Análise de Algoritmos - Ordenação

Gustavo de Souza Silva
Guilherme de Souza Silva
Arthur Xavier
Schumaiquer Souto

Faculdade de Computação
Universidade Federal de Uberlândia

1 de agosto de 2017

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Sumário

Capítulo 1

Introdução

Este relatório tem como objetivo fazer a análise de diversos algoritmos já conhecidos de ordenação. O intuito deste trabalho é comprovar que as provas matemáticas realmente acontecem em um ambiente real de execução.

1.1 Codificação

O arquivo vetor.c mantém todas as funções a respeito do vetor, como geração, preenchimento, etc.

Listagem 1.1: Arquivo referente ao vetor

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  #include "vetor.h"
5
6  #define MAX(x,y) ( \
7      { __auto_type __x = (x); __auto_type __y = (y); \
8        __x > __y ? __x : __y; })
9
10 #define TROCA(v, i, j, temp) (  \
11     { (temp) = v[(i)];      \
12       v[(i)] = v[(j)];      \
13       v[(j)] = (temp);})
14
15 /*
16 typedef enum ordem {ALEATORIO, CRESCENTE, DECRESCENTE} Ordem;
17 typedef enum modificador {TOTALMENTE, PARCIALMENTE} Modificador;
18 typedef int Percentual;
19 */
20
21 double rand_double(double min, double max)
22 { // Retorna números em ponto flutuante aleatórios uniformemente
23   // distribuídos no intervalo fechado [min,max].
24     return min + (rand() / (RAND_MAX / (max-min)));
25 }
26
27 int rand_int(int min, int max){
```

```
28   // Retorna números inteiros aleatórios uniformemente distribuídos
29   // no intervalo fechado [min,max].
30   // Para maiores informações:
31   // https://stackoverflow.com/questions/2509679/how-to-generate-a-random-
         number-from-within-a-range
32   unsigned long num_baldes = (unsigned long) max-min+1;
33   if (num_baldes<1){
34     fprintf(stderr, "Intervalo invalido\n");
35     exit(-1);
36   }
37   unsigned long num_rand  = (unsigned long) RAND_MAX+1;
38   unsigned long tam_balde = num_rand / num_baldes;
39   unsigned long defeito   = num_rand % num_baldes;
40   long x;
41   do
42     x = random();
43   while (num_rand - defeito <= (unsigned long)x);
44   return x / tam_balde + min;
45 }
46
47
48 static void inline preenche_vetor_int(int * v, int n, int k, int q, int r,
       int incr){
49   int i, j;
50       i=0;
51       while (i < n) {
52         for(j=i; j < i+q; j++)
53           v[j] = k;
54
55         i = i + q;
56         if (r > 0) {
57           v[j] = k;
58           i = i + 1;
59           r = r - 1;
60         }
61         k = k + incr;
62       }
63 }
64
65
66 int * gera_vetor_int(int n, Modificador c, Ordem o, Percentual p,
67                      int minimo, int maximo){
68   int i,j; // índices
69   int a = maximo - minimo + 1; // amplitude do intervalo
70   int q = n / a;  // número mínimo de valores repetidos
71   int r = n % a;  // r elementos terão o número (q+1) valores repetidos
72   int k;          // valor do elemento atualmente sob consideração
73   int * v;        // vetor[0..n-1] a ser preenchido
74
75   CONFIRME(n >= 1, "O número de elementos deve ser estritamente positivo.\
         n");
76   CONFIRME(maximo >= minimo, "O valor máximo deve ser maior que o mínimo.\
         n");
77   CONFIRME(0 <= p && p <= 100, "O percentual deve estar entre [0,100]\n");
78
79   v = (int *) calloc(n, sizeof(int)); // aloca um vetor com n inteiros
80   CONFIRME(v != NULL, "calloc falhou\n");
81
82   switch (o) {
```

```
83      case CRESCENTE:
84        preenche_vetor_int(v, n, minimo, q, r, 1);
85        break;
86      case DECRESCENTE:
87        preenche_vetor_int(v, n, maximo, q, r, -1);
88        break;
89      case ALEATORIO:
90        for(i=0; i<n; i++) v[i] = rand_int(minimo,maximo);
91        break;
92      default: CONFIRME(false, "Ordem Inválida\n");
93    }
94
95    switch (c) {
96    case PARCIALMENTE:
97      q = (p * n) / 200;
98      for(i=0;i<q;i++)
99        TROCA(v,i,n-i-1,k);
100     break;
101   case TOTALMENTE: break;
102   default: CONFIRME(false, "Modificador do vetor desconhecido");
103   }
104
105   return v;
106 }
107
108
109 static void inline preenche_vetor_double(double * v, int n, double inicial
      ,
110                                          double delta, double sinal)
111 {
112   int i;
113   for(i=0;i<n;i++)
114     v[i]= inicial + sinal*i*delta;
115 }
116
117
118 double * gera_vetor_double(int n, Modificador c, Ordem o, Percentual p,
119                            double minimo, double maximo){
120   int i; // índice
121   double a = maximo - minimo;     // amplitude do intervalo
122   double delta;
123   double * v;       // vetor[0..n-1] a ser preenchido
124   double temp;
125   int q;
126
127   CONFIRME(n >= 1, "O número de elementos deve ser estritamente positivo.\
        n");
128   CONFIRME(maximo >= minimo, "O valor máximo deve ser maior que o mínimo.\
        n");
129   CONFIRME(0 <= p && p <= 100, "O percentual deve estar entre [0,100]\n");
130
131   delta =  a / MAX(n-1.0, 1.0);    // incremento nos elementos do vetor
132   v = (double *) calloc(n, sizeof(double)); // aloca um vetor com n
        doubles
133   CONFIRME(v != NULL, "callocfalhou\n");
134
135   switch (o) {
136     case CRESCENTE:
137       preenche_vetor_double(v, n, minimo, delta, 1);
```

```
138       break;
139     case DECRESCENTE:
140       preenche_vetor_double(v, n, maximo, delta, -1);
141       break;
142     case ALEATORIO:
143       for(i=0; i<n; i++) v[i] = rand_double(minimo,maximo);
144       break;
145     default: CONFIRME(false, "Ordem Inválida\n");
146   }
147 
148   switch (c) {
149   case PARCIALMENTE:
150     q = (p * n) / 200;
151     for(i=0;i<q;i++)
152       TROCA(v,i,n-i-1,temp);
153     break;
154   case TOTALMENTE: break;
155   default: CONFIRME(false, "Modificador do vetor desconhecido");
156   }
157 
158   return v;
159 }
160 
161 void escreva_vetor_int(int * v, int n, char * arq){
162   int i;
163   FILE* fd = NULL;
164 
165   fd = fopen(arq, "w");
166   CONFIRME(fd!= NULL, "escreva_vetor_int: fopen falhou\n");
167 
168   // Na primeira linha está o número de elementos
169   fprintf(fd, "%d\n", n);
170   for(i=0; i<n; i++)
171     fprintf(fd, "%d\n", v[i]);
172   fclose(fd);
173 }
174 
175 void escreva_vetor_double(double * v, int n, char * arq){
176   int i;
177   FILE* fd = NULL;
178 
179   fd = fopen(arq, "w");
180   CONFIRME(fd!= NULL, "escreva_vetor_double: fopen falhou\n");
181 
182   // Na primeira linha está o número de elementos
183   fprintf(fd, "%d\n", n);
184   for(i=0; i<n; i++)
185     fprintf(fd, "%f\n", v[i]);
186   fclose(fd);
187 }
188 
189 int * leia_vetor_int(char * arq, int * n){
190   int i;
191   FILE* fd = NULL;
192   int * v;
193 
194   fd = fopen(arq, "r");
195   CONFIRME(fd!= NULL, "leia_vetor_int: fopen falhou\n");
196 
```

```
197   // Leia o número de elementos do vetor
198   CONFIRME(fscanf(fd, "%d\n", n) == 1,
199            "leia_vetor_int: erro ao ler o número de elementos do vetor\n")
                 ;
200
201   v = (int *) calloc(*n, sizeof(int)); // aloca um vetor com n inteiros
202   CONFIRME(v != NULL, "leia_vetor_int: calloc falhou\n");
203
204   i=0;
205   while(fscanf(fd, "%d\n", &v[i]) == 1) i++;
206   fclose(fd);
207
208   return v;
209 }
210
211 double * leia_vetor_double(char * arq, int * n){
212   int i;
213   FILE* fd = NULL;
214   double * v;
215
216   fd = fopen(arq, "r");
217   CONFIRME(fd!= NULL, "leia_vetor_int: fopen falhou\n");
218
219   // Leia o número de elementos do vetor
220   CONFIRME(fscanf(fd, "%d\n", n) == 1,
221            "leia_vetor_double: erro ao ler o número de elementos do vetor\
                 n");
222
223
224  // Aloca um vetor com n doubles
225   v = (double *) calloc(*n, sizeof(double));
226   CONFIRME(v != NULL, "leia_vetor_int: calloc falhou\n");
227
228   i=0;
229   while(fscanf(fd, "%lf\n", &v[i]) == 1) i++;
230   fclose(fd);
231
232   return v;
233 }
234
235 bool esta_ordenado_int(Ordem o, int * v, int n){
236   int i;
237
238   CONFIRME(n > 0,
239            "estaOrdenado_int: o número de elementos deve ser maior que
                 zero.\n");
240   if (n == 1) return true;
241   switch (o) {
242     case CRESCENTE:
243       for(i=0;i<n;i++)
244         if (v[i-1] > v[i])
245           return false;
246       break;
247     case DECRESCENTE:
248       for(i=0;i<n;i++)
249         if (v[i-1] < v[i])
250           return false;
251       break;
252     default: CONFIRME(false, "estaOrdenado_int: Ordem Inválida\n");
```

```
253   }
254   return true;
255 }
256
257
258 bool esta_ordenado_double(Ordem o, double * v, int n){
259   int i;
260
261   CONFIRME(n > 0,
262            "estaOrdenado_double: o número de elementos deve ser maior que
                 zero.\n");
263   if (n == 1) return true;
264   switch (o) {
265     case CRESCENTE:
266       for(i=1;i<n;i++)
267         if (v[i-1] > v[i]){
268           printf("valor V[%d] = %lf eh maior que V[%d] = %lf",i-1,v[i-1],i
                  ,v[i]);
269           return false;
270         }
271       break;
272     case DECRESCENTE:
273       for(i=1;i<n;i++)
274         if (v[i-1] < v[i]){
275           printf("valor V[%d] = %lf eh menor que V[%d] = %lf",i-1,v[i-1],i
                  ,v[i]);
276           return false;
277           }
278       break;
279     default: CONFIRME(false, "estaOrdenado_double: Ordem Inválida\n");
280   }
281   return true;
282 }
283
284 void imprime_vetor_int(int * v, int n){
285   int i;
286
287   for(i=0; i < n; i++)
288     printf("v[%d] = %d\n", i, v[i]);
289   printf("\n");
290 }
291
292 void imprime_vetor_double(double * v, int n){
293   int i;
294
295   for(i=0; i < n; i++)
296     printf("v[%d] = %lf\n", i, v[i]);
297   printf("\n");
298 }
```

Este arquivo serve para gerar os vetores e salva-los em arquivos.

Listagem 1.2: Geração dos vetores

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <math.h>
5 #include <sys/types.h>
```

```c
#include <sys/stat.h>
#include <unistd.h>

#include "vetor.h"

#define POT2(n) (1 << (n))


void gera_e_salva_vet(int n, Modificador m, Ordem o, Percentual p){
  int * v = NULL;
  char nome_do_arquivo[64];
  char sufixo[10];

  switch (o){
    case ALEATORIO:
      sprintf(nome_do_arquivo, "vIntAleatorio_%d", n);
      break;
    case CRESCENTE:
      sprintf(nome_do_arquivo, "vIntCrescente_%d", n);
      break;
    case DECRESCENTE:
      sprintf(nome_do_arquivo, "vIntDecrescente_%d", n);
      break;
    default: CONFIRME(false,
                      "gera_e_salva_vet: Ordenação desconhecida");
  }

  if (p > 0)
    sprintf(sufixo, "_P%2d.dat", p);
  else
    strcpy(sufixo, ".dat");

  v = gera_vetor_int(n, m, o, p, 1, n);
  strcat(nome_do_arquivo, sufixo);
  escreva_vetor_int(v, n, nome_do_arquivo);
  free(v);
}


int main(int argc, char *argv[]){
  int n = 0;
  int p = 0;
  char diretorio[256];

  struct stat st = {0};


  if (argc == 2)
    strcpy(diretorio, argv[1]);
  else
    strcpy(diretorio, "./vetores");

  if (stat(diretorio, &st) == -1) { // se o diretorio não existir,
    mkdir(diretorio, 0700);         // crie um
  }

  CONFIRME(chdir(diretorio) == 0, "Erro ao mudar de diretório");

  for(n = POT2(4); n <= POT2(14); n <<= 1){
```

```
65      gera_e_salva_vet(n, TOTALMENTE, ALEATORIO,    0);
66      gera_e_salva_vet(n, TOTALMENTE, CRESCENTE,    0);
67      gera_e_salva_vet(n, TOTALMENTE, DECRESCENTE, 0);
68
69      for(p=10; p <= 50; p += 10){
70        gera_e_salva_vet(n, PARCIALMENTE, CRESCENTE,    p);
71        gera_e_salva_vet(n, PARCIALMENTE, DECRESCENTE, p);
72      }
73      printf("Vetores para n = %d gerados.\n", n);
74    }
75
76    CONFIRME(chdir("..") == 0, "Erro ao mudar de diretório");
77
78    exit(0);
79  }
```

Este arquivo contém os algoritmos de ordenação pedidos.

