UNIVERSIDADE FEDERAL
DE MINAS GERAIS

DOCUMENTAÇÃO - TRABALHO PRÁTICO 2

FECHO CONVEXO

ALUNO: FELIPPE VELOSO MARINHO
MATRÍCULA: 2021072260
DISCIPLINA: ESTRUTURA DE DADOS
PROFESSOR: MARCIO COSTA SANTOS

1 - Introdução

O objetivo deste trabalho é documentar a implementação de um sistema que calcula o fecho convexo de um conjunto de pontos utilizando algoritmos de ordenação. O sistema foi desenvolvido utilizando as linguagens C/C++ e baseia-se nos conceitos estudados na disciplina de Estruturas de Dados.

O sistema recebe como entrada um arquivo contendo uma sequência de pontos no plano cartesiano. Existem quatro configurações possíveis para o cálculo do fecho convexo:

1. Graham + Mergesort
2. Graham + Insertion Sort
3. Graham + Counting Sort, Bucket Sort ou Radix Sort
4. Jarvis

Ao executar as quatro configurações, o programa mede o tempo de execução de cada uma delas, desconsiderando o tempo necessário para a leitura do arquivo. Além disso, o programa determina o fecho convexo dos pontos de entrada. Os pontos que representam os vértices do fecho convexo e os tempos de execução de cada método são impressos como saída na saída padrão. Os tempos são exibidos em segundos, com três casas decimais.

Para garantir a robustez do programa, foram implementados vários mecanismos de segurança, incluindo o uso de exceções, registros de memória e monitoramento de desempenho.

2 - Método

2.1 - Configurações da Máquina

- Windows 10 Pro x64
- WSL 2 com Ubuntu 20.04.06 LTS
- Processador: AMD Ryzen 5 5500U 2100 Mhz
- RAM: 8,00 GB (utilizável: 7,80 GB)
- Compilador: MinGW GCC Build 2 - 9.2.0
- Linguagem: C++

2.2 - Estruturas de Dados

No presente trabalho, foi adotada uma estrutura de dados específica para a implementação do algoritmo de fecho convexo, utilizando uma pilha de arranjo de pontos. Essa estrutura é semelhante àquela apresentada nos slides disponibilizados aos alunos.

A pilha de pontos foi utilizada para armazenar os pontos pertencentes ao fecho convexo. Após a execução das funções de Graham, responsáveis por definir o fecho convexo, os pontos resultantes foram armazenados na pilha. A pilha foi definida com um tamanho máximo (MAXTAM) de 10000 posições.

A implementação da pilha de pontos segue um modelo básico, com operações de empilhar e desempilhar para adicionar e remover elementos, respectivamente. Além disso, foram incluídas as funções NextToTop, Limpa e Topo. A função NextToTop retorna o ponto abaixo do topo da pilha a Limpa tem a finalidade de resetar a pilha, enquanto a função Topo retorna o índice do elemento que está no topo da pilha.

Essa abordagem utilizando uma pilha de arranjo de pontos permitiu o armazenamento adequado dos pontos do fecho convexo e a sua posterior impressão quando necessário.

2.3 - Classes e suas Funções

Para a implementação do programa, foi utilizada a classe Pilha. As outras classes do programa são point, jarvis, graham, sort, tmp, e conversao.

A classe "Exceptions" foi incorporada ao projeto com o objetivo de aumentar a robustez do sistema, e será detalhada no tópico 4 - Estratégia de Robustez.

A classe `**Point**` representa um ponto no plano cartesiano, e possui as seguintes funções:

- **`Point()`:** Construtor padrão da classe `Point`. Não recebe nenhum parâmetro. Utilizado para criar um objeto `Point` com coordenadas x e y inicializadas com zero.
- **`Point(int x, int y)`:** Construtor da classe `Point` que recebe dois parâmetros inteiros x e y. Utilizado para criar um objeto `Point` com as coordenadas fornecidas.
- **`~Point()`:** Destrutor da classe `Point`. Não possui implementação neste caso específico.
- **`int getX()`:** Função que retorna a coordenada x do ponto.
- **`int getY()`:** Função que retorna a coordenada y do ponto.
- **`static Point p0`:** Declaração de uma variável estática `p0` do tipo `Point`. Provavelmente utilizada para armazenar um ponto de referência.
- **`int orientation(Point p, Point q, Point r)`:** Função que calcula a orientação de três pontos, p, q e r, retornando um valor inteiro que indica se os pontos estão em sentido horário, anti-horário ou são colineares.
- **`int getOrientation(const Point &p, const Point &q, const Point &r)`:** Função semelhante à anterior, mas recebe os pontos p, q e r como parâmetros passados por referência.
- **`int distSq(Point p1, Point p2)`:** Função que calcula o quadrado da distância entre dois pontos p1 e p2, retornando um valor inteiro.
- **`void swap(Point &p1, Point &p2)`:** Função que troca os valores de dois pontos p1 e p2, passados por referência.
- **`double computeAngle(Point p)`:** Função que calcula o ângulo entre este ponto e outro ponto p fornecido como parâmetro, retornando um valor em ponto flutuante (double).

- **`int distSquare(Point p)`:** Função que calcula o quadrado da distância entre este ponto e outro ponto p fornecido como parâmetro, retornando um valor inteiro.
- **`int verifyCollinearPoints(Point *points, int n)`:** Função que verifica se um conjunto de pontos é colinear. Recebe um array de pontos e o número total de pontos como parâmetros, retornando um valor inteiro que indica se os pontos são colineares ou não.
- **`bool isPointInHull(const Point &point, Point hull[], int n)`:** Função que verifica se um ponto está dentro do fecho convexo. Recebe um ponto, um array de pontos representando o fecho convexo e o número total de pontos no fecho convexo como parâmetros, retornando um valor booleano que indica se o ponto está dentro do fecho convexo ou não.

Essas funções são utilizadas para diversas operações relacionadas ao cálculo do fecho convexo, como cálculo de distâncias, orientações, verificação de pontos colineares, entre outras.

A classe `***Sort***` possui diversas funções relacionadas a algoritmos de ordenação e manipulação de pontos. Vou explicar cada uma delas:

- **`bool comparePoints(Point a, Point b)`:** Função de comparação utilizada no algoritmo de ordenação MergeSort. Recebe dois pontos `a` e `b` como entrada e retorna um valor booleano que indica se `a` é menor que `b`.
- **`void merge(Point *points, int inicio, int meio, int fim, Point p0)`:** Implementação do algoritmo de ordenação MergeSort. Recebe um array de pontos `points`, os índices `inicio`, `meio` e `fim` que delimitam as partes do array a serem ordenadas, e o ponto de referência `p0` utilizado para comparar os pontos durante a ordenação.
- **`void mergeSort(Point *points, int left, int right, Point p0)`:** Implementação recursiva do algoritmo de ordenação MergeSort. Recebe um array de pontos `points`, os índices `left` e `right` que delimitam as partes do array a serem ordenadas, e o ponto de referência `p0` utilizado para comparar os pontos durante a ordenação.
- **`int compare(Point p1, Point p2, Point p0)`:** Função utilizada para comparar três pontos `p1`, `p2` e `p0`. Retorna um valor inteiro que indica a ordem dos pontos, com base na sua posição em relação a `p0`.
- **`Point nextToTop(Pilha &stk)`:** Função que retorna o segundo elemento do topo de uma pilha `stk` de pontos.
- **`int findMax(Point array[], int size)`:** Função que encontra o índice do ponto com o maior valor de coordenada y em um array `array` de pontos de tamanho `size`. Retorna o índice do ponto com o maior valor de y.
- **`void Insertion(Point *points, int n, Point p0)`:** Implementação do algoritmo de ordenação por inserção (Insertion Sort). Recebe um array de pontos `points`, o número de pontos `n` e o ponto de referência `p0`.
- **`void Radix(Point *points, int n)`:** Implementação do algoritmo de ordenação Radix Sort. Recebe um array de pontos `points` e o número de pontos `n`.
- **`void countingSortByDigit(Point* points, int n, int exp)`:** Implementação do algoritmo de ordenação Counting Sort por dígito. Recebe um array de pontos `points`, o número de pontos `n` e o expoente `exp` que representa a posição do dígito a ser considerado durante a ordenação.

