

RELATÓRIO DO TRABALHO PRÁTICO 2: RESOLVEDOR DE FECHO CONVEXO

Diogo Tuler Chaves - 2022043663

1. Introdução

Essa documentação lida com o problema de implementar um resolvidor de fecho convexo dado uma quantidade inicial de pontos.

Para a implementação de um programa com essa funcionalidade foram usados dois algoritmos clássicos, o “Jarvis March” e o “Graham scan”. Além disso, o “Graham scan”, por usar um método de ordenação ao longo de seu processo de resolução, foi implementado usando três algoritmos de ordenação diferentes: “Bucket Sort”, “Merge Sort” e o “Insertion Sort”. Por fim, foi comparado o tempo entre as 4 formas de resolução.

Foram criados 3 TAD básicos, “Ponto”, “Reta” e “Fecho_Convexo” que serão explicados mais a fundo posteriormente neste documento. Ademais, ficou claro que para a resolução do “Graham scan” seria necessário um TAD “Pilha” para ir armazenando os pontos. Tendo em vista que esse TAD teria de ser criado independente, fiz dele um “template” de classe para que pudesse ser capaz de lidar tanto com “Ponto” quanto com “Reta” e estruturei o código para não precisar de mais nenhum TAD. Em outras palavras, o código funciona inteiramente utilizando apenas esses dois tipos de pilhas e vetores(ponteiros) alocados dinamicamente, possuindo complexidade assintótica igual ou muito similar à se fossem criados outros TAD, como o “Vetor”.

Vale ressaltar que esse trabalho foi realizado em C++ e compilado com g++ em um “Lenovo Lenovo IdeaPad S145-15 IWL” com o processador “Intel® Core™ i5-8265 U CPU @ 1.60GHz × 8” com “NV 118 / Mesa Intel® UHD Graphics 620 (WHL GT2)” e com um sistema operacional “Ubuntu 22.04.2 LTS 64-bit”

2. Métodos

2.1. Organização

- **bin:** Contém o arquivo executável “main”, resultado da compilação dos arquivos .o da pasta obj pelo G++.
- **include:** Contém os arquivos “.hpp” com as declarações de funções modularizadas ou de Tipos Abstratos de Dados. Além disso, alguns arquivos acabam por conter implementações, fato que será detalhado abaixo:
 - **Pilha.hpp:** É o arquivo onde é definida e implementada a classe Elemento_da_Pilha, a classe Pilha(ambas usando template) e o struct Pilha_vazia.
 - **(template<class T>) Class Elemento_da_Pilha:** Classe que representa um elemento do tipo T da lista encadeada na memória. Foi usado um “template” pois a classe “Pilha” foi usada para abrigar elementos do tipo “Ponto” e elementos do tipo “Reta”. Pelo fato de ser um template toda a classe teve que ser implementada nesse arquivo “.hpp”. Os atributos dessa classe são:
 - **Elemento_da_Pilha():** Construtor da classe Elemento_da_Pilha.
 - **T Elemento:** Os elementos do tipo T que vão ser enfileirados.
 - **Elemento_da_Pilha<T>* Proximo:** Um ponteiro para o próximo elemento da pilha.