Listagem 1.3: Métodos de ordenação

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "vetor.h"
4  #include <math.h>
5  void intercala(int * v,int p, int q, int r);
6  static void inline troca(int *A, int i, int j){
7    int temp;
8    temp = A[i];
9    A[i] = A[j];
10   A[j] = temp;
11 }
12
13 void ordena_por_bolha(int *A, int n){
14   int i,j;
15
16   if (n<2) return;
17
18   for(i=0; i<n; i++)
19     for(j=0; j<n-1; j++)
20       if (A[j] > A[j+1])
21         troca(A,j,j+1);
22 }
23
24
25 void ordena_por_shell(int *A, int n){
26   // Sequência de lacunas de Marcin Ciura
27   // Ref: https://en.wikipedia.org/wiki/Shellsort
28   int lacunas[] = {701, 301, 132, 57, 23, 10, 4, 1};
29   int *lacuna;
30   int i, j, temp;
31
32   for(lacuna=lacunas; *lacuna > 0; lacuna++){
33     for(i=*lacuna; i < n; i++){
34       // adicione A[i] aos elementos que foram ordenados
35       // guarde A[i] em temp e crie um espaço na posição i
36       temp = A[i];
37       // Desloque os elementos previamente ordenados até
38       // que a posição correta para A[i] seja encontrada
39       for(j=i; j >= *lacuna && A[j - *lacuna] > temp; j -= *lacuna){
```

```
40          A[j] = A[j - *lacuna];
41        }
42        // Coloque temp (o A[i] original) em sua posição correta
43        A[j] = temp;
44      }
45    }
46 }
47
48
49 void ordena_intercala(int * v,int p,int r)
50 {
51     int q;
52     if (p < r) {
53         q = (p + r) / 2; // retorna o chão dessa operação
54         ordena_intercala (v, p, q);
55         ordena_intercala(v, q + 1, r);
56         intercala(v, p, q, r);
57     }
58 }
59
60
61 void intercala(int *V, int p, int q, int r) {
62   int inicio1 = p ;
63   int inicio2 = q+1 ;
64   int aux = 0;
65   int B[r-p+1];
66
67   while(inicio1<=q && inicio2<=r){
68     if(V[inicio2] >= V[inicio1]){
69       B[aux] = V[inicio1];
70       inicio1++;
71     }else{
72       B[aux] = V[inicio2];
73       inicio2++;
74     }
75     aux++;
76   }
77
78   while(inicio1<=q){
79     B[aux] = V[inicio1];
80     aux++;
81   inicio1++;
82   }
83
84   while(inicio2<=r){
85     B[aux] = V[inicio2];
86     aux++;
87   inicio2++;
88   }
89
90   for(aux=p;aux<=r;aux++)
91     V[aux] = B[aux-p];
92 }
93
94
95 void insertion(int *v, int tam)
96 {
97     int chave,i,j;
98     for(j=1;j<tam;j++)
```

```
99      {
100         chave = v[j];
101         i = j - 1;
102         while (i >= 0 && v[i] > chave)
103         {
104             v[i+1] = v[i];
105             i = i-1;
106         }
107   v[i+1] = chave;
108     }
109 }
110 
111 void heap(int *a, int n) {
112     int i = n / 2, pai, filho, t;
113     for (;;) {
114         if (i > 0) {
115             i--;
116             t = a[i];
117         } else {
118             n--;
119             if (n == 0) return;
120             t = a[n];
121             a[n] = a[0];
122         }
123         pai = i;
124         filho = i * 2 + 1;
125         while (filho < n) {
126             if ((filho + 1 < n)  &&  (a[filho + 1] > a[filho]))
127                 filho++;
128             if (a[filho] > t) {
129                 a[pai] = a[filho];
130                 pai = filho;
131                 filho = pai * 2 + 1;
132             } else {
133                 break;
134             }
135         }
136         a[pai] = t;
137     }
138 }
139 
140 void quick(int *vetor, int inicio, int fim){
141 
142     int pivo, aux, i, j, meio;
143 
144     i = inicio;
145     j = fim;
146 
147     meio = (int) ((i + j) / 2);
148     pivo = vetor[meio];
149 
150     do{
151         while (vetor[i] < pivo) i = i + 1;
152         while (vetor[j] > pivo) j = j - 1;
153 
154         if(i <= j){
155             aux = vetor[i];
156             vetor[i] = vetor[j];
157             vetor[j] = aux;
```

```
158             i = i + 1;
159             j = j - 1;
160         }
161     }while(j > i);
162 
163     if(inicio < j) quick(vetor, inicio, j);
164     if(i < fim) quick(vetor, i, fim);
165 }
166 
167 void coutingsort(int *A, int tamanho){
168     int k = 10;
169     int aux;
170     int *C = (int*)calloc(k+1,sizeof(int));
171     int *B = (int*)malloc(tamanho*sizeof(int));
172 
173     for(int j = 0;j<tamanho;j++){
174         C[A[j]]++;
175     }
176     for(int i=1;i<=k;i++){
177         C[i] = C[i] + C[i-1];
178     }
179     for(int j=0;j<tamanho;j++){
180         B[C[A[j]]-1] = A[j];
181         C[A[j]]--;
182     }
183     for(int i=0;i<tamanho;i++){
184         A[i] = B[i];
185     }
186 }
187 
188 int pegaMax(int *arr, int n) //pegar o maior valor no array;
189 {
190     int mx = arr[0];
191     for (int i = 1; i < n; i++)
192         if (arr[i] > mx)
193             mx = arr[i];
194     return mx;
195 }
196 
197 void couting_radix(int *A, int tamanho, int exp){ //couting adaptado para
        ir de digito a digito
198     int k = tamanho;
199     int aux;
200     int *C = (int*)calloc(k+1,sizeof(int));
201     int *B = (int*)malloc(tamanho*sizeof(int));
202 
203     for(int j = 0;j<tamanho;j++){
204         C[(A[j]/exp)%10]++;
205     }
206     for(int i=1;i<=k;i++){
207         C[i] = C[i] + C[i-1];
208     }
209     for(int j=tamanho-1;j>=0;j--){
210         B[C[ (A[j]/exp)%10]-1] = A[j];
211         C[(A[j]/exp)%10]--;
212     }
213     for(int i=0;i<tamanho;i++){
214         A[i] = B[i];
215     }
```

```
216 }
217
218 void radixsort(int *arr, int n)
219 {
220     // Encontrar o nro máximo nos valores
221     int m = pegaMax(arr, n);
222     //For do radix ir digito a digito
223     for (int exp = 1; m/exp > 0; exp *= 10)
224         couting_radix(arr, n, exp);
225 }
226
227 void insertiondouble(double *v, int tam)
228 {
229     int i,j;
230     double chave;
231     for(j=1;j<tam;j++)
232     {
233         chave = v[j];
234         i = j - 1;
235         while (i >= 0 && v[i] > chave)
236         {
237             v[i+1] = v[i];
238             i = i-1;
239         }
240   v[i+1] = chave;
241     }
242 }
243
244 void bucketsort(double *A,int tamanho){
245     bucket *C = (bucket*)malloc(10*sizeof(bucket));
246     int j,i;
247     for(int i=0;i<10;i++){ //Inicialização dos topos dos baldes
248         C[i].topo = 0.0;
249         C[i].balde = (double*)malloc((int)(tamanho)*sizeof(double));
250     }
251     for(i = 0;i<tamanho;i++){ //Verifica em que balde o elem deve ficar
252         j = 10-1;
253         while(1){
254             if(j<0){
255                 break;
256             }
257             if(A[i]>=j*10){
258                 C[j].balde[C[j].topo] = A[i];
259                 (C[j].topo)++;
260                 break;
261             }
262             j--;
263         }
264     }
265     for(i=0;i<10;i++){ //ordena os baldes
266         if(C[i].topo){
267             insertiondouble(C[i].balde,C[i].topo);
268         }
269     }
270     i=0;
271     for(j=0;j<10;j++){ //coloca os elementos dos baldes de volta no vetor
272         for(int k=0;k<C[j].topo;k++){
273             A[i]=C[j].balde[k];
274             i++;
```

```
275         }
276     }
277     for(i=0;i<10;i++){
278         free(C[i].balde);
279     }
280     free(C);
281 }
```

O arquivo ensaios.c serve para automatizar e calcular os tempos de cada método de ordenação.

Listagem 1.4: Automatização dos experimentos

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdint.h>
5  #include <time.h>
6  #include <float.h>
7  #include <math.h>
8  #include <sys/types.h>
9  #include <sys/stat.h>
10 #include <unistd.h>
11
12 #include "vetor.h"
13 #include "ordena.h"
14
15 #define BILHAO 1000000000L
16
17 #define CRONOMETRA(funcao,vetor,n) {                              \
18    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &inicio);              \
19    funcao(vetor,0,n-1);                                               \
20    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &fim);                 \
21    tempo_de_cpu_aux = BILHAO * (fim.tv_sec - inicio.tv_sec) +     \
22                    fim.tv_nsec - inicio.tv_nsec;                  \
23    }
24
25 int main(int argc, char *argv[]){
26   int * v = NULL;
27   int n = 0;
28   uint64_t tempo_de_cpu_aux = 0;
29   int tamanho = 0,count = 0;
30   //clock_t inicio, fim;
31   struct timespec inicio, fim;
32   uint64_t tempo_de_cpu = 0.0;
33   char msg[256];
34   char nome_do_arquivo[128];
35   char **arquivos;
36   int k=0,h = 0;
37   arquivos = (char**)malloc(200*sizeof(char*));
38   for(int i=0;i<200;i++){
39     arquivos[i] = (char*)malloc(128*sizeof(char));
40   }
41
42   for(int i=0;i<11;i++){
43     sprintf(nome_do_arquivo,"vetores/vIntAleatorio_%d.dat",(int)pow(2,i
         +4%15));
44     strcpy(arquivos[k], nome_do_arquivo);
45     k++;
```

```
46    }
47    for(int i=0;i<11;i++){
48      sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d.dat",(int)pow(2,i
          +4%15));
49      strcpy(arquivos[k], nome_do_arquivo);
50      k++;
51
52    }
53    for(int i=0;i<11;i++){
54      sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d_P10.dat",(int)pow(2,
          i+4%15));
55      strcpy(arquivos[k], nome_do_arquivo);
56      k++;
57    }
58    for(int i=0;i<11;i++){
59      sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d_P20.dat",(int)pow(2,
          i+4%15));
60      strcpy(arquivos[k], nome_do_arquivo);
61      k++;
62    }
63    for(int i=0;i<11;i++){
64      sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d_P30.dat",(int)pow(2,
          i+4%15));
65      strcpy(arquivos[k], nome_do_arquivo);
66      k++;
67    }
68    for(int i=0;i<11;i++){
69      sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d_P40.dat",(int)pow(2,
          i+4%15));
70      strcpy(arquivos[k], nome_do_arquivo);
71      k++;
72    }
73    for(int i=0;i<11;i++){
74      sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d_P50.dat",(int)pow(2,
          i+4%15));
75      strcpy(arquivos[k], nome_do_arquivo);
76      k++;
77    }
78    for(int i=0;i<11;i++){
79      sprintf(nome_do_arquivo,"vetores/vIntDecrescente_%d.dat",(int)pow(2,i
          +4%15));
80      strcpy(arquivos[k], nome_do_arquivo);
81      k++;
82    }
83    for(int i=0;i<11;i++){
84      sprintf(nome_do_arquivo,"vetores/vIntDecrescente_%d_P10.dat",(int)pow
          (2,i+4%15));
85      strcpy(arquivos[k], nome_do_arquivo);
86      k++;
87    }
88    for(int i=0;i<11;i++){
89      sprintf(nome_do_arquivo,"vetores/vIntDecrescente_%d_P20.dat",(int)pow
          (2,i+4%15));
90      strcpy(arquivos[k], nome_do_arquivo);
91      k++;
92    }
93
94    for(int i=0;i<11;i++){
95        sprintf(nome_do_arquivo,"vetores/vIntDecrescente_%d_P30.dat",(int)
```

```
          pow(2,i+4%15));
96    strcpy(arquivos[k], nome_do_arquivo);
97    k++;
98  }
99  for(int i=0;i<11;i++){
100   sprintf(nome_do_arquivo,"vetores/vIntDecrescente_%d_P40.dat",(int)pow
          (2,i+4%15));
101   strcpy(arquivos[k], nome_do_arquivo);
102   k++;
103 }
104 for(int i=0;i<11;i++){
105   sprintf(nome_do_arquivo,"vetores/vIntDecrescente_%d_P50.dat",(int)pow
          (2,i+4%15));
106   strcpy(arquivos[k], nome_do_arquivo);
107   k++;
108 }
109   printf("%d\n",k);
110 //strcpy(nome_do_arquivo, "vetores/vIntCrescente_131072.dat");
111 // Leia o vetor a partir do arquivo
112 //v = leia_vetor_int(nome_do_arquivo, &n);
113 printf("%s\n",arquivos[11]);
114 for(int i=0;i<k;i++){
115   tempo_de_cpu = 0.0;
116   if(h > 10){
117       h = 0;
118   }
119   for(int j=0;j<3;j++){
120       v = leia_vetor_int(arquivos[i],&n);
121       tamanho = (int)pow(2,h+4%15);
122       /*inicio = clock();
123       //ordena_por_bolha(v,n);
124       insertion(v,tamanho);
125       fim = clock();*/
126 CRONOMETRA(ordena_intercala, v,tamanho);
127       //tempo_de_cpu += ((double) (fim - inicio)) / CLOCKS_PER_SEC;
128 tempo_de_cpu += tempo_de_cpu_aux;
129   }
130   if(esta_ordenado_int(CRESCENTE,v,n) && count < 11){
131       printf("Tempo do vetor aleatorio tamanho %d: %llu\n",tamanho,(long
            long unsigned int)tempo_de_cpu/(uint64_t) 3);
132   }
133   else if(esta_ordenado_int(CRESCENTE,v,n) && count < 22){
134       printf("Tempo do vetor Crescente tamanho %d: %llu\n",tamanho,(long
            long unsigned int)tempo_de_cpu/(uint64_t) 3);
135   }
136   else if(esta_ordenado_int(CRESCENTE,v,n) && count < 33){
137       printf("Tempo do vetor Crescente P10 tamanho %d: %llu\n",tamanho,(
          long long unsigned int)tempo_de_cpu/(uint64_t) 3);
138   }
139   else if(esta_ordenado_int(CRESCENTE,v,n) && count < 44){
140       printf("Tempo do vetor Crescente P20 tamanho %d: %llu\n",tamanho,(
          long long unsigned int)tempo_de_cpu/(uint64_t) 3);
141   }
142   else if(esta_ordenado_int(CRESCENTE,v,n) && count < 55){
143       printf("Tempo do vetor Crescente P30 tamanho %d: %llu\n",tamanho,(
          long long unsigned int)tempo_de_cpu/(uint64_t) 3);
144   }
145   else if(esta_ordenado_int(CRESCENTE,v,n) && count < 66){
146       printf("Tempo do vetor Crescente P40 tamanho %d: %llu\n",tamanho,(
```

```
            long long unsigned int)tempo_de_cpu/(uint64_t) 3);
147     }
148     else if(esta_ordenado_int(CRESCENTE,v,n) && count < 77){
149         printf("Tempo do vetor Crescente P50 tamanho %d: %llu\n",tamanho,(
                long long unsigned int)tempo_de_cpu/(uint64_t) 3);
150     }
151     else if(esta_ordenado_int(CRESCENTE,v,n) && count < 88){
152         printf("Tempo do vetor Decrescente tamanho %d: %llu\n",tamanho,(
                long long unsigned int)tempo_de_cpu/(uint64_t) 3);
153     }
154     else if(esta_ordenado_int(CRESCENTE,v,n) && count < 99){
155         printf("Tempo do vetor Decrescente P10 tamanho %d: %llu\n",tamanho
                ,(long long unsigned int)tempo_de_cpu/(uint64_t) 3);
156     }
157     else if(esta_ordenado_int(CRESCENTE,v,n) && count < 110){
158         printf("Tempo do vetor Decrescente P20 tamanho %d: %llu\n",tamanho
                ,(long long unsigned int)tempo_de_cpu/(uint64_t) 3);
159     }
160     else if(esta_ordenado_int(CRESCENTE,v,n) && count < 121){
161         printf("Tempo do vetor Decrescente P30 tamanho %d: %llu\n",tamanho
                ,(long long unsigned int)tempo_de_cpu/(uint64_t) 3);
162     }
163     else if(esta_ordenado_int(CRESCENTE,v,n) && count < 132){
164         printf("Tempo do vetor Decrescente P40 tamanho %d: %llu\n",tamanho
                ,(long long unsigned int)tempo_de_cpu/(uint64_t) 3);
165     }
166     else if(esta_ordenado_int(CRESCENTE,v,n) && count < 143){
167         printf("Tempo do vetor Decrescente P50 tamanho %d: %llu\n",tamanho
                ,(long long unsigned int)tempo_de_cpu/(uint64_t) 3);
168     }
169     else{
170         printf("Erro em ordenção do vetor %d, arquivo %s\n",i,arquivos[i])
                ;
171     }
172     h++;
173     count++;
174   }
175   //imprime_vetor_int(v,16384);
176   free(v);
177   exit(0);
178 }
```

1.1.1 Comandos

Os seguintes passos devem ser seguidos para criação dos vetores que serão utilizados no experimento:
1 - Compilar o arquivo vetor.c;

```
> gcc -O3 -c vetor.c
```

2 - Compilar o programa que gera os vetores e os coloca no diretório determinado;

```
> gcc -O3 vetor.o gera_vets.c -o gera_vets.exe
```

3 - Para usá-lo digite

```
> ./gera_vets.exe
```

Os passos a seguir são para execução do experimento
1 - Verifique a existência do diretório contendo os vetores, e então digite o seguinte comando:

```
> gcc -O3 -c ordena.c
```

2 - Agora é necessário compilar o arquivo de ensaio e tudo que será utilizado

```
> gcc -O3 vetor.o ordena.o ensaios.c -o ensaios.exe -lm
```

3 - Para executar digite:

```
> ./ensaios.exe
```

1.2 Máquina de teste

Todos os testes foram realizados na mesma máquina com as seguintes configurações, e usando apenas um núcleo:
AMD FX-8350 4.0GHZ
16GB Memória DDR3-1600
HDD 2TB 7200RPM
Placa de video Nvidia GTX1050Ti
Sistema Operacional: Ubuntu 16.04

Capítulo 2

Grafo

balbalbalbal

2.1 Busca Largura

É um algoritmo de busca em grafos utilizado para realizar uma busca ou travessia num grafo e estrutura de dados do tipo árvore. Você começa pelo vértice raiz e explora todos os vértices vizinhos. Então, para cada um desses vértices mais próximos, exploramos os seus vértices vizinhos inexplorados e assim por diante, até que ele encontre o alvo da busca.