Essas funções são responsáveis por realizar diferentes algoritmos de ordenação e auxiliar nas operações de comparação e manipulação de pontos necessárias para o cálculo do fecho convexo.

A classe `***Tmp***` possui duas funções:

- **`float Tmp::diffUserTime(struct rusage *start, struct rusage *end)`:** Esta função calcula a diferença de tempo de usuário entre duas estruturas `rusage`. Recebe como entrada um ponteiro para a estrutura `start` que representa o início do intervalo de tempo e um ponteiro para a estrutura `end` que representa o final do intervalo de tempo. A função retorna um valor float que representa a diferença de tempo de usuário em segundos.
- **`float Tmp::diffSystemTime(struct rusage *start, struct rusage *end)`:** Esta função calcula a diferença de tempo de sistema entre duas estruturas `rusage`. Recebe como entrada um ponteiro para a estrutura `start` que representa o início do intervalo de tempo e um ponteiro para a estrutura `end` que representa o final do intervalo de tempo. A função retorna um valor float que representa a diferença de tempo de sistema em segundos.

Essas funções são utilizadas para calcular a diferença de tempo entre dois pontos específicos durante a execução de um programa. Elas recebem as estruturas `rusage`, que são usadas para medir o tempo de usuário e tempo de sistema do programa, e retornam a diferença de tempo em segundos.

A classe `**Jarvis**` possui as seguintes funções:

- **`Jarvis::Jarvis()`**: Construtor da classe `Jarvis`. É chamado quando um objeto `Jarvis` é criado. Não possui entrada nem saída.
- **`Jarvis::~Jarvis()`**: Destrutor da classe `Jarvis`. É chamado quando um objeto `Jarvis` é destruído. Não possui entrada nem saída.
- **`int Jarvis::convexHull(Point points[], int n, Point hull[], bool returnOnlyTwoPoints)`:** Função que encontra o fecho convexo de um conjunto de pontos. Recebe como entrada um array de pontos `points`, o número de pontos `n`, um array `hull` para armazenar o fecho convexo encontrado e um parâmetro `returnOnlyTwoPoints` que indica se deve retornar apenas dois pontos caso todos os pontos do array sejam colineares. Retorna um valor inteiro representando o número de pontos no fecho convexo.
- **`void Jarvis::drawConvexHull(Point points[], int n, Point hull[], int nPoints)`:** Função para desenhar o fecho convexo de um conjunto de pontos. Recebe como entrada um array de pontos `points`, o número de pontos `n`, um array `hull` com os pontos do fecho convexo e o número de pontos no fecho convexo `nPoints`. Não possui saída.

Essas funções são utilizadas para encontrar e desenhar o fecho convexo de um conjunto de pontos. A função `convexHull` implementa o algoritmo de Marcha de Jarvis para encontrar o fecho convexo e retorna o número de pontos no fecho convexo. A função `drawConvexHull` utiliza a biblioteca gráfica `graphics.h` para desenhar os pontos e as linhas do fecho convexo em uma janela de visualização. Foi utilizada uma regra de 3 para que todos os desenhos permaneçam dentro das limitações de tela da biblioteca graphics.h, devido a isso ao usar a flag “-d” para desenhar, no terminal é printado alguns erros de fora dos limites mesmo com a animação do convex hull sendo feita.

