Instituto Federal de Educação Ciência e Tecnologia
do Norte de Minas Gerais - IFNMG
Bacharelado em Ciência da Computação

Disciplina de Laboratório de PAA

Lab 1: Ordenações.

Thiago Emanuel Silva Antunes Lopes

Sumário

Capítulo 1

Introdução

Este relatório tem o objetivo de detalhar o funcionamento de dois algoritmos de ordenação, são eles o mergesort e o heapsort, analisando também suas complexidades e tempo de execução com gráficos. A atividade foi proposta pelo Professor Alberto Miranda na disciplina Laboratório de Projeto de analise de algoritmos e realizado pelo discente Thiago Emanuel Silva Antunes Lopes.

Capítulo 2

Algoritmos de ordenação

Na atividade foi pedido a implementação dos dois algoritmos, sendo eles o mergesort e heapsort. A partir das implementações é feita a analise dos algoritmos.

A implementação abaixo foi feita para os dois algoritmos de ordenação, é nescessário comentar a chamada de uma função para analisar a outra. Linhas: 123 ou 124.

2.1 Implementação

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <sys/time.h>
5
6
7  // MERGE SORT //
8  void merge(int* vetor, int* vetoraux,int primeiro,int meio,int ultimo)
       {
9
10     int i,j,n;
11     i=primeiro;
12     j=meio+1;
13     n=0;
14     while(i<=meio && j<=ultimo) ///compara se um dos lados ja acabou
15     {
```

```
16          if(vetor[i]>vetor[j])
17          {
18              vetoraux[n++]=vetor[j];
19              j++;
20          }
21          else
22          {
23              vetoraux[n++]=vetor[i];
24              i++;
25          }
26      }
27      while(j<=ultimo) ///descarrega o vetor lado2
28      {
29          vetoraux[n]=vetor[j];
30          j++;
31          n++;
32      }
33      while(i<=meio) ///descarrega o vetor lado1
34      {
35          vetoraux[n]=vetor[i];
36          i++;
37          n++;
38      }
39          for(n=0;n<ultimo-primeiro+1;n++)
40                  vetor[n+primeiro]=vetoraux[n];
41  }
42  void mergesort(int* vetor, int* vetoraux,int primeiro,int ultimo) {
43      if(ultimo>primeiro)
44      {
45          int meio;
46          meio=(primeiro+ultimo)/2;
47          mergesort(vetor, vetoraux,primeiro,meio);
48          mergesort(vetor, vetoraux,meio+1,ultimo);
49          merge(vetor, vetoraux,primeiro,meio,ultimo);
50      }
51  }
```

```
52
53 // HEAP SORT //
54
55 void heapify(int *vetor, int n, int i)
56 {
57     int maior = i, e=2*i + 1, d=2*i + 2;
58
59     if (e < n && vetor[e] > vetor[maior])
60         maior = e;
61
62     if (d < n && vetor[d] > vetor[maior])
63         maior = d;
64
65     if (maior != i) ///verificar se o maior nao    a raiz
66     {
67         ///fazendo a troca
68         int temporario;
69         temporario = vetor[i];
70         vetor[i]=vetor[maior];
71         vetor[maior]=temporario;
72
73         heapify(vetor, n, maior);
74     }
75 }
76
77 void heap(int *vetor, int n)
78 {
79     ///constroi heap
80     for (int i=n/(2-1); i>=0; i--)
81         heapify(vetor, n, i);
82
83     for (int i=n-1; i>=0; i--)
84     {
85         ///fazendo a troca
86         int temporario;
87         temporario = vetor[0];
```

```
88          vetor[0]=vetor[i];
89          vetor[i]=temporario;
90
91          heapify(vetor, i, 0);
92      }
93  }
94
95
96  // FUNCAO TEMPO //
97  long double getNow()
98  {
99      struct timeval now;
100     long double valor = 1000000;
101
102     gettimeofday(&now,NULL);
103
104     return ((long double)(now.tv_sec*valor)+(long double)(now.tv_usec))
            ;
105 }
106
107 void temp(int *vetor, int *vetoraux)
108 {
109     long double t=0, ti, tf;
110     int i=10;
111     while(t<60)
112     {
113         vetor=(int *) malloc(i * sizeof (int));
114             vetoraux=(int *) malloc(i * sizeof (int));
115         for(int j=0;j<i;j++)
116         {
117             vetor[j]=rand()%100000000;
118         }
119
120         ti = getNow();
121
122         /// COMENTE A CHAMADA DA FUNCAO QUE NAO DESEJA EXECUTAR A
```