2.2 Busca Largura - Grafo Esparso

Tabela gerada utilizando Busca em largura com grafo esparso de tamanho n, sendo = (2^k), k = 4...14.

Tabela 2.1: *Busca Largura com grafo Esparso*

Número de Elementos	**Tempo de execução em nanosegundos**
128	7943
256	7472
512	15414
1024	32102

2.2.1 Gráfico Busca Largura - Grafo Esparso

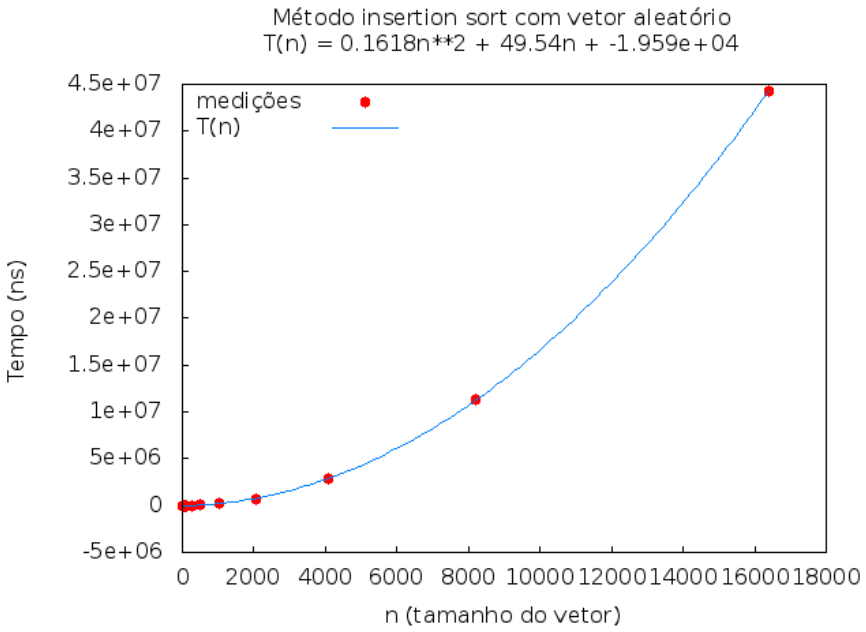

Figura 2.1: *Busca Largura - Grafo Esparso*

2.3 Busca Largura - Grafo Denso

Tabela gerada utilizando Busca em Largura num grafo Denso com n, sendo = (2^k), k = 4...14.

Tabela 2.2: *Busca em Largura Grafo Denso*

Número de Elementos	Tempo de execução em nanosegundos
128	67546
256	318565
512	1216634
1024	4527417

2.3.1 Busca em Largura - Grafo Denso

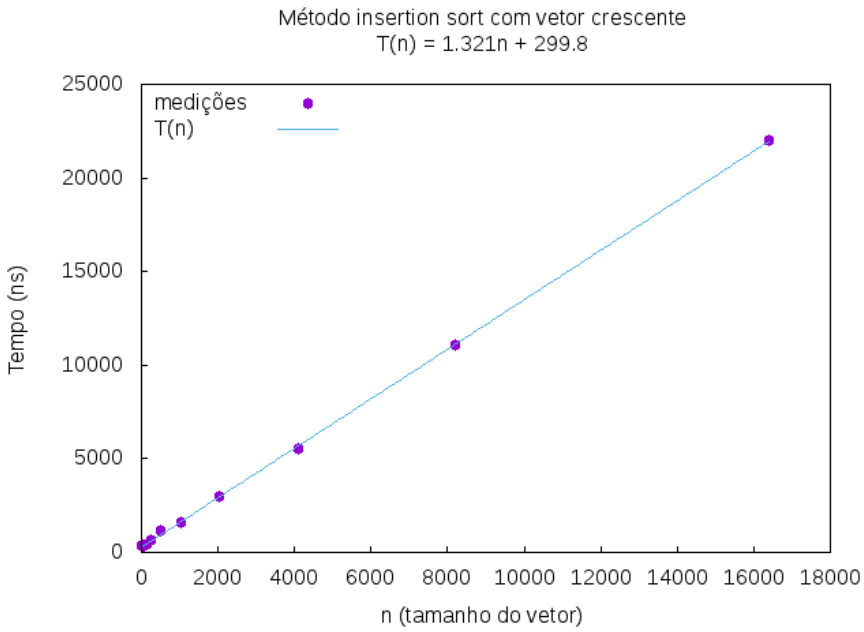

Figura 2.2: *Busca em Largura - Grafo Denso*

2.4 Busca Profundidade

É um algoritmo usado para realizar uma busca ou travessia numa árvore, estrutura de árvore ou grafo. O algoritmo começa num nó raiz (selecionando algum nó como sendo o raiz, no caso de um grafo) e explora tanto quanto possível cada um dos seus ramos, antes de retroceder(backtracking).

2.5 Busca Profundidade - Grafo Esparso

Tabela gerada utilizando Busca Profundidade com um Grafo Esparso, sendo n = (2^k), de k = 4..14.

Tabela 2.3: *Busca Profundidade com Grafo Esparso*

Número de Elementos	Tempo de execução em nanosegundos
128	6619
256	7975
512	21265
1024	36223

2.5.1 Busca Profundidade - Grafo Esparso

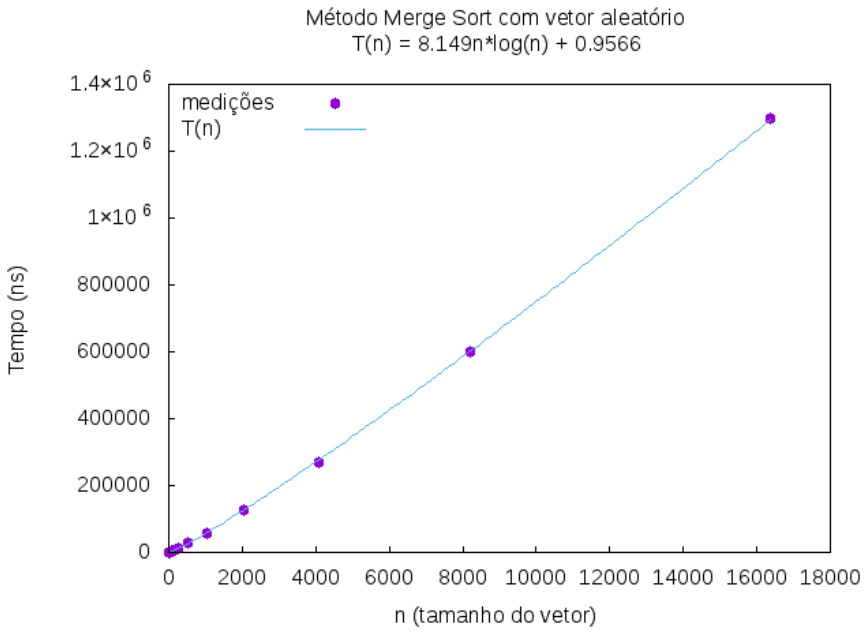

Figura 2.3: *Busca Profundidade - Grafo Esparso*

2.6 Busca Profundidade - Grafo Denso

Tabela gerada utilizando Busca em Profundidade com um Grafo Denso, sendo n = (2^k), de k = 4..14.

Tabela 2.4: *Busca Profundidade em um Grafo Denso*

Número de Elementos	Tempo de execução em nanosegundos
128	72161
256	330916
512	1380078
1024	4735134

2.6.1 Busca Profundidade - Grafo Denso

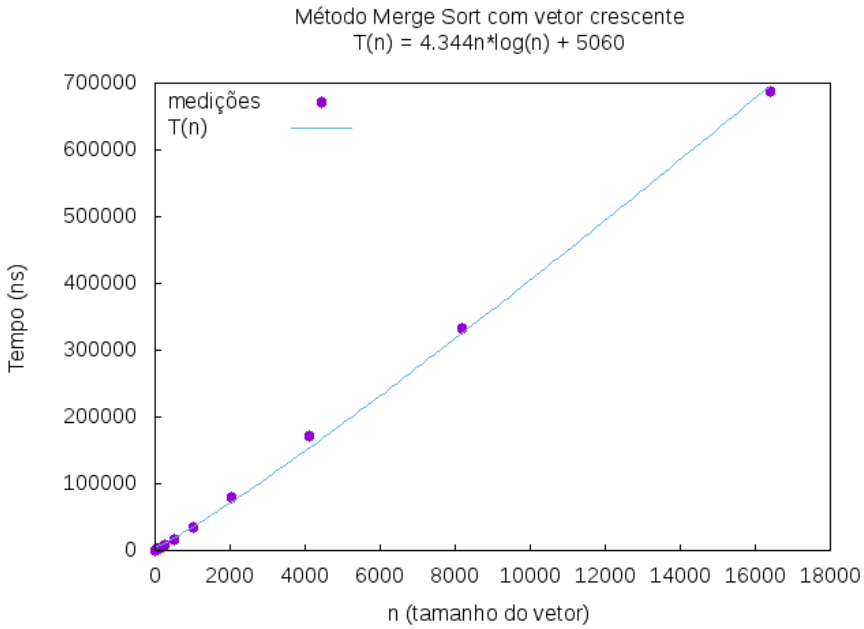

Figura 2.4: *Busca Profundidade - Grafo Denso*

2.7 Ordenação Topologica

blabalblalbalbalb

2.8 Ordenação Topologica - Grafo Esparso

Tabela gerada utilizando Ordenação Topologica com um Grafo Esparso, sendo n = (2^k), de k = 4..14.

Tabela 2.5: *Ordenação Topologica com Grafo Esparso*

Número de Elementos	Tempo de execução em nanosegundos
128	13549
256	15347
512	30745
1024	57293

2.8.1 Ordenação Topologica - Grafo Esparso

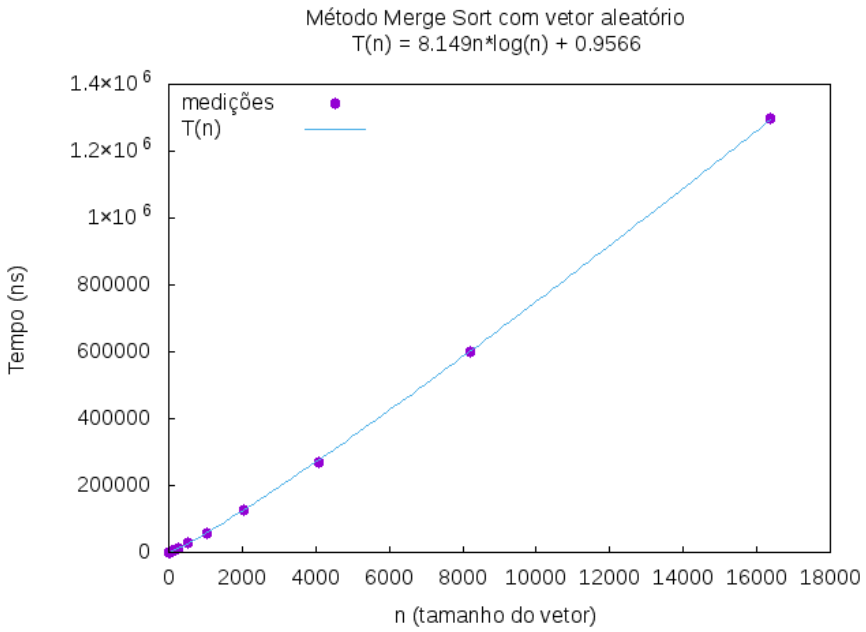

Figura 2.5: *Ordenação Topologica - Grafo Esparso*

2.9 Ordenação Topologica - Grafo Denso

Tabela gerada utilizando Ordenação Topologica com um Grafo Denso, sendo n = (2^k), de k = 4..14.

Tabela 2.6: *Ordenação Topologica com Grafo Denso*

Número de Elementos	Tempo de execução em nanosegundos
128	93864
256	349061
512	1344655
1024	4865168

2.9.1 Ordenação Topologica - Grafo Denso

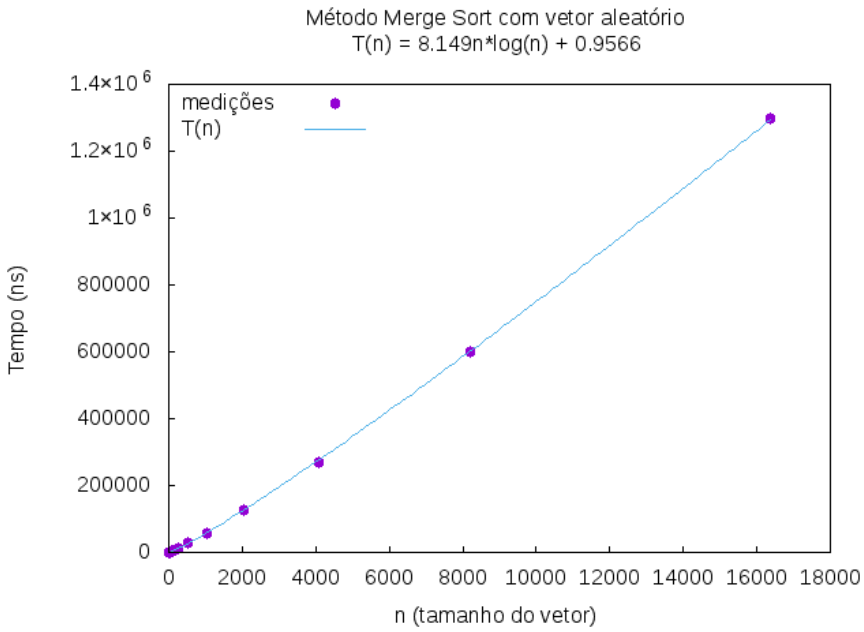

Figura 2.6: *Ordenação Topologica - Grafo Denso*

Capítulo 3

Guloso

blablabla

3.1 Huffman

blabalbal Vetor de string aleatorias com frequencias aleatorias Tempo (Onlgn)

3.1.1 Vetor aleatorio

Tabela gerada utilizando Huffman com vetor de string aleatorias com frequencias em tempo O(nlogn) de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente.