Essa classe, denominada "Graham", implementa algoritmos para encontrar o fecho convexo de um conjunto de pontos. Ela possui três funções principais:

- "**MergeConvexHull**":
  - Descrição: Essa função utiliza o algoritmo de ordenação "mergesort" para ordenar os pontos em relação ao ponto mais à esquerda. Em seguida, ela encontra o fecho convexo desses pontos usando o algoritmo de Graham.
  - Entrada: Um array de pontos (do tipo "Point") e o número de pontos no array.
  - Saída: Nenhum retorno específico.
- "**InsertionConvexHull**":
  - Descrição: Essa função utiliza o algoritmo de ordenação "insertionsort" para ordenar os pontos em relação ao ponto mais à esquerda. Em seguida, ela encontra o fecho convexo desses pontos usando o algoritmo de Graham.
  - Entrada: Um array de pontos (do tipo "Point") e o número de pontos no array.
  - Saída: Nenhum retorno específico.
- "**RadixConvexHull**":
  - Descrição: Essa função encontra o ponto mais à esquerda e ordena os pontos restantes pelo ângulo polar em relação a esse ponto. Em seguida, ela remove os pontos intermediários colineares e encontra o fecho convexo dos pontos usando o algoritmo de **Graham**.
  - Entrada: Um array de pontos (do tipo "Point") e o número de pontos no array.
  - Saída: Nenhum retorno específico.

Além dessas funções, a classe também possui um construtor padrão e um destrutor padrão. O construtor não recebe nenhum parâmetro e o destrutor não possui implementação específica.
A classe depende de outras classes e estruturas como "Sort", "Pilha" e "Point" para realizar as operações necessárias.

A última classe usada foi a memlog utilizada nas aulas anteriores que realiza registros de acessos à memória, permitindo o monitoramento e registro de leituras e escritas realizadas em posições de memória específicas.

3 - Análise de Complexidade

As funções a serem analisadas abaixo são as mais relevantes para o projeto, não serão consideradas nas análises as funções das estruturas de dados e nem as funções auxiliares usadas no programa.

A função `**MergeConvexHull**` utiliza o algoritmo de ordenação mergesort para ordenar um conjunto de pontos e encontrar o fecho convexo. A complexidade de tempo do mergesort é O(n log n), onde n é o número de pontos. Além disso, a função realiza algumas operações adicionais, como encontrar o ponto mais à esquerda, remover pontos colineares e empilhar os pontos que formam o fecho convexo. Essas operações adicionais têm complexidade de tempo O(n), onde n é o número de pontos. Portanto, a complexidade de tempo total da função `MergeConvexHull` é O(n log n).

A função `**InsertionConvexHull**` utiliza o algoritmo de ordenação insertionsort para ordenar um conjunto de pontos e encontrar o fecho convexo. A complexidade de tempo do insertionsort é O(n^2), onde n é o número de pontos. Assim como na função anterior, há operações adicionais de encontrar o ponto mais à esquerda, remover pontos colineares e

empilhar os pontos que formam o fecho convexo, com complexidade de tempo O(n). Portanto, a complexidade de tempo total da função `InsertionConvexHull` é O(n^2).

A função `**RadixConvexHull**` utiliza o algoritmo de ordenação radixsort para ordenar um conjunto de pontos e encontrar o fecho convexo. A complexidade de tempo do radixsort depende do número de dígitos dos números a serem ordenados. Considerando que os pontos possuem coordenadas inteiras, a quantidade de dígitos é limitada e, portanto, a complexidade de tempo do radixsort é O(d * (n + k)), onde d é o número de dígitos, n é o número de pontos e k é a base do sistema de numeração utilizado (nesse caso, k = 10). Além disso, há operações adicionais de encontrar o ponto mais à esquerda, remover pontos colineares e empilhar os pontos que formam o fecho convexo, com complexidade de tempo O(n). Assim, a complexidade de tempo total da função `RadixConvexHull` é O(d * (n + k)). Em relação à complexidade de espaço, as funções utilizam algumas variáveis locais para armazenar os pontos, pilhas, e outras informações temporárias. A quantidade de espaço utilizado depende do número de pontos e de outras variáveis auxiliares utilizadas. No entanto, como essas variáveis não são proporcionais ao tamanho do conjunto de pontos, a complexidade de espaço das funções é considerada O(1), ou seja, espaço constante.