```
            ORDENACAO
123         heap(vetor, i);
124         mergesort(vetor, vetoraux,0,i-1);
125
126         tf = getNow();
127         t=(tf-ti)/1000000;
128             printf("\n %d elementos = %Lf segundos\n", i, t);
129         //cout<<"\n"<<fixed<< i <<" elementos  = "<< t <<" segundos\n\
            n";
130         i=i*2;
131         free(vetor);
132     }
133
134 }
135
136 int main()
137 {
138     srand(time(NULL));
139     int* vetor,*vetoraux;
140     //int n;
141     //Tamanho do vetor
142     //scanf("%d", &n);
143
144
145     temp(vetor, vetoraux);
146
147     return 0;
148 }
```

2.2 Gráficos da execução dos algoritmos

2.2.1 Mergesort

Tabela Merge:

Quantidade de elementos	Tempo (s)
10	0.000002
20	0.000005
40	0.000008
80	0.000016
160	0.000031
320	0.000067
640	0.000141
1280	0.000310
2560	0.000668
5120	0.001443
10240	0.000813
20480	0.001258
40960	0.002694
81920	0.006055
163840	0.012223
327680	0.025424
655360	0.053308
1310720	0.111334
2621440	0.232450
5242880	0.490090
10485760	1.018367
20971520	2.173764
41943040	4.436050
83886080	9.256506
167772160	19.169308
335544320	39.880480
671088640	111.874418

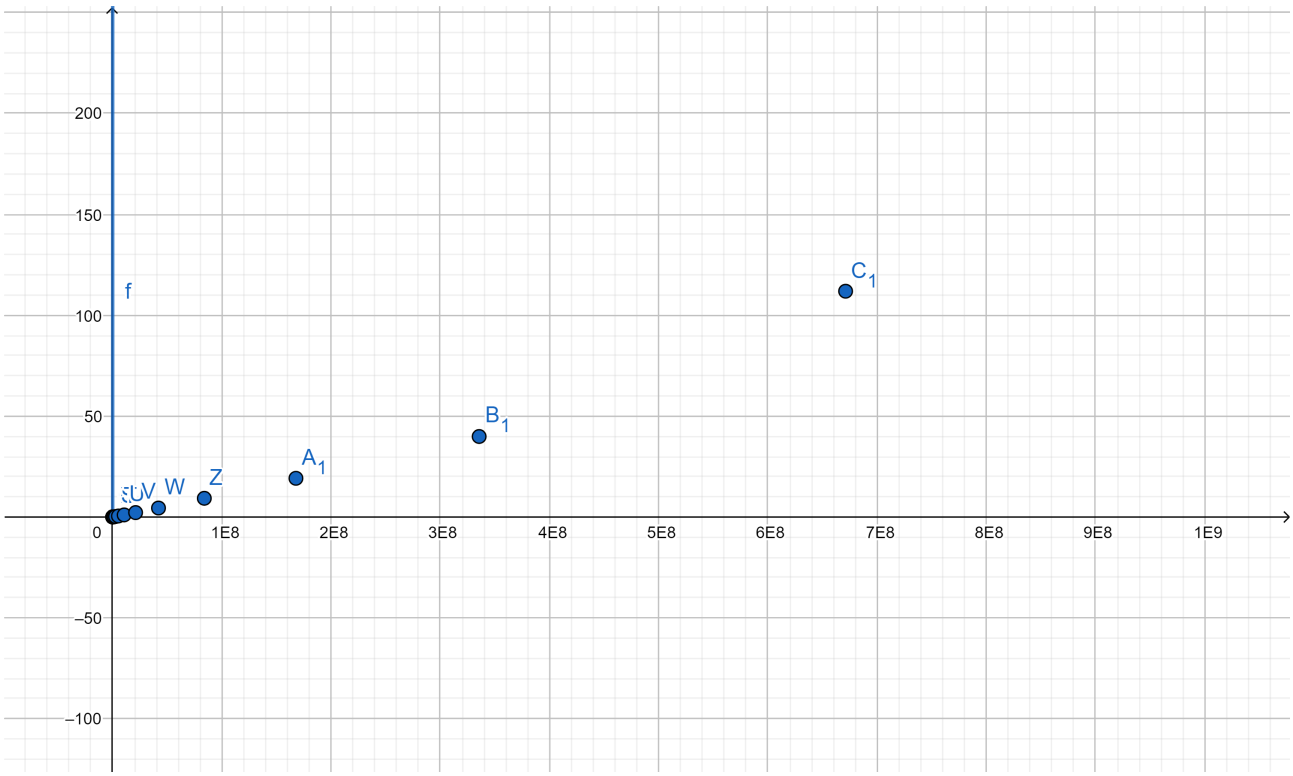


Figura 2.1: (Tamanho, Tempo(s))

2.2.2 Heapsort

Tabela Heap:

Quantidade de elementos	Tempo (s)
10	0.000001
20	0.000001
40	0.000002
80	0.000004
160	0.000008
320	0.000018
640	0.000038
1280	0.000082
