



Universidade Federal  
de São João del-Rei

# TRABALHO PRÁTICO 1

Lucas Gonçalves Nojiri  
Arthur Antunes Santos Silva

Este trabalho prático tem por objetivo exercitar conceitos e práticas dos algoritmos sobre hipercampos. Baseado no código em C para a disciplina de AEDS3 do curso de Ciência da Computação da Universidade Federal de São João del Rei.

São João del Rei  
Abril de 2023

# Sumário

<b>1</b>	<b>Introdução</b>	<b>2</b>
1.1	Requisitos . . . . .	2
<b>2</b>	<b>Mapeamento do código</b>	<b>3</b>
2.0.1	main . . . . .	3
2.0.2	header . . . . .	3
2.0.3	arquivos . . . . .	3
2.0.4	pontos . . . . .	4
<b>3</b>	<b>Complexidade</b>	<b>5</b>
3.1	Complexidade das funções . . . . .	5
3.1.1	main.c . . . . .	5
3.1.2	arquivos.c . . . . .	5
3.1.3	pontos.c . . . . .	6
<b>4</b>	<b>Testes</b>	<b>7</b>
4.1	Performance e Resultados . . . . .	7
4.1.1	Testes com 2 entradas. . . . .	8
4.1.2	Testes com 4 entradas. . . . .	8
4.1.3	Testes com 10 entradas. . . . .	9
4.1.4	Testes com 100 entradas. . . . .	9
4.1.5	Testes com 1000 entradas. . . . .	10
4.1.6	Testes com 50000 entradas. . . . .	10
4.1.7	Testes com 100000 entradas. . . . .	11
<b>5</b>	<b>Conclusão</b>	<b>12</b>
<b>6</b>	<b>Referências</b>	<b>13</b>

---

# 1 Introdução

Neste trabalho vamos implementar um sistema que possui âncoras e pontos para criarmos um Hipercampo e localizar os pontos dos quais queremos realizar os testes, dadas as duas âncoras e dois pontos  $A = (X_A, 0)$  e  $B = (X_B, 0)$  para a formação de um segmento horizontal, tal que  $0 < X_A < X_B$ , de modo que se forme um conjunto  $P$  de  $N$  pontos na forma  $(X, Y)$ , tal que  $X > 0$  e  $Y > 0$ .

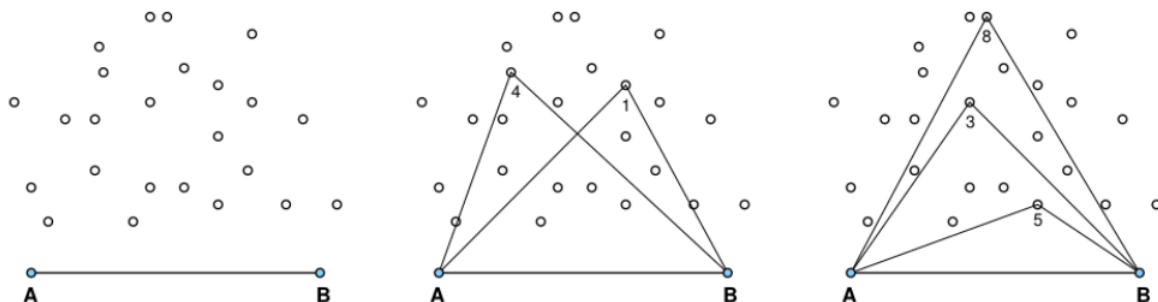


Figura 1: Hipercampo

## 1.1 Requisitos

As implementações devem ser feitas na linguagem C (C++ pode ser usado para o tratamento de strings), usando a biblioteca padrão da linguagem. Para realizar a ligação de um ponto  $v$  que pertence a  $P$ , é preciso desenhar os dois segmentos de reta  $(v, A)$  e  $(v, B)$ . Para que desta forma seja possível ligar vários pontos, mas de modo que os segmentos se interceptem apenas nas âncoras.

