# Trabalho Prático 2

**Frederico Ferri – 2020026931**

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

ferrifrederico040@gmail.com

## 1. Introdução

Este projeto lida com a determinação do menor polígono convexo que encapsula um conjunto de pontos contidos em um eixo cartesiano. Para concretizar a funcionalidade proposta, o código desenvolvido utiliza dois algoritmos que realizam essa função: o algoritmo de feixe convexo de Graham e de Jarvis.

O primeiro algoritmo requer uma ordenação adequada dos pontos em relação ao ponto de origem, sendo necessária a implementação de algum algoritmo de ordenação. Neste projeto, foi implementado três algoritmos que cumprem esse propósito: insertionSort, mergeSort e radixSort. Por fim, tanto o algoritmo de Jarvis quanto o de Graham necessitam de estruturas de dados do tipo Pilha, o que foi implementado também.

Cada um dos algoritmos é executado e tem seu tempo de execução medido, o que nos determina o principal objetivo deste trabalho prático: analisar o comportamento de algoritmos e estruturas de dados com base na quantidade de elementos a serem processados.

## 2. Método

### 2.1. Configurações da máquina

Sistema operacional: WSL - Ubuntu 20.04 LTS
Linguagem de programação: C++
Compilador: G++ / GNU Compiler Collection.
Processador: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
Memória RAM: 8 GB

## 2.2. Estruturas de dados

Na implementação deste trabalho, a estrutura de dados utilizada é a Pilha, que permite aos dois algoritmos de geração de feixe convexo operar perante os pontos analisados. O TAD Pilha armazena objetos de tipo Ponto e possui tamanho máximo equivalente a quantidade de pontos lidos no arquivo de entrada. Para realizar essa operação, as posições da pilha são alocadas dinamicamente.

À medida que os dois algoritmos são executados, os pontos são empilhados e desempilhados, e o feixe convexo definitivo é armazenado nessa pilha de forma ordenada. Essa estrutura de dados é interessante para esse caso de implementação, pois garante aos algoritmos fácil adição/remoção de elementos, além de facilitar o acesso a eles.

## 2.3. Classes

Foram utilizadas duas classes no projeto: classe Pilha e classe Ponto.

A classe Ponto define os pontos a serem tratados, tendo como parâmetros as suas coordenadas no plano cartesiano. Essa classe facilita o acesso aos valores que referenciam os pontos, o que permite a outros algoritmos operarem sobre o conjunto de pontos.

A classe Pilha armazena os pontos tratados pelos algoritmos de feixe convexo, possuindo como parâmetros um array de objetos tipo Ponto com tamanho n (equivalente ao número de pontos lidos no arquivo de entrada), além de possuir parâmetros que definem o tamanho da pilha e o elemento topo da pilha.

## 2.4. Funções

O código possui quatro funções principais, **grahamHull(), jarvisHull(), insertionSort(), mergeSort(), radixSort(), radius(), distSq() e orderHelp().**

A função **grahamHull()** é a função de implementação do algoritmo de Graham, que consiste em encontrar o ponto com menor Y do conjunto de pontos e, a partir deste ponto, desenvolver o feixe convexo.

A função **jarvisHull()** é a função de implementação do algoritmo de Jarvis, que consiste em encontrar o ponto com menor X do conjunto de pontos e, a partir deste ponto, desenvolver o feixe convexo.

A função **radius()** calcula o ângulo em radianos entre dois pontos. Essa função é crucial para o funcionamento dos métodos de ordenação, pois o valor retornado é utilizado como referência para a ordenação dos pontos.

A função **distSq()** calcula a distância euclidiana entre dois pontos. Essa função é também crucial para ordenação de pontos, pois será utilizada par diferenciar pontos com mesmo ângulo.

A função **orderHelp**() opera sobre o array ordenado. Para pontos que possuem o mesmo ângulo, essa função garante que pontos com distância euclidiana menor estejam afrente de outros com distância maior. Essa ação requer o uso da função distSq().

A função **insertionSort()** é a implementação do método de ordenação insertion sort, que consiste em comparar um elemento da parte não ordenada e o inserir na posição correta da parte ordenada. Esse processo se repete até a lista de pontos estar totalmente ordenada de acordo com o valor do ângulo em comparação com o ponto inicial, disponibilizado por radius().

A função **mergeSort()** é a implementação do método de ordenação merge sort, que consiste em dividir a lista de elementos em dois e repetir o processo até que as sublistas possuam tamanho = 1. Em seguida, as listas são mescladas em pares e combinadas, gerando listas maiores até a lista retornar ao seu tamanho original. Os pontos são ordenados de acordo com o valor do ângulo em comparação com o ponto inicial, disponibilizado por radius().

A função **radixSort()** é a implementação do método de ordenação radix sort, que consiste em classificar os pontos tendo como base o dígito menos significativo até que todos os dígitos sejam considerados. Cada elemento é colocado em um

compartimento correspondente ao seu dígito e recolocado em uma lista de acordo com a lista de compartimentos. Os pontos são ordenados de acordo com o valor do ângulo em comparação com o ponto inicial, disponibilizado por radius().