Tabela 3.1: *Huffman com vetor aleatório*

Número de Elementos	**Tempo de execução em nanosegundos**
16	4599
32	8150
64	22736
128	42886
256	95255
512	170503
1024	438873
2048	1031620
4096	2592036
8192	5381493
16384	10506005

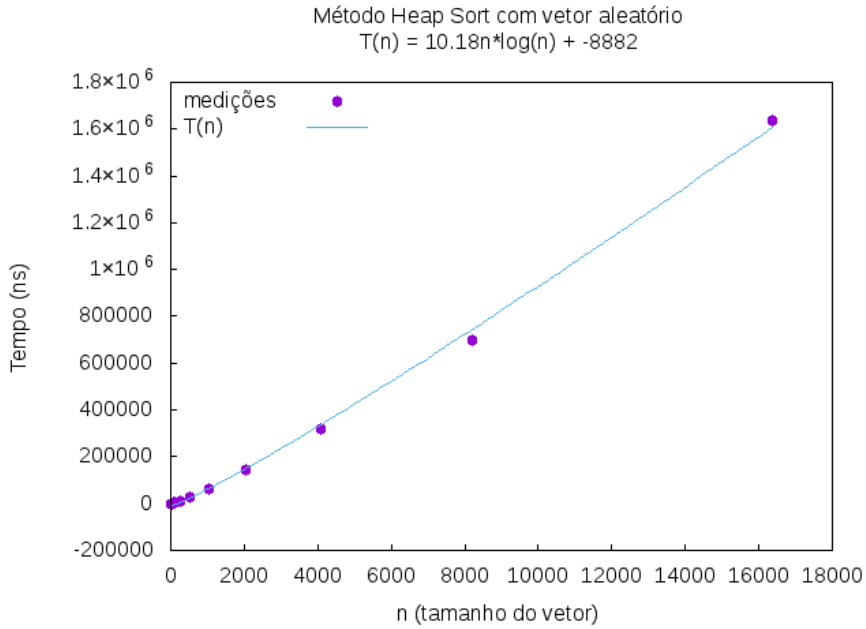

Figura 3.1: *Huffman - Vetor Aleatório*

3.2 Seleção de Atividade Interativo

blabalbal

3.2.1 Vetor crescente

Tabela gerada utilizando Seleção de Atividade Interativo com vetor de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos crescente.

Tabela 3.2: *Seleção de Atividade Interativo com vetor crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	615
32	680
64	769
128	802
256	764
512	836
1024	1058
2048	1294
4096	1685
8192	2594
16384	4578

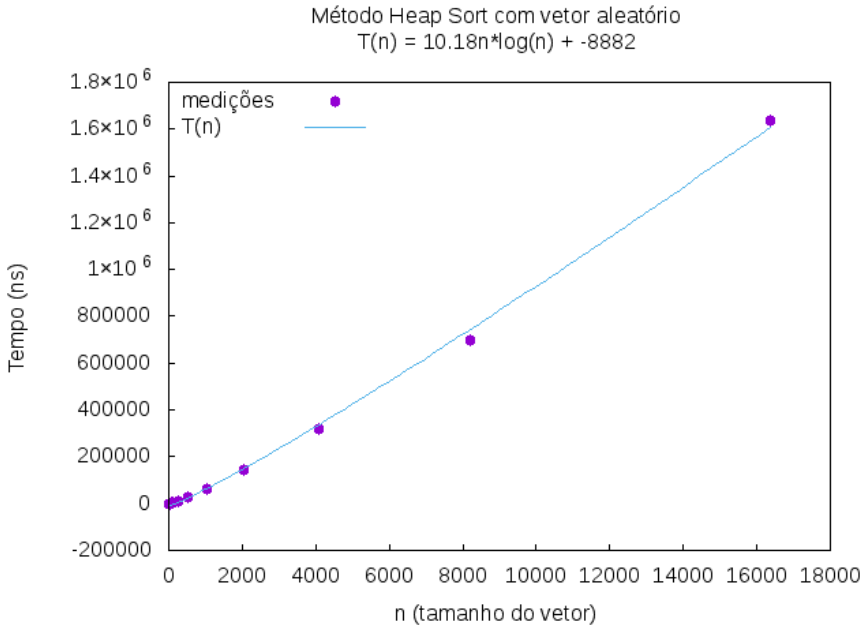

Figura 3.2: *Seleção de Atividade Interativo - Vetor crescente*

3.3 Seleção de Atividade Bottom Up

blabalbal

3.3.1 Vetor crescente

Tabela gerada utilizando Seleção de Atividade Bottom Up com vetor de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos crescente.

Tabela 3.3: *Seleção de Atividade Bottom Up com vetor crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	615
32	680
64	769
128	802
256	764
512	836
1024	1058
2048	1294
4096	1685
8192	2594
16384	4578

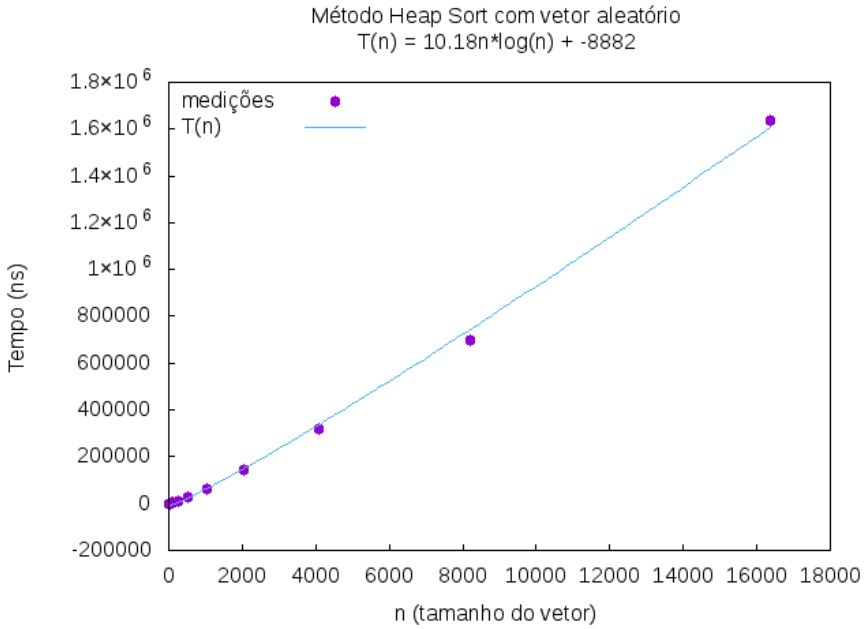

Figura 3.3: *Seleção de Atividade Bottom Up - Vetor crescente*

Capítulo 4

Programação Dinâmica

Programação dinamica blablabla

4.1 Corte Haste

Falar sobre corte de hastes comum +dados falar o que deu errado. Será resolvido utilizando programação dinâmica.

4.2 Corte Haste Bottom Up

falar um pouco sobre bottomup (n muito).

4.2.1 Vetor aleatorio

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente.

Tabela 4.1: *Corte Haste Bottom Up com vetor aleatório*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

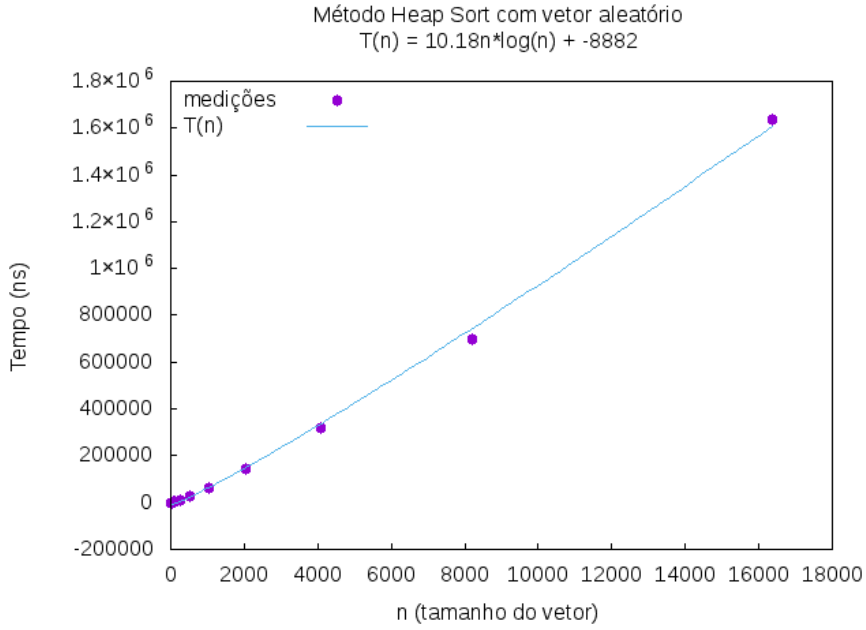

Figura 4.1: *Corte Haste Bottom Up - Vetor Aleatório*

4.2.2 Vetor Crescente

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente.

Tabela 4.2: *Corte Haste Bottom Up com Vetor Crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

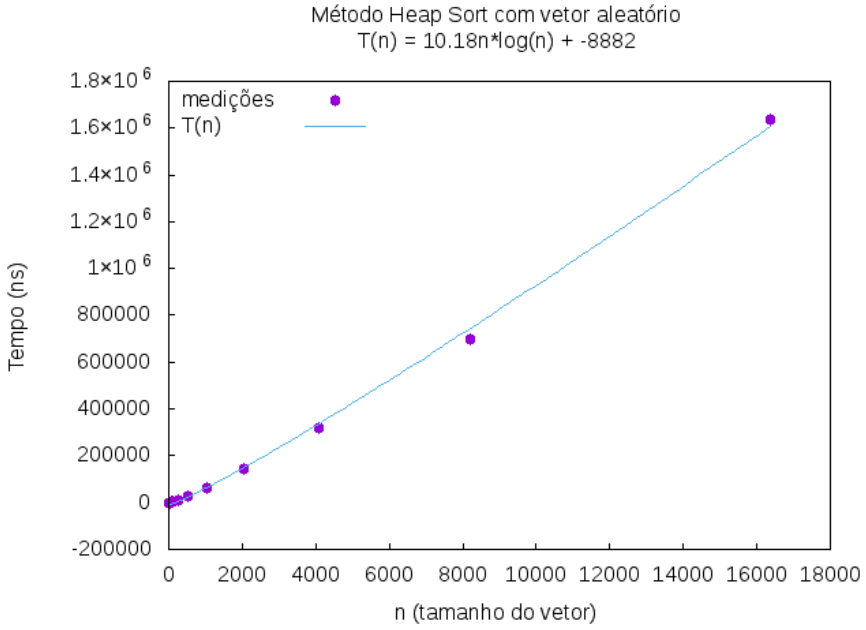

Figura 4.2: *Corte Haste Bottom Up - Vetor Crescente*

4.2.3 Vetor Crescente P10

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P10.

Tabela 4.3: *Corte Haste Bottom Up com Vetor Crescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

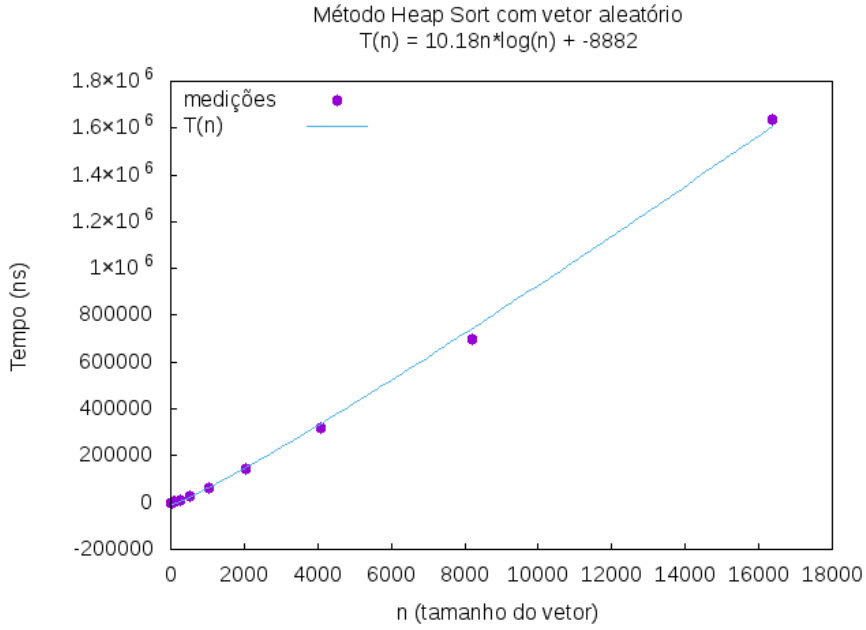

Figura 4.3: *Corte Haste Bottom Up - Vetor Crescente P10*

4.2.4 Vetor Crescente P20

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P20.

Tabela 4.4: *Corte Haste Bottom Up com Vetor Crescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

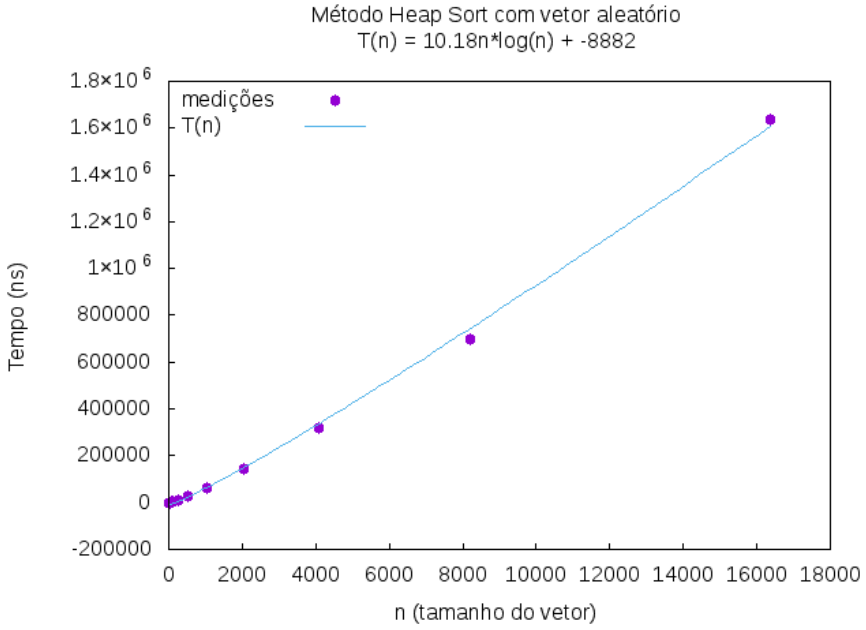

Figura 4.4: *Corte Haste Bottom Up - Vetor Crescente P20*

4.2.5 Vetor Crescente P30

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P30.

Tabela 4.5: *Corte Haste Bottom Up com Vetor Crescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

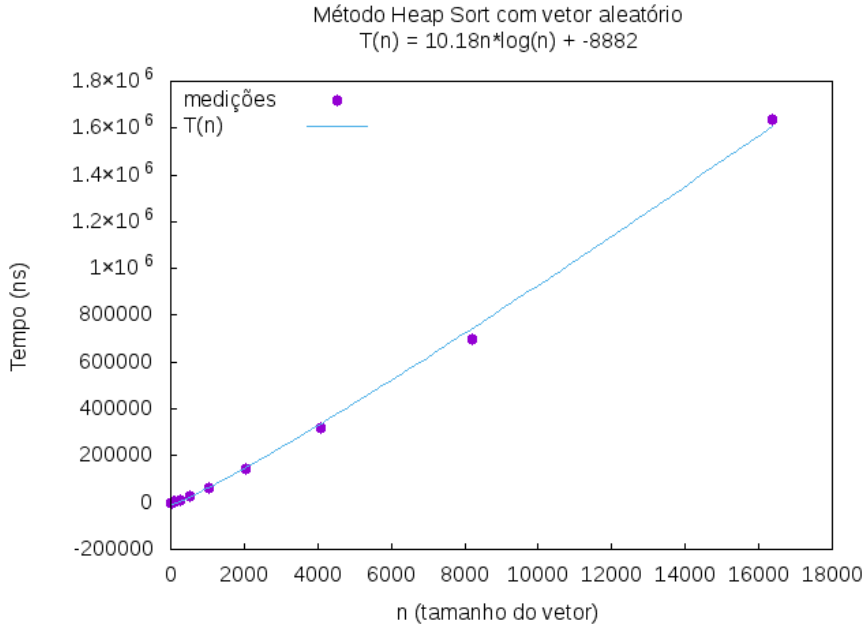

Figura 4.5: *Corte Haste Bottom Up - Vetor Crescente P30*

4.2.6 Vetor Crescente P40

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P40.