---

## 2 Mapeamento do código

### 2.0.1 main

Esta secção é responsável pelo funcionamento do programa sendo a função principal do sistema, dentro dela estão as funções `File(argc,argv, input,output)`, `Leitura_input(input)`, `Solucao( )` e `plot( )`. A função `File` está sendo responsável por testar se os nomes dos arquivos foram indicados, caso forem, será possível abri-los. A função `Leitura` passa as coordenadas dos pontos para a lista `pontos`. A função `Solucao` encontra o maior numero de pontos dentro das linhas.

### 2.0.2 header

Esta secção é responsável por armazenar todas as funções usadas no código do trabalho, possui um struct `Ponto`, o número de pontos e as âncoras usadas no programa. Dentro estão inseridas estas funções, `float Coeficiente_Reta( )`, `void selectionSort( )`, `void Solucao( )`, `void Busca_Maior( )`, `int Interceptacao( )`, `int File( )`, `void Leitura_input( )`, `float Resolve_coef( )` e `int plot( )`, e também todas as bibliotecas necessárias para o funcionamento do código.

### 2.0.3 arquivos

Esta secção é responsável por gerenciar os arquivos que serão utilizados para o funcionamento do código.

#### 1. `int File(int argc, char **argv, char *input, char *output)`

Arquivos que possuem seus parâmetros e suas entradas, como exigido nas especificações do trabalho prático, é necessário que o programa receba dois parâmetros pela linha de comando, utilizando a primitiva `getopt`, utilizando a função `while` as entradas serão lidas até chegar ao `-o`, caso o arquivo de entrada `-i` vamos copiar o nome do arquivo para a entrada para `"input"`, caso o arquivo de entrada `-o` vamos copiar o nome do arquivo para a entrada para `"output"` e caso não for informado se algumas das entradas `-i` ou `-o`, aparecerá uma mensagem de erro e sairá do programa.

#### 2. `void Leitura_input(char *input)`

Faz a leitura das entradas e verifica se os valores são aceitos. São feitas as seguintes verificações: se o número de pontos excedeu o limite de pontos, se `A` é maior que `B`, se o valor de `B` excede o limite de pontos e se `A` é menor que 0. Após isso são lidos os valores do arquivo e são passados as coordenadas do ponto.

---

### 2.0.4 pontos

Esta secção é responsável por ordenar os pontos mapeados no hipercampo.

1. **float Coeficiente\_Reta(float x2, float y2, float x1, float y1)**

Essa função que realiza operações para encontrar o coeficiente da reta entre 2 pontos com o uso da fórmula  $m = y2 - y1 / x2 - x1$ . Por uso desta fórmula é possível representar a reta de forma algébrica, sendo possível obter informações importantes sobre o comportamento da reta representada no plano cartesiano.

2. **float Resolve\_coef(Ponto points[4])**

Função criada como auxiliar, para resolver todos os casos do coeficiente de reta, sendo eles:  $x_a < x_b$ ,  $x_a > x_b$ , e quando o coeficiente poderá ser  $\text{coef} > \text{coeficiente}$  de a ou  $\text{coef} < \text{coeficiente}$  de b.

3. **void selectionSort(Ponto \*arr, int n)**

Foi usado o método de "selectionsort" para a ordenação dos pontos, o selection sort é um algoritmo de ordenação que funciona selecionando o menor elemento na lista e movendo-o para o início da lista não ordenada, este processo é repetido até que todos os elementos estejam em ordem crescente.

4. **void Solucao()**

Com os pontos ordenados de forma crescente, a função "Solucao" chama a função "Busca\_Maior" dentro de si fazendo a procura do ponto mais alto ou maior ponto do hipercampo, ao criar o inteiro "Maior" é possível atribuir o valor global armazenando o número máximo de pontos que estão ligados com a interseção das âncoras.

5. **void Busca\_Maior(int \*pt\_alto)**

Essa função faz a busca do maior ponto ou ponto "mais alto" do Hipercampo. Ao criar o inteiro "Maior" é possível atribuir o valor global armazenando o número máximo de pontos que estão ligados com a interseção das âncoras. É feito o coeficiente de reta de A e B, em seguida é chamada a função "Interceptacao" caso o maior ponto "i" for maior que o "n". E assim a função termina percorrendo todos os pontos abaixo do maior ponto de n.

