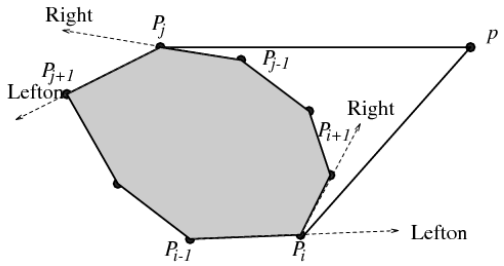


# Algoritmo Incremental

- **Observação:**

  **Como determinar, quando $p \notin Q$, os pontos de tangência com o conv(Q)?**

  - É possível utilizar o predicado **Lefton** para encontrar o ponto de tangência mais baixo de $p_i$.
    - ***p* está à esquerda de $p_{i-1}p_i$ e à direita de $p_ip_{i+1}$**




  - Para o ponto de tangência mais alto o raciocínio é análogo.
  - Este teste também pode ser feito em tempo ***O(n)***

# Algoritmo Incremental

- Dados de Entrada: Um polígono convexo Q com $m$ vértices e um ponto $p \notin Q$
- Dados de Saída: Os dois vértices de tangencia de Q em relação a $p$
- **Algoritmo Vértices Tangentes (Q, $p$):**

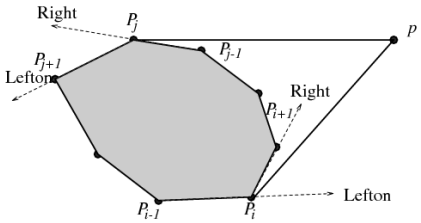
for $i = 1, \ldots, m$ do
  if Lefton($p_{i-1}$, $p_i$, $p$) $\neq$ Lefton($p_i$, $p_{i+1}$, $p$) then
    $p_i$ é um vértice de tangencia



# Algoritmo Incremental

- **Observação:**

**Como construir o novo polígono a partir do polígono Q, do ponto $p$ e dos de tangencia, por exemplo $p_i$ e $p_j$?**




Novo polígono Q = {$p_0$, $p_1$, …, $p_i$, $p$, $p_j$, $p_{j+1}$, …, $p_n$}

- Se os vértices de Q estiveram representados numa lista duplamente ligada (em sentido anti-horário) então a actualização de Q poderá ser feita através de uma simples sequencia de remoções e uma inserção

## Algoritmo Incremental

- Dados de Entrada: um conjunto finito $S=\{p_1,\ldots,p_n\}$ de $n$ pontos no plano
- Dados de Saída: as arestas que definem conv($S$), orientadas no sentido positivo
- **Algoritmo Incremental (ideia geral):**

$Q_3 \leftarrow \text{conv}(\{p_1, p_2, p_3\})$
for $k = 4, \ldots, n$ do
  $Q_k \leftarrow \text{conv}(Q_{k-1} \cup \{p_k\})$
return $Q_n$



## Algoritmo Incremental

$Q_3 \leftarrow \text{conv}(\{p_1, p_2, p_3\})$
for $k = 4, \ldots, n$ do
  $Q_k \leftarrow \text{conv}(Q_{k-1} \cup \{p_k\})$
return $Q_n$

- **Análise:**
  - Decidir se $p \in Q$ é ***O(n)***
  - Se $p \notin Q$, então os vértices de tangência podem ser encontrados em ***O(n)***
  - O número de remoções da lista circular que armazena os vértices de Q pode ser actualizada através de operações de inserções e remoções.
    - O número total de remoções é menor que o número total de inserções que é igual a $n$.

A complexidade de tempo do Algoritmo Incremental é ***O($n^2$)***

# Algoritmo MergeHull

(outro algoritmo de divisão e conquista)

- **Generalidades:**
  - O MergeHull é um *algoritmo de divisão e conquista* que encontra o invólucro convexo de um conjunto de *n* pontos no plano em tempo $O(n \log n)$
  - A técnica de divisão e conquista é a única conhecida que ao ser estendida para o problema do invólucro convexo tri-dimensional atinge a mesma complexidade, i.e. $O(n \log n)$
  - Este método foi desenvolvido por Preparata e Hong em 1977. Por causa da sua similaridade com o MergeSort, o algoritmo é muitas vezes chamado de MergeHull
  - Assumiremos que os pontos do conjunto *S* estão em posição geral e qualquer dois pontos não têm a mesma *x*-coordenada.
  - **Para conseguirmos construir o invólucro convexo em $O(n \log n)$ é preciso que a fase *combinar* do algoritmo seja feita em $O(n)$.**