2560	0.000176
5120	0.000379
10240	0.000818
20480	0.001813
40960	0.003181
81920	0.006845
163840	0.014551
327680	0.031517
655360	0.069282
1310720	0.152195
2621440	0.369074
5242880	0.836048
10485760	2.210076
20971520	4.911016
41943040	10.791628
83886080	25.075090
167772160	53.781858
335544320	118.230318

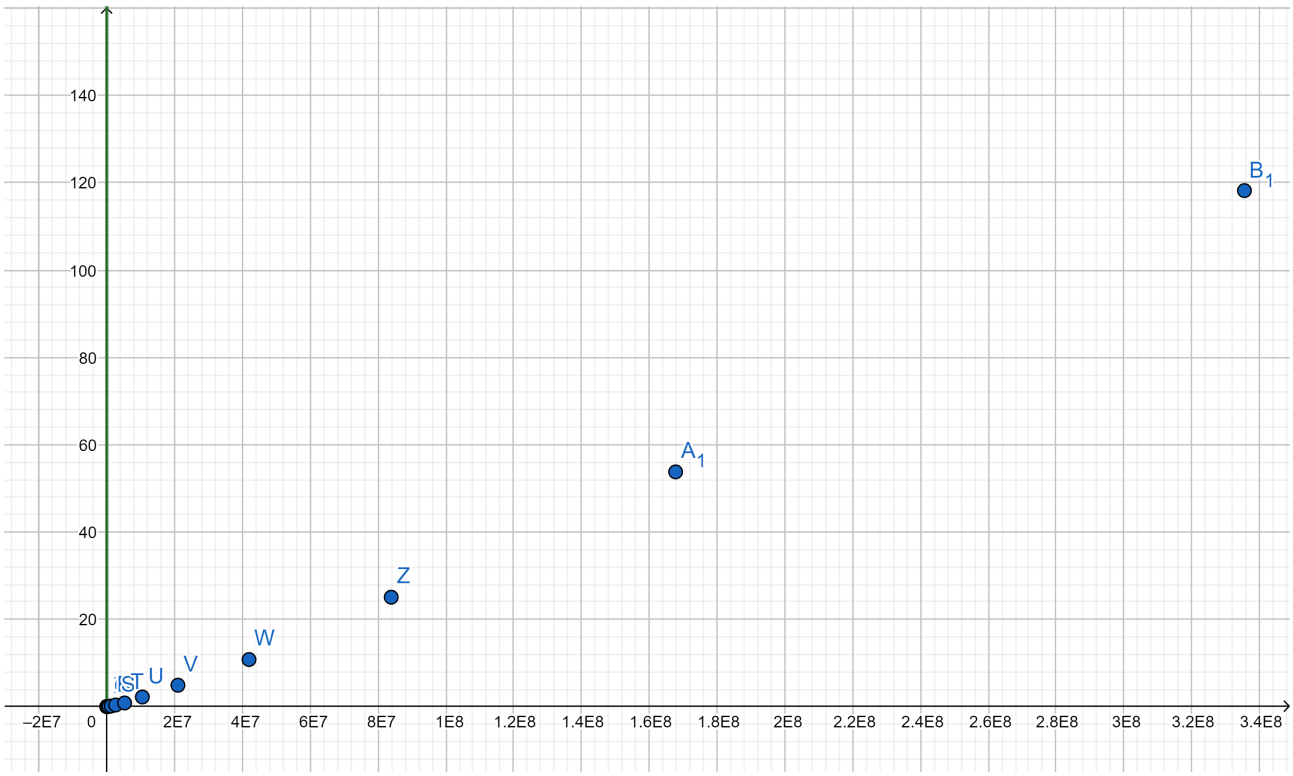


Figura 2.2: (Tamanho, Tempo(s))

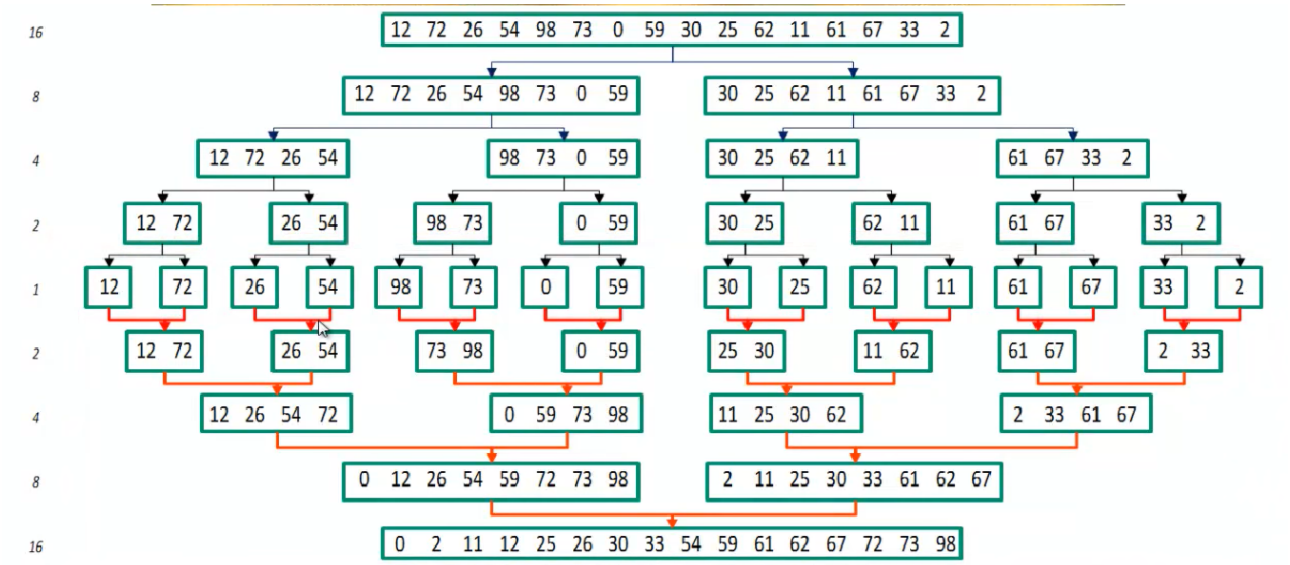
2.3 Análise de complexidade

2.3.1 Mergesort

O Mergesort é um algoritmo de intercalação, em que cada passo dado é consumido um elemento de dois vetores, exemplo, um elemento de a ou b. Se o vetor a tiver tamanho x, e b tamanho y, logo a intercalação entre os vetores exige no máximo x+y passos.

Na primeira faze do merge, onde somente é dividido o vetor ao meio até que fique no tamanho 1, não dá trabalho computacional significante.

Exemplo da execução do merge



Na próxima fase, começa a intercalação comparando os vetores. No exemplo da figura acima, é intercalado primeiramente dois elementos de tamanho um, dois em dois, 8 intercalações no total, então é feito 16 passos para esse nível, formando novos vetores com 2 elementos cada.
O processo é repetido no novo nível, com apenas 4 intercalações e como os vetores são de tamanho 2, então são feitos novamente 16 passos. A partir dessa análise é possível concluir q serão sempre os 16 passos em todos os níveis, até que se tenha somente um vetor.