A complexidade de tempo e espaço das funções da classe **Jarvis** pode ser analisada da seguinte forma:

`Jarvis::convexHull`:

- Complexidade de tempo: O algoritmo utilizado é conhecido como "Marcha de Jarvis" ou "Algoritmo do Envelope Convexo". A complexidade de tempo desse algoritmo é O(nh), onde n é o número de pontos de entrada e h é o número de pontos no fecho convexo resultante. O laço externo percorre todos os pontos do fecho convexo, o que implica em O(h) iterações. Dentro desse laço, o algoritmo busca o próximo ponto do fecho convexo em O(n) iterações. Portanto, a complexidade de tempo é O(nh).
- Complexidade de espaço: A função utiliza uma quantidade constante de espaço adicional para as variáveis locais, como p0, leftmost, p, q, idx, etc. Além disso, os parâmetros points, hull e returnOnlyTwoPoints são passados por referência, portanto, não há consumo adicional de espaço para eles. A complexidade de espaço é, portanto, O(1).

`Jarvis::drawConvexHull`:

- Complexidade de tempo: A função percorre os pontos de entrada duas vezes. Primeiro, encontra o ponto com as maiores coordenadas para determinar a escala de proporção em O(n) iterações. Em seguida, encontra o ponto mais à esquerda para começar a desenhar em O(n) iterações. Em seguida, desenha os pontos que estão fora do fecho convexo em O(n) iterações. Finalmente, desenha as retas que formam o fecho convexo em O(n) iterações e desenha os pontos do fecho convexo em O(nPoints) iterações, onde nPoints é o número de pontos no fecho convexo. Portanto, a complexidade de tempo é O(n + n + n + n + nPoints) = O(4n + nPoints) = O(n + nPoints).
- Complexidade de espaço: A função utiliza uma quantidade constante de espaço adicional para as variáveis locais, como p0, xMax, yMax, xMaxPoint, yMaxPoint, xScale, yScale, gd, gm, msg, etc. Além disso, os parâmetros points, hull e nPoints são passados por referência, portanto, não há consumo adicional de espaço para eles. A complexidade de espaço é, portanto, O(1). Em resumo, a função `Jarvis::convexHull` tem complexidade de tempo O(nh) e complexidade de espaço

O(1), enquanto a função `Jarvis::drawConvexHull` tem complexidade de tempo O(n + nPoints) e complexidade de espaço O(1).

A função `**merge**` tem complexidade de tempo O(n), onde n é a quantidade de elementos a serem mesclados. Isso ocorre porque ela percorre as sublistas uma vez, comparando os elementos e mesclando-os em um único vetor. Em relação à complexidade de espaço, a função utiliza dois arrays temporários `pointStart` e `pointEnd` para armazenar as sublistas ordenadas. Cada um desses arrays tem tamanho p1 e p2, respectivamente, que são as quantidades de elementos das sublistas. Portanto, a complexidade de espaço é O(p1 + p2).

A função `**mergeSort**` tem complexidade de tempo O(n log n), onde n é o número total de elementos a serem ordenados. Isso ocorre porque ela divide repetidamente a lista de elementos pela metade e, em seguida, mescla as sublistas ordenadas. Em relação à complexidade de espaço, a função utiliza espaço adicional para os arrays temporários `pointStart` e `pointEnd`, que têm tamanho p1 e p2, respectivamente. Além disso, a recursão consome espaço na pilha de chamadas. No pior caso, em que a recursão atinge o nível mais profundo, a complexidade de espaço é O(log n).

A função `**Insertion**` tem complexidade de tempo O(n^2), onde n é o número total de elementos a serem ordenados. Isso ocorre porque ela percorre a lista de elementos e, para cada elemento, realiza uma comparação e possivelmente uma troca com os elementos anteriores, movendo-os para a frente. Em relação à complexidade de espaço, a função não utiliza espaço adicional além do necessário para as variáveis locais, portanto a complexidade de espaço é O(1).

A função `**findMax**` tem complexidade de tempo O(n), onde n é o número total de elementos no array. Isso ocorre porque ela percorre o array uma vez para encontrar o valor máximo. Em relação à complexidade de espaço, a função não utiliza espaço adicional além do necessário para as variáveis locais, portanto a complexidade de espaço é O(1).

