Trabalho Prático 2 – Fecho Convexo

Gustavo Dias Apolinário - 2022035911

Universidade Federal de Minas Gerais (UFMG) Belo Horizonte – MG – Brasil

gustavo-apolinario@ufmg.br

1. Introdução

O problema proposto foi a implementação de um programa que determina o menor polígono convexo que encapsule todos os pontos presentes em uma lista com um conjunto de pontos no plano cartesiano que representam uma peça de tecido. Para isso, foram implementadas 4 configurações para determinar esse fecho convexo: Graham Scan + MergeSort, Graham Scan + InsertionSort, Graham Scan +BucketSort e Jarvis March.

O programa foi desenvolvido na linguagem C++, a partir do compilador GCC da GNU Compiler Collection. Sistema Operacional: Linux Ubuntu 22.04.01 LTS.

1.1 Estruturas de Dados

O programa utilizou cinco estruturas de dados: Ponto, Reta, Fecho Convexo, Pilha, Tipo Célula.

O Ponto, a Reta e o Fecho Convexo, foram implementados por razões naturais de necessidade de representar o problema do mundo real. Sendo o ponto composto por suas coordenadas e seu ângulo polar (referente ao ponto mais baixo em y das lista), a reta por dois pontos, o inicial e o final, e o fecho composto por uma série de pontos.

A pilha foi escolhida devido sua utilidade no algoritmo de Graham Scan. No algoritmo em questão, constantemente avaliamos a angulação, ou em sentidos práticos, para que lado, esquerda ou direita, que a ligação entre três pontos se manifesta. Caso ocorra uma curvatura a direita, eliminamos o último ponto adicionado, pois ele não faz parte do fecho. Essa repetição de aderir um ponto para teste e depois remover o último adicionado se enquadra perfeitamente com o conceito de pilha em que o primeiro a ser removido é o último a ter entrado. A estrutura Tipo Célula é apenas auxiliar para o funcionamento da Pilha.

1.2 Funções

Principais Funções:

distSq: Retorna o quadrado da distância de dois pontos.

Find_max_Y: retorna o maior valor de y dentre todos os pontos

Find_max_PA: retorna o maior ângulo polar dentre todos os pontos

Reta_gradiente: retorna o gradiente da reta

Polar_angle: recebe duas retas e retorna o ângulo entre elas

```
Insercao(Ponto * v, int n,int id)
```

Método de ordenação por Inserção. Recebe como parâmetros, vetor v, tamanho do vetor n e id para identificar se ordena em função de y ou do ângulo polar. O método de inserção basicamente ordena o vetor dentro dele mesmo. Ele confere se os elementos anteriores a ele em posição são menores se for passa esse elemento superior para frente e coloca o menor na posição antiga do maior, ou seja, troca de lugar, e faz isso para todos os elementos anteriores em index.

```
void BucketSort(Ponto arr[], int n, int max, int id)
```

Método de ordenação por BucketSort. Esse método foi escolhido dentre as possibilidade: CountingSort,BucketSort e Radix Sort, pois é o único que trabalho bem com números fracionários que é o caso dos ângulos. Parâmetro:o vetor com pontos, tamanho, elemento máximo para a criação dos baldes e o id que mantém sua função.

Importante destacar que as funções Find_max são utilizadas **apenas** para servir como parâmetro do max do BucketSort. O BucketSort consiste em criar “baldes” com intervalos definidos, seja o intervalo igual a x, teremos no primeiro balde todos os números do vetor entre (0 e x-1), o segundo balde entre (x e 2x-1) e assim por diante. Depois, de atribuir cada número no seu respectivo balde, ordenamos cada um separadamente e depois juntamos na ordem dos baldes.

```
void merge(Ponto arr[], int l, int m, int r, int id)
```

Função utilizada pelo MergeSort para concatenar vetores, sendo arr o vetor, l ,m e r as partições dos vetores. O primeiro sendo de [l,l+1...,m-1,m] e o segundo [m+1...r], o resultado concatenado é [l...r]. O id para indicar se a ordenação é em função de y ou ângulo. Ela compara os elementos dos dois vetores e vê qual deles é o menor, para ser colocado no vetor resultante, será mais evidente na explicação seguinte.

```
void mergeSort(Ponto arr[], int l, int r, int id)
```

Função principal da ordenação, recebe de qual índice começa a ordenação e em qual termina seguindo a mesma lógica do merge, com o id mantendo sua função.