- **(template<class T>) Pilha:** Classe que representa uma pilha de elementos do tipo T. Foi usado um "template" para que não fosse necessário implementar duas pilhas de elementos diferentes ("Ponto" e "Reta") que possuem as mesmas funções. Pelo fato de ser um template toda a classe teve que ser implementada nesse arquivo ".hpp". Os atributos dessa classe são:
 - **Pilha():** Construtor da classe Pilha.
 - **bool Pilha_Vazia():** Função que informa se a Pilha está vazia.
 - **void Empilha(T valor):** Função que adiciona elementos do tipo T no topo da pilha.
 - **T Desempilha():** Função que retorna o elemento do tipo T que está no topo da pilha e o apaga da pilha
 - **void Destroi_Pilha():** Destroi a Pilha de forma recursiva.
 - **~Pilha():** Destrutor da classe Pilha que chama a função "Destroi_Pilha()".
 - **Elemento_da_Pilha<T> * Topo:** É o topo da pilha, sendo um elemento do tipo Elemento_da_Pilha<T>.
- **struct Pilha_vazia:** Será abordado na parte 4 deste relatório sendo uma exceção usada nos mecanismos de programação defensiva implementados.
- **Tempo.hpp:** É o arquivo onde estão definidas funções que auxiliam na hora de analisar o tempo.
 - **double Compare_Clock_Time(struct timespec Time_Clock_Begin, struct timespec Time_Clock_End):** Auxiliar para "Time_Clock_Combine".
 - **double Time_Clock_Combine(struct timespec Time_Clock):** Devolve a comparação de tempo em s.
- **Ponto.hpp:** Arquivo onde está definido a classe "Ponto".
 - **Class Ponto:** Classe que representa as coordenadas (X,Y), ou seja, um ponto no plano cartesiano.
 - **Ponto():** Construtor da classe padrão.
 - **void Monta_Ponto(int Eixo_X, int Eixo_Y):** Monta o ponto quando temos os valores de X e Y.
 - **string Retorna_Ponto():** Função que retorna o ponto no formato "(X,Y)" em uma string.
 - **float Distancia(Ponto Ponto_2):** Retorna a distância euclidiana entre dois pontos.
 - **float Angulo(Ponto Ponto_2):** Retorna o ângulo entre dois pontos.
 - **int X & int Y:** Todo ponto tem coordenadas (X,Y).
- **Reta.hpp:** Arquivo onde está definido a classe "Reta".
 - **Class Reta:** Classe que representa uma equação $Ax+B$, ou seja, uma reta no plano cartesiano.
 - **Reta():** Construtor da classe.
 - **void Monta_Reta(Ponto Ponto_1, Ponto Ponto_2):** Quando queremos construir a reta que passa entre dois pontos.
 - **string Retorna_Reta():** Função que retorna a reta no formato " $Ax+B$ " em uma string
 - **Ponto Ponto_1 & Ponto Ponto_2:** Pontos que deram origem a reta.
 - **float A & float B:** Reta tem o formato $Ax + B$.
 - **float Angulo:** Angulo formado pela reta.

- **Leitura_Entrada.hpp:** Arquivo com funções auxiliares para a leitura e análise da entrada.
 - **structs Muitos_Argumentos & Poucos_Argumentos & Arquivo_Invalido :** Serão abordados na parte 4 deste relatório sendo exceções usadas nos mecanismos de programação defensiva implementados.
 - **ifstream Abre_Arquivo(int argc, char ** argv):** Função para abrir o arquivo lido na linha de comando enviando exceções quando necessário.
 - **struct Vetor_de_Ponto_e_seu_Tamanho:** Struct para armazenar os dados de entrada em um vetor de pontos e seu tamanho.
 - **Pontos_Quantidade* Le_Arquivo(ifstream* Arquivo_de_Entrada):** Função para ler o arquivo e retornar uma Pilha de pontos.
- **Fecho_Convexo.hpp:** Arquivo onde está definido a classe “Fecho_Convexo”.
 - **structs Algoritmo_Invalido & Ordenacao_Invalida:** Serão abordados na parte 4 deste relatório sendo exceções usadas nos mecanismos de programação defensiva implementados.
 - **Class Fecho_Convexo:** Classe que representa o fecho convexo.
 - **Fecho_Convexo():** Construtor da Classe.
 - **void Montar_Fecho_Convexo(string Algoritmo, string Ordenacao, Ponto* Vetor_Entrada, int Tamanho_Entrada):** Função para montar o fecho convexo dado um vetor de pontos. Nessa função é necessário escolher o algoritmo de achar o fecho e o algoritmo de ordenação desejado. Chama as funções "Montar_com_jarvis_march" e "Montar_com_graham_scan" de acordo com a opção escolhida
 - **void Imprimir_Fecho_Convexo():** Função para imprimir o fecho armazenado.
 - **void Montar_com_jarvis_march(Ponto* Vetor_Entrada, int Tamanho_Entrada):** Chamada pela função "Montar_Fecho_Convexo" monta o fecho utilizando o algoritmo de Jarvis March. Esse algoritmo a partir do ponto com menor x são analisadas as retas formadas entre os pontos e sua orientação para determinar o fecho convexo
 - **int Orientacao(Ponto Ponto_ultimo, Ponto Ponto_atual, Ponto Ponto_analise):** Função que retorna a orientação entre duas retas formadas pelos pontos fornecidos.
 - **void Montar_com_graham_scan(Ponto* Vetor_Entrada, int Tamanho_Entrada, string Ordenacao):** Chamada pela função "Montar_Fecho_Convexo" monta o fecho utilizando o algoritmo de Graham scan. Esse algoritmo funciona primeiramente achando o ponto com o menor Y e depois ordenando os outros pontos em relação ao ângulo formado pelas retas que passam entre cada um deles e o ponto de menor Y. Depois disso, em ordem, vão sendo empilhados os possíveis pontos que podem ser o fecho, analisando a orientação das retas que vão se formando, desempilhando e empilhando novos pontos quando necessário se baseando na análise da orientação das retas. Como entrada é necessário afirmar qual algoritmo de ordenação vai ser utilizado. Essa função chama uma das

funções "Merge_sort_Graham_scan", "Insertion_sort_Graham_scan" e "Bucket_sort_Graham_scan" de acordo com o método escolhido.