6. **int Interceptacao(float ma, float mb, float xa, float ya, float xb, float yb())**

Verifica as colisões e se as retas se interceptam no hipercampo. Nessa função são analisados os coeficientes de  $x_a$  e  $x_b$  para a verificação das interseções entre o maior de i e o maior de n.

7. **int plot()**

Essa função é responsável por gerar os gráficos de funções matemáticas e outros conjuntos de dados.

---

## 3 Complexidade

### 3.1 Complexidade das funções

Esta secção é responsável por fornecer a ordem de complexidade sobre as funções que atuam dentro do código.

#### 3.1.1 main.c

---

```
1      int main(int argc, char **argv){
2          File(argc,argv, input,output);
3          Leitura_input(input);
4          Solucao();
```

---

A complexidade das funções que estão dentro da main( ) são explicadas separadamente.

---

```
1      getrusage(RUSAGE_SELF,&buff);
```

---

A função getrusage() por ser utilizada para obter informações sobre o tempo de CPU e memória utilizada, apresentando uma complexidade  $O(1)$ , pois ela serve apenas para medir o tempo de execução utilizado no programa, sendo em todos os casos constante.

#### 3.1.2 arquivos.c

##### 1. int File(int argc, char \*\*argv, char \*input, char \*output)

O código possui complexidade  $O(n)$ ,  $n$  sendo o número de argumentos, existe apenas um loop while que percorre a linha de comando apenas uma vez, assim podemos dizer que sua complexidade é  $O(1)$ , caso não for inserido nenhum argumento a complexidade também será  $O(1)$ , desta forma o melhor caso teria a complexidade de  $O(2)$ .

##### 2. void Leitura\_input(char \*input)

O código possui complexidade  $O(n)$ ,  $n$  sendo o número de pontos lidos no arquivo de entrada, pois o código lê cada ponto do arquivo em um loop while e armazena os pontos em um array alocado dinamicamente, o restante são entradas mais simples com complexidade  $O(1)$ .

---

**3.1.3 pontos.c****1. float Coeficiente\_Reta(float x2, float y2, float x1, float y1)**

O código possui complexidade  $O(1)$ , sendo constante e também não depende do tamanho das entradas, apenas realiza algumas operações básicas seguindo a fórmula e retorna o resultado.

**2. void selectionSort(Ponto \*arr, int n)**

O código possui complexidade  $O(n^2)$ , este código implementa o algoritmo Selection Sort ordenando de forma crescente todos os elementos, no pior dos casos sendo  $n$  o número de elementos do array, o loop externo percorre  $n - 1$  vezes.

$$(n-1) + (n-2) + \dots + 1 = n(n-1)/2$$

**3. void Solucao()**

O código possui complexidade  $O(n^2)$ , sendo dependente também da função Busca\_maior e SelectionSort, onde a função SelectionSort tem a complexidade de  $O(n^2)$ . Sendo dominada pela função Busca\_maior possuindo 2 loops aninhados, o loop externo percorre todos os pontos da lista, e o interno percorre os pontos anteriores da lista.

**4. void Busca\_Maior(int \*pt\_alto)**

O código possui complexidade  $O(n^2)$ , sendo  $n$  o número de pontos da lista, neste código possui 2 loops aninhados, o loop externo percorre todos os pontos da lista, e o interno percorre os pontos anteriores da lista. O uso de outras funções auxiliares como a função interceptação possui complexidade  $O(1)$ , contribuindo para a complexidade do código.

**5. int Interceptacao(float ma, float mb, float xa, float ya, float xb, float yb)**

O código possui complexidade  $O(1)$ , o código sempre executa o mesmo número de operações não dependendo do tamanho da entrada, usando apenas operações matemáticas simples como comparações e o cálculo do coeficiente da reta com complexidade de  $O(1)$ .