O mergeSort divide o vetor em partes menores e depois juntas eles de volta. Basicamente, a ideia é dividir o problema em algo menor para facilitar a solução, quando temos vetores com apenas 1 de tamanho juntamos eles. Essa junção mantém a ordem, pois ela confere se o elemento do primeiro vetor da junção é menor que o elemento mais a esquerda do segundo vetor, se for menor é colocado primeiro no vetor resultante se não for, o outro é colocado, sempre que um elemento é selecionado para o resultante, se compara apenas os elementos a direita deste no vetor original, pois se ele foi escolhido significa que todos os anteriores a ele que são menores já foram também.

```
int ccw(Ponto a, Ponto b, Ponto c){
```

Informa se a orientação da reta está no sentido horário, anti-horário ou se os pontos são colineares, importante para saber se houve uma curva à esquerda ou não na ligação dos pontos. Isso é feito a partir de uma fórmula que calcula o produto cruzado entre dois vetores AB e AC.

```
void jarvismarch(Ponto points[], int n)
```

Implementação do algoritmo de Jarvis March de ordenação, recebendo como parâmetro um vetor de pontos e o tamanho deste. Para começar o Jarvis seleciona o ponto com menor y. Depois ele tenta formar uma ligação com os outros pontos e só realiza a ligação com aquele que tiver o menor ângulo no sentido anti-horário,após ligar com um novo ponto ele parte desse ponto e repete o processo sequencialmente até fechar o fecho.

```
void grahamscanI(Ponto * pontos, int n){
void grahamscanM(Ponto * pontos, int n){
void grahamscanB(Ponto * pontos, int n){
```

Três implementações de Graham Scan, a única mudança entre elas é como serão ordenados os pontos tanto em função de y ou do ângulo.

Os Graham´s Scan consistem em primeiro usar o método de ordenação correspondente para ordenar em função de y, deixando o ponto mais baixo na primeira posição do vetor. Depois são criadas as retas saindo do ponto mais baixo para os outros pontos e a partir delas são atribuídos os ângulos. Após isso, se ordena o vetor em função dos ângulos, mas mantém a primeira posição com o mais baixo. Por fim, usamos a pilha e a função para ver se houve uma curva não a esquerda, como explicado anteriormente e da pilha são passados para o fecho convexo.

2. Análise de Complexidade

Jarvis March:

A **Complexidade de Tempo** do Jarvis March é **O(n*m)**, sendo n o tamanho da entrada e m o tamanho da saída. Evidentemente, o pior caso é O(n^2) em que todos os pontos de entrada estão no fecho convexo. A complexidade do Jarvis é O(n*m), pois, como explicado anteriormente, para cada ponto do fecho convexo se testa todos os outros pontos, dessa forma ficamos com n*m.

Complexidade de espaço: Geralmente, seria **O(m)**, pois se armazenaria os pontos que pertencem ao fecho em um vetor auxiliar. Porém, armazenamos direto na classe Fecho Convexo, e por esse passo de armazenar no Fecho é compartilhado por todos os métodos, devido ao problema proposto e a solução adequada proposta, será considerado uma complexidade de espaço de O(1), pois todos terão essa complexidade O(m) pré-definida, o que não faz sentido.

Graham Scan genérico:

Complexidade de Tempo: Seja n o número de pontos de entrada. O algoritmo leva tempo **max(O(ordenação),O(n))**. A primeira etapa (encontrar o ponto mais baixo) leva tempo igual ao do método de ordenação. A segunda etapa (atribuir os ângulos ao pontos) leva tempo O(n).A terceira etapa (ordenar em função dos ângulos) leva tempo igual ao do método de ordenação. A quarta etapa de retirar os elementos que são colineares mantendo apenas o colinear mais longe custa O(n). Na última etapa, cada elemento é empurrado e removido no máximo uma vez. Portanto etapa de processar pontos um por um leva tempo O(n), já que as operações de pilha levam tempo O(1). A complexidade total é O(ordenação) + O(n) + O(ordenação) + O(n) + O(n), ou seja, é max(O(ordenação),O(n)).

Complexidade de espaço: O(n), devido a pilha que é utilizada. Claro que assim como a complexidade de tempo ela pode ser aumentada devido ao método de ordenação, mas a complexidade de espaço de cada método será explicada individualmente.

Insertion Sort:

Por ser um algoritmo adaptável, vamos analisar o melhor caso, pior caso e caso médio:

Melhor caso: O(n) -> O melhor caso seria quando a entrada já está ordenada. Nesse caso o algoritmo irá passar por cada elemento do vetor e não fazer nada.