Tabela 4.6: *Corte Haste Bottom Up com Vetor Crescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

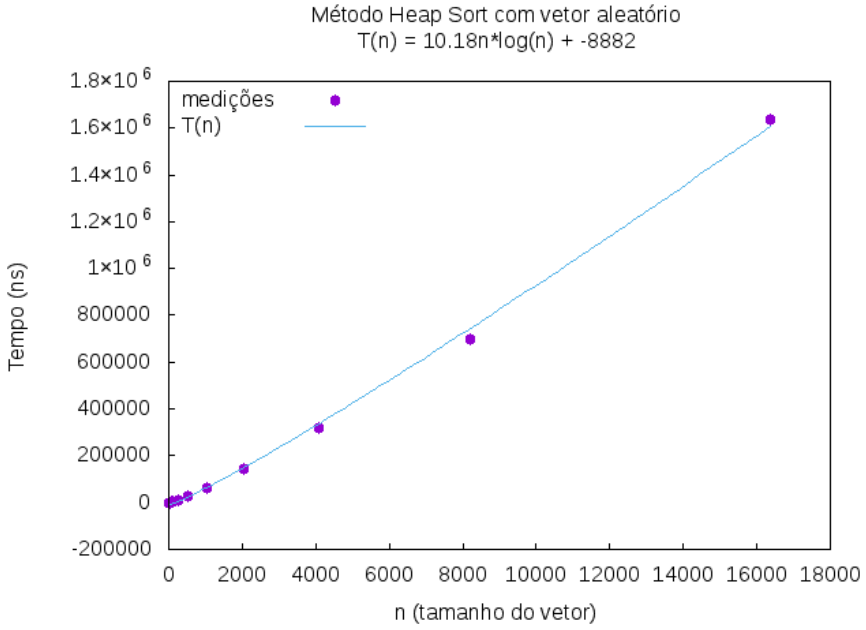

Figura 4.6: *Corte Haste Bottom Up - Vetor Crescente P40*

4.2.7 Vetor Crescente P50

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P50.

Tabela 4.7: *Corte Haste Bottom Up com Vetor Crescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

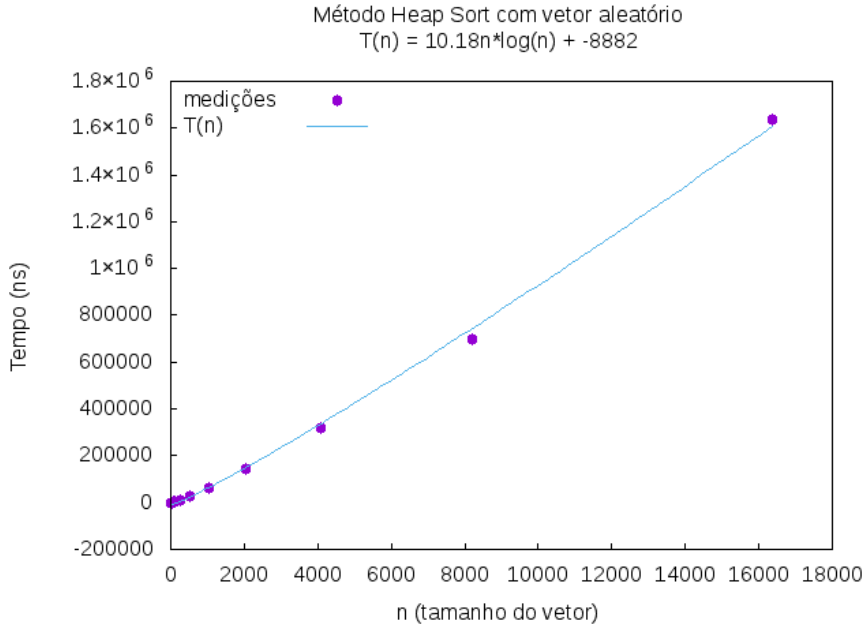

Figura 4.7: *Corte Haste Bottom Up - Vetor Crescente P50*

4.2.8 Vetor Decrescente

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente.

Tabela 4.8: *Corte Haste Bottom Up com Vetor Decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

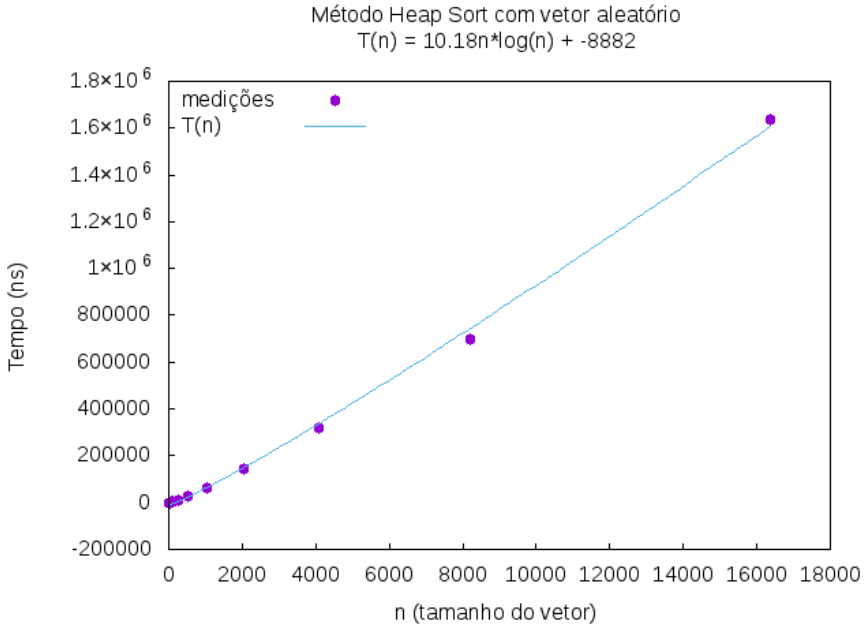

Figura 4.8: *Corte Haste Bottom Up - Vetor Decrescente*

4.2.9 Vetor Decrescente P10

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P10.

Tabela 4.9: *Corte Haste Bottom Up com Vetor Decrescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

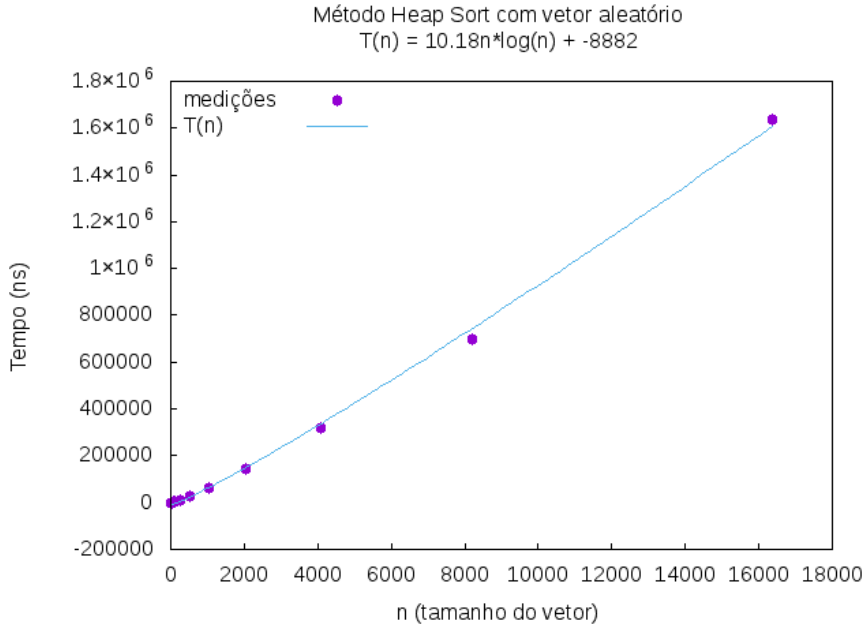

Figura 4.9: *Corte Haste Bottom Up - Vetor Decrescente P10*

4.2.10 Vetor Decrescente P20

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P20.

Tabela 4.10: *Corte Haste Bottom Up com Vetor Decrescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

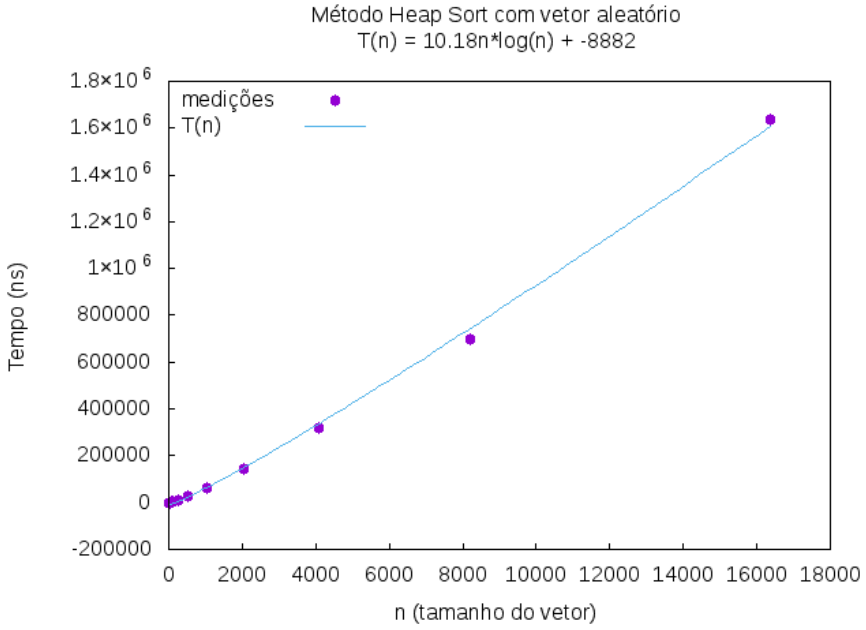

Figura 4.10: *Corte Haste Bottom Up - Vetor Decrescente P20*

4.2.11 Vetor Decrescente P30

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P30.

Tabela 4.11: *Corte Haste Bottom Up com Vetor Decrescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

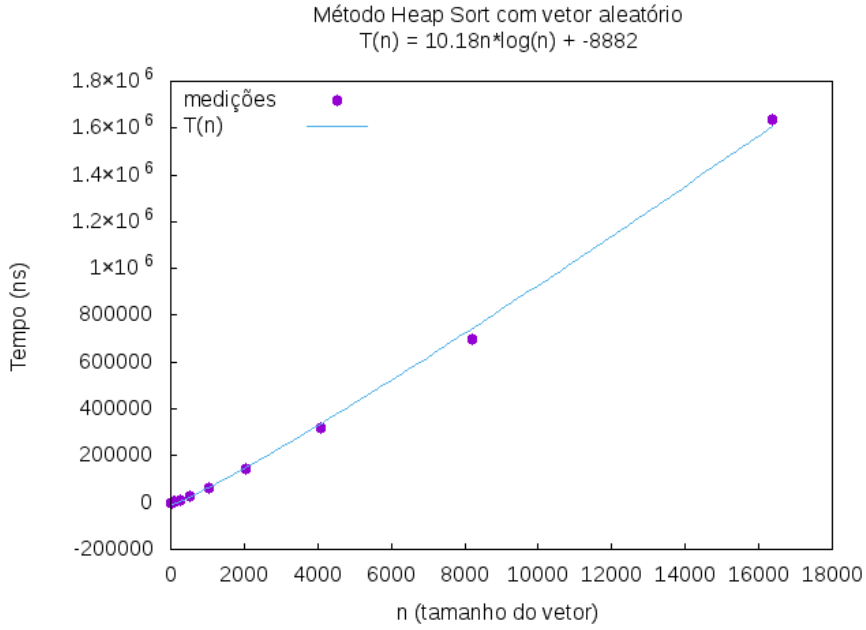

Figura 4.11: *Corte Haste Bottom Up - Vetor Decrescente P30*

4.2.12 Vetor Decrescente P40

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P40.

Tabela 4.12: *Corte Haste Bottom Up com Vetor Decrescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

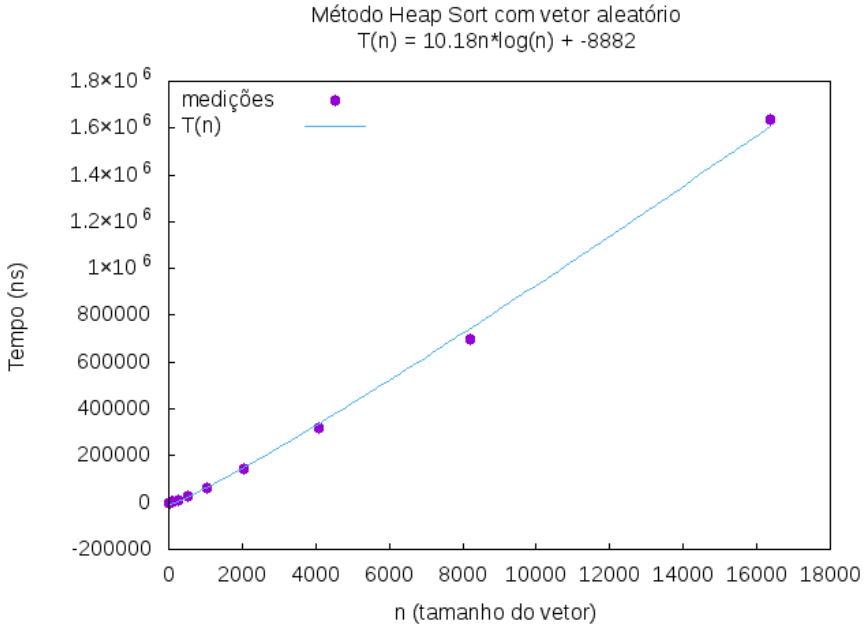

Figura 4.12: *Corte Haste Bottom Up - Vetor Decrescente P40*

4.2.13 Vetor Decrescente P50

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P50.

Tabela 4.13: *Corte Haste Bottom Up com Vetor Decrescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

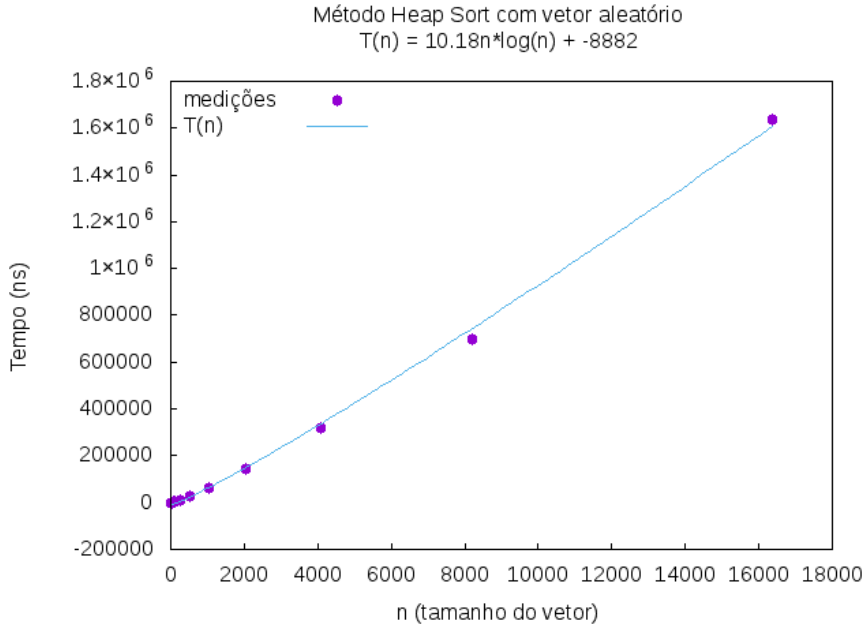

Figura 4.13: *Corte Haste Bottom Up - Vetor Decrescente P50*

4.3 Corte Haste Comum

blablabal

Tabela gerada utilizando Corte Haste Comum com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente, crescente, crescente P10, crescente P20, crescente P30, decrescente, decrescente P10, decrescente P20, decrescenteP30.