# Algoritmo MergeHull

(outro algoritmo de divisão e conquista)

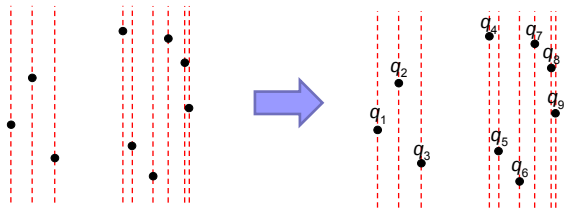
- Dados de Entrada: um conjunto finito $S=\{p_1,\ldots,p_n\}$ de *n* pontos no plano
- Dados de Saída: as arestas que definem conv(*S*), orientadas no sentido positivo
- **Algoritmo MergeHull (S)**

  (1) if $|S| \leq k_0$, ($k_0$ – constante pequena) then
  construir conv(S); return
  (2) (**Dividir**) Particionar S em $S_1$ e $S_2$;
  $|S_1|=\lceil n/2 \rceil$; $|S_2|=\lfloor n/2 \rfloor$;
  (3) (**Conquistar**) Recursivamente construir
  conv($S_1$) e conv($S_2$)
  (4) (**Combinar**) A partir de conv($S_1$) e conv($S_2$)
  construir conv($S_1 \cup S_2$)

# Algoritmo MergeHull

(outro algoritmo de divisão e conquista)

- **Descrição do Algoritmo**
  - **1ª Fase: Pre-Processamento**
    - Para facilitarmos a fase *conquistar*, antes de aplicarmos o algoritmo MergeHull, ordenamos o conjunto de pontos em relação as suas *x*-coordenadas.

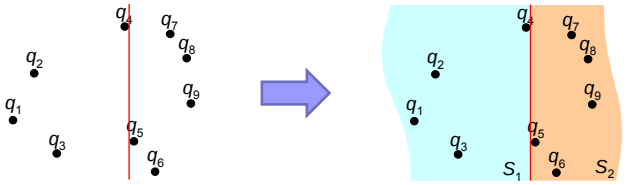





# Algoritmo MergeHull

(outro algoritmo de divisão e conquista)

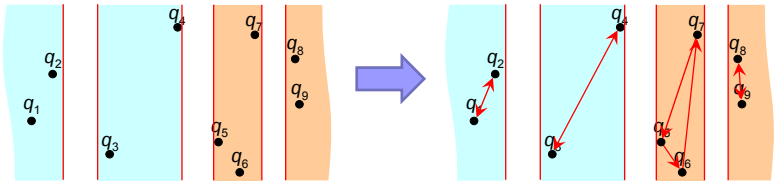
- **Descrição do Algoritmo**
  - **2ª Fase: Dividir**
    - Dividir $S$ em $S_1$ e $S_2$ segundo a mediana das x coordenadas e tais que $|S_1|=\lceil n/2 \rceil$ e $|S_2|=\lfloor n/2 \rfloor$. Esta escolha de conjuntos $S_1$ e $S_2$ e pela hipótese de que $S$ não tem dois pontos com a mesma *x*-coordenada garante que $\text{conv}(S_1) \cap \text{conv}(S_2) = \varnothing$.



# Algoritmo MergeHull

(outro algoritmo de divisão e conquista)

- **Descrição do Algoritmo**
  - **3ª Fase: Conquistar**
    - Corresponde a chamadas recursivas onde cada sub-problema será resolvido independentemente, até que o número de pontos seja suficientemente pequeno ($k_0$). Por exemplo se $k_0 = 3$, o invólucro convexo é um triângulo pela hipótese de posição geral.

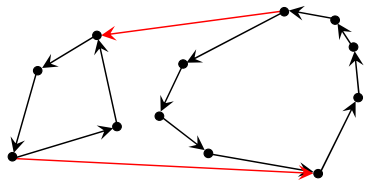





# Algoritmo MergeHull

(outro algoritmo de divisão e conquista)

- **Descrição do Algoritmo**
  - **4ª Fase: Combinar**
    - Atenção: Precisamos ser cuidadosos para realizarmos este passo em tempo linear.
    - Nesta fase vamos procurar duas rectas tangentes a conv($S_1$) e conv($S_2$): uma tangenciando os invólucros convexos por *baixo* e outra tangenciando por *cima.*
    - A partir destas tangentes é fácil construirmos o conv($S_1 \cup S_2$) em tempo linear.