Portanto o numero de passos totais, será (n), que é o tamanho do vetor, multiplicado pela altura da arvore de intercalação. Como em cada nível o tamanho dos vetores é dobrado, logo a altura é o quanto 2 elevado a uma variável da o tamanho do vetor de entrada, no caso (n), isto é log(n) na base 2.

Provando:

$T(n) = T(n/2) + T(n/2) + n$

$T(n) = 2T(n/2) + n$

$T(n/2) = 2T(n/4) + n/2$

$T(n) = 2(2T(n/4) + n/2) + n$

$T(n) = 4T(n/4) + 2n$

$T(n/4) = 2T(n/8) + n/4$

$T(n) = 4(2T(n/8) + n/4) + 2n$

$T(n) = 8T(n/8) + 3n$

$T(n) = 16T(n/16) + 4n$

Isso acontece ate $T(1) = 1$, ou seja, o tamanho do vetor ser 1

$T(n) = 2^i T(n/2^i) + in$

$Sendo : n/2^i = 1$

$Logo : lg(n)(base2) = i$

$T(n) = nT(1) + nlg(n)$

$T(n) = n + nlg(n)$

Θ(nlg(n))

2.3.2 Heapsort

Na função heapify, as declarações de variáveis fora dos ”if's”e os ”if's”são $\Theta(1)$. Enquanto as operações feitas dentro dos ”if's”são O(1). A cada chamada recursiva da função heapify desce nível por nível, então ela é <= T(altura - 1)

A função fica: T(h)<=T(h-1)+Θ(1)

O consumo de tempo para um nó i depende da altura da arvore

Como a altura de um nó i é: h = log(n/i)

Então:

T(h)= O(h)

= O(log(n/i))

= O(log(n/i))

= O(log(n))

Na função heap, o primeiro for começa de n/2 e vai até 0, então ele é Θ(n/2), dentro desse for é chamada a função heapify, que como foi visto é O(log(n)), sendo assim fica Θ(n/2log(n))