Tabela 4.14: *Corte Haste Bottom Up com vetor aleatório*

Número de Elementos	Tempo de execução em nanosegundos
16	535410
16	538372
16	539477
16	529753
16	541344
16	538372
16	539477
16	529753
16	541344

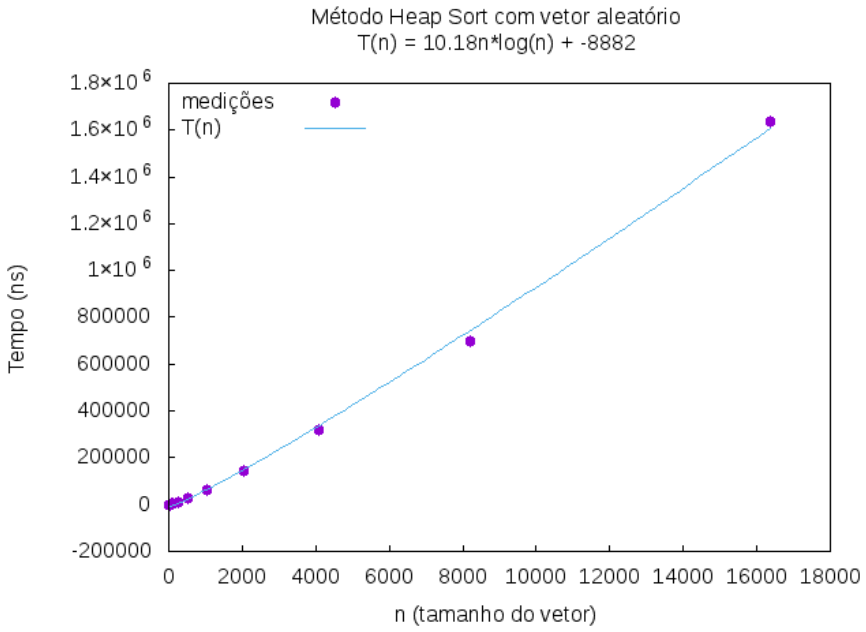

Figura 4.14: *Corte Haste Comum*

4.4 Corte Haste Memoizada

blablbablabla

4.4.1 Vetor aleatorio

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente.

Tabela 4.15: *Corte Haste Memoizada com vetor aleatório*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	6253

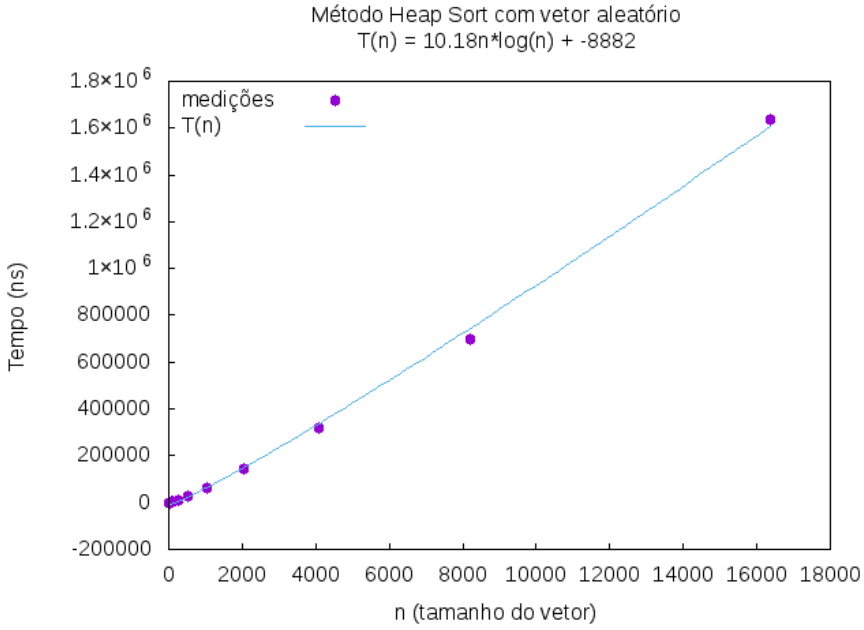

Figura 4.15: *Corte Haste Memoizada - Vetor Aleatório*

4.4.2 Vetor Crescente

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente.

Tabela 4.16: *Corte Haste Memoizada com Vetor Crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	6253

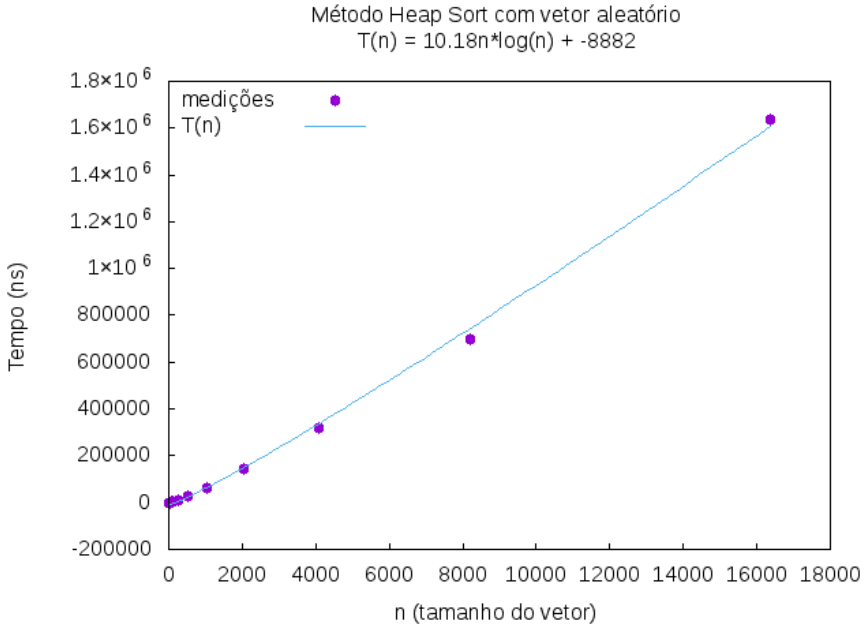

Figura 4.16: *Corte Haste Memoizada - Vetor Crescente*

4.4.3 Vetor Crescente P10

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P10.

Tabela 4.17: *Corte Haste Memoizada com Vetor Crescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	6253

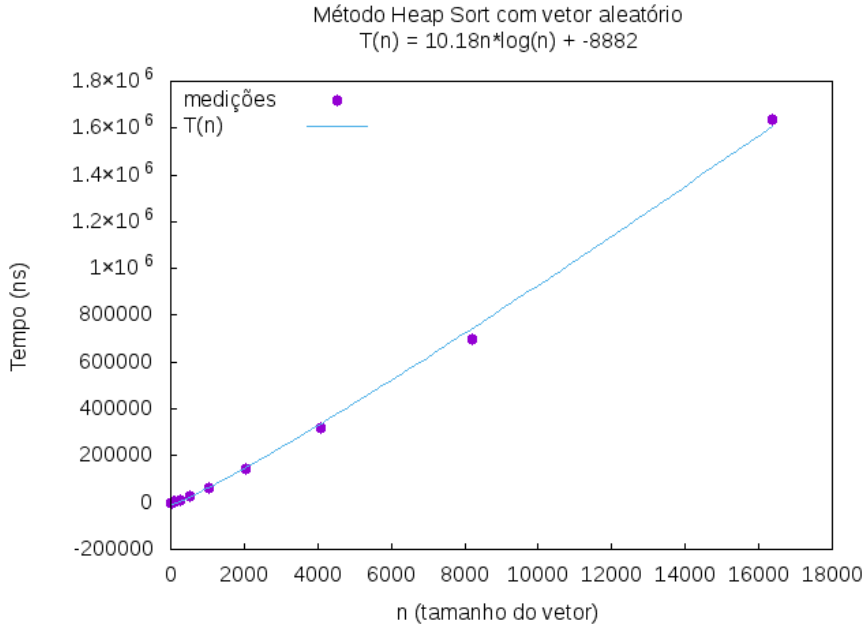

Figura 4.17: *Corte Haste Memoizada - Vetor Crescente P10*

4.4.4 Vetor Crescente P20

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P20.

Tabela 4.18: *Corte Haste Memoizada com Vetor Crescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	6253

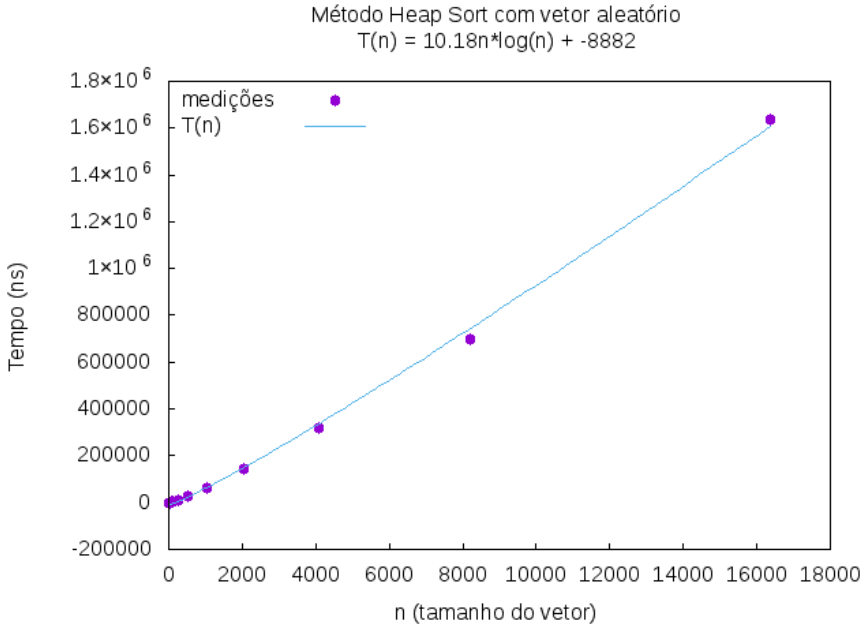

Figura 4.18: *Corte Haste Memoizada - Vetor Crescente P20*

4.4.5 Vetor Crescente P30

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P30.

Tabela 4.19: *Corte Haste Memoizada com Vetor Crescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	6253

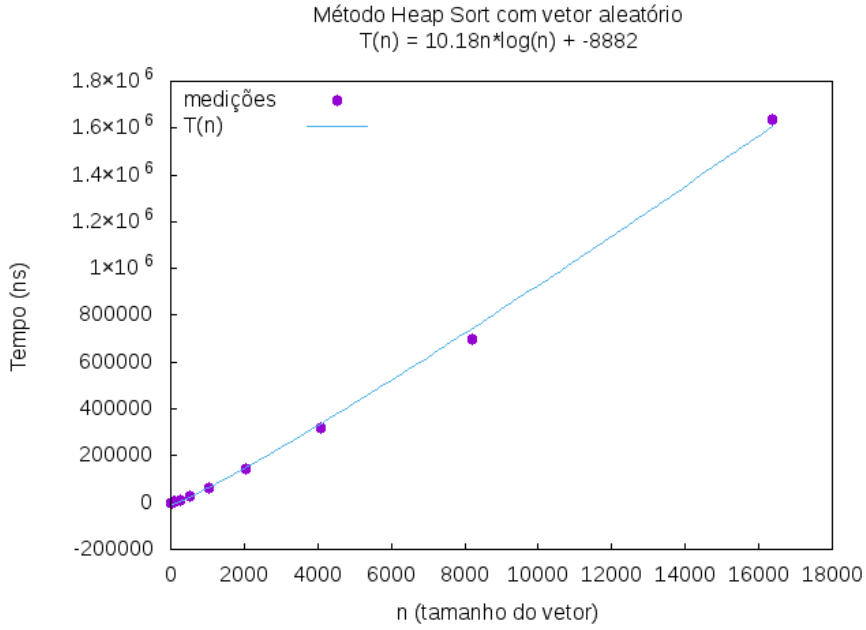

Figura 4.19: *Corte Haste Memoizada - Vetor Crescente P30*

4.4.6 Vetor Crescente P40

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P40.

Tabela 4.20: *Corte Haste Memoizada com Vetor Crescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	6253

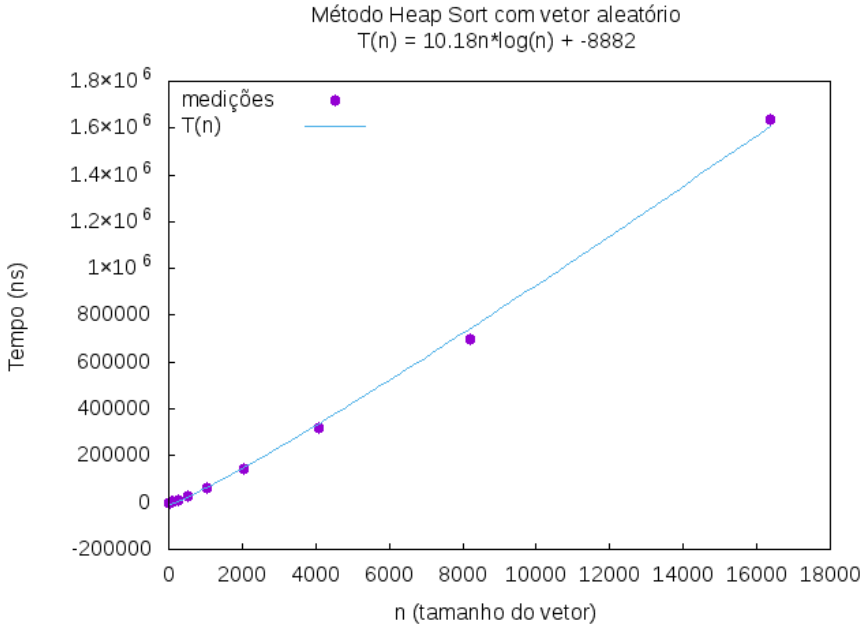

Figura 4.20: *Corte Haste Memoizada - Vetor Crescente P40*

4.4.7 Vetor Crescente P50

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P50.

Tabela 4.21: *Corte Haste Memoizada com Vetor Crescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	6253

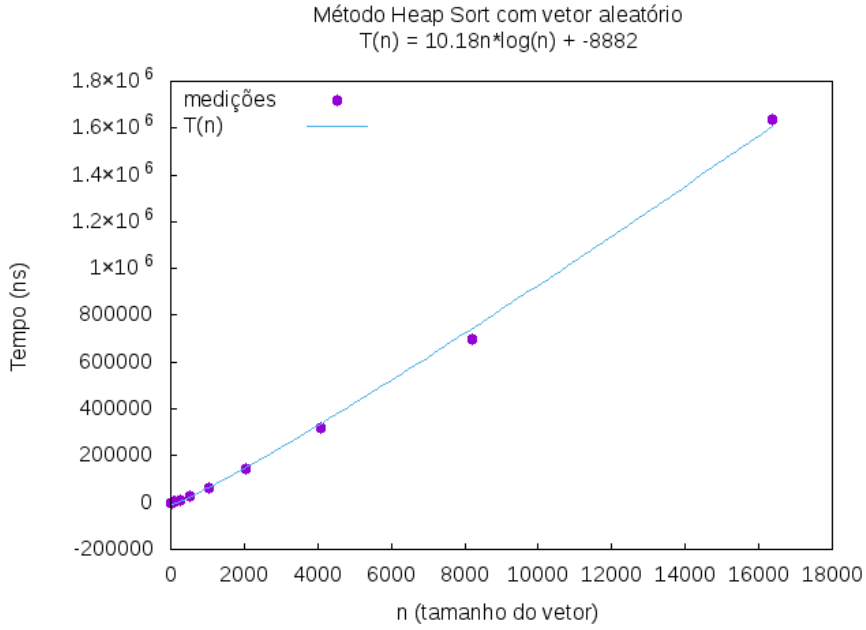

Figura 4.21: *Corte Haste Memoizada - Vetor Crescente P50*

4.4.8 Vetor Decrescente

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente.