Pior caso: O(n^2) -> O pior caso seria o vetor está ordenado em ordem decrescente. Nesse caso, o algoritmo terá sempre que fazer o máximo de movimentações em cada casa do vetor, pois o elemento a ser analisado é sempre menor que todos os anteriores.

Caso Médio: O(n^2) -> O caso médio é um vetor embaralhado aleatoriamente, nesse caso é mais provável que para grandes vetores o Insertion Sort se comporte próximo ao pior caso.

Complexidade de espaço: O(1).A ordenação é feita no próprio vetor sem auxílio de espaço, além de poucas variáveis constantes O(1).

Bucket Sort:

O Bucket Sort utiliza so Insertion Sort então ele também é adaptável:

Para ficar mais claro o custo do Bucket seguirei o modelo de explicação do Graham por etapa.

A primeira etapa é descobrir quantos elementos estarão em cada bucket, para evitar desperdício de memória futura. Para isso o custo é O(n) já que passamos por cada elemento do vetor e decidimos qual o bucket dele. A segunda etapa é criar o espaço de memória de cada bucket que é O(num_buckets) que necessariamente é menor ou igual a n. O terceiro passo é zerar a quantidade o vetor responsável por calcular a quantidade em cada bucket que é O(num_buckets). A quarta etapa é atribuir cada elemento ao bucket correto, o que é O(n).

Melhor caso: O(n) -> O melhor caso do Insertion Sort ocorre quando os elementos são distribuídos uniformemente entre os buckerts, pois será mais fácil a ordenação individual de cada um que será realizada pelo Insertion Sort.

Pior caso: O(n^2) -> O pior caso é um bucket ficar com todos os elementos e será igual um Insertion Sort normal, sendo o pior caso do Insertion a ordenação ao contrário, ou seja, o pior caso do bucketsort são todos em um único balde e ordenados de forma decrescente.

Caso Médio: O(n)//O(k^2) -> O esperado do BucketSort é que ele tenha alguns buckets com mais elementos do que outros, mas no geral eles estejam minimamente bem distribuídos, é o mais provável de ocorrer, além disso esse baldes dividem o problema, de maneira que facilita para o Insertion. Portanto, se espera O(n).

Contudo, se o n aumentar muito e o máximo não aumentar proporcionalmente para dividir esse n em mais baldes é provável que os baldes fiquem todos muito cheios e isso acarretará em O(k^2), sendo k o tamanho médio de elementos em cada balde, pois é o esperado do Insertion Sort em vetores muito grandes uma complexidade quadrática.

Complexidade de espaço: O(n+k), sendo n o tamanho da entrada e k o número de buckets. Precisamos reservar um espaço k para armazenar os buckets e um espaço n para armazenar os elementos da entrada.

Merge Sort:

A **Complexidade de Tempo** do Merge Sort é a mesma independente do caso, **O(n log n),** ou seja, mesmo se o vetor já estiver ordenado ele faz o mesmo processo. Para calcular a complexidade do Merge Sort precisamos utilizar o Teorema Mestre. A equação de recorrência da função pode ser expressa assim:

$T(n) = 2T(n/2) + n$. Temos: $a = 2$, $b = 2$, $f(n) = n$ e $n^{\log_b a} = n^{\log_2 2} = n$. Dessa forma, surge o segundo caso do Teorema Mestre porque, $f(n) = \Theta(n^{\log_b a}) = \Theta(n)$, assim, $T(n) = \Theta(n \log n)$.

Complexidade de espaço: O(n), pois todos os elementos da entrada são copiados para vetores auxiliares, logo precisamos de um espaço equivalente a entrada.

3. Estratégias de Robustez

O programa está robusto, porém não há um número excessivo de tratamento de exceções ou condições problemáticas no código. Isso ocorre, pois o algoritmo, apesar de possuir uma lógica complexa e ser eficiente, não é extravagante e não é limitado a casos muito específicos.

Entretanto, existem certas medidas de proteção e robustez necessárias.

Divisão por zero: são feitas poucas divisões no programa, a única que existe a possibilidade do denominador ser zero é na função reta_gradiente, a qual calcula o gradiente da reta, porém essa fórmula utiliza como denominador a seguinte expressão: x do ponto inicial da reta- x do ponto final da reta. Porém, se esse x for o mesmo teremos 0 e a divisão daria problema. Entretanto, utilizamos essa função de reta_gradiente como auxiliar da função que calcula o ângulo de cada ponto e no sentido prático das duas funções juntas, uma reta cujo x inicial e x final são idênticos, implica em um ângulo de 90 graus, já que o ponto inicial das retas é o ponto mais baixo das entradas.