**6. int plot()**

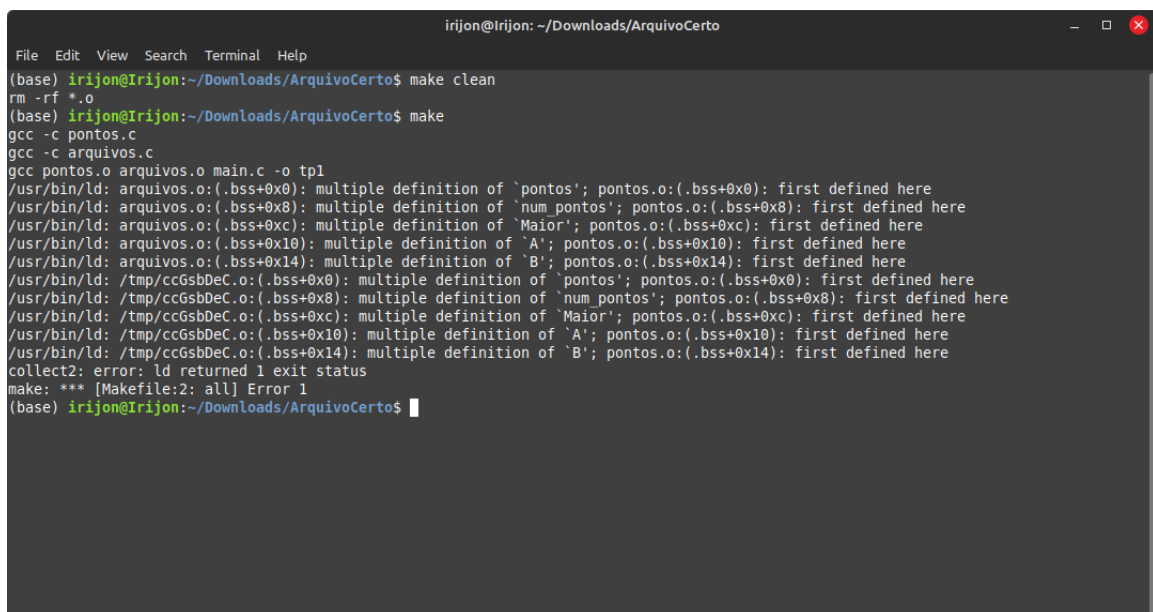
O código possui complexidade  $O(1)$ , pois não há nenhum loop ou operação que dependa do tamanho de alguma entrada.

---

## 4 Testes

### 4.1 Performance e Resultados

Foi feito um planejamento para os testes dos algoritmos, inicialmente foi definido que seriam feitos testes com 7 entradas diferentes. Foram analisados o tempo e o uso de memória, além disso foram realizados testes com outros tipos de algoritmos de ordenação para os pontos analisados. Os testes foram usando o "Selection Sort", porém o "Bubble Sort" também foi testado, mas o resultado do maior ponto foi incorreto. Foi feita uma tentativa com o "Quick Sort", mas ao rodar o algoritmo era sempre encontrado o erro de "segmentation fault", outro erro encontrado foi uma incompatibilidade com linuxmint, impedindo que o código fosse executado.



```

irijon@Irijon: ~/Downloads/ArquivoCerto
File Edit View Search Terminal Help
(base) irijon@Irijon:~/Downloads/ArquivoCerto$ make clean
rm -rf *.o
(base) irijon@Irijon:~/Downloads/ArquivoCerto$ make
gcc -c pontos.c
gcc -c arquivos.c
gcc pontos.o arquivos.o main.c -o tp1
/usr/bin/ld: arquivos.o(.bss+0x0): multiple definition of `pontos'; pontos.o(.bss+0x0): first defined here
/usr/bin/ld: arquivos.o(.bss+0x8): multiple definition of `num pontos'; pontos.o(.bss+0x8): first defined here
/usr/bin/ld: arquivos.o(.bss+0xc): multiple definition of `Maior'; pontos.o(.bss+0xc): first defined here
/usr/bin/ld: arquivos.o(.bss+0x10): multiple definition of `A'; pontos.o(.bss+0x10): first defined here
/usr/bin/ld: arquivos.o(.bss+0x14): multiple definition of `B'; pontos.o(.bss+0x14): first defined here
/usr/bin/ld: /tmp/ccGsbDeC.o(.bss+0x0): multiple definition of `pontos'; pontos.o(.bss+0x0): first defined here
/usr/bin/ld: /tmp/ccGsbDeC.o(.bss+0x8): multiple definition of `num pontos'; pontos.o(.bss+0x8): first defined here
/usr/bin/ld: /tmp/ccGsbDeC.o(.bss+0xc): multiple definition of `Maior'; pontos.o(.bss+0xc): first defined here
/usr/bin/ld: /tmp/ccGsbDeC.o(.bss+0x10): multiple definition of `A'; pontos.o(.bss+0x10): first defined here
/usr/bin/ld: /tmp/ccGsbDeC.o(.bss+0x14): multiple definition of `B'; pontos.o(.bss+0x14): first defined here
collect2: error: ld returned 1 exit status
make: *** [Makefile:2: all] Error 1
(base) irijon@Irijon:~/Downloads/ArquivoCerto$

