

# Geometria Computacional

## Algoritmo de Graham

---

Prof. Edson Alves

Faculdade UnB Gama

## 1. Algoritmo de Graham

# Algoritmo de Graham

---

# Algoritmo de Graham

- O algoritmo de Graham (*Graham Scan*, no original), foi proposto por Ronald Graham em 1972

# Algoritmo de Graham

- O algoritmo de Graham (*Graham Scan*, no original), foi proposto por Ronald Graham em 1972
- Ele inicialmente ordena todos os  $N$  pontos de  $P$  de acordo com o ângulo que eles formam com um ponto pivô fixado previamente

# Algoritmo de Graham

- O algoritmo de Graham (*Graham Scan*, no original), foi proposto por Ronald Graham em 1972
- Ele inicialmente ordena todos os  $N$  pontos de  $P$  de acordo com o ângulo que eles formam com um ponto pivô fixado previamente
- A escolha padrão para o pivô é o ponto de menor coordenada  $y$

# Algoritmo de Graham

- O algoritmo de Graham (*Graham Scan*, no original), foi proposto por Ronald Graham em 1972
- Ele inicialmente ordena todos os  $N$  pontos de  $P$  de acordo com o ângulo que eles formam com um ponto pivô fixado previamente
- A escolha padrão para o pivô é o ponto de menor coordenada  $y$
- Caso exista mais de um ponto com coordenada  $y$  mínima, escolhe-se o de maior coordenada  $x$  dentre eles

# Algoritmo de Graham

- O algoritmo de Graham (*Graham Scan*, no original), foi proposto por Ronald Graham em 1972
- Ele inicialmente ordena todos os  $N$  pontos de  $P$  de acordo com o ângulo que eles formam com um ponto pivô fixado previamente
- A escolha padrão para o pivô é o ponto de menor coordenada  $y$
- Caso exista mais de um ponto com coordenada  $y$  mínima, escolhe-se o de maior coordenada  $x$  dentre eles
- Se  $P$  é armazenado em um vetor, o algoritmo pode ser simplificado movendo-se o pivô para a primeira posição



# Implementação da escolha do pivô

```
34 template<typename T>
35 class GrahamScan {
36 private:
37     static Point<T> pivot(vector<Point<T>>& P)
38     {
39         size_t idx = 0;
40
41         for (size_t i = 1; i < P.size(); ++i)
42             if (P[i].y < P[idx].y or (equals(P[i].y, P[idx].y) and P[i].x > P[idx].x))
43                 idx = i;
44
45         swap(P[0], P[idx]);
46
47         return P[0];
48     }
```

## Ordenação dos pontos de acordo com o ângulo

- A ordem é dada, de forma crescente, pelo valor do ângulo que é formado entre um ponto e o eixo  $x$  positivo do pivô. Quando dois pontos formam um ângulo igual, a prioridade é dada ao que está mais perto do pivô

## Ordenação dos pontos de acordo com o ângulo

- A ordem é dada, de forma crescente, pelo valor do ângulo que é formado entre um ponto e o eixo  $x$  positivo do pivô. Quando dois pontos formam um ângulo igual, a prioridade é dada ao que está mais perto do pivô
- Para realizar a ordenação dos pontos é preciso definir um operador booleano que receba dois pontos  $P$  e  $Q$  e retorne verdadeiro se  $P$  antecede  $Q$  de acordo com a ordenação proposta

## Ordenação dos pontos de acordo com o ângulo

- A ordem é dada, de forma crescente, pelo valor do ângulo que é formado entre um ponto e o eixo  $x$  positivo do pivô. Quando dois pontos formam um ângulo igual, a prioridade é dada ao que está mais perto do pivô
- Para realizar a ordenação dos pontos é preciso definir um operador booleano que receba dois pontos  $P$  e  $Q$  e retorne verdadeiro se  $P$  antecede  $Q$  de acordo com a ordenação proposta
- Como é necessário o conhecimento do pivô para tal ordenação, há três possibilidades para a implementação deste operador:

## Ordenação dos pontos de acordo com o ângulo

- A ordem é dada, de forma crescente, pelo valor do ângulo que é formado entre um ponto e o eixo  $x$  positivo do pivô. Quando dois pontos formam um ângulo igual, a prioridade é dada ao que está mais perto do pivô
- Para realizar a ordenação dos pontos é preciso definir um operador booleano que receba dois pontos  $P$  e  $Q$  e retorne verdadeiro se  $P$  antecede  $Q$  de acordo com a ordenação proposta
- Como é necessário o conhecimento do pivô para tal ordenação, há três possibilidades para a implementação deste operador:
  1. implementar o operador  $<$  da classe `Point`, tornando o pivô um membro da classe para que o operador tenha acesso a ele;

## Ordenação dos pontos de acordo com o ângulo

- A ordem é dada, de forma crescente, pelo valor do ângulo que é formado entre um ponto e o eixo  $x$  positivo do pivô. Quando dois pontos formam um ângulo igual, a prioridade é dada ao que está mais perto do pivô
- Para realizar a ordenação dos pontos é preciso definir um operador booleano que receba dois pontos  $P$  e  $Q$  e retorne verdadeiro se  $P$  antecede  $Q$  de acordo com a ordenação proposta
- Como é necessário o conhecimento do pivô para tal ordenação, há três possibilidades para a implementação deste operador:
  1. implementar o operador  $<$  da classe `Point`, tornando o pivô um membro da classe para que o operador tenha acesso a ele;
  2. tornar o pivô uma variável global;