- **void Merge_sort_Graham_scan(Reta* Vetor_Entrada, int Inicio, int Fim):** Chamada pela função "Montar_com_graham_scan" ordena o vetor de retas passado como argumento de acordo com o ângulo de cada reta utilizando o "MergeSort". Esse método usa a recursão para dividir o vetor ao máximo e, posteriormente, une essas divisões de modo que o vetor fique em ordem no final.
 - **void Merge(Reta* Vetor_Entrada, int Inicio, int Metade, int Fim):** Chamada pela função "Merge_sort_Graham_scan" auxilia na ordenação por "MergeSort".
 - **void Insertion_sort_Graham_scan(Reta* Vetor_Entrada, int Tamanho_Entrada):** Chamada pela função "Montar_com_graham_scan" ordena o vetor de retas passado como argumento de acordo com o ângulo de cada reta utilizando o "InsertionSort". Esse método passa por cada elemento da esquerda para direita e ordena os elementos à direita do elemento atual, movendo-os para a direita se forem maiores, até encontrar a posição correta para o inserir.
 - **void Bucket_sort_Graham_scan(Reta* Vetor_Entrada, int Tamanho_Entrada):** Chamada pela função "Montar_com_graham_scan" ordena o vetor de retas passado como argumento de acordo com o ângulo de cada reta utilizando o "BucketSort". Esse método divide o vetor em “baldes” de acordo com a faixa que seu ângulo ocupa, depois ordena cada balde e junta mantendo a ordenação.
 - **Pilha<Ponto> Fecho:** Uma pilha de pontos que representa o fecho convexo em si.
-
- **obj:** Contém os arquivos .cpp da pasta src traduzidos para linguagem de máquina pelo compilador GCC e com a extensão .o (object).
 - **src:** Contém as implementações dos arquivos .hpp da pasta “include” em arquivos .cpp, além disso contém a função main que utiliza as funções de Leitura_Entrada.hpp para organizar a entrada e a saída de dados. Nestes arquivos estão programadas toda a lógica e a robustez contra erros do programa.

3. Análise de Complexidade

Para uma análise melhor do código será feita uma análise da complexidade apenas das funções mais expressivas e que realmente influenciam no custo do código.

- **Reta & Ponto .hpp/.cpp**
 - Todas suas funções são $O(1)$
- **Leitura_Entrada .cpp/.hpp**
 - **Pontos_Quantidade* Le_Arquivo(ifstream* Arquivo_de_Entrada):** Possui dois laços que passam pelos n pontos de entrada, logo sua complexidade de tempo é $2n$ o que equivale a $\Theta(n)$. Em relação a espaço, é criada uma pilha de pontos de tamanho n

que é convertida em um vetor de pontos de n posições, logo sua complexidade de espaço $\Theta(n)$.

- **Pilha .hpp/.cpp**

- **void Empilha(T valor) & T Desempilha():** Tanto o custo de espaço quanto o de tempo são constantes sendo $\Theta(1)$.
- **~Pilha():** Sendo a variável 'k' o tamanho da fila no final do escopo, essa função chama a função "T Desempilha()" k vezes, logo sua complexidade de tempo é $\Theta(k)$ e a função theta da sua complexidade de espaço é dada pelo tamanho da "Elemento_da_Pilha<T>* Auxiliar" multiplicado por k.