No segundo for, ele inicia de n-1 e vai até 0, então é Θ(n), dentro do for tem operações O(1) e a chamada da função heapify, portanto fazendo as multiplicações temos Θ(nlog(n)).

Somando a complexidade dos dois for temos:

Θ((3/2)nlog(n))

= Θ(nlog(n))

2.4 Estimativa de tamanho do vetor para ordenação em 24 horas

Para fazer a estimativa de tempo de execução do algoritmo para qualquer tamanho é necessário se calcular uma constante que é multiplicada pela função do algoritmo. O exercício proposto foi para fazer a estimativa do tamanho do vetor que sua ordenação seja realizada em 24 horas

para os dois algoritmos de ordenação.

2.4.1 Mergesort

A partir dos dados tabelados do merge, vamos calcular sua constante. Para isso, vamos utilizar o tempo para ordenar 3 vetores, e fazer a média das constantes.
Para vetor de tamanho 83886080:

$$Const * 83886080 * \log_2 83886080 = 9.256506$$
$$Const = 4,192176 * 10^{-9}$$

Para vetor de tamanho 167772160:

$$Const * 167772160 * \log_2 167772160 = 19.169308$$
$$Const = 4,181915 * 10^{-9}$$

Para vetor de tamanho 167772160:

$$Const * 335544320 * \log_2 335544320 = 39.880480$$
$$Const = 4,196504 * 10^{-9}$$

Fazendo a média:

$$Const = \frac{4,192176 * 10^{-9} + 4,181915 * 10^{-9} + 4,196504 * 10^{-9}}{3}$$
$$Const = 4,190198 * 10^{-9}$$