```

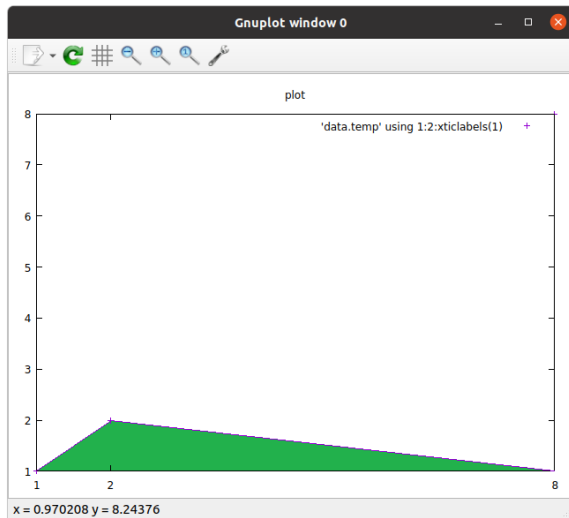
Figura 2: Erro make.

Por fim com a análise dos resultados foi observado que quanto mais entradas maior o tempo e uso de memória. Os testes de 2, 4, 10 e 100 entradas tiveram uso de memória semelhantes e tempos iguais. Já as entradas 1000, 50000 e 100000 obtiveram tempo e uso de memória completamente diferentes.

Pode se dizer que os testes foram considerados bem sucedidos. Abaixo estarão os capturas dos testes. Usamos a função do "Gnuplot" em nosso algoritmo para fazer a plotagem do ponto mais alto no hipercampo. (Por erro das entradas da função plot, valores "Maior" que forem menores que as âncoras são representados atrás das âncoras A e B, como no exemplo da figura 3.)



## 4.1.1 Testes com 2 entradas.



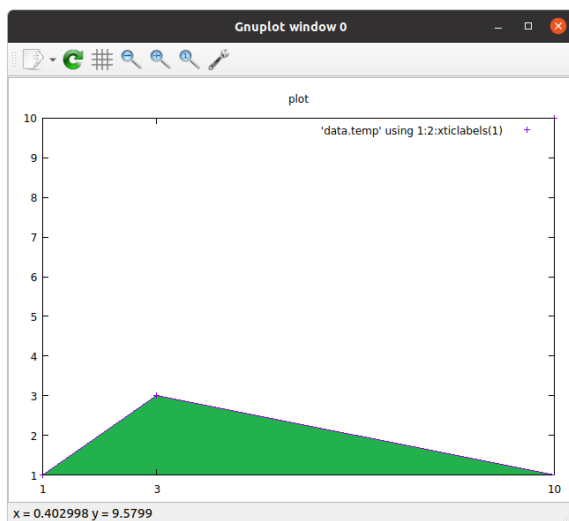
(a) Plot teste\_2

```
-----  
Maior valor possível = 1  
Tempo = 0.000000 segundos  
Uso de memória = 680 Kb  
-----
```