- **Fecho_Convexo.hpp:**

- **int Orientacao(Ponto Ponto_ultimo,Ponto Ponto_atual,Ponto Ponto_analise):** Essa função apenas realiza uma operação sem loop e armazena apenas um float auxiliar, logo seu custo de tempo e espaço é $\Theta(1)$.
- **void Montar_Fecho_Convexo(string Algoritmo, string Ordenacao,Ponto* Vetor_Entrada,int Tamanho_Entrada):** Sua complexidade depende do algoritmo escolhido:
 - **void Montar_com_jarvis_march(Ponto* Vetor_Entrada,int Tamanho_Entrada):** Seu custo de espaço é $\Theta(1)$, ou seja, constante, pois nenhum vetor auxiliar é alocado. Já sua complexidade de tempo é dada por $\Theta(n * k)$ sendo n o tamanho da entrada e k o tamanho do fecho convexo.
 - **void Montar_com_graham_scan(Ponto* Vetor_Entrada,int Tamanho_Entrada,string Ordenacao):** Seu de espaço é $2n$ (n o tamanho da entrada) pois são criados um vetor de retas e um de pontos auxiliares, logo sua complexidade de espaço é dada por $\Theta(n)$ independente do método de ordenação. Já sua complexidade de tempo depende do algoritmo de ordenação escolhido, sendo igual a $\Theta(n)$ mais a complexidade do método de ordenação escolhido que, de fato, acaba sobressaindo na hora de calcular o custo assintótico total:
 - **void Merge_sort_Graham_scan(Reta* Vetor_Entrada, int Inicio,int Fim):** O Merge Sort possui uma complexidade de tempo que equivale a $\Theta(n \log n)$, sendo n o tamanho do vetor. Cada vez que duas partições vão ser juntadas é criado um vetor auxiliar para armazená-las, isso equivale a uma complexidade de espaço igual a $\Theta(n)$.
 - **void Insertion_sort_Graham_scan(Reta* Vetor_Entrada,int Tamanho_Entrada):** O Insertion Sort possui uma complexidade de tempo que equivale a $\Theta(n^2)$, sendo n o tamanho do vetor. Em relação a espaço, é criada apenas uma variável auxiliar, logo seu custo é $\Theta(1)$.
 - **void Bucket_sort_Graham_scan(Reta* Vetor_Entrada,int Tamanho_Entrada):** O Bucket Sort possui uma complexidade de tempo que depende da forma com que os valores estão distribuídos. Ela é limitada por $\Omega(n)$ e $O(n^2)$, sendo n o tamanho do vetor. Em relação a espaço são criados n baldes se n for menor que 1000, e 1000 caso contrário, logo sua complexidade é $O(1000)$.

4. Estratégias de Robustez

4.1. Mecanismos defensivos e de tolerância a falhas

Tendo em vista as possíveis falhas, foram usados bloco try/catch nas partes sensíveis do código e diferentes estruturas de dados que representavam os diferentes erros. Elas são:

- **Struct Pilha_vazia** = Struct para enviar um erro no try catch quando temos uma pilha vazia e queremos desempilhar. Usado todas as vezes que alguma pilha foi desempilhada sem conferir se esta está vazia.
- **struct Muitos_Argumentos** = Struct para enviar um erro no try catch quando temos um argv com mais argumentos que o necessário. Usado na função Abre_Arquivo.
- **struct Poucos_Argumentos** = Struct para enviar um erro no try catch quando temos um argv sem os argumentos necessários. Usado na função Abre_Arquivo.
- **struct Arquivo_Invalido** = Struct para enviar um erro no try catch quando temos como entrada um nome de arquivo inválido. Usado na função Abre_Arquivo
- **struct Algoritmo_Invalido** = Struct para enviar um erro no try catch quando a função "Montar_Fecho_Convexo" é chamada e o algoritmo escolhido não é "Grahamscan" nem "Jarvismarch"

Além dessas estratégias foram implementados pequenos 'if' dentro de algumas funções para evitar possíveis erros.