Com a constante calculada, agora vamos fazer a estimativa do tamanho do vetor que sua ordenação leve 24 horas.
Como foi utilizado o tempo em segundos, vamos transformar as 24 horas em segundos. Isso dá

um total de 86400 segundos.

A equação fica:

$$4,190198 * 10^{-9} * n \log_2 n = 86400$$

$$n \log_2 n = 2,061955 * 10^{13}$$

Para encontrar o n que substituindo na equação dê $2,061955^{13}$, é feita uma busca binária.

O resultado dessa busca é aproximadamente: $5,293996 * 10^{11}$

2.4.2 Heapsort

Agora a partir dos dados tabelados do heap, vamos calcular sua constante. Para isso, vamos utilizar o tempo para ordenar 3 vetores, e fazer a média das constantes.

Para vetor de tamanho 41943040:

$$Const * 41943040 * \log_2 41943040 = 10.791628$$

$$Const = 1,0160856 * 10^{-8}$$

Para vetor de tamanho 83886080 :

$$Const * 83886080 * \log_2 83886080 = 25.075090$$

$$Const = 1,1356249 * 10^{-8}$$

Para vetor de tamanho 167772160:

$$Const * 167772160 * \log_2 167772160 = 53.781858$$

$$Const = 1,1732878 * 10^{-8}$$

Fazendo a média:

$$Const = \frac{1,0160856 * 10^{-8} + 1,1356249 * 10^{-8} + 1,1732878 * 10^{-8}}{3}$$

$$Const = 1,1083337 * 10^{-8}$$

Com a constante calculada, agora vamos fazer a estimativa do tamanho do vetor que sua ordenação leve 24 horas.

Como foi utilizado o tempo em segundos, vamos transformar as 24 horas em segundos. Isso dá um total de 86400 segundos.