## Ordenação dos pontos de acordo com o ângulo

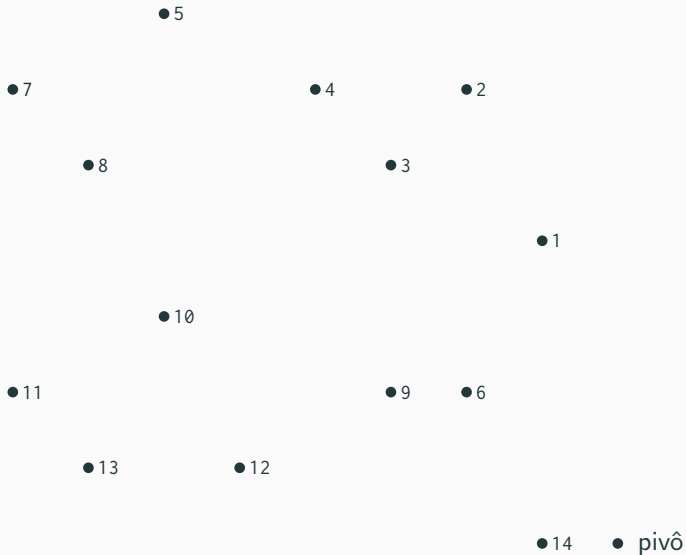
- A ordem é dada, de forma crescente, pelo valor do ângulo que é formado entre um ponto e o eixo  $x$  positivo do pivô. Quando dois pontos formam um ângulo igual, a prioridade é dada ao que está mais perto do pivô
- Para realizar a ordenação dos pontos é preciso definir um operador booleano que receba dois pontos  $P$  e  $Q$  e retorne verdadeiro se  $P$  antecede  $Q$  de acordo com a ordenação proposta
- Como é necessário o conhecimento do pivô para tal ordenação, há três possibilidades para a implementação deste operador:
  1. implementar o operador  $<$  da classe `Point`, tornando o pivô um membro da classe para que o operador tenha acesso a ele;
  2. tornar o pivô uma variável global;
  3. usar uma função lambda no terceiro parâmetro da função `sort()`, capturando o pivô por referência ou cópia

## Ordenação dos pontos de acordo com o ângulo

- A ordem é dada, de forma crescente, pelo valor do ângulo que é formado entre um ponto e o eixo  $x$  positivo do pivô. Quando dois pontos formam um ângulo igual, a prioridade é dada ao que está mais perto do pivô
- Para realizar a ordenação dos pontos é preciso definir um operador booleano que receba dois pontos  $P$  e  $Q$  e retorne verdadeiro se  $P$  antecede  $Q$  de acordo com a ordenação proposta
- Como é necessário o conhecimento do pivô para tal ordenação, há três possibilidades para a implementação deste operador:
  1. implementar o operador  $<$  da classe `Point`, tornando o pivô um membro da classe para que o operador tenha acesso a ele;
  2. tornar o pivô uma variável global;
  3. usar uma função lambda no terceiro parâmetro da função `sort()`, capturando o pivô por referência ou cópia
- O ângulo pode ser obtido através da função `atan2()` da biblioteca `math.h` da linguagem C/C++



## Exemplo de ordenação por ângulo



# Implementação da rotina de ordenação dos pontos

```
50  static void sort_by_angle(vector<Point<T>>& P)
51  {
52      auto P0 = pivot(P);
53
54      sort(P.begin() + 1, P.end(), [&](const Point<T>& A, const Point<T>& B) {
55          // pontos colineares: escolhe-se o mais próximo do pivô
56          if (equals(D(P0, A, B), 0))
57              return A.distance(P0) < B.distance(P0);
58
59          auto alfa = atan2(A.y - P0.y, A.x - P0.x);
60          auto beta = atan2(B.y - P0.y, B.x - P0.x);
61
62          return alfa < beta;
63      });
64  }
```

## Identificação do envoltório convexo

- Após a ordenação dos pontos, o algoritmo procede empilhando três pontos de  $P$ : inicialmente os pontos cujos índices são  $n - 1, 0$  e  $1$

## Identificação do envoltório convexo

- Após a ordenação dos pontos, o algoritmo procede empilhando três pontos de  $P$ : inicialmente os pontos cujos índices são  $n - 1, 0$  e  $1$
- O invariante a ser mantido é que os três elementos do topo de pilha estão em sentido anti-horário ( $D > 0$ )

## Identificação do envoltório convexo

- Após a ordenação dos pontos, o algoritmo procede empilhando três pontos de  $P$ : inicialmente os pontos cujos índices são  $n - 1, 0$  e  $1$
- O invariante a ser mantido é que os três elementos do topo de pilha estão em sentido anti-horário ( $D > 0$ )
- Para cada um dos demais pontos  $Q_i$  de  $P$ , com  $i = 2, 3, \dots, n - 1$ , verifica-se se este ponto mantém o sentido anti-horário com os dois elementos do topo da pilha

## Identificação do envoltório convexo

- Após a ordenação dos pontos, o algoritmo procede empilhando três pontos de  $P$ : inicialmente os pontos cujos índices são  $n - 1, 0$  e  $1$
- O invariante a ser mantido é que os três elementos do topo de pilha estão em sentido anti-horário ( $D > 0$ )
- Para cada um dos demais pontos  $Q_i$  de  $P$ , com  $i = 2, 3, \dots, n - 1$ , verifica-se se este ponto mantém o sentido anti-horário com os dois elementos do topo da pilha
- Em caso afirmativo, o ponto é inserido na pilha