Arquivo não lido: caso o nome do arquivo não seja encontrado o programa informa esse erro e finaliza.

Tamanho maior que o limite: Caso a entrada seja superior à 10000 pontos o programa informa que o tamanho excede o máximo e finaliza.

Confere se **n>=3** para que haja formação de fecho.

Acesso em endereço nulo: a pilha sofreu algumas alterações nos ponteiros para servir melhor nosso problema, entretanto, isso gerou um risco de acesso em endereço nulo, portanto antes de acessar verificamos se o ponteiro é nulo e sempre inicializa os ponteiros e nulo e depois de desalocar também.

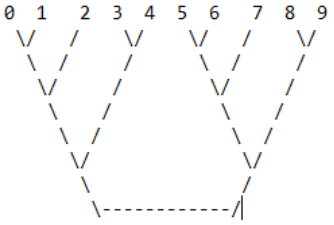
A função desempilha da pilha, verifica se a **pilha está vazia** e retorna um erro se estiver.

4. Análise Experimental

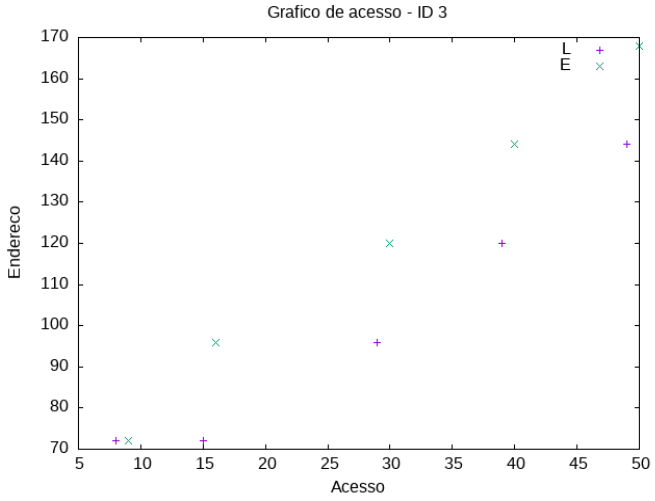
Vamos analisar cada método individualmente, para entender se nossas expectativas de memória e tempo são correspondidas.

Entrada: (27, 11) (69, 8) (6, 82) (19, 82) (33, 96) (45, 13) (38, 82) (6, 22) (68, 79) (66, 68).

Insertion Sort:.



É esperado que o Insertion Sort, sempre que realizar uma leitura, realizar também uma escrita, pois temos dois casos em que um elemento é lido. O primeiro, ele é o **elemento ao ser analisado**, ou seja, se lê o elemento com o intuito de compará-lo, caso ele seja o maior até agora se escreve ele na mesma posição em que ele já estava, se não for se escreve na posição adequada para ele.



O segundo caso, ele é **maior que o elemento que o elemento à direita** que está sendo analisado, pois nesse momento é preciso ler o elemento para ele ser passado para uma posição mais ao final do vetor e por isso mais uma escrita é feita. Logo, nessa lógica, sempre haverá duplas de leituras e escritas, de modo que a quantidade de duplas é 1(a primeira em que ele é analisado) + n (número de elementos menor que ele que estão à direita). Portanto, o gráfico está de acordo e a distância do endereço também, note que a primeira leitura e escrita são próximas, pois na primeira leitura ele está na posição “correta”, então se escreve no mesmo endereço, já nas outras ele está efetivamente sendo trocado. Cada troca responde aos pontos: (45, 13), (6, 22), (68, 79), (66, 68), pois o ID 3, corresponde ao ponto: (19, 82). Esse caso foi uma ordenação em função de y.