A função `**Radix**` tem complexidade de tempo O(d * (n + k)), onde n é o número total de elementos a serem ordenados, k é o intervalo de dígitos (10 no caso dos dígitos decimais) e d é o número máximo de dígitos dos elementos. No caso do algoritmo radix sort, o número de dígitos d é determinado pelo número máximo de dígitos do maior elemento do array. Em relação à complexidade de espaço, a função utiliza espaço adicional para o array `output`, que tem tamanho n, e o array `count`, que tem tamanho k. Além disso, a recursão consome espaço na pilha de chamadas. No pior caso, em que a recursão atinge o nível mais profundo, a complexidade de espaço é O(n + k + d).

A função `**countingSortByDigit**` tem complexidade de tempo O(n + k), onde n é o número total de elementos a serem ordenados e k é o intervalo de dígitos (10 no caso dos dígitos decimais). Isso ocorre porque ela realiza duas etapas principais: a contagem da frequência dos dígitos e a construção do array de saída ordenado pelos dígitos. Em relação à complexidade de espaço, a função utiliza espaço adicional para o array `output`, que tem tamanho n, e o array `count`, que tem tamanho k. Além disso, a função não utiliza espaço adicional além do necessário para as variáveis locais. Portanto, a complexidade de espaço é O(n + k).

4 - Estratégias de Robustez

De modo a garantir a robustez e a corretude do programa após a entrada do usuário passada pelo argumento, foi empregado o uso de “exceptions”, semelhante aos

“msgasserts” visto em C.

Foi criada uma classe com diversas exceções personalizadas onde são disparadas caso ocorra uma excessão no código.

Algumas das exceções colocadas incluem:

- As excessões de arquivo não encontrado, caso os ponteiros passados como parâmetros sejam apontados para o nulo;
- As excessões das classes pilhas, que verificam se a pilha está vazia ou cheia;
- Excessões de coordenadas inválidas, caso as coordenadas passadas sejam maior que a biblioteca graphics permita, apesar que usando a gambiarra da regra de 3 ela ficou em desuso.
- Excessão de tamanho de array inválido, caso seja menor que 2 pontos;

Uma potencial fragilidade na resistência da aplicação é que a entrada do programa deve estar em conformidade com os formatos fornecidos para testes no moodle. Isso ocorre porque toda a leitura do tipo de expressões se baseia nessas especificações. No entanto, para o contexto do trabalho prático, essa questão pode ser deixada de lado.

5 - Análise Experimental

Para a análise de desempenho computacional, assim como no TP 1, utilizei a ferramenta gprof. O objetivo dessa análise foi identificar os métodos que mais afetaram o desempenho do programa e entender como o tamanho e o tipo da expressão influenciam no tempo de execução, através da contagem do número de chamadas realizadas.

Ao analisar um exemplo específico, com a entrada de 10 pontos, observamos que houve um grande número de chamadas para as funções Point e ~Point. Em particular, foram registradas 40066 chamadas para a função Point e 240764 chamadas para a função ~Point. Esses valores evidenciam a importância dessas operações no tempo total de execução.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	240764	0.00	0.00	Point::~Point()
0.00	0.00	0.00	40066	0.00	0.00	Point::Point()
0.00	0.00	0.00	123	0.00	0.00	Point::orientation(Point, Point, Point, int)
0.00	0.00	0.00	100	0.00	0.00	Point::computeAngle(Point, int)
0.00	0.00	0.00	92	0.00	0.00	Point::getY(int)
0.00	0.00	0.00	47	0.00	0.00	Point::getX(int)
0.00	0.00	0.00	46	0.00	0.00	Pilha::Empilha(Point, int)
0.00	0.00	0.00	20	0.00	0.00	Pilha::NextToTop(Pilha, int)
0.00	0.00	0.00	19	0.00	0.00	Pilha::Desempilha(int)
0.00	0.00	0.00	10	0.00	0.00	Sort::compare(Point, Point, Point, int)
0.00	0.00	0.00	9	0.00	0.00	Sort::merge(Point*, int, int, int, Point, int)
0.00	0.00	0.00	9	0.00	0.00	Sort::nextToTop(Pilha&, int)
0.00	0.00	0.00	9	0.00	0.00	Point::Point(int, int)
0.00	0.00	0.00	6	0.00	0.00	Point::isPointInHull(Point const&, Point*, int, int)
0.00	0.00	0.00	4	0.00	0.00	Point::distSquare(Point, int)
0.00	0.00	0.00	3	0.00	0.00	Point::swap(Point&, Point&, int)
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN3Tmp12diffUserTimeEP6rusageS1_
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN4Sort13comparePointsE5PointS0_i
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN5Pilha7EmpilhaE5Pointi
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN5Point2p0E
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN6GrahamC2Ev
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I_main
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I_ml
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I_screen
0.00	0.00	0.00	1	0.00	0.00	Sort::Radix(Point*, int)
0.00	0.00	0.00	1	0.00	0.00	Sort::Insertion(Point*, int, Point, int)
0.00	0.00	0.00	1	0.00	0.00	Sort::mergeSort(Point*, int, int, Point, int)
0.00	0.00	0.00	1	0.00	0.00	Point::verifyCollinearPoints(Point*, int, int)