## Identificação do envoltório convexo

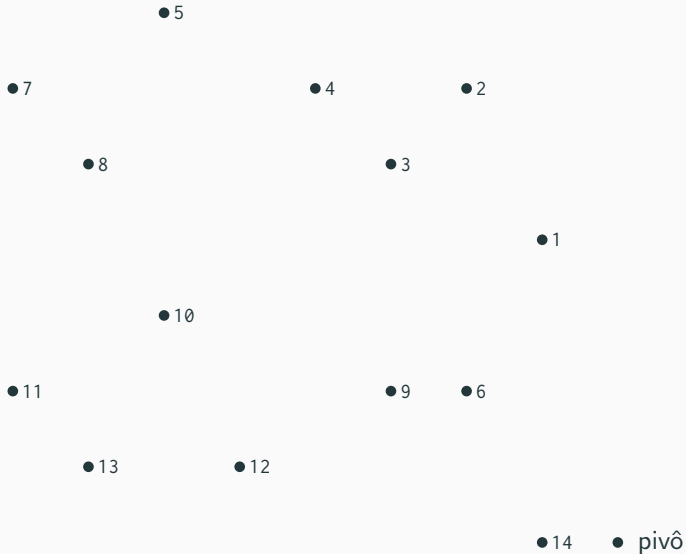
- Após a ordenação dos pontos, o algoritmo procede empilhando três pontos de  $P$ : inicialmente os pontos cujos índices são  $n - 1, 0$  e  $1$
- O invariante a ser mantido é que os três elementos do topo de pilha estão em sentido anti-horário ( $D > 0$ )
- Para cada um dos demais pontos  $Q_i$  de  $P$ , com  $i = 2, 3, \dots, n - 1$ , verifica-se se este ponto mantém o sentido anti-horário com os dois elementos do topo da pilha
- Em caso afirmativo, o ponto é inserido na pilha
- Caso contrário, remove-se o topo da pilha e se verifica o invariante para  $Q_i$  novamente

## Identificação do envoltório convexo

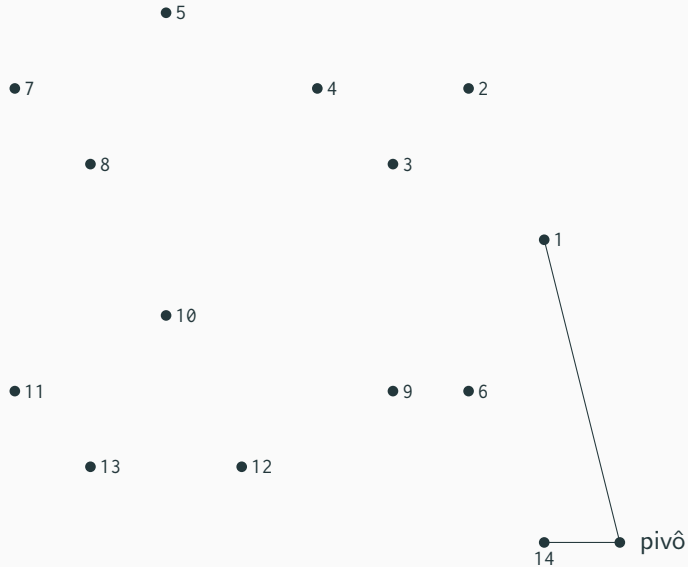
- Após a ordenação dos pontos, o algoritmo procede empilhando três pontos de  $P$ : inicialmente os pontos cujos índices são  $n - 1, 0$  e  $1$
- O invariante a ser mantido é que os três elementos do topo de pilha estão em sentido anti-horário ( $D > 0$ )
- Para cada um dos demais pontos  $Q_i$  de  $P$ , com  $i = 2, 3, \dots, n - 1$ , verifica-se se este ponto mantém o sentido anti-horário com os dois elementos do topo da pilha
- Em caso afirmativo, o ponto é inserido na pilha
- Caso contrário, remove-se o topo da pilha e se verifica o invariante para  $Q_i$  novamente
- Como cada ponto é ou inserido ou removido uma única vez, este processo tem complexidade  $O(N)$ , e o algoritmo como um todo tem complexidade  $O(N \log N)$ , devido à ordenação



