# Trabalho Prático 2
# Fecho Convexo

**Lucas Albano Olive Cruz - 2022036209**

Departamento de Ciência da Computação (DCC) - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

lucasalbano@dcc.ufmg.br

## 1. Introdução

O problema proposto se resume a determinar o fecho convexo de um conjunto de pontos para otimizar a compra de tecidos para uma indústria têxtil. O fecho convexo é o menor polígono que envolve todos os pontos.

Existem dois algoritmos famosos para determinar o fecho convexo: Marcha de Jarvis e Scan de Graham. O segundo se utiliza de uma fase de ordenação dos pontos para a qual podemos utilizar diferentes algoritmos para concluí-la.

Portanto, neste trabalho utilizaremos ao todo 4 métodos para resolver o problema: Scan de Graham com ordenação por Merge Sort, Insertion Sort e Bubble Sort e a Marcha de Jarvis. Ainda, avaliaremos o desempenho para cada um dos métodos.

## 2. Método

Como descrito utilizaremos 4 métodos ao todo para determinar o fecho convexo de um conjunto de pontos. Também, serão utilizados TADs para representar os pontos e o fecho convexo.

### 2.1. Estruturas de Dados

Para representar o fecho convexo iremos utilizar uma pilha simples com alocação dinâmica que terá como elementos os pontos que serão outro TAD.

#### 2.1.1. Pontos

O TAD Ponto é bastante simples, é uma classe que armazena dois inteiros, as coordenadas X e Y. Possui construtor, destrutor e métodos auxiliares getters e setters.

#### 2.1.2. Fecho Convexo

O fecho convexo como já descrito é o subconjunto de pontos que envolve todo o conjunto de pontos fornecidos. Estes pontos serão armazenados em uma pilha simples com alocação dinâmica.

Uma pilha, como estudado ao longo da matéria, é uma estrutura com a característica que o último item a ser inserido é o primeiro a ser retirado, exatamente como em uma pilha de pratos ou livros. Com alocação dinâmica queremos dizer que nós não pré-alocamos uma quantidade fixa de memória, com isso podemos armazenar a quantidade exata de pontos que desejamos sem desperdiçar memória.

Assim, nossa pilha possui dois atributos: um ponteiro para o topo da pilha e um inteiro que representa seu tamanho. Possui funções para empilhar e desempilhar pontos, construtor e destrutor, getters e setters, além de uma função auxiliar NextToTop() que retorna o elemento logo abaixo do topo da pilha sem fazer alterações na pilha, este método será de grande ajuda ao executar nossos algoritmos. Por fim, uma função para imprimir os elementos armazenados na pilha.

## 2.2. Algoritmos

Para efetivamente determinar o fecho convexo do conjunto de pontos fornecidos para o programa iremos utilizar dois algoritmos famosos: Scan de Graham e Marcha de Jarvis.

### 2.2.1. Scan de Graham

O primeiro passo do Scan de Graham é encontrar o ponto com a menor coordenada Y, ou seja, o ponto mais baixo do conjunto. Em caso de empate, o ponto com menor coordenada X é escolhido.

O próximo passo é ordenar os demais pontos baseados no ângulo que formam com ponto inicial de forma crescente. Esse passo nos dá a liberdade de escolher qualquer método de ordenação. Para o nosso problema iremos adotar 3 métodos diferentes: MergeSort, InsertionSort e BucketSort. Um detalhe importante é que não precisamos calcular o ângulo propriamente dito, podemos apenas comparar a orientação dos vetores formados pelo ponto inicial e os dois pontos que estamos comparando e utilizar a norma dos vetores como método de desempate.

O algoritmo procede removendo do array todos os pontos que formam um ângulo igual com o ponto de menor coordenada Y. Caso tenhamos menos que 3 pontos após este passo nós não temos um fecho convexo.

Continuamos criando uma pilha e adicionando os 3 primeiros pontos do array. Em seguida, o algoritmo irá processar todos os outros pontos do array observando se os últimos 3 pontos formam uma curva à esquerda ou à direita. Caso um ponto forme uma curva à direita o removemos da pilha. Ao fim teremos uma pilha com os pontos pertencentes ao fecho convexo.

Novamente, não precisamos calcular o ângulo para determinar curvas a direita ou esquerda, basta somente calcular a orientação dos vetores formados pelos pontos.

### 2.1.2. Marcha de Jarvis

A ideia da Marcha de Jarvis é simples, partimos do ponto mais à esquerda, ou seja, o ponto com menor coordenada X e continuamos envolvendo os pontos no sentido anti-horário.

