Trabalho Prático 2

Resolvedor de Fecho Convexo

Luiz Fernando Gonçalves Rocha - 2022043582

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

luizfgr@ufmg.br

Introdução

O problema consiste em, dado um conjunto de pontos, encontrar o seu fecho convexo. O fecho convexo de um conjunto de pontos é dado pelo subconjunto de pontos que define um polígono que cobre todos os pontos no conjunto dado.

O problema pode ser resolvido de diversas formas. Neste trabalho, conforme a especificação, serão implementados e analisados dois algoritmos: a Varredura de Graham e o Marchar de Jarvis. Sendo o primeiro algoritmo dependente de uma ordenação, ele foi implementado e testado com três tipos diferentes de algoritmos de ordenação: insertion sort, merge sort e bucket sort.

Método

Classes

Foram implementadas as classes como foi exigido na documentação: ponto, reta e fecho convexo.

Tanto o ponto quanto a reta são formados por dois inteiros. A primeira estrutura tem as coordenadas x e y do ponto e a segunda tem os coeficientes a e b, de forma que a reta é expressa na forma ax + b = y, sendo as possíveis soluções (x, y) os pontos pertencentes à reta.

O fecho convexo, na prática, é apenas um vetor. Quando o fecho é construído usando um vetor de pontos, o fecho desse vetor é calculado e atribuído ao vetor privado da classe. Por conveniência, foi adicionada à classe um método “imprime()”, que imprime na tela os pontos que constituem o fecho.

Também foram implementadas as classes “vetor” e “par”, porém, já que a sua implementação não é o foco deste trabalho, vou omitir os detalhes sobre essas classes;

O restante do trabalho foi implementado com uma série de funções.

Funções

Todos os algoritmos de fato foram implementados em funções externas às classes.

Primeiro, temos a Varredura de Graham. Ela foi implementada seguindo exatamente os passos do pseudocódigo dados no livro *Algoritmos: Teoria e Prática*[1], do Cormen, sendo a única adição de fato a remoção de pontos colineares, para que a saída fique similar aos exemplos fornecidos.

Em resumo, o que o algoritmo faz é:

1. É dado um conjunto de pontos Q. Será calculado o fecho desses pontos.
2. É removido o ponto p_0 mais baixo do conjunto (menor coordenada y).
3. O restante dos pontos em Q é ordenado em termos de seu ângulo polar em relação a p_0, sendo que em empates de ângulo é mantido somente o ponto mais distante de p_0. Seja V o vetor de pontos em ordem crescente de ângulo.
4. É criada uma pilha S e são empilhados nela p_0 e os dois pontos com menor ângulo.
5. Para cada ponto q restante em V, em ordem crescente de ângulo:
   5.1. Enquanto o ângulo formado pelos dois pontos no topo da pilha e por q define uma rotação no sentido horário, desempilhamos o topo da pilha, já que ele não estará no fecho. (Estamos montando o fecho procurando os pontos externos no sentido anti-horário, logo, se a rotação feita é no sentido horário, há uma concavidade e, portanto, o ponto no topo da pilha não está no fecho)
   5.2. Colocamos q no topo da pilha.
6. S é a pilha de pontos que determina o fecho.

A implementação feita utiliza vetores em vez de pilhas e conjuntos, uma vez que essa abordagem se mostrou mais conveniente, especialmente ao operar sobre o fecho para remover pontos colineares.

A função também recebe, opcionalmente, um inteiro que define o método de ordenação a ser utilizado. Por padrão, é utilizado o bucket sort, que apresentou o melhor desempenho para entradas grandes.

Em seguida, temos o Marchar de Jarvis. Essa função, similarmente à anterior, recebe um vetor de pontos Q e retorna outro vetor S, formado pelos pontos que constituem o fecho.

Basicamente, a função faz o seguinte:

1. Seja Q o conjunto de pontos cujo fecho convexo será calculado e S o fecho convexo.
2. Encontramos o ponto p_x em Q com a menor coordenada x e o inserimos em S, já que esse ponto trivialmente estará no fecho.
3. Seja p o último ponto a ser adicionado no fecho.
4. Enquanto p é diferente de p_x:
   4.1. Escolhemos um ponto qualquer q do conjunto Q.
   4.2. Para cada outro ponto r em Q, verificamos o ângulo pqr. Se o giro é no sentido anti-horário, fazemos q := r. (Ao final, teremos o ponto “mais anti-horário” de Q partindo de p)
   4.3. Inserimos q em S.