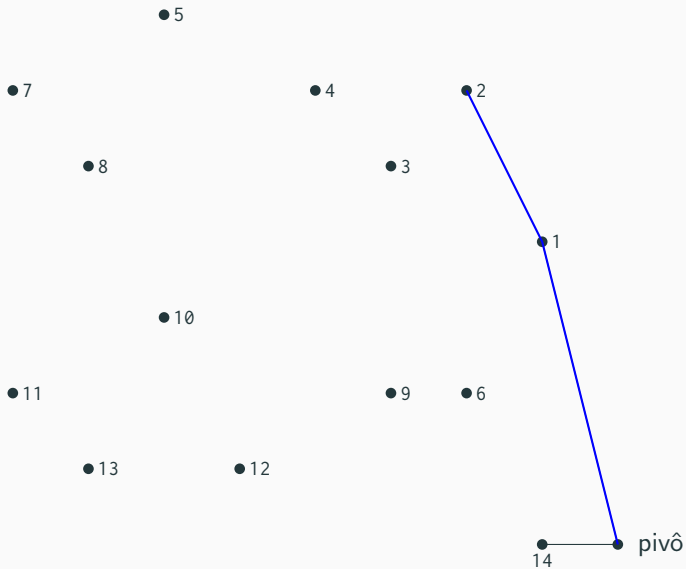
# Visualização do algoritmo de Graham



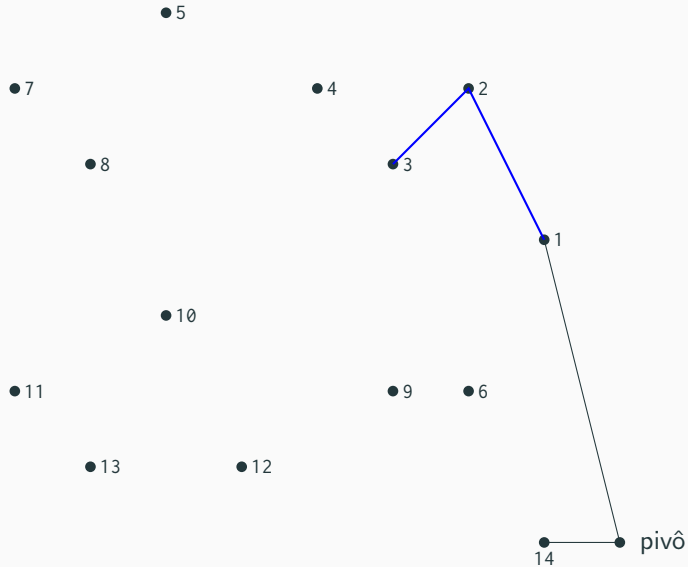
# Visualização do algoritmo de Graham



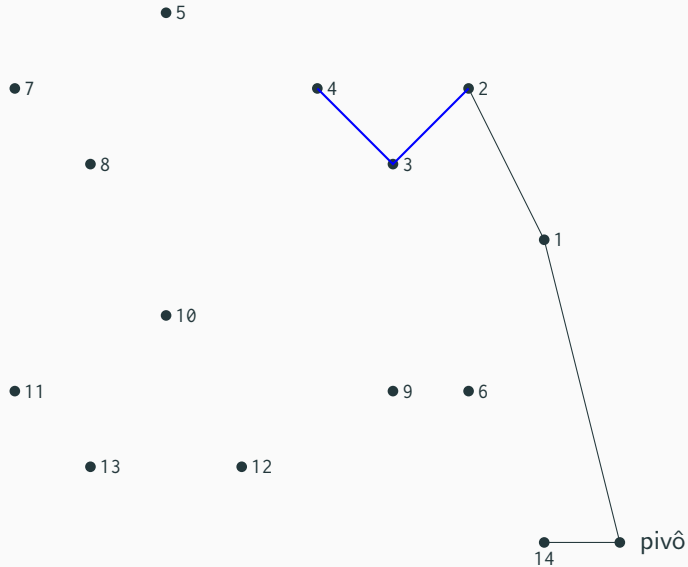
# Visualização do algoritmo de Graham



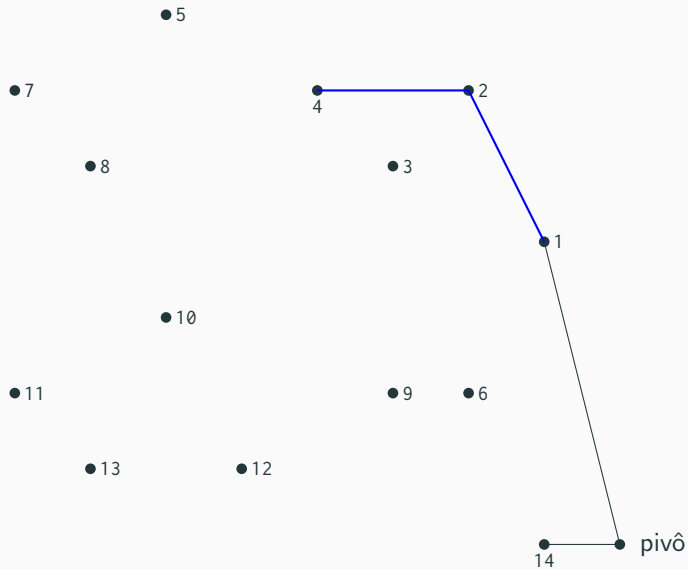
# Visualização do algoritmo de Graham



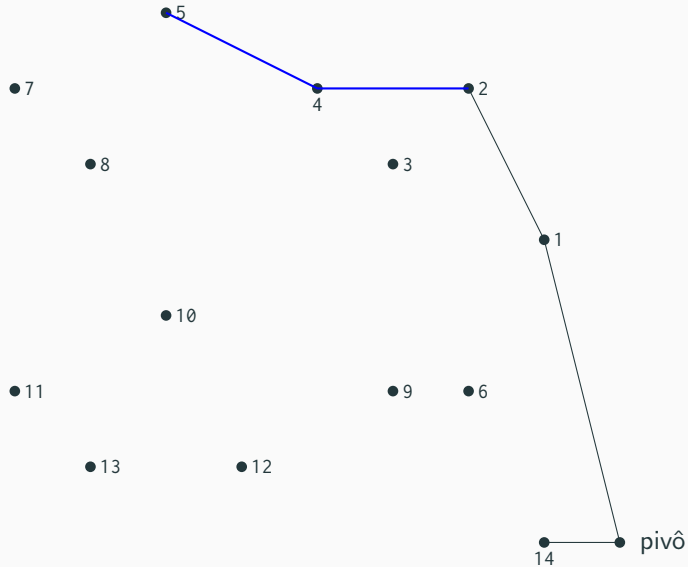
# Visualização do algoritmo de Graham



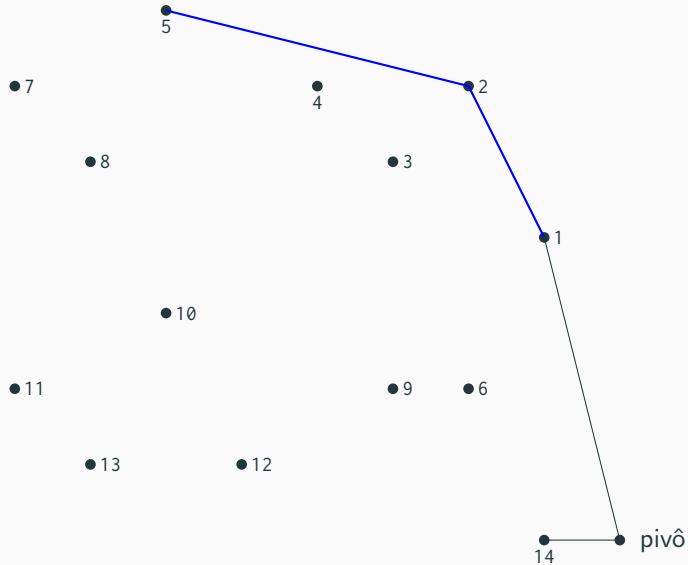
# Visualização do algoritmo de Graham



# Visualização do algoritmo de Graham

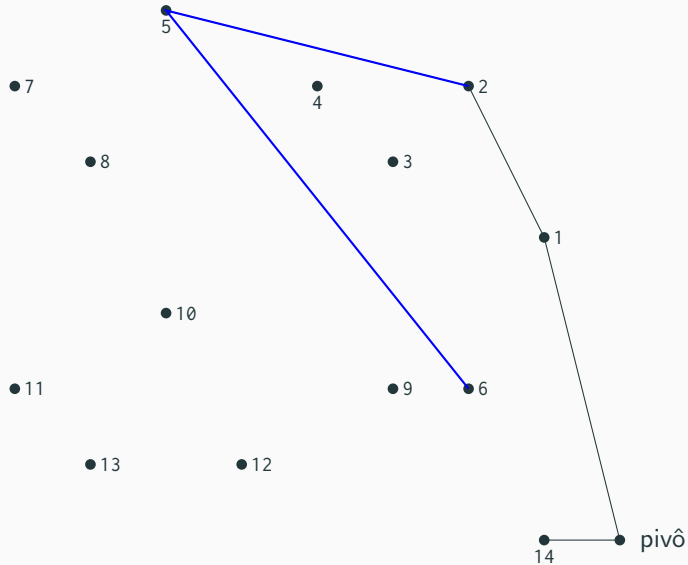


# Visualização do algoritmo de Graham

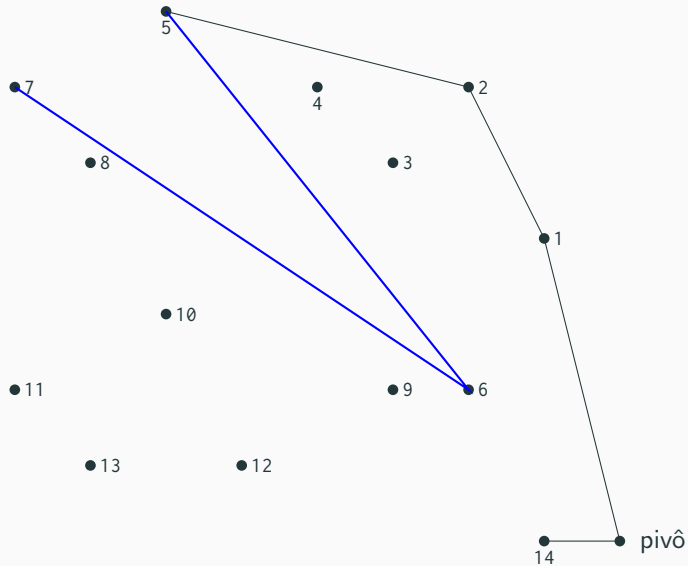