Para isso utilizaremos sempre 3 pontos para efetuar as comparações, os chamaremos de p, q e r. Inicialmente p é justamente o ponto mais à esquerda.

O próximo ponto q é o ponto, tal que a tripla (p, q, r) seja anti-horária para qualquer outro ponto r. Para encontrar isso, simplesmente inicializamos q como o próximo ponto, então percorremos todos os pontos. Para qualquer ponto i, se i é mais anti-horário, ou seja, a orientação (p, i, q) é anti-horária, então atualizamos q como i. Nosso valor final de q será o ponto mais anti-horário.

Então, armazenamos q na pilha como o próximo ponto após p e definimos p como q para a próxima iteração.

Ao fim, retornamos ao ponto inicial e nossa pilha conterá o fecho convexo.

# 3. Análise de Complexidade

Nesse tópico iremos analisar a complexidade de tempo e espaço dos algoritmos implementados.

## 3.1. Marcha de Jarvis

O loop interno do algoritmo verifica cada ponto no conjunto, e o loop externo se repete para cada ponto no fecho convexo. Portanto, o tempo total de execução é $O(nm)$ onde $n$ é o tamanho da entrada e $m$ é a quantidade de elementos do fecho convexo.

Logo, o pior caso ocorre quando todos os pontos estão no fecho convexo, ou seja, $m = n$ e a complexidade é $O(n^2)$.

Como lemos um array e armazenamos todos os pontos do fecho convexo em uma pilha a nossa complexidade de espaço é $O(m)$.

## 3.2. Scan de Graham

A primeira etapa (encontrar o ponto mais baixo) leva tempo O(n). A segunda etapa (ordenar os pontos) depende do algoritmo de ordenação utilizado, logo, para o MergeSort temos $O(nlogn)$, para o InsertionSort $O(n^2)$ e para o BucketSort $O(n)$ considerando os casos médios para todos os algoritmos. Na terceira etapa, cada elemento é empilhado e desempilhado no máximo uma vez. Assim, a etapa para processar os pontos um a um leva tempo O(n), assumindo que as operações de empilhamento levam tempo O(1).

Portanto, a complexidade de tempo será definida pelo algoritmo de ordenação utilizado. Para os algoritmos utilizados neste trabalho teremos a complexidade como $O(n^2)$, $O(nlogn)$ e $O(n)$ para os casos médios dos algoritmos InsertionSort, MergeSort e BucketSort respectivamente.

A complexidade de espaço será definida pelo tamanho da pilha necessária para armazenar os pontos do fecho convexo, logo $O(n)$.

# 4. Estratégias de Robustez

O programa tem uma entrada muito bem definida. Um arquivo com dois inteiros por linha que representam as coordenadas X e Y no plano cartesiano de um ponto separados por espaço. Portanto, qualquer entrada que fuja desse formato resultará em um erro de leitura que será retornado no terminal para o usuário do programa.

Também, retornamos um erro caso o arquivo passado não exista. Caso o usuário invoque o programa sem passar nenhum arquivo ele recebe como saída instruções de uso do programa no terminal.

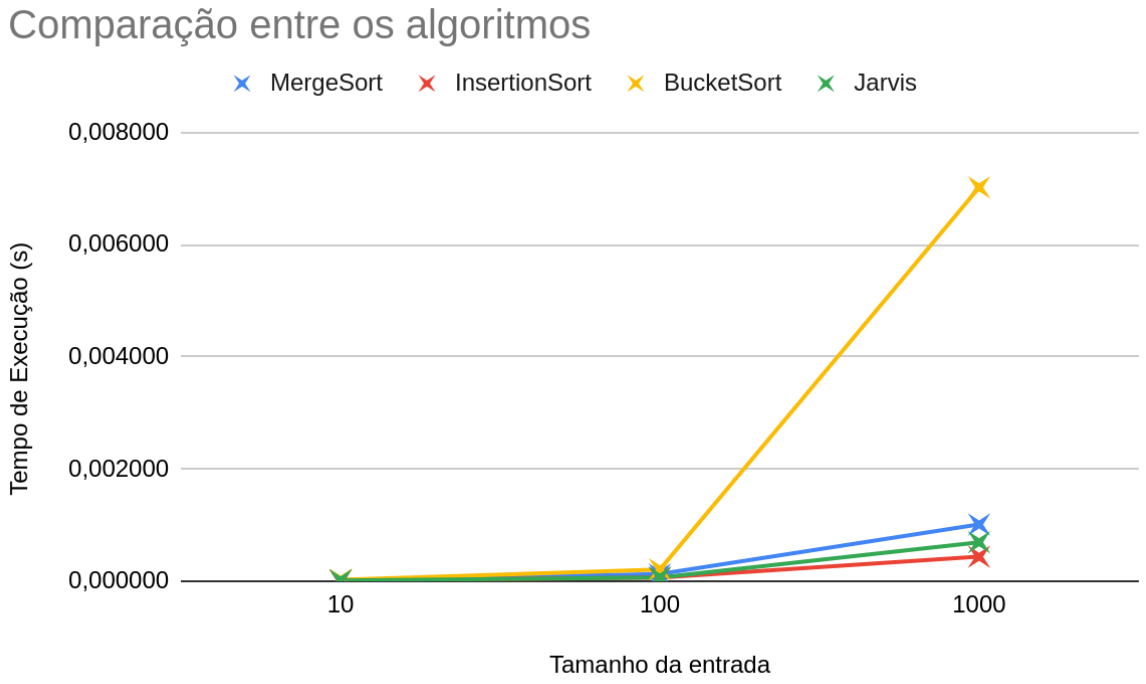
Os passos dos algoritmos são determinísticos, logo erros em cálculos matemáticos como divisão por zero não ocorrerão para o domínio do problema.