    4.4. p := q.
5. S é o fecho de Q.

Na implementação feita para esse trabalho, foram utilizados vetores em vez de conjuntos e, na instrução 4.1, foi simplesmente tomado o elemento na posição 0 do vetor.

Análise de Complexidade

De tempo

A complexidade da Varredura de Graham está dominada pelo método de ordenação utilizado, uma vez que temos n - 3 iterações do passo 5 e o passo 5.1, naturalmente, pode apenas fazer n - 3 pops no vetor, já que o fecho precisa de pelo menos três pontos e cada ponto é inserido exatamente uma vez, o que resulta em 2(n - 3) operações. Sendo essa parte O(n), a ordenação domina a complexidade, já que ela será O(n) ou pior.

Foram utilizados três métodos de ordenação:

Com o insertion, a ordenação fica $O(n^2)$ no pior caso e no caso médio e O(n) no melhor caso. Sendo assim, o algoritmo como um todo fica $O(n^2)$.

Com o merge, a ordenação sempre fica $\Theta(n \log n)$, sendo essa também a complexidade do algoritmo resultante.

Com o bucket, a ordenação fica O(n) no melhor caso, assumindo uma distribuição uniforme dos ângulos dos pontos, portanto, o algoritmo como um todo fica O(n). No pior caso (todos os elementos menos um em um bucket, caímos de volta no caso de insertion, já que a ordenação em cada bucket foi feita com esse algoritmo).

A complexidade do Marchar de Jarvis, por sua vez, depende da quantidade de pontos presente no fecho. Note que o loop iniciado no passo 4 é executado h vezes, sendo h a quantidade de pontos no fecho convexo de Q. Dentro desse loop, temos outro que executa n - 1 vezes, sendo n a quantidade de pontos em Q. Sendo assim, o programa é O(nh).

No pior caso, n = h e temos $O(n^2)$.
No melhor caso, h = 3 e temos O(n).

De espaço

Na varredura de Graham, as operações foram realizadas em uma cópia do vetor de pontos. Essa decisão foi tomada porque, já que a main calcula o fecho quatro vezes, é interessante manter o vetor de pontos naquele escopo intacto, de modo que a construção do fecho com um método não afete a construção do fecho com outro. Sendo assim, temos, por enquanto, que a Varredura é O(n).

Em seguida, é criado outro vetor, dessa vez de pares ponto-ângulo, que será ordenado. A complexidade permanece O(n).

É feita, então, a ordenação desse vetor.

No caso do insertion sort, a ordenação é feita inplace, de modo que a complexidade de espaço dessa parte é O(1).

O mergesort, por sua vez, foi implementado com o uso de memória auxiliar. No ponto mais alto da recursão, o merge cria um vetor de tamanho n para misturar os dois lados do vetor que está sendo ordenado. Dessa forma, o mergesort é O(n).

Finalmente, temos o bucket sort. Ele cria n / 10 buckets, que, no total, tem n elementos. Logo, a complexidade é O(n).

Por fim, feita a ordenação, é criado o vetor S, que é preenchido com no mínimo três e no máximo n pontos.

Sendo assim, a Varredura de Graham sempre tem complexidade de espaço O(n).

O Marchar de Jarvis também trabalha com uma cópia do vetor de pontos, logo, também começamos com O(n).

No seu interior, a única estrutura criada dependente do tamanho do vetor é o fecho, que tem entre três e n pontos, ou seja, ainda temos uma complexidade O(n).

Sendo assim, a complexidade de espaço do Marchar de Jarvis sempre é O(n).

Estratégias de Robustez

O código foi verificado e não existe a possibilidade de divisão por 0. A única função que poderia oferecer esse risco é a de cálculo de ângulo, mas foi colocada uma condição que contorna a divisão caso o divisor seja 0.

O programa lança exceções nos seguintes casos:

1. A escolha de método de ordenação é inválida na Varredura de Graham.
2. Foi solicitado o ângulo de dois pontos iguais.
3. Não foi possível abrir o arquivo de entrada.
4. Não foi possível ler os argumentos na linha de comando.
5. A quantidade de pontos solicitada ou presente no arquivo é menor que 3.
6. Uma linha da entrada possui um caractere que não é espaço nem número.