A equação fica:

$$1,1083337 * 10^{-8} * n \log_2 n = 86400$$

$$n \log_2 n = 7,7954861 * 10^{12}$$

Para encontrar o n que substituindo na equação dê $7,7954861 * 10^{12}$, é feita uma busca binária. O resultado dessa busca é aproximadamente: $2,073633279 * 10^{11}$

2.4.3 Gráfico para estimativas

Gráfico de execução dos algorítimos em 24 horas

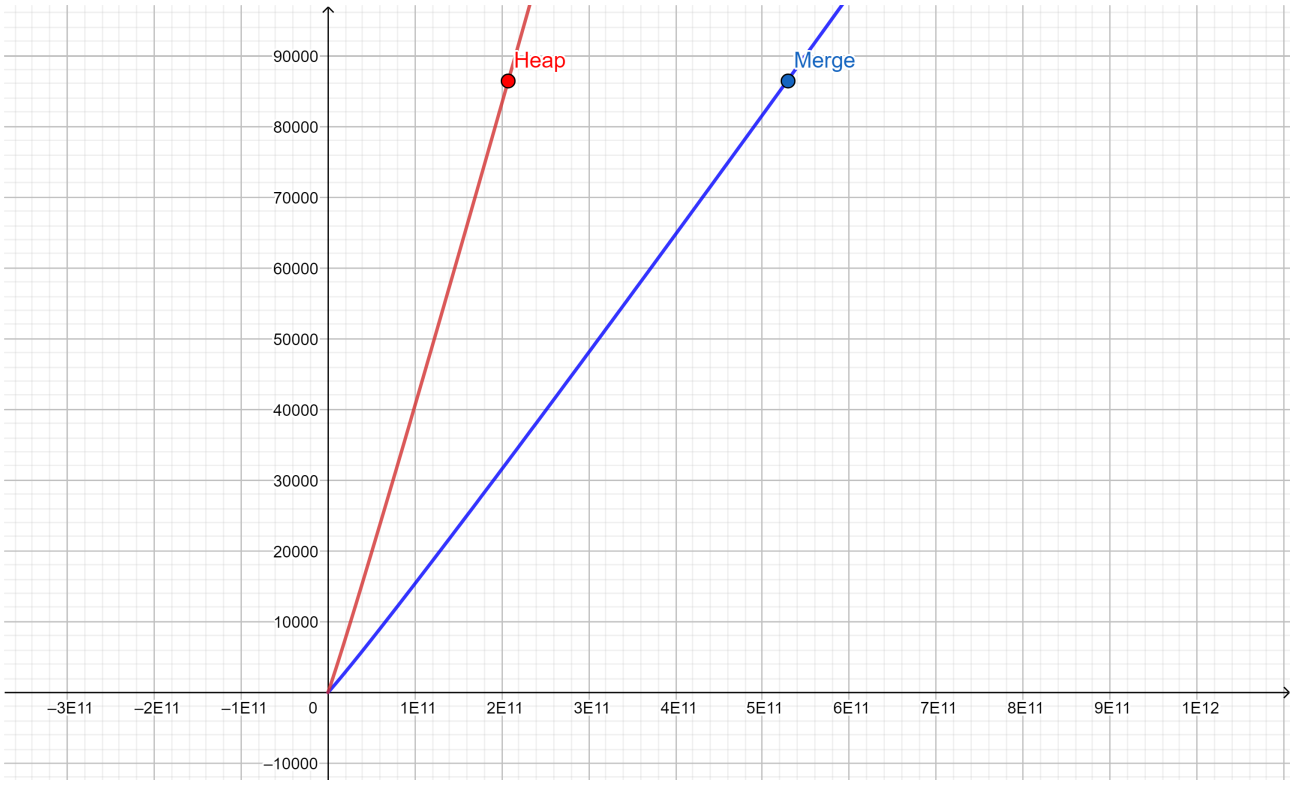


Figura 2.3: (Tamanho, Tempo(s)

2.4.4 Gráfico das funções

Gráfico das funções multiplicadas pelas constantes. heap em vermelho e merge em azul.

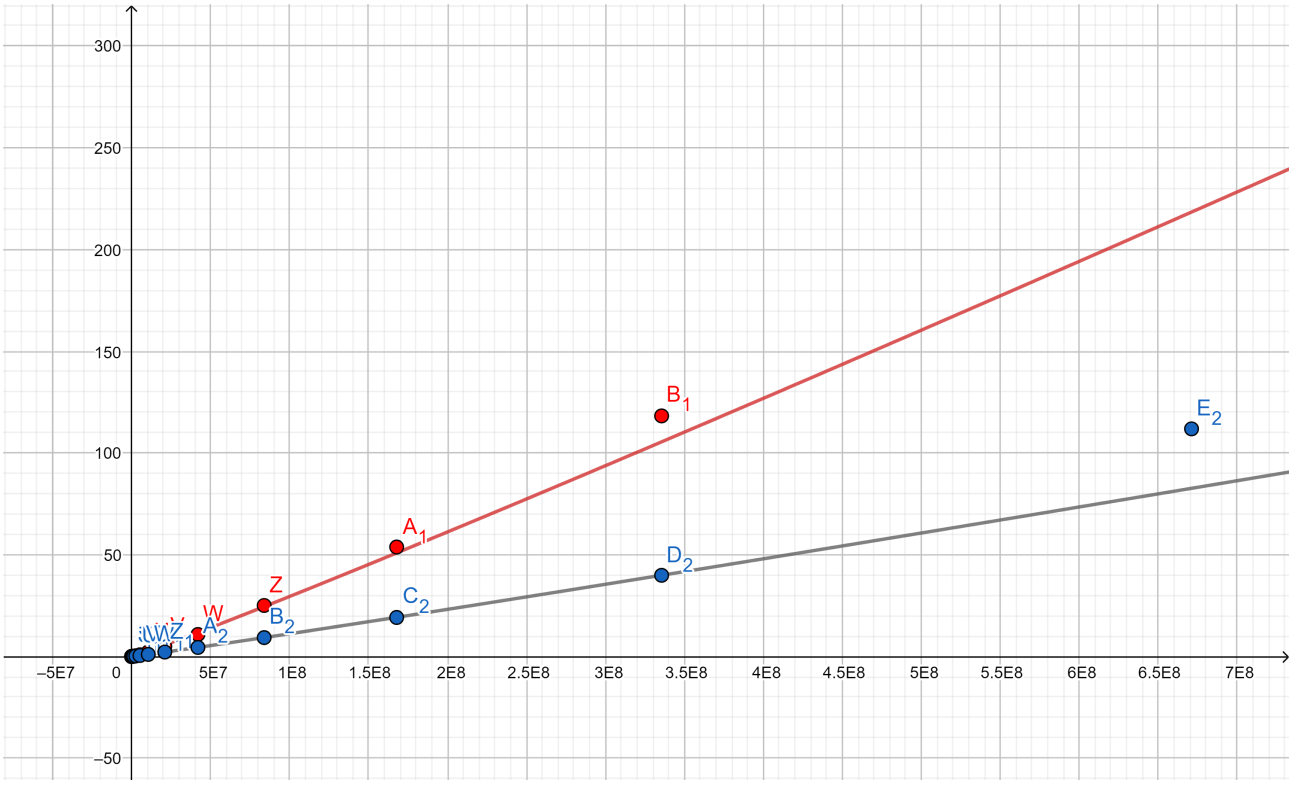


Figura 2.4: (Tamanho, Tempo(s)

Capítulo 3

Conclusão

3.1 Qual a melhor melhor implementação?

A partir da execução dos algoritmos foi possível perceber que mesmo tendo mesmas complexidades, na prática possuem tempos de execução diferentes.
O Merge possui complexidade n + nlog(n), enquanto o Heap (3/2)nlog(n).
Como foi apresentado nos gráficos, o Heap para pequenas entradas é mais rápido que o Merge, porém com o incremento do tamanho das entradas o Merge acaba ficando mais rápido que o Heap.
Isso acontece pelo fato do Merge dividir o vetor em pequenos pedaços que ficam melhor alocados na memória, facilitando as comparações, enquanto o Heap trabalha com o vetor sem dividi-lo. Portanto para vetores muito grandes o Merge é um algoritmo mais eficiente que o Heap.