5. Análise Experimental

5.1. Análise de Tempo

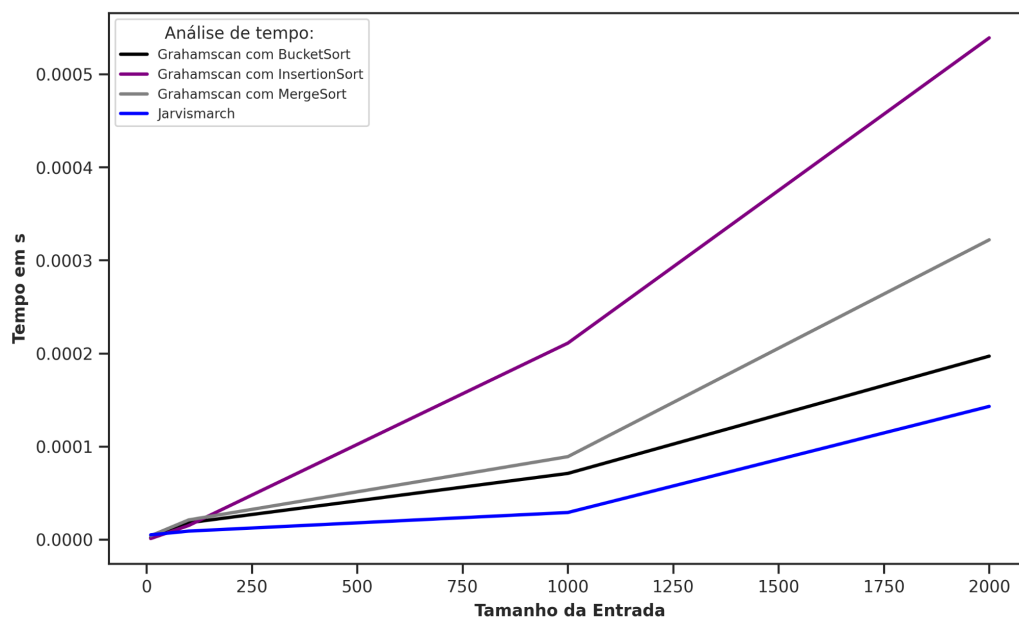


Gráfico do tempo de execução do programa

Para obtermos uma análise mais precisa do tempo gasto pelo algoritmo, utilizamos o tempo de relógio para criar um arquivo CSV que contém informações sobre o tamanho da entrada de teste (fornecidos no Moodle), e uma média do tempo que cada algoritmo gasta para cada tamanho de entrada diferente. Dessa forma, conseguimos uma medição mais acurada do desempenho dos algoritmos em diferentes cenários de entrada. Após a criação do CSV foi implementado um código em Python para a criação do gráfico acima.

Podemos perceber que, no “Graham scan”, temos uma diferença absurda do tempo dependendo do tipo de ordenação. O que mais gasta tempo é obviamente o que usa o Insertion sort e o que menos gasta é o que usa o Bucket sort, estando de acordo com o descrito na análise de complexidade. Em todos os exemplos usados o tamanho do fecho acabou ficando pequeno comparado a saída, logo faz sentido o fato do “Jarvis March” demorar menos tempo que os demais. Logo a análise experimental de tempo está de acordo com o esperado para o código.