Além dessas exceções, o vetor, que é amplamente usado neste trabalho, faz checagem de limites e lança uma exceção se ocorrer algum problema na alocação de memória.

Sendo assim, todos os problemas mais frequentes estão cobertos.

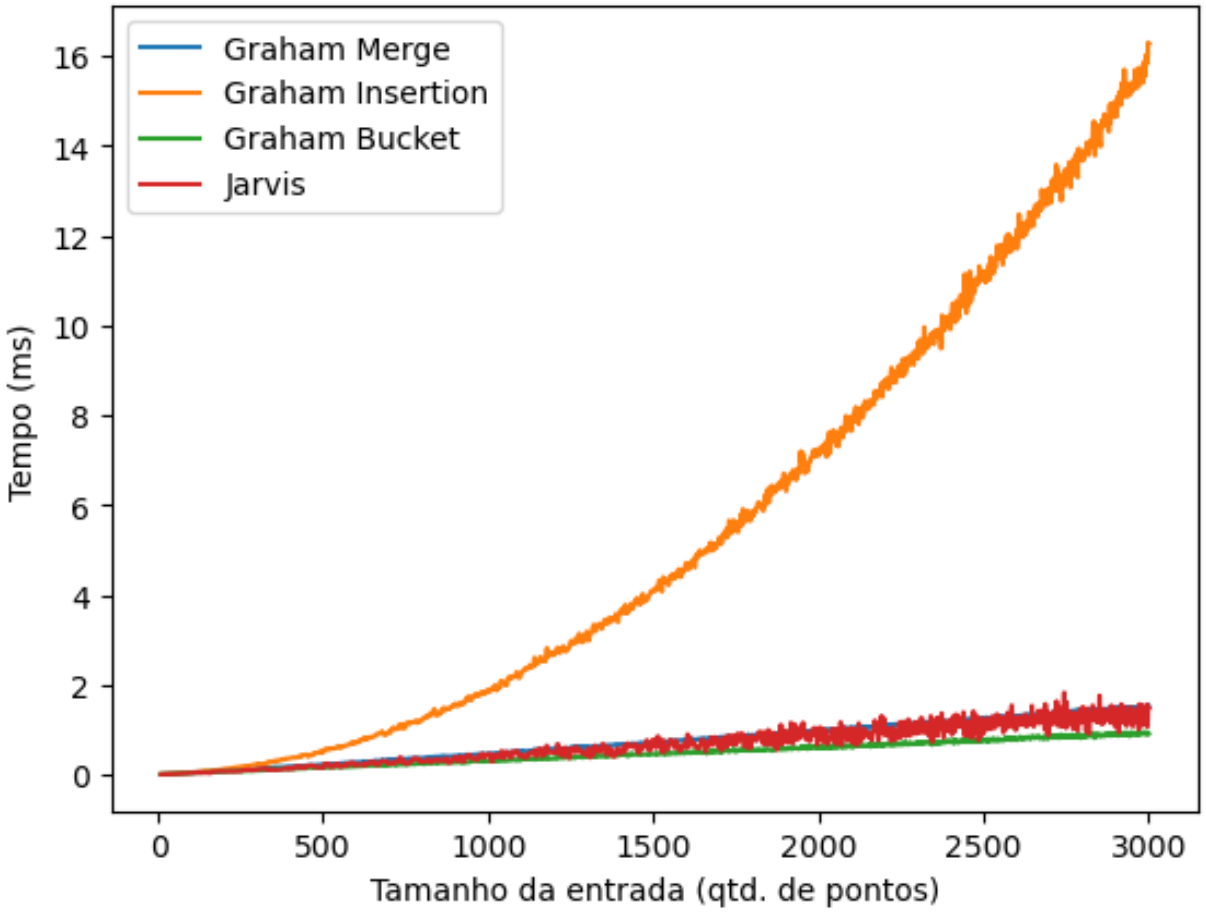
Análise Experimental

Para a análise experimental, foram realizados um total de 99835 testes, sendo que cada teste calculou o fecho convexo de um conjunto aleatório de pontos com um tamanho entre 10 e 3000.

Para o tamanho do conjunto de pontos, foram tomados 1990 inteiros entre 10 e 3000 inclusive os extremos, espaçados de forma aproximadamente uniforme.