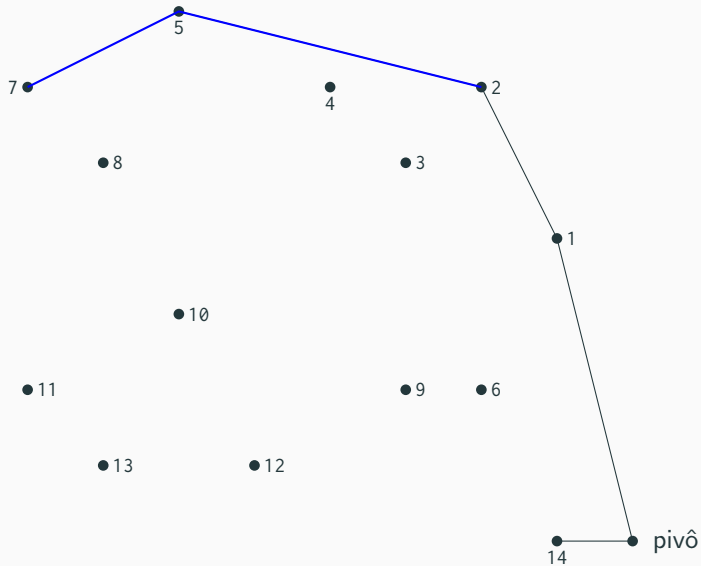
# Visualização do algoritmo de Graham



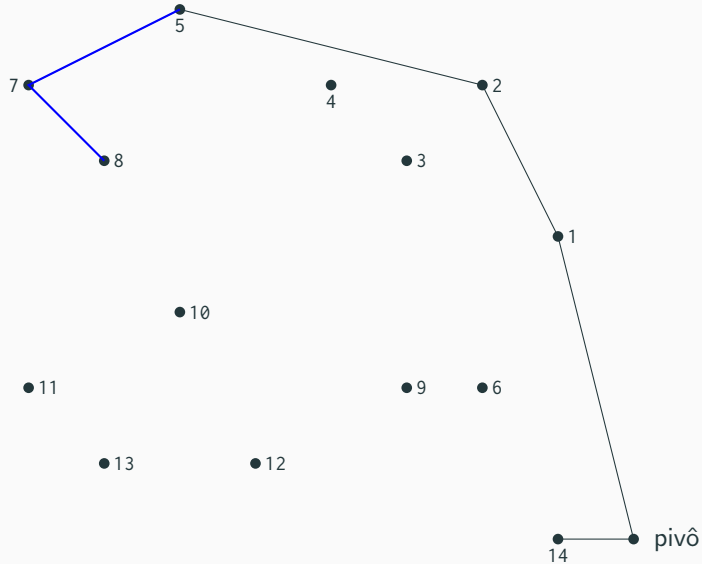
# Visualização do algoritmo de Graham



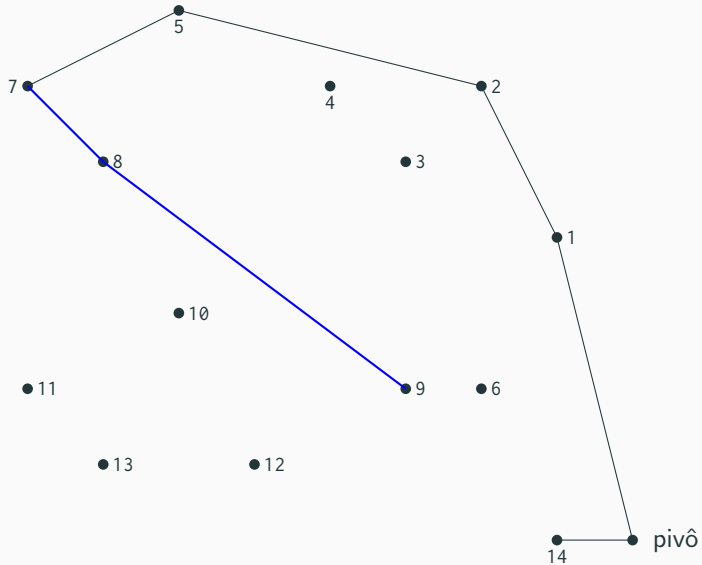
# Visualização do algoritmo de Graham



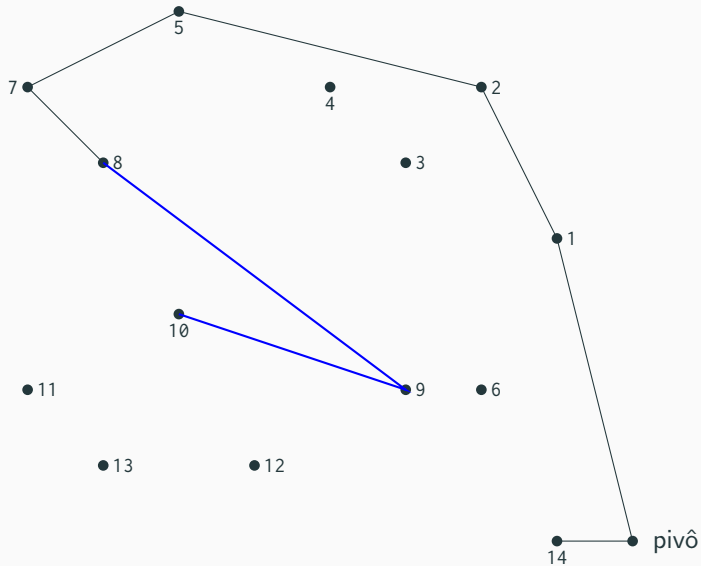
# Visualização do algoritmo de Graham



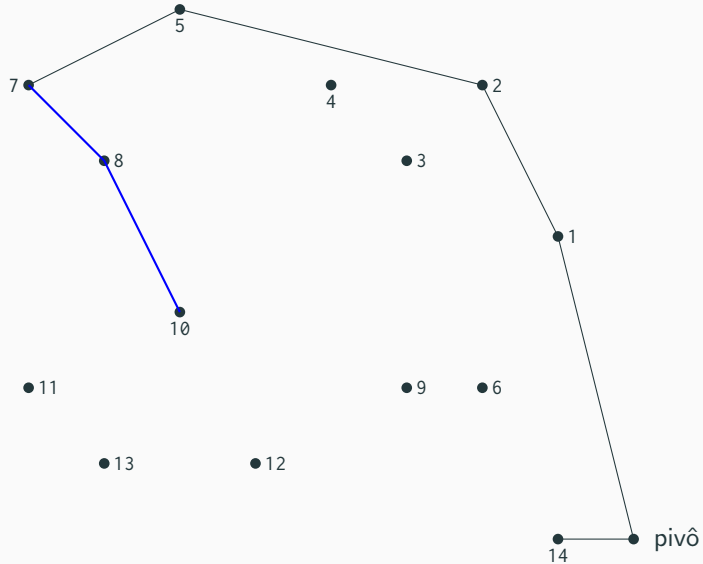
# Visualização do algoritmo de Graham



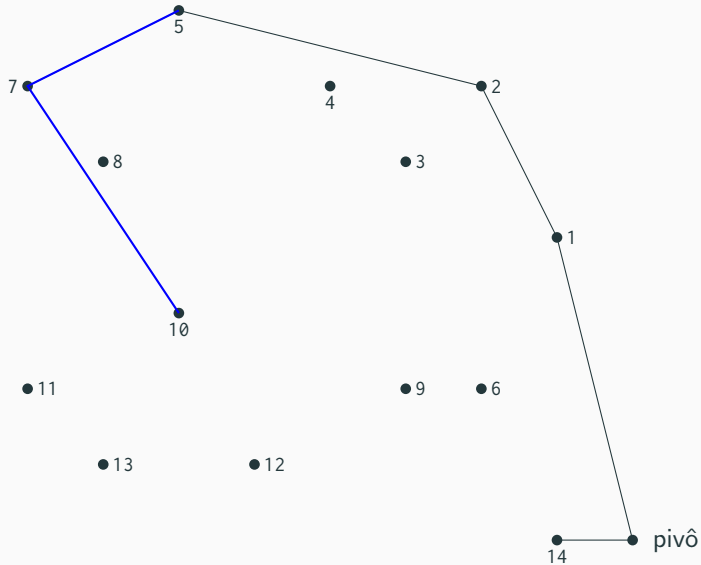
# Visualização do algoritmo de Graham



# Visualização do algoritmo de Graham

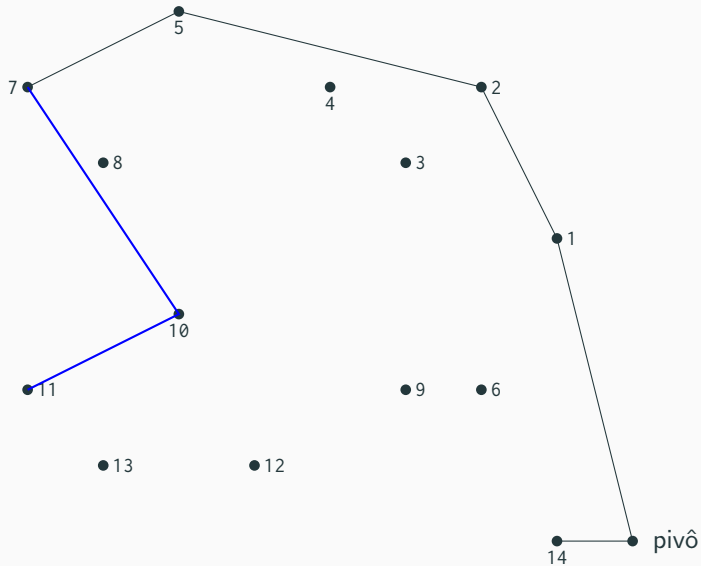


# Visualização do algoritmo de Graham

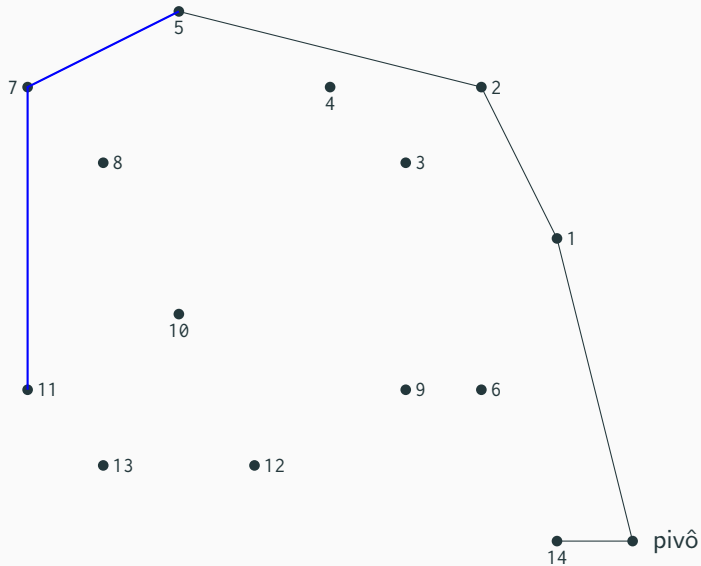