Porém, algo curioso ocorre quando é testada a entrada de 1000 pontos. As chamadas de Point e ~Point possuem valores similares as chamadas de 10 pontos.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	240735	0.00	0.00	Point::~Point()
0.00	0.00	0.00	40055	0.00	0.00	Point::Point()
0.00	0.00	0.00	1405	0.00	0.00	leMemLog(long, long, int)
0.00	0.00	0.00	132	0.00	0.00	escreveMemLog(long, long, int)
0.00	0.00	0.00	123	0.00	0.00	Point::orientation(Point, Point, Point, int)
0.00	0.00	0.00	100	0.00	0.00	Point::computeAngle(Point, int)
0.00	0.00	0.00	92	0.00	0.00	Point::getY(int)
0.00	0.00	0.00	47	0.00	0.00	Point::getX(int)
0.00	0.00	0.00	46	0.00	0.00	Pilha::Empilha(Point, int)
0.00	0.00	0.00	20	0.00	0.00	Pilha::NextToTop(Pilha, int)
0.00	0.00	0.00	19	0.00	0.00	Pilha::Desempilha(int)
0.00	0.00	0.00	10	0.00	0.00	Sort::compare(Point, Point, Point, int)
0.00	0.00	0.00	9	0.00	0.00	Sort::merge(Point*, int, int, int, Point, int)
0.00	0.00	0.00	9	0.00	0.00	Sort::nextToTop(Pilha&, int)
0.00	0.00	0.00	9	0.00	0.00	Point::Point(int, int)
0.00	0.00	0.00	4	0.00	0.00	Point::distSquare(Point, int)
0.00	0.00	0.00	3	0.00	0.00	Point::swap(Point&, Point&, int)
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN3Tmp12diffUserTimeEP6rusageS1_
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN4Sort13comparePointsE5PointS0_i
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN5Pilha7EmpilhaE5Pointi
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN5Point2p0E
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN6GrahamC2Ev
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I_main
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I_ml
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I_screen
0.00	0.00	0.00	1	0.00	0.00	Sort::Radix(Point*, int)
0.00	0.00	0.00	1	0.00	0.00	Sort::Insertion(Point*, int, Point, int)
0.00	0.00	0.00	1	0.00	0.00	Sort::mergeSort(Point*, int, int, Point, int)
0.00	0.00	0.00	1	0.00	0.00	Point::verifyCollinearPoints(Point*, int, int)

As mudanças são percebidas quando vemos a quantidade de chamadas das funções de Empilha, Desempilha, getX e getY além da própria comparação de tempo de execução, sendo o primeiro exemplo a execução com 10 pontos e a subsequente com 1000 pontos.

```
JARVIS: 0.167s
GRAHAM + MERGESORT: 4.080s
GRAHAM + INSERTIONSORT: 0.465s
GRAHAM + RADIX: 0.038s
Geração de log finalizada!
```

```
JARVIS: 0.000s
GRAHAM + MERGESORT: 0.000s
GRAHAM + INSERTIONSORT: 0.000s
GRAHAM + RADIX: 0.000s
Geração de log finalizada!
```

Existem vários fatores que contribuem para essa situação. Um dos principais é o fato de a pilha utilizada na aplicação ter um tamanho fixo (MAXTAM) de 100000. Essa escolha foi feita para suportar grandes entradas, mas se fosse utilizada uma pilha com alocação dinâmica, provavelmente obteríamos resultados de chamadas semelhantes em Point e ~Point.