(b) Valores no terminal

Figura 3

## 4.1.2 Testes com 4 entradas.



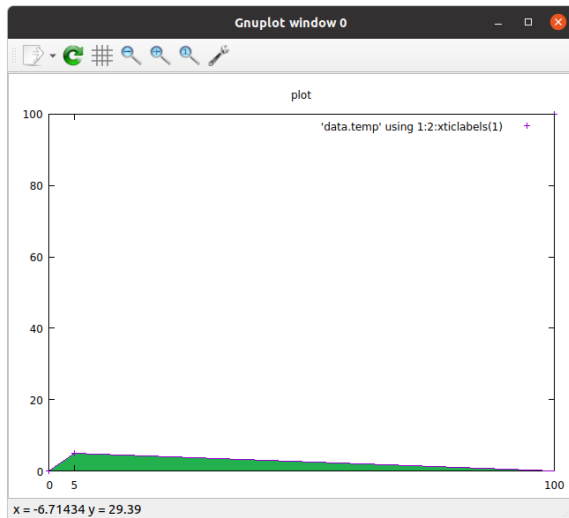
(a) Plot teste\_4

```
-----  
Maior valor possível = 3  
Tempo = 0.000000 segundos  
Uso de memória = 680 Kb  
-----
```

(b) Valores no terminal

Figura 4

## 4.1.3 Testes com 10 entradas.



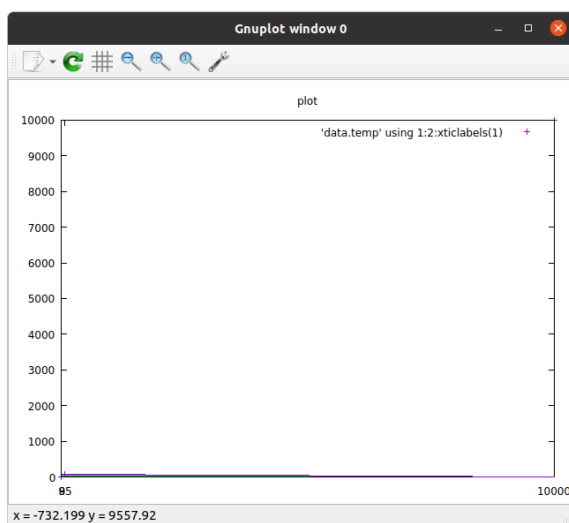
(a) Plot teste\_10

```
Maior valor possível = 5
Tempo = 0.000000 segundos
Uso de memória = 684 Kb
```

(b) Valores no terminal

Figura 5

## 4.1.4 Testes com 100 entradas.



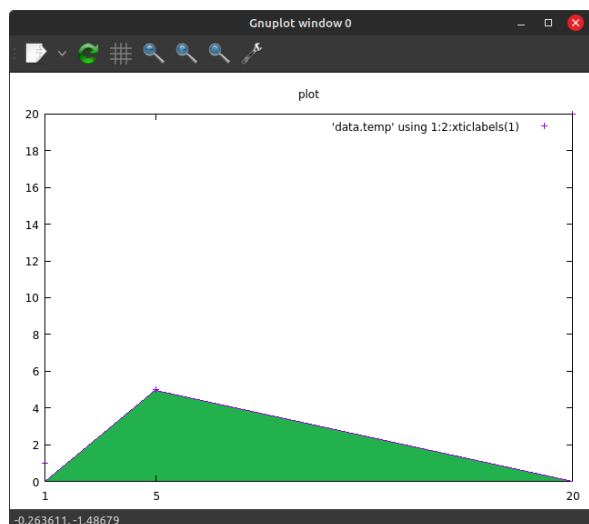
(a) Plot teste\_100

```
Maior valor possível = 85
Tempo = 0.000000 segundos
Uso de memória = 680 Kb
```