5.2. Análise de Memória

```
diogo@diogo-Lenovo:~/Documentos/Faculdade/Códigos/Materias/Periodo3/ESTRUTURA_DE_DADOS/TP2/Codigo/Entrega$ valgrind bin/fecho
==268656== Memcheck, a memory error detector
==268656== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==268656== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==268656== Command: bin/fecho
==268656==
ERRO: Não foram passados argumentos como entrada
==268656==
==268656== HEAP SUMMARY:
==268656==    in use at exit: 0 bytes in 0 blocks
==268656==   total heap usage: 3 allocs, 3 frees, 73,356 bytes allocated
==268656==
==268656== All heap blocks were freed -- no leaks are possible
==268656==
==268656== For lists of detected and suppressed errors, rerun with: -s
==268656== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Como podemos ver, ao analisar o código com o software “valgrind”, esse código não apresenta nenhum vazamento de memória. Todas as alocações dinâmicas feitas utilizaram a função “New” do C++ e foram devidamente deslocadas.

5.3. Análise de Resultado

Para conseguir analisar a acurácia do programa foram realizados inúmeros testes nos algoritmos de achar o fecho e nos de ordenação específicos. Além disso, todas as medidas de programação defensiva foram testadas para uma maior certeza sobre a resistência a falhas do programa, sendo alteradas quando necessário.

Por fim, todos os exemplos postados foram testados e validados. Além disso foram gerados diversos pontos aleatórios para testar o programa com diversas entradas diferentes. Como o código será testado não será incluído capturas de telas contendo esses resultados, tendo em vista que os testes irão demonstrar isso.

6. Conclusões

Em conclusão, no trabalho foi desenvolvido um sistema para achar o fecho convexo das






coordenadas fornecidas como entrada.

Neste trabalho fui introduzido ao conceito de fecho convexo que, com certeza, possui uma utilidade grande para o desenvolvimento de sistemas. Além disso, fui apresentado a dois algoritmos diferentes para calculá-lo, ambos com suas vantagens e desvantagens. Foi bastante interessante trabalhar com ambos e analisar suas semelhanças e diferenças, acredito que todo problema possui diversas formas de ser resolvido e cabe ao programador analisar ganhos e perdas para decidir qual solução será implementada.

Além disso, foi implementada formas de ordenações diferentes, sendo muito interessante analisar a enorme diferença de tempo que elas possuem. Ordenar algo é sempre vital, logo esse trabalho foi muito importante para expandir meus conhecimentos e noções sobre essas formas de ordenação, me deixando mais apto para escolher alguma quando necessário. Outrossim, tal como o último Trabalho Prático, foi importante a implementação e a utilização do TAD “Pilha”, que fez com que eu usasse minha criatividade para não ter que implementar outro TAD, expandindo bastante meus conhecimentos sobre alocação dinâmica e sobre a utilização de pilhas.

Ademais, o relatório desempenhou um papel fundamental na aquisição de conhecimento, pois permitiu uma análise profunda do código, seus resultados e seu custo. Tais noções são vitais na vida de um programador, tendo esse trabalho agregado bastante nisso.

7. Bibliografia

- [CPlusPlus.com](#), Acesso em: 2 de junho de 2023.
-  Convex hulls: Graham scan - Inside code, Acesso em: 2 de junho de 2023.
-  Convex hulls: Jarvis march algorithm (gift-wrapping) - Inside code, Acesso em: 3 de junho de 2023.
-  Bucket Sort | GeeksforGeeks, Acesso em: 8 de junho de 2023.
-  Insertion Sort | GeeksforGeeks, Acesso em: 4 de junho de 2023.
-  Merge Sort | GeeksforGeeks, Acesso em: 4 de junho de 2023.

8. Instruções para compilação e execução

- Para compilar é necessário dar apenas o comando “Make” no terminal.
- Para rodar é necessário escrever bin/fecho <nome_do_arquivo> no terminal. Vale ressaltar que o arquivo tem que estar na pasta raiz do projeto.
- Para limpar é necessário dar o comando “Make Clean” no terminal

```
diogo@diogo-Lenovo:~/Documentos/Faculdade/Códigos/Materias/Periodo3/ESTRUTURA_DE_DADOS/TP2/Codigo/Entrega$ make clean
rm -f bin/fecho obj/main.o obj/Ponto.o obj/Reta.o obj/Leitura_Entrada.o obj/Fecho_Convexo.o obj/Tempo.o *.csv
diogo@diogo-Lenovo:~/Documentos/Faculdade/Códigos/Materias/Periodo3/ESTRUTURA_DE_DADOS/TP2/Codigo/Entrega$ tree
.
├── bin
├── include
│   ├── Fecho_Convexo.hpp
│   ├── Leitura_Entrada.hpp
│   ├── Pilha.hpp
│   ├── Ponto.hpp
│   ├── Reta.hpp
│   └── Tempo.hpp
├── Makefile
├── obj
├── src
│   ├── Fecho_Convexo.cpp
│   ├── Leitura_Entrada.cpp
│   ├── main.cpp
│   ├── Ponto.cpp
│   ├── Reta.cpp
│   └── Tempo.cpp
└── TESTE.txt

4 directories, 14 files
diogo@diogo-Lenovo:~/Documentos/Faculdade/Códigos/Materias/Periodo3/ESTRUTURA_DE_DADOS/TP2/Codigo/Entrega$ make
g++ -Wall -c -Iinclude -o obj/main.o src/main.cpp
g++ -Wall -c -Iinclude -o obj/Ponto.o src/Ponto.cpp
g++ -Wall -c -Iinclude -o obj/Reta.o src/Reta.cpp
g++ -Wall -c -Iinclude -o obj/Leitura_Entrada.o src/Leitura_Entrada.cpp
g++ -Wall -c -Iinclude -o obj/Fecho_Convexo.o src/Fecho_Convexo.cpp
g++ -Wall -c -Iinclude -o obj/Tempo.o src/Tempo.cpp
g++ -o bin/fecho obj/main.o obj/Ponto.o obj/Reta.o obj/Leitura_Entrada.o obj/Fecho_Convexo.o obj/Tempo.o -ln
diogo@diogo-Lenovo:~/Documentos/Faculdade/Códigos/Materias/Periodo3/ESTRUTURA_DE_DADOS/TP2/Codigo/Entrega$ bin/fecho TESTE.txt
FECHO CONVEXO:
```