Tabela 4.22: *Corte Haste Memoizada com Vetor Decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	6253

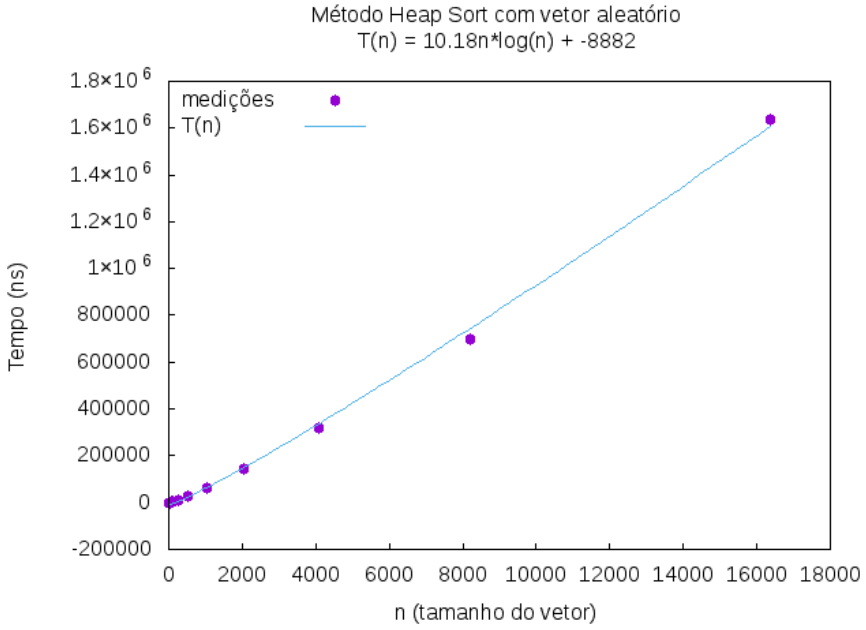

Figura 4.22: *Corte Haste Memoizada - Vetor Decrescente*

4.4.9 Vetor Decrescente P10

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P10.

Tabela 4.23: *Corte Haste Memoizada com Vetor Decrescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	6253

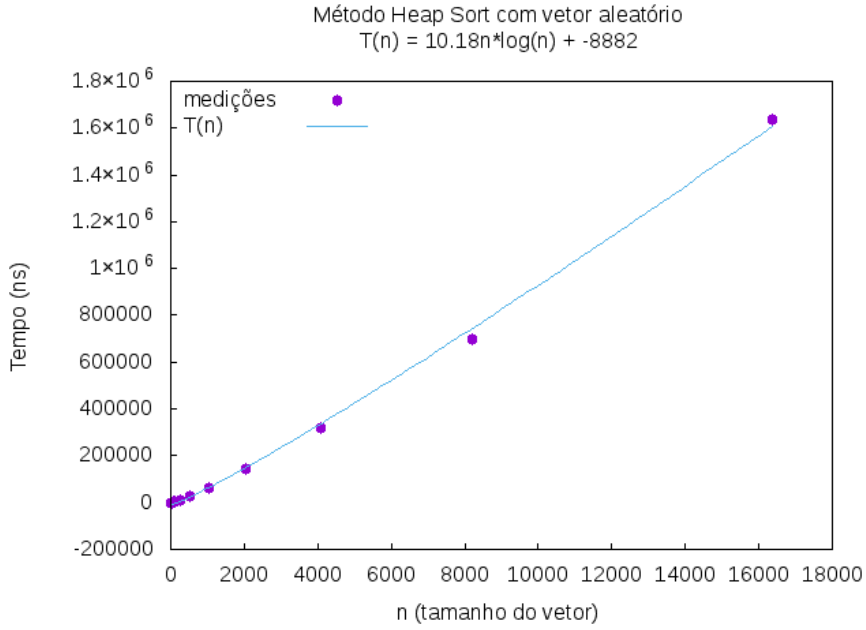

Figura 4.23: *Corte Haste Memoizada - Vetor Decrescente P10*

4.4.10 Vetor Decrescente P20

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P20.

Tabela 4.24: *Corte Haste Memoizada com Vetor Decrescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	6253

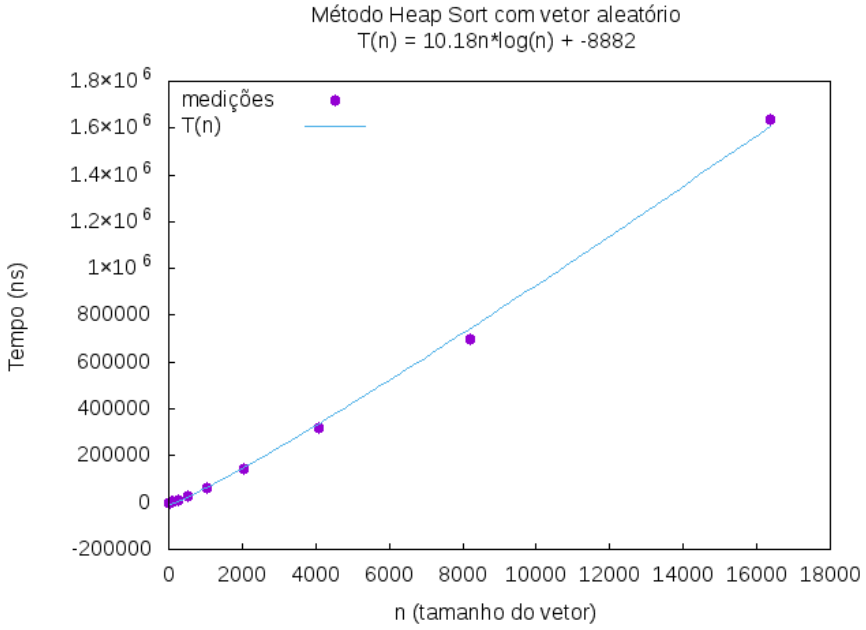

Figura 4.24: *Corte Haste Memoizada - Vetor Decrescente P20*

4.4.11 Vetor Decrescente P30

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P30.

Tabela 4.25: *Corte Haste Memoizada com Vetor Decrescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	6253

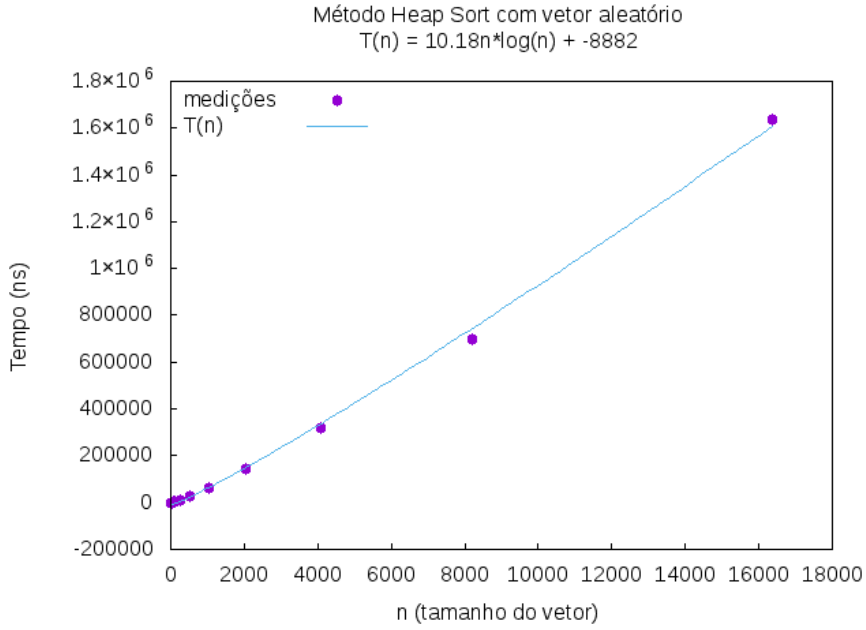

Figura 4.25: *Corte Haste Memoizada - Vetor Decrescente P30*

4.4.12 Vetor Decrescente P40

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P40.

Tabela 4.26: *Corte Haste Memoizada com Vetor Decrescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	6253

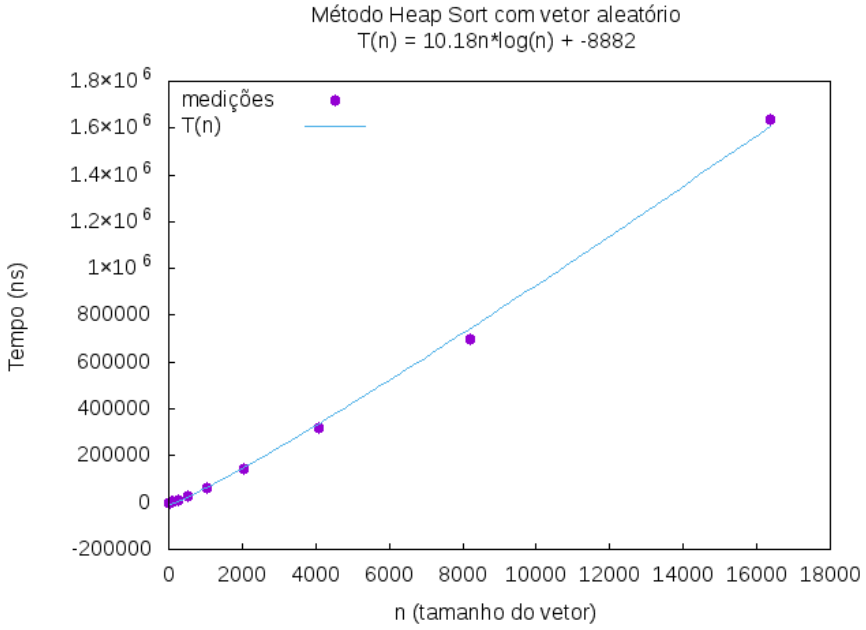

Figura 4.26: *Corte Haste Memoizada - Vetor Decrescente P40*

4.4.13 Vetor Decrescente P50

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P50.

Tabela 4.27: *Corte Haste Memoizada com Vetor Decrescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	6253

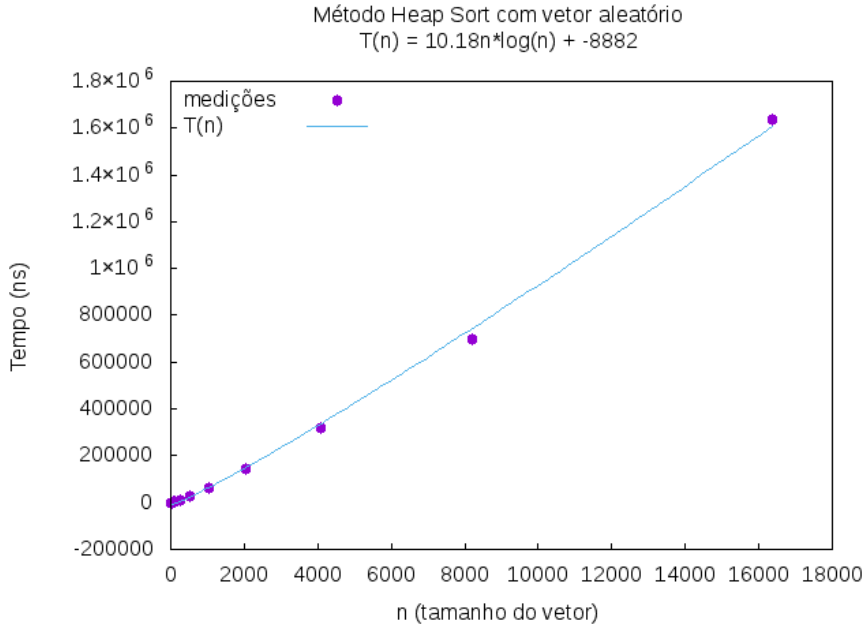

Figura 4.27: *Corte Haste Memoizada - Vetor Decrescente P50*

4.5 Parentização Bottom Up

bLABLABALBALBAL

4.5.1 Vetor aleatorio

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente.

Tabela 4.28: *Parentização Bottom Up com vetor aleatório*

Número de Elementos	Tempo de execução em nanosegundos
16	8230
32	19874
64	138865
128	1119535
256	7674247
512	70240583
1024	1582607779

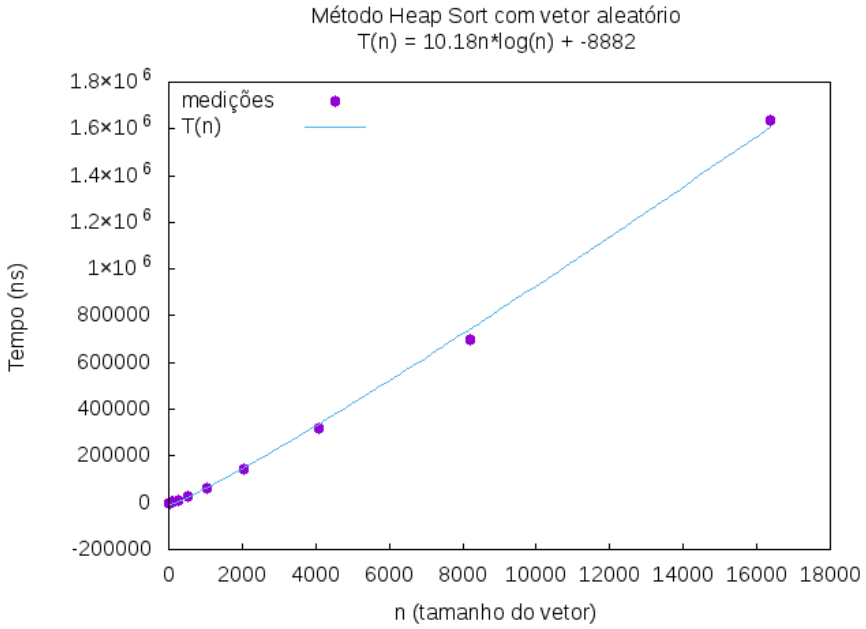

Figura 4.28: *Parentização Bottom Up - Vetor Aleatório*

4.5.2 Vetor Crescente

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente.