(b) Valores no terminal

Figura 6

## 4.1.5 Testes com 1000 entradas.



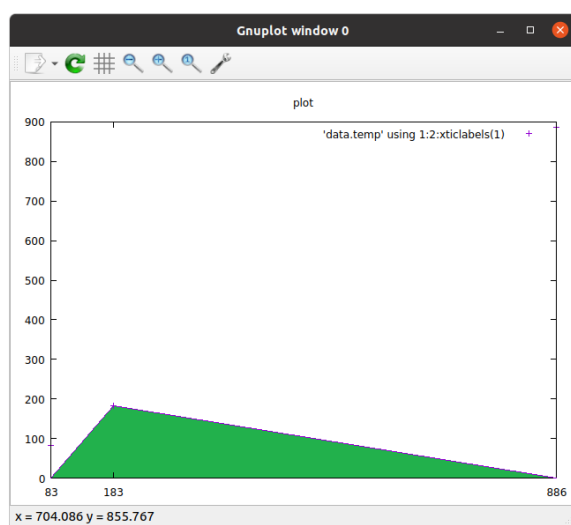
(a) Plot teste\_1000

```
Maior valor possível = 5
Tempo = 0.015625 segundos
Uso de memória = 692 Kb
```

(b) Valores no terminal

Figura 7

## 4.1.6 Testes com 50000 entradas.



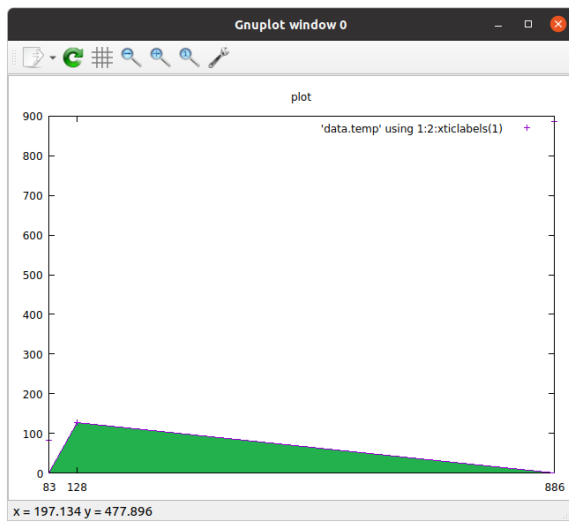
(a) Plot teste\_50000

```
Maior valor possível = 183
Tempo = 11.890625 segundos
Uso de memória = 1860 Kb
```

(b) Valores no terminal

Figura 8

## 4.1.7 Testes com 100000 entradas.



(a) Plot teste\_100000

```
-----  
Maior valor possível = 128  
-----  
Tempo = 3.109375 segundos  
-----  
Uso de memória = 1276 Kb  
-----
```

(b) Valores no terminal

Figura 9

## 5 Conclusão

Neste trabalho, aprendemos como é o funcionamento de um sistema de hipercampos com a implementação em C .

Resumidamente neste trabalho foram apresentados os conhecimentos para a implementação de um sistema Hipercampo com âncoras e pontos, a linguagem C foi escolhida para realizar estas implementações utilizando a biblioteca padrão da linguagem, como os testes mostram é necessário ligarmos o ponto  $v$  ao ponto  $P$  desenhando os 2 segmentos da reta  $(v,A)$  e  $(v,B)$  que fiquem entre as âncoras que os interceptam, dessa forma é possível encontrarmos o ponto mais alto garantindo que a estrutura do sistema não se altere e permaneça rodando de forma correta, apresentando uma solução correta e eficiente, além de trazer um desafio na área de programação, também nos trouxe muito conhecimento sobre a parte da implementação e como desenvolver um hipercampos em C.

---

## 6 Referências

<https://mundoeducacao.uol.com.br/matematica/equacao-reduzida-reta.htm>  
[https://daemoniolabs.wordpress.com/2011/10/07/usando-com-as-funcoes-getopt-e-getopt\\_long-em-c/](https://daemoniolabs.wordpress.com/2011/10/07/usando-com-as-funcoes-getopt-e-getopt_long-em-c/)  
<https://www.geeksforgeeks.org/c-program-to-read-contents-of-whole-file/>  
<https://mundoeducacao.uol.com.br/matematica/calculo-coeficiente-angular.htm>

---