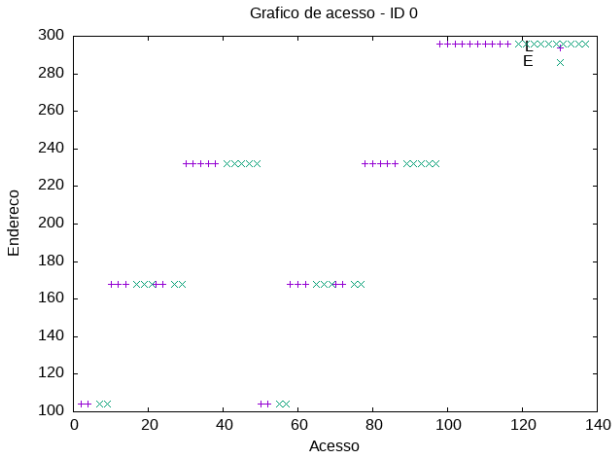
Bucket Sort:

Não foi possível utilizar a biblioteca analisamemlog no Bucket Sort devido a problema de analise na memória em heap, portanto não há gráficos a serem analisados. A análise será melhor detalhada quando for feito os comparativos entre os tempos.

Merge Sort:

Para entender os resultados do MergeSort, é preciso entender o comportamento dele no sentido prático, então para a mesma entrada de tamanho 10 que foi usada no Insertion, se tem o seguinte comportamento:

Para cada fechamento entre dois caminhos significa que o programa ordenou aquele conjunto, ou seja, primeiro se ordena de 0 a 1, depois de 0 a 2, depois de 0 a 4, depois de 5 a 6... até o fechamento que a ordenação total. Assim, fica claro o mecanismo de dividir para conquistar e quais são essas divisões, o que é o mais importante para a análise seguinte fazer sentido e comprovarmos se o desempenho foi o esperado.



Esse é o gráfico de leitura e escrita no vetor inicial. Repare que apesar de usarmos o endereço do vetor, ele altera durante o código, o que causa uma noção errada, porém isso pode ocorrer, devido a recursão e a passagem do vetor como parâmetro para a função merge auxiliar do mergesort. O endereço do vetor pode até alterar de acordo com a profundidade da recursão, mas isso não produz nenhuma mudança nos elementos do vetor, só é preciso estar atento a esse detalhe para não haver confusão.

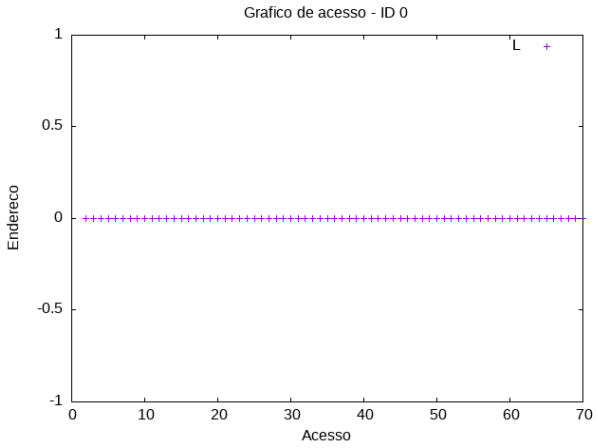
Percebemos então 9 grupos de leituras e escritas, sendo sempre a mesma quantidade de leituras e escritas para cada grupo. Esse comportamento é justamente o processo descrito pela imagem anterior dos caminhos de ordenação. Para cada intervalo ordenado é necessário ler os valores para colocá-los em vetores auxiliares, depois reescrevemos em cima do vetor original de acordo com a ordem correta entre cada vetor, por isso o número de escritas e leituras é sempre o mesmo e as leituras sempre precedem as escritas, já que estamos separando e juntando o mesmo grupo. É possível notar que o tamanho de cada grupo corresponde ao tamanho do caminho!

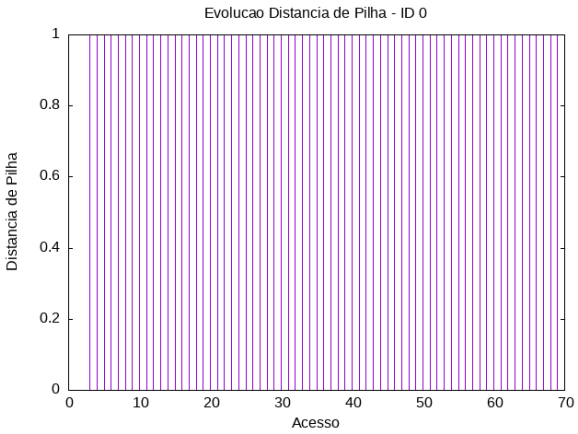
Jarvis:

O comportamento do Jarvis é extremamente simples. Infelizmente, também não foi possível usar o analisamem para a parte de escrita no fecho, pois dava o mesmo erro de não foi possível alocar na pilha igual ao Bucket Sort, porém foi possível analisar os outros aspectos.