# Algoritmo MergeHull

(outro algoritmo de divisão e conquista)

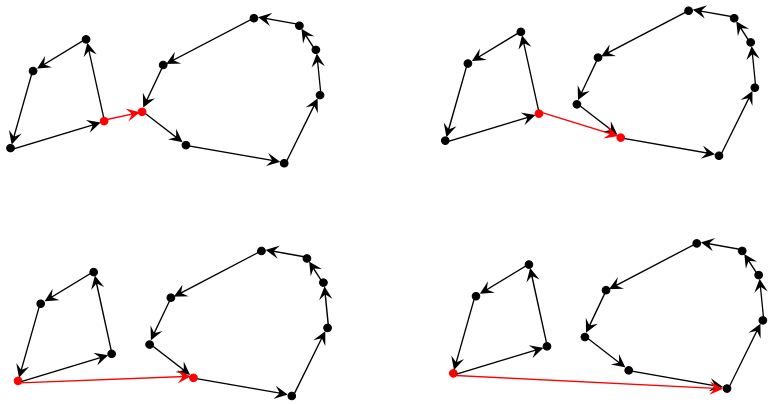
- **Descrição do Algoritmo**
  - **4ª Fase: Combinar** (cont…)
    - Denotaremos por $u_i$ e $v_j$ os vértices de conv($S_1$) e conv($S_2$), respectivamente.
    - Seja $t = u_i v_j$ o segmento de recta que determina a tangente inferior. O problema é que não sabemos de antemão quais os vértices $u_i$ de conv($S_1$) e $v_j$ de conv($S_2$) que são extremidades de $t$.
    - A procura do segmento $t$ começa por iniciar $t$ como sendo o segmento formado pelo vértice $u_i$ mais à direita de conv($S_1$) e pelo vértice $v_j$ mais à esquerda de conv($S_2$), e "deixar $t$ cair", primeiro "escorregando" de um lado e depois do outro, e assim sucessivamente até que as extremidades do segmento procurado sejam encontradas



# Algoritmo MergeHull

(outro algoritmo de divisão e conquista)

- **Descrição do Algoritmo**
  - **4ª Fase: Combinar** (cont…)



# Algoritmo MergeHull

(outro algoritmo de divisão e conquista)

- **Descrição do Algoritmo**
  - **4ª Fase: Combinar** (cont…)

**Algoritmo Tangente Inferior**

Entrada: dois invólucros convexos, CH($S_1$) e CH($S_2$), separados por uma recta vertical

Saída: o segmento tangente inferior a CH($S_1$) e CH($S_2$)

```
u_i ← vértice mais a direita de CH(S_1)
v_j ← vértice mais a esquerda de CH(S_2)
WHILE t = u_i v_j não é tangente inferior de CH(S_1) e CH(S_2)
  WHILE t não é tangente inferior de CH(S_1)
    i ← i -1
  WHILE t não é tangente inferior de CH(S_2)
    j ← j+1
return t
```

**Observação:** no algoritmo acima, a frase "**$t = u_i v_j$ não é tangente inferior de CH($S_1$)**" é equivalente a dizermos **$u_{i-1}$** está à direita de **$u_i v_j$**



# Algoritmo MergeHull

(outro algoritmo de divisão e conquista)

- **Análise:**
  - A complexidade temporal do pré-processamento (ordenação dos pontos) que é $O(n \log n)$.
  - A divisão é $O(n)$
  - No caso combinar os ciclos são executados $O(n)$ vezes.
  - Se usarmos uma lista circular duplamente ligada para representar os invólucros convexos então, dadas as tangentes inferior e superior o novo invólucro convexo pode ser construído em $O(n)$.

    Portanto, a complexidade total do passo combinar é $O(n)$.

A complexidade de tempo do algoritmo MergeHull é $O(n \log n)$.

## Cota inferior do problema conv(*S*)

- **Teorema:** O problema da determinação do invólucro convexo de um conjunto *S* com *n* pontos não pode ser resolvido em menos que $O(n \log n)$



## Demonstrações

- A implementação de grande parte dos algoritmos estudados pode ser apreciada em:

  http://www.cse.unsw.edu.au/~lambert/java/3d/