Para cada tamanho, foram realizados 5 * $\log_2$(tamanho) testes, para que posteriormente fosse tomada a média e o ruído fosse reduzido.

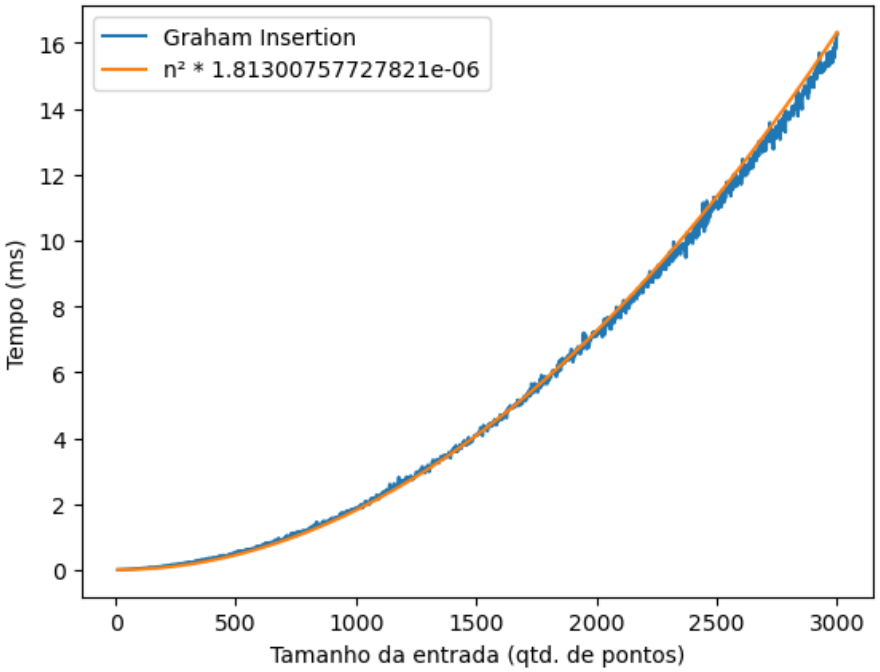
Temos o seguinte gráfico para o desempenho:



Claramente, o pior desempenho é o da Varredura de Graham usando o insertion sort, e o melhor desempenho é a Varredura de Graham usando o bucket sort.

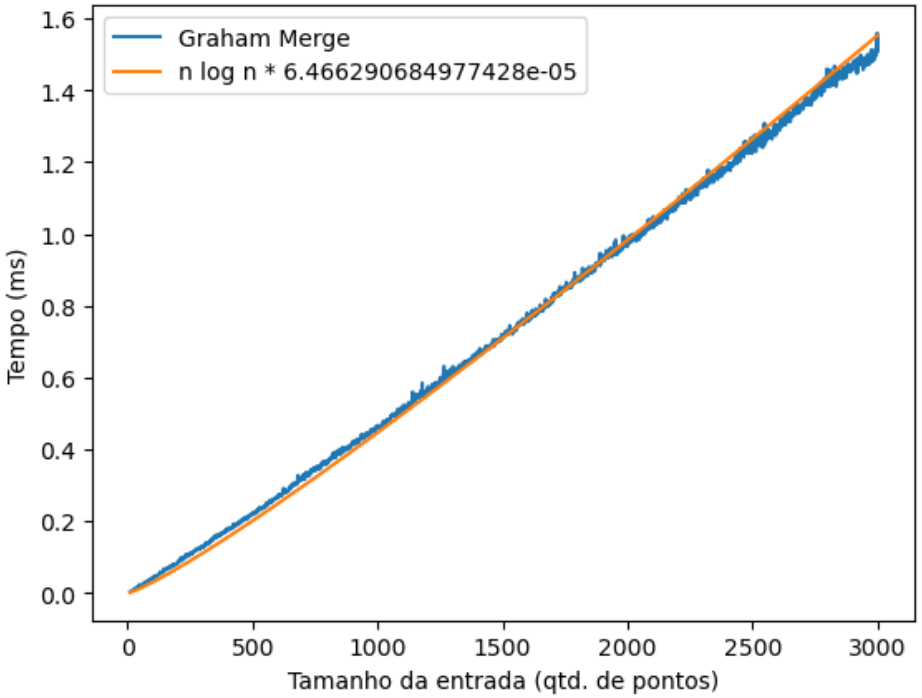
O desempenho parece refletir de forma bastante precisa o que foi encontrado na análise de complexidade. Vamos comparar o desempenho experimental com as curvas teóricas.