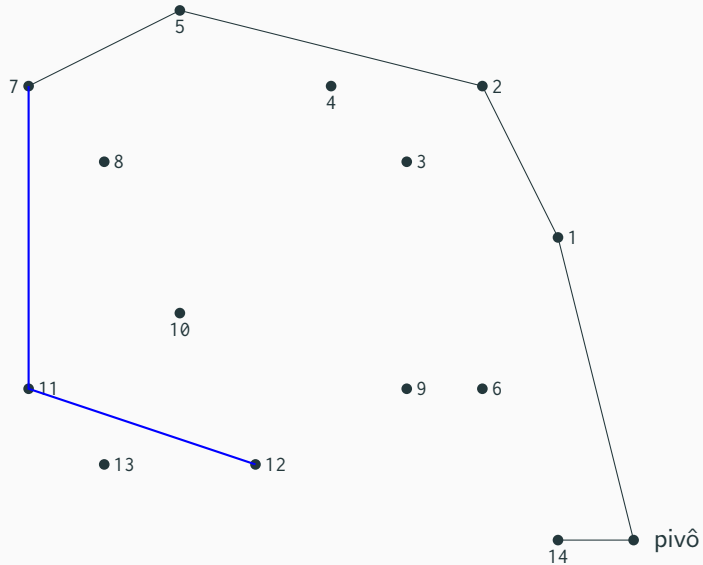
# Visualização do algoritmo de Graham



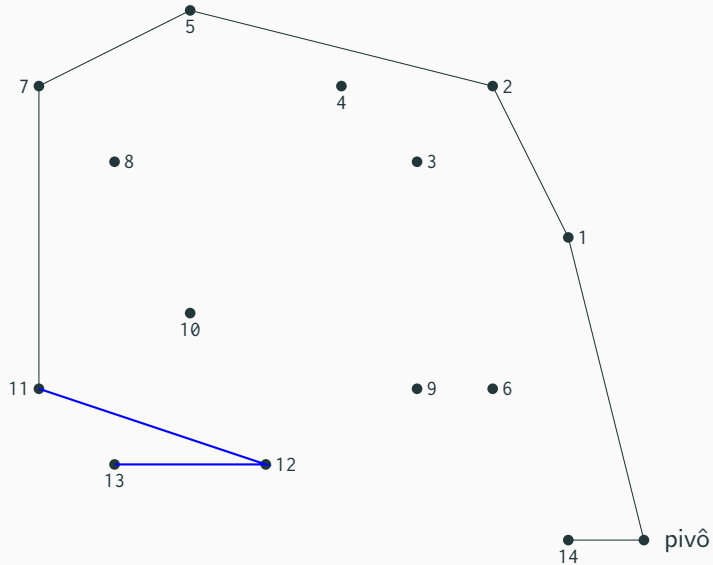
# Visualização do algoritmo de Graham



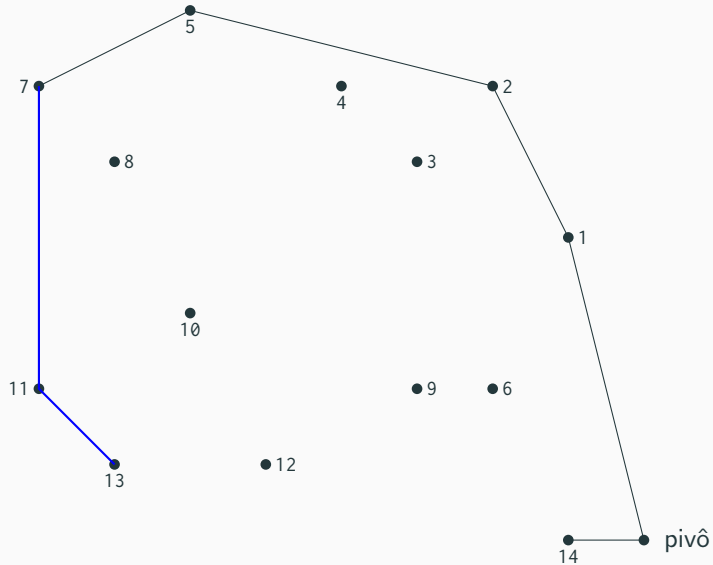
# Visualização do algoritmo de Graham



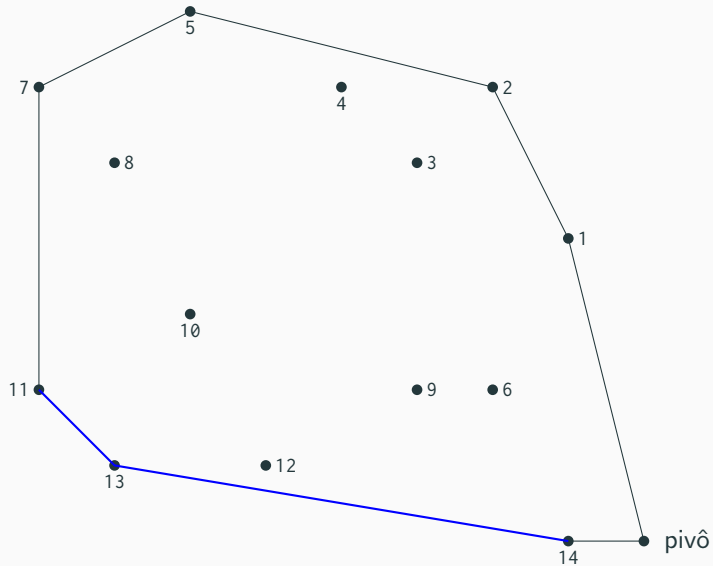
# Visualização do algoritmo de Graham



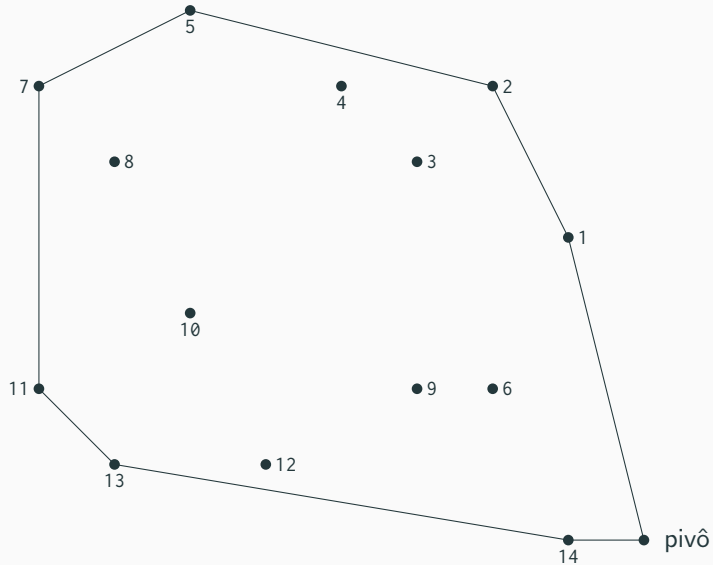
# Visualização do algoritmo de Graham



# Visualização do algoritmo de Graham



# Visualização do algoritmo de Graham



# Implementação da rotina de envoltório convexo

```
67  static vector<Point<T>> convex_hull(const vector<Point<T>>& points)
68  {