O curioso é a simplicidade do Jarvis sendo retratada. O Jarvis basicamente lê todos os pontos do vetor em busca do menor y. Escreve na variável L o ponto que possuir o menor y, o número de escritas nessa variável depende da entrada, porém no caso teste que da entrada de tamanho 10, só se escreve duas vezes, sendo um gráfico pouco informativo.

Depois de encontrar o ponto de menor y, ele testa a orientação dos pontos e seleciona aquele com menor ângulo no sentido anti-horário, esse ponto com menor ângulo é escrito pelo fecho (o qual a analisamem não consegue criar o gráfico). O importante é notar como o Jarvis soluciona o problema em tão poucos passos e **somente** com leitura, enquanto a escrita fica por conta do fecho e por isso seu gráfico tem o formato:





Como todos os acessos são sequenciais a distância de pilha se mantém em 1 por todo o processo.

Tempo de execução:

Tamanho entradas: 10//100//1000//2500//5000

Graham + Merge: 50.400 // 42.500 // 425.000//1179.600// 2260.400 microsegundos

Graham + Insertion:8.100//37.500//1315.200//11453.700//30916.200 microsegundos

Graham + Bucket (Insertion): 10.500 // 46.900 // 347.700//1082.000//2700.500 microsegundos

Graham + Bucket(MergeSort): 11.000 // 38.300 // 359.300//1082.000//1792.400 microsegundos

Jarvis: 1.200 // 11.700 // 112.700//306.500//623.300 microsegundos

O tempo de execução nos explica muitos aspectos e confirma tudo que foi visto até o momento. Para fins de análise, alterei o código para o Graham Bucket com implementação do MergeSort para ordenar cada balde individualmente, **a entrega será feita em função do Insertion**. Como não foi possível analisar o Bucket individualmente, devido à questão do heap, ele terá uma ênfase maior nesse tópico.

Bucket Sort:

É importante lembrar da explicação do Caso Médio do Bucket Sort e sua relação com o Insertion Sort. É nítido que para entradas maiores (especialmente a entrada de 5000), o Bucket(Insertion) é muito pior que o Bucket(MergeSort), isso ocorre por conta do caso O(k^2), repare que é diferente de O(n^2) do Insertion Sort e não tem uma curvatura exponencial tão acentuada, devido uma limitação natural da relação do máximo da função e o k que não existe em n, mas a diferença do BucketM e BucketL é considerável.

Insertion Sort:

Mostrou com clareza que possui o pior desempenho e cresce muito rápido em função da entrada, o que está complemente de acordo com o O(n^2) previsto anteriormente.

Graham + Merge:

É possível notar que o crescimento do MergeSort é quase linear para essas entradas. Obviamente o log n continua presente e isso é observável na comparação com o Bucket(MergeSort), o qual ainda está dependente do log n devido sua relação com o Merge, mas por usar entradas menores acaba sendo levemente mais eficiente. Se provando ser um algoritmo muito bom em termos de tempo de execução.

Jarvis:

Teve o melhor desempenho sem dúvida alguma, com certeza, devido à sua simplicidade e o número menor de passos e não necessitar da ordenação por um método igual aos outros. Além disso, o pior caso do Jarvis é extremamente raro para o nosso problema de pontos, pois dificilmente todos os pontos de uma entrada estarão no fecho convexo.

5. Conclusões

É possível concluir que o programa foi capaz de solucionar o problema proposto por completo, com as 4 configurações implementadas. Além disso, foi possível entender como cada método funciona, a complexidade de tempo e espaço deles e observar como a memória é acessada. A partir desse estudo foi possível observar o tempo de execução de cada configuração e comprovar as afirmações anteriores, de maneira que ficou claro qual método é o melhor para cada contexto.

6. Bibliografia

- Geeksforgeeks.or

- Stable Sort (Youtube Channel)

- https://www.cs.auckland.ac.nz/software/AlgAnim/convex_hull.html

-Wikipedia (SAM)

7. Instruções Para Compilação e Execução

A biblioteca gráfica SFML é obrigatória para o funcionamento do programa!

- Abrir o terminal
- Digitar o comando para atualizar pacotes: sudo apt-get update
- Digitar o comando: sudo apt-get install libsfml-dev
- Acessar a pasta TP
  Comando: cd / <caminho para a pasta /TP >
- Compilar com o Makefile Comando: make
- Executar com o Makefile Comando: make run
- Finalizado, caso deseja limpar: Comando: make clean

- Importante ressaltar que os arquivos de texto a serem lidos devem estar na raiz do projeto.