Primeiro, vejamos como o desempenho do Graham com insertion sort fica ao lado do gráfico de n^2, que é o custo médio teórico, ajustado com uma constante:



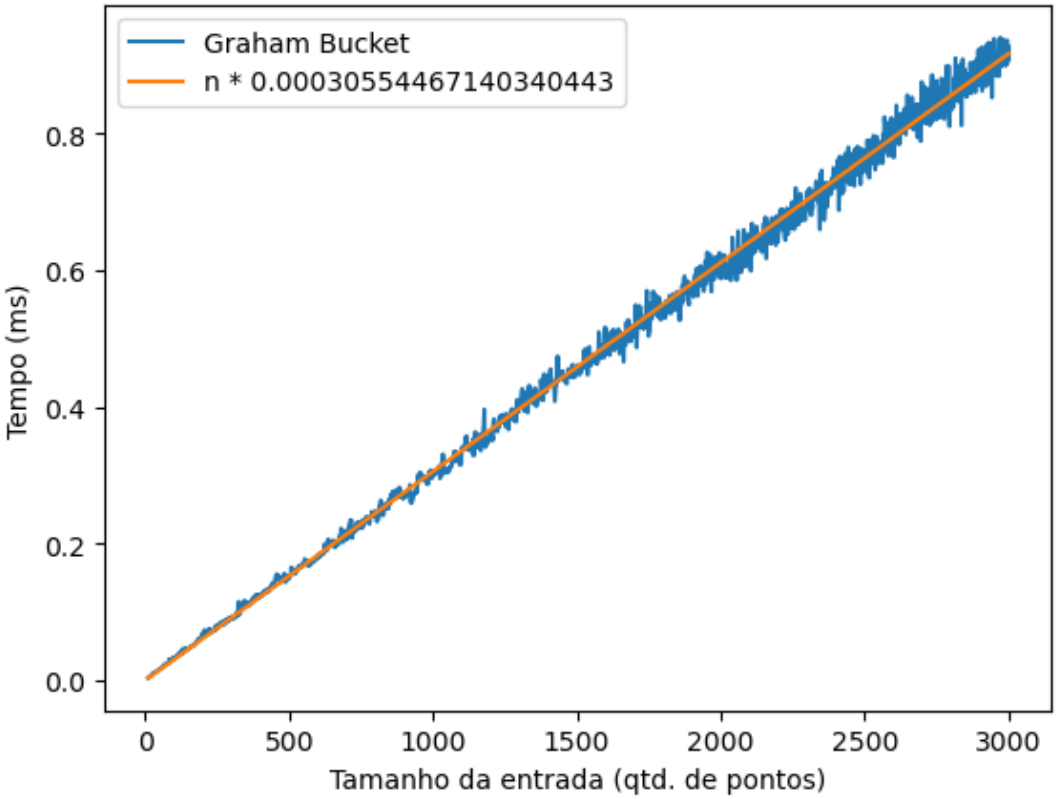
Os gráficos ficam muito próximos, como esperado.

Façamos o mesmo para o Graham com mergesort ao lado do gráfico de n log n:



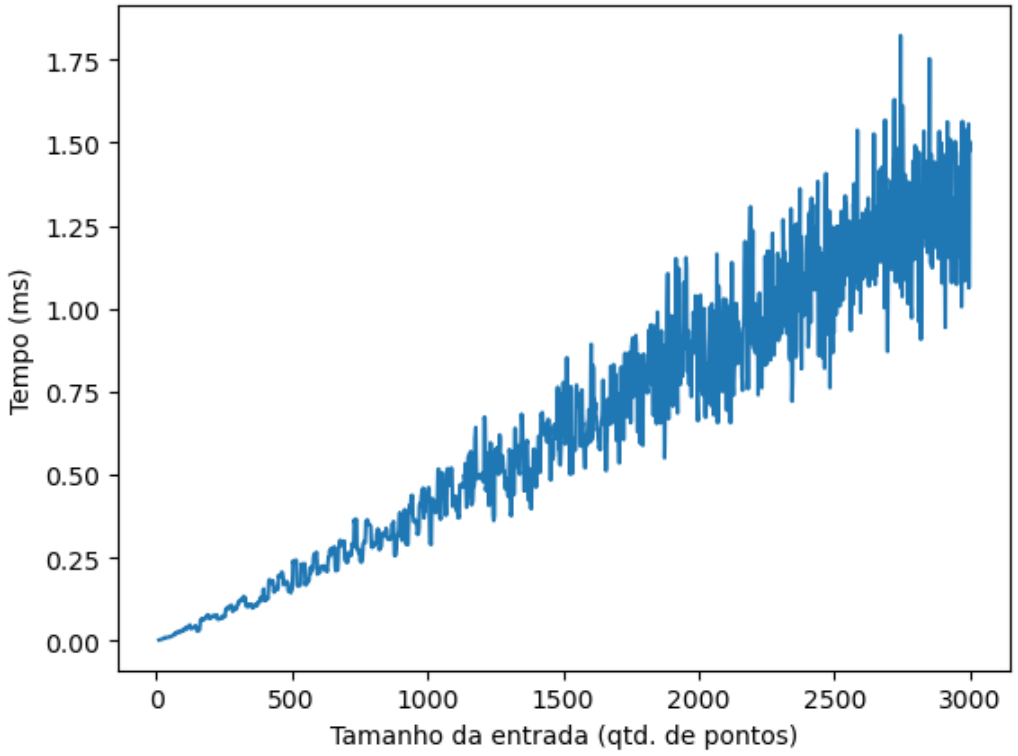
Novamente, o resultado condiz de forma satisfatória com o esperado.

Vejamos o Graham com bucket sort ao lado de n:



Nesse caso, o tempo parece flutuar mais aos arredores do custo teórico. Ainda assim, o desempenho é satisfatório e dentro do esperado, uma vez que o bucket sort conta com a uniformidade da distribuição dos valores no vetor para que sua complexidade seja O(n).

Por fim, vejamos como se comporta o Jarvis:



O Jarvis apresenta um comportamento bem mais irregular. Isso certamente ocorre devido ao fato de seu custo ser muito dependente da quantidade de pontos no fecho (o algoritmo é O(nh), sendo h a quantidade de pontos no fecho resultante).

Conclusões

O trabalho consistiu na implementação robusta e na análise teórica e experimental de diferentes algoritmos para o cálculo do fecho convexo de um conjunto de pontos.

Dentre as implementações realizadas, a mais eficiente é a que utiliza o bucket sort. Apesar de o bucket ser dependente da uniformidade da entrada, a natureza dos testes faz com que o seu desempenho seja maximizado.

O Jarvis, apesar de ser mais eficiente que o bucket algumas vezes, é prejudicado pela irregularidade inerente ao método: a complexidade depende não só do tamanho da entrada, mas também da natureza da saída. Sendo assim, em geral, o Jarvis tende a ficar um pouco atrás do bucket nessas implementações e nesse conjunto de testes.

Bibliografia

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Algoritmos: Teoria e Prática.* 3a edição. Elsevier, 2012. ISBN 9788535236996

[2] GeeksforGeeks. [s.d.]. Convex Hull using Jarvis' Algorithm or Wrapping - GeeksforGeeks. [online] Disponível em: <https://www.geeksforgeeks.org/convex-hull-using-jarvis-algorithm-or-wrapping/> [Acesso em 26 de abril de 2023].

[3] GeeksforGeeks. [s.d.].Convex Hull using Graham Scan - GeeksforGeeks. [online] Disponível em: <https://www.geeksforgeeks.org/convex-hull-using-graham-scan/> [Acesso em 28 de abril de 2023].

Instruções para compilação e execução

Para compilar e executar o código, basta digitar “make fecho entrada=<entrada>”, substituindo <entrada> pelo nome do arquivo de entrada a ser utilizado. Caso não seja especificada uma entrada, i.e, seja digitado apenas “make fecho”, o programa procurará o arquivo “entrada.txt” na pasta TP. Caso haja algum problema ao ler o arquivo, o programa lançará uma exceção.

O executável bin/main possui algumas opções na linha de comando, que podem ser vistas com o comando ./bin/main -h.

Também foram implementados testes para os algoritmos de ordenação. Para compilá-los e executá-los basta digitar “make teste” na linha de comando.