69      vector<Point<T>> P(points);
70      auto N = P.size();
71
72      // Corner case: com 3 vértices ou menos, P é o próprio convex hull
73      if (N <= 3)
74          return P;
75
76      sort_by_angle(P);
77
78      vector<Point<T>> ch;
79      ch.push_back(P[N - 1]);
80      ch.push_back(P[0]);
81      ch.push_back(P[1]);
82
83      size_t i = 2;
```



# Implementação da rotina de envoltório convexo

```
85     while (i < N)
86     {
87         auto j = ch.size() - 1;
88
89         if (D(ch[j - 1], ch[j], P[i]) > 0)
90             ch.push_back(P[i++]);
91         else
92             ch.pop_back();
93     }
94
95     // O envoltório é um caminho fechado: o primeiro ponto é igual ao último
96     return ch;
97 }
```

1. **DE BERG**, Mark. *Computational Geometry: Algorithms and Applications*, Springer, 3rd edition, 2008.
2. **GRAHAM**, R. L. *An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set*. Information Processing Letters vol. 1 (4), pg. 132-133, 1972.
3. **HALIM**, Felix; **HALIM**, Steve. *Competitive Programming 3*, 2010.
4. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, 2018 (Open Access).
5. **O'ROURKE**, Joseph. *Computational Geometry in C*, Cambridge University Press, 2nd edition, 1998.
6. Wikipedia. [Graham scan](#), acesso em 09/06/2019.