Caso o conjunto de pontos não gere um fecho convexo, por exemplo, caso sejam passados menos que 3 pontos ou os pontos passados sejam todos idênticos, o programa retorna um erro e informa que os pontos fornecidos são inválidos.

# 5. Análise Experimental

Para a análise experimental iremos aproveitar o requisito do trabalho de medir o tempo de execução dos algoritmos e iremos os comparar utilizando entradas de 10, 100 e 1000 pontos. Usaremos uma média de 10 observações para definir o tempo gasto por cada algoritmo para cada entrada.

Dessa forma, obtemos os seguintes resultados:




Observamos uma discrepância em relação ao algoritmo de Scan de Graham usando o BucketSort. Isso muito possivelmente se deve ao passo de definir a qual bucket um elemento pertence, pois para isso é necessário calcular o ângulo deste. Portanto, o BucketSort não seria indicado para este tipo de problema em que é custoso determinar o bucket ao qual um elemento pertence.

Os demais algoritmos aparentam manter um comportamento estável e linear, os quais demandariam mais testes para determinar com maior precisão seus desempenhos.

# 6. Conclusões

Conseguimos então implementar 4 métodos para determinar o fecho convexo de um conjunto de pontos. Foi possível analisar o funcionamento de cada um dos métodos de ordenação utilizados, assim como o funcionamento da estrutura de dados pilha. Observamos que os métodos mais eficientes para resolver o problema é utilizar a Marcha de Jarvis ou o Scan de Graham com um método de ordenação de menor custo possível.

Os maiores desafios encontrados foram adaptar os algoritmos de ordenação para usar uma comparação relativa à orientação dos vetores formados pelos pontos.

# 7. Bibliografia

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Pages 949–955 of section 33.3: Finding the convex hull.

GeeksforGeeks, 2022. Convex Hull using Jarvis' Algorithm or Wrapping. Disponível em: https://www.geeksforgeeks.org/convex-hull-using-jarvis-algorithm-or-wrapping/. Acesso em: 05 jun. 2023.

GeeksforGeeks, 2023. Convex Hull using Graham Scan. Disponível em: https://www.geeksforgeeks.org/convex-hull-using-graham-scan/. Acesso em: 05 jun. 2023.

CHAIMOWICZ, L. and PRATES, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

# Apêndice A - Instruções para Compilação e Execução

Para compilar o arquivo primeiro descompacte o arquivo .zip e acesse a pasta raiz do projeto. Na pasta raiz execute o comando `make` para compilar. O executável será armazenado na pasta `bin` com o nome `programa`.

Para executar o programa acesse diretamente o programa com `./bin/programa` ou use o comando `make run`.

Ao executar o programa ele exibirá no terminal instruções de utilização. Sendo mais específico, o programa deve ser executado passando na linha de comando o caminho para um arquivo .txt com as coordenadas X e Y separadas por espaço com cada linha sendo referente a um ponto do conjunto que se deseja determinar os vértices do fecho convexo. Exemplo: `./programa /entradas/teste100.txt`

Instruções adicionais do utilitário makefile:

- `make clean` - Deleta os arquivos .o e o executável.
- `make teste` - Deleta os arquivos .o e o executável, em seguida, compila o programa novamente e o executa.