Por último, devido a não ter conseguido rodar a biblioteca analisamen, não consegui análises de melhores qualidades. Porém realmente tentei muito utilizar a biblioteca de forma correta porém a escassez de documentação acabou fazendo ser um esforço enorme sem compensação.

6 - Conclusões

O Trabalho Prático 2 abordou uma variedade de problemas relacionados ao fecho convexo, exigindo a aplicação de habilidades e conhecimentos adquiridos durante o curso, bem como conhecimentos de outras disciplinas correlatas. A implementação das funções proporcionou uma compreensão aprofundada das estruturas de dados utilizadas, dos algoritmos de ordenação e dos métodos de análise de tempo de execução. Além disso, foi uma oportunidade de testar diferentes tipos de estruturas de dados, contribuindo para o aprendizado. No entanto, vale ressaltar que a restrição de prazo para a entrega do trabalho demandou um considerável investimento de tempo. A documentação do trabalho

desempenhou um papel fundamental na consolidação do conhecimento sobre análise de complexidade e desempenho computacional, bem como na manutenção da padronização do código em conformidade com boas práticas e eficiência.

Além dos aspectos mencionados, o desafio adicional de desenhar na tela utilizando a biblioteca graphics foi uma experiência interessante e prazerosa. Embora não tenha sido possível utilizar o analisamen dentro do prazo, considero que as tentativas de aprendizado também foram valiosas.

Em resumo, o trabalho teve um impacto extremamente positivo na aplicação prática dos conhecimentos adquiridos ao longo da disciplina, proporcionando uma aprendizagem significativa.

7 - Bibliografia

Tutorial usado para usar a biblioteca graphics: TutorialGraphics.h

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.

8 - Instruções para compilação e execução

Para a compilação do arquivo é utilizado o comando make, que irá compilar todos os arquivos do programa e gerar os arquivos fontes e objetos na estrutura de pasta pedida no enunciado do Trabalho Prático. O comando make clean faz a limpeza dos arquivos objetos e do executável.

Para a execução existem duas maneiras:

- Ir no make file e adicionar o caminho do arquivo ao lado da variável $(TARGET) e executar o comando make run:

```
# Regra para compilar e executar o programa, coloque o caminho até o arquivo
# entre aspas após o comando. Exemplo: $(TARGET) "c:/Users/usuario/Desktop/en
run: all
    @$(TARGET) "../../ENTRADATESTE.txt"
    @$(GPROF_DIR) $(TARGET) gmon.out > $(REPORT)
    @echo ""
    @echo -e -n $(ANSI_GREEN)
    @echo -n "√ "
    @echo $(ANSI_GREEN) "Relatório gerado com sucesso!" $(ANSI_DEFAULT)
    @echo ""
```

- A outra maneira, a mais convencional, é feita após a compilação, passando por argumento o caminho até o arquivo com a expressão á ser lida e executando o arquivo executável gerado na pasta bin. Além de ser possíve, após a compilaçãol a passagem dos argumentos ‘-d’ para realizar a animação do fecho convexo e ‘-l’ para gerar os logs utilizados para gerar os gráficos na biblioteca analisamen.

```
√  Concluído!

Para executar o programa diretamente, digite:
----- ./bin/fecho 'ENTRADA.txt' -----

Para executar o programa e desenhar o fecho, digite:
----- ./bin/fecho 'ENTRADA.txt' -d -----

Para executar o programa e gerar os Logs para gráficos da Análise Experimental, digite:
----- ./bin/fecho 'ENTRADA.txt' -l -----
```

-