# 3. Análise de complexidade

**Radius(): Complexidade de tempo:** O(1), pois realiza sempre apenas um cálculo do ângulo em radianos.

**Radius(): Complexidade de espaço:** O(1), pois opera sempre sobre dois valores dos pontos.

**distSq(): Complexidade de tempo:** O(1), pois realiza sempre apenas um cálculo da distância euclidiana.

**distSq(): Complexidade de espaço:** O(1), pois opera sempre sobre dois valores dos pontos.

**orderHelp(): Complexidade de tempo:** O(n), tendo em vista que realiza n-1 comparações sobre o array de pontos.

**orderHelp(): Complexidade de espaço:** O(1), tendo em vista que opera sobre o próprio array, não precisando de espaço adicional.

**insertionSort(): Complexidade de tempo:** O(n^2), tendo em vista que realiza n-1 comparações sobre o array de pontos.

**insertionSort(): Complexidade de espaço:** O(1), tendo em vista que opera sobre o próprio array, não precisando de espaço adicional.

**mergeSort(): Complexidade de tempo:** O(n log n), tendo em vista que realiza n-1 comparações sobre o array de pontos.

**mergeSort(): Complexidade de espaço:** O(n), tendo em vista que precisa de espaço adicional para realocar os valores nas etapas de merge.

**radixSort(): Complexidade de tempo:** O(n), tendo em vista que opera sobre todas as posições do array.

**radixSort(): Complexidade de espaço:** O(n), tendo em vista que precisa de espaço adicional para armazenar os valores.

**grahamHull(): Complexidade de tempo:** O(n^2), tendo em vista que, para cada ponto, deve comparar com todos os elementos sua respectiva orientação.

**grahamHull(): Complexidade de espaço:** O(n), tendo em vista que precisa de alocar uma pilha de tamanho n para realizar as operações.

**jarvisHull(): Complexidade de tempo:** O(n^2), tendo em vista que, para cada ponto, deve comparar com todos os elementos sua respectiva orientação.

**jarvisHull(): Complexidade de espaço:** O(n), tendo em vista que precisa de alocar uma pilha de tamanho n para realizar as operações.

# 4. Análise de robustez

O código está completamente comentado, além de possuir variáveis e funções com nomes de fácil compreensão. Além disso, pelo fato de estarmos lidando com arquivos externos, há funções que verificam a abertura adequada desses arquivos e garantem, com isso, uma leitura correta dos dados. Outro exemplo da robustez do código está na função radius(), que possui artifícios que impedem divisões por zero. Por fim, podemos ver a robustez o código na implementação da pilha, que não permite remover elementos em uma pilha vazia nem adicionar mais elementos em uma pilha cheia.

# 5. Análise experimental

No código, a execução dos algoritmos de cálculo de feixo tem seu tempo de execução medido.
Para número de pontos = 100:

GRAHAM+MERGESORT: 0.001s
GRAHAM+INSERTIONSORT: 0.002s
GRAHAM+RADIX: 0.000s
JARVIS: 0.000s

Para número de pontos = 1000:

GRAHAM+MERGESORT: 0.100s
GRAHAM+INSERTIONSORT: 0.203s
GRAHAM+RADIX: 0.003s
JARVIS: 0.001s

Os resultados estão de acordo com a análise de complexidade. Nota-se que ao aumentar o número de elementos em 10 vezes, a execução dos dois algoritmos é aumentado em 100x. A função radixSort possui um comportamento linear, apresentando maior eficiência de tempo em comparação com os demais métodos.

Nota-se também que o algoritmo jarvis é mais eficiente no quesito tempo de execução.

# 6. Conclusões

O projeto desenvolvido atende de forma eficiente e objetiva a proposta de encontrar polígonos que encapsulem por completo um conjunto de pontos. Isso foi possível por meio da aplicação de conceitos apresentados em sala de aula, principalmente a respeito da implementação de estruturas de dados e algoritmos eficientes.

Durante o desenvolvimento houve dificuldades, principalmente a respeito da ordenação de pontos com ângulos equivalentes, que necessitavam de ser ordenados seguindo duas ordens de prioridade. Com o intuito de solucionar esse problema, os algoritmos de ordenação utilizados no projeto tiveram de ser aprimorados para atender às especificidades do problema, aumentando o grau de sofisticação do código.

# 7. Bibliografia


Convex Hull using Graham Scan - GeeksforGeeks

Insertion Sort - Data Structure and Algorithm Tutorials - GeeksforGeeks

Merge Sort - Data Structure and Algorithms Tutorials - GeeksforGeeks

Radix Sort - Data Structures and Algorithms Tutorials - GeeksforGeeks

(215) Convex hulls: Jarvis march algorithm (gift-wrapping) - Inside code - YouTube

Estrutura de Dados – Pilhas e filas. Professores: Luiz Chaimowicz e Raquel Prate – Departamento de ciência da computação – Universidade Federal de Minas Gerais

Estrutura de Dados –Biblioteca memlog Aplicativo analisamem. Professores: Marcio Santos e Wagner Meira Jr – Departamento de ciência da computação – Universidade Federal de Minas Gerais

# 8. Instruções de compilação e execução

Diretamente do diretório **/TP2** digite o comando

```
make
```

Em seguida, digite a seguinte linha de comando:

```
./fecho EXEMPLO
```

Onde EXEMPLO é o nome de um arquivo a ser lido, com a extensão. Um exemplo de entrada seria:

```
./fecho ENTRADA100.txt
```

O arquivo .txt deve estar contido no diretório raiz, ou então deve ser informada a localidade deste arquivo ao digitar o comando.

Para limpar o diretório **/obj** que contém os arquivos .o do projeto, basta digitar

```
make clean
```