Tabela 4.29: *Parentização Bottom Up com vetor Crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	8230
32	19874
64	138865
128	1119535
256	7674247
512	70240583
1024	1582607779

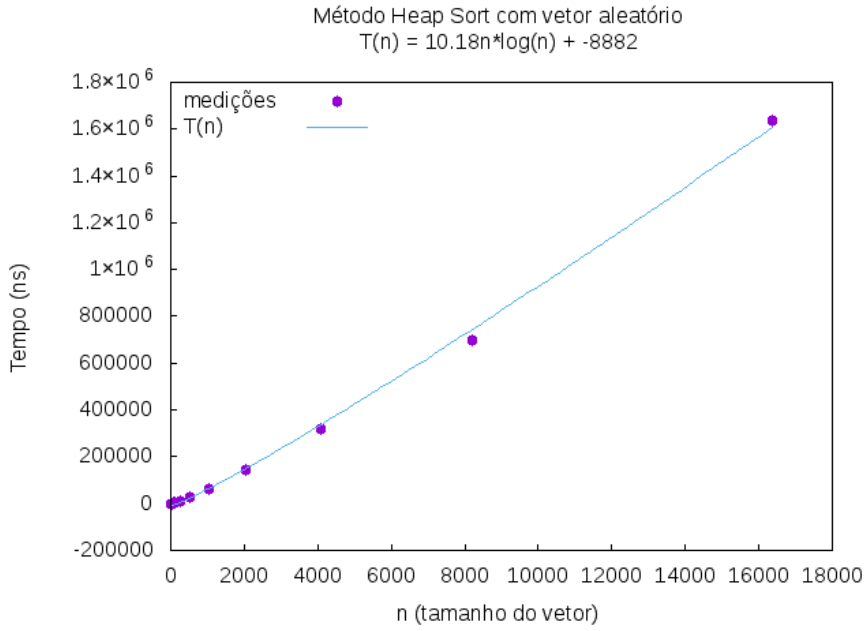

Figura 4.29: *Parentização Bottom Up - Vetor Crescente*

4.5.3 Vetor Crescente P10

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P10.

Tabela 4.30: *Parentização Bottom Up com vetor Crescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	8230
32	19874
64	138865
128	1119535
256	7674247
512	70240583
1024	1582607779

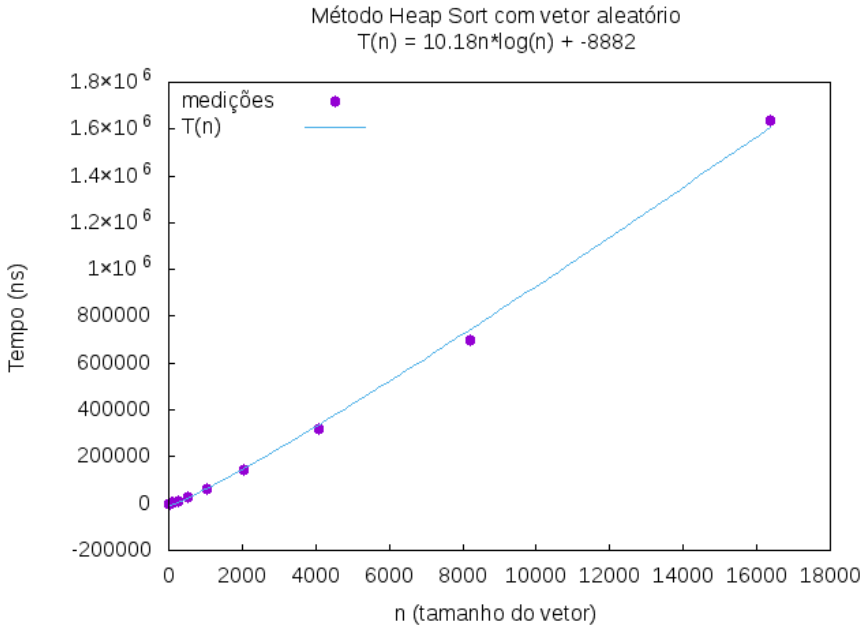

Figura 4.30: *Parentização Bottom Up - Vetor Crescente P10*

4.5.4 Vetor Crescente P20

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P20.

Tabela 4.31: *Parentização Bottom Up com vetor Crescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	8230
32	19874
64	138865
128	1119535
256	7674247
512	70240583
1024	1582607779

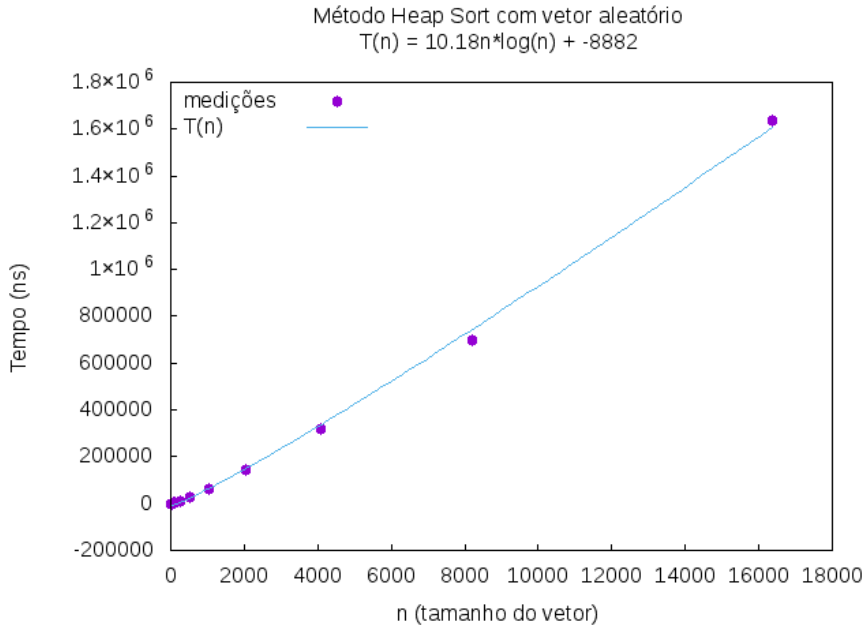

Figura 4.31: *Parentização Bottom Up - Vetor Crescente P20*

4.5.5 Vetor Crescente P30

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P30.

Tabela 4.32: *Parentização Bottom Up com vetor Crescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	8230
32	19874
64	138865
128	1119535
256	7674247
512	70240583
1024	1582607779

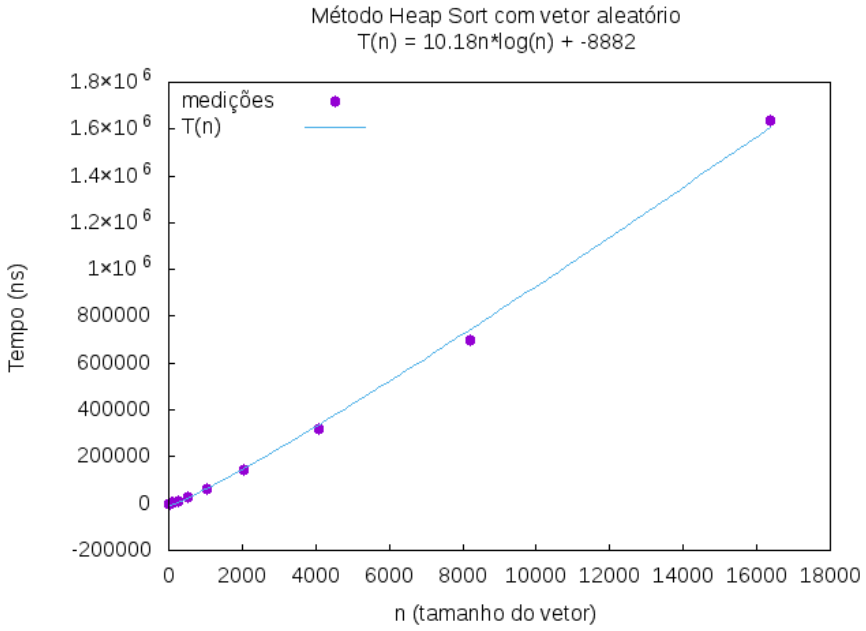

Figura 4.32: *Parentização Bottom Up - Vetor Crescente P30*

4.5.6 Vetor Crescente P40

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P40.

Tabela 4.33: *Parentização Bottom Up com vetor Crescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	8230
32	19874
64	138865
128	1119535
256	7674247
512	70240583
1024	1582607779

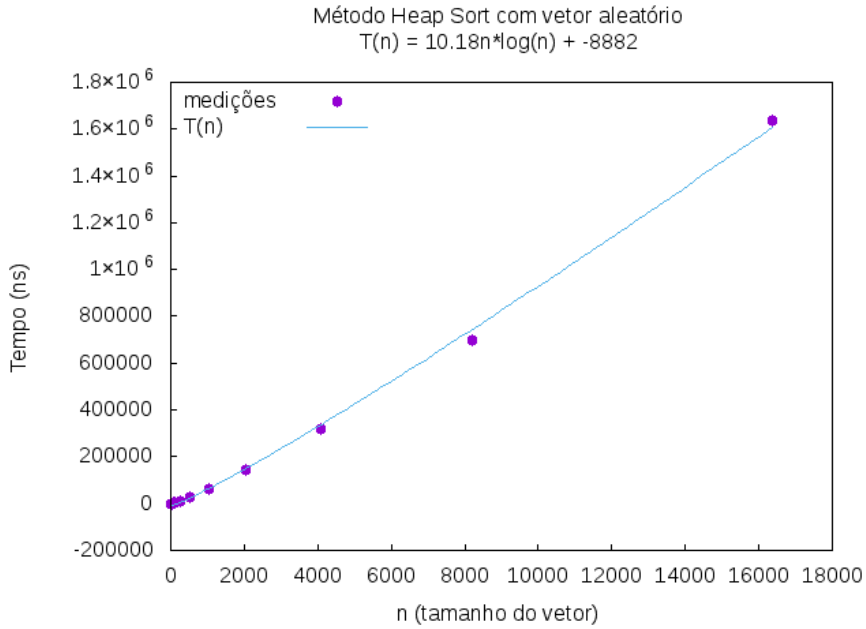

Figura 4.33: *Parentização Bottom Up - Vetor Crescente P40*

4.5.7 Vetor Crescente P50

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P50.

Tabela 4.34: *Parentização Bottom Up com vetor Crescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	8230
32	19874
64	138865
128	1119535
256	7674247
512	70240583
1024	1582607779

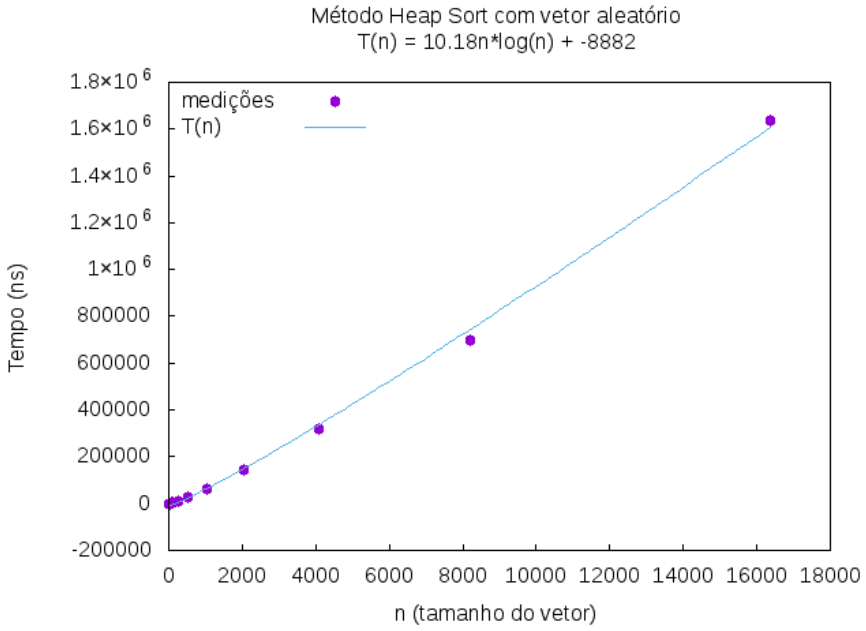

Figura 4.34: *Parentização Bottom Up - Vetor Crescente P50*

4.5.8 Vetor Decrescente

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente.

Tabela 4.35: *Parentização Bottom Up com vetor Decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	8230
32	19874
64	138865
128	1119535
256	7674247
512	70240583
1024	1582607779

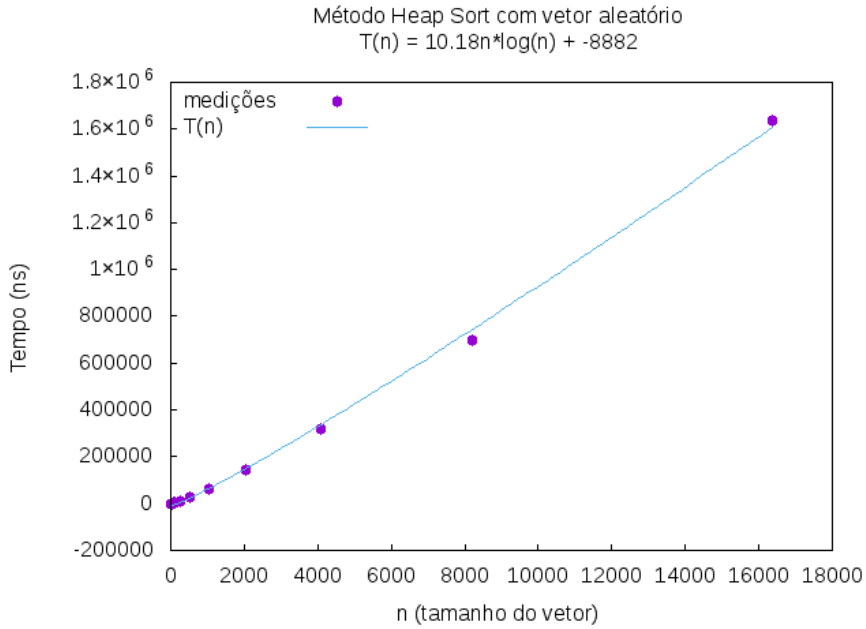

Figura 4.35: *Parentização Bottom Up - Vetor Decrescente*

4.5.9 Vetor Decrescente P10

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P10.

Tabela 4.36: *Parentização Bottom Up com vetor Decrescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	8230
32	19874
64	138865
128	1119535
256	7674247
512	70240583
1024	1582607779

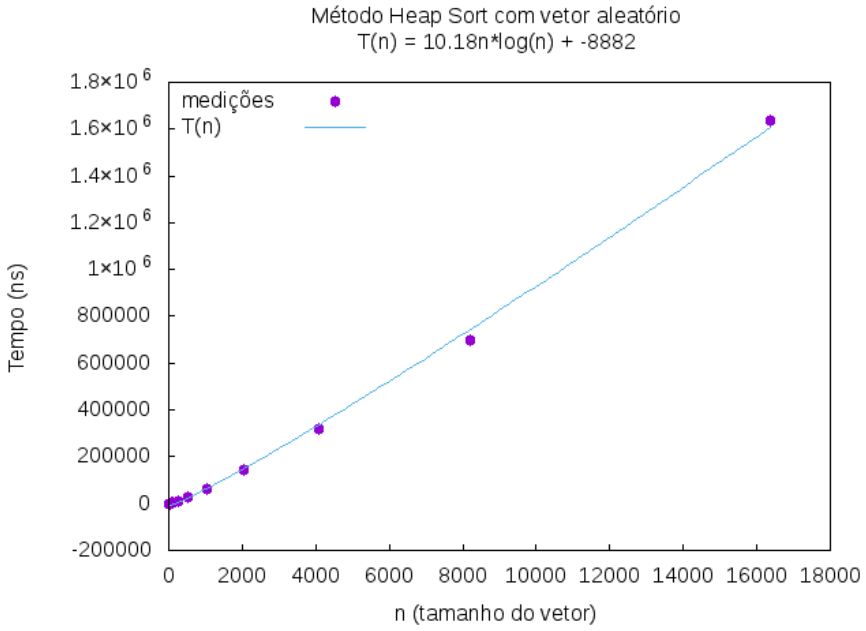

Figura 4.36: *Parentização Bottom Up - Vetor Decrescente P10*

4.5.10 Vetor Decrescente P20

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P20.

Tabela 4.37: *Parentização Bottom Up com vetor Decrescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	8230
32	19874
64	138865
128	1119535
256	7674247
512	70240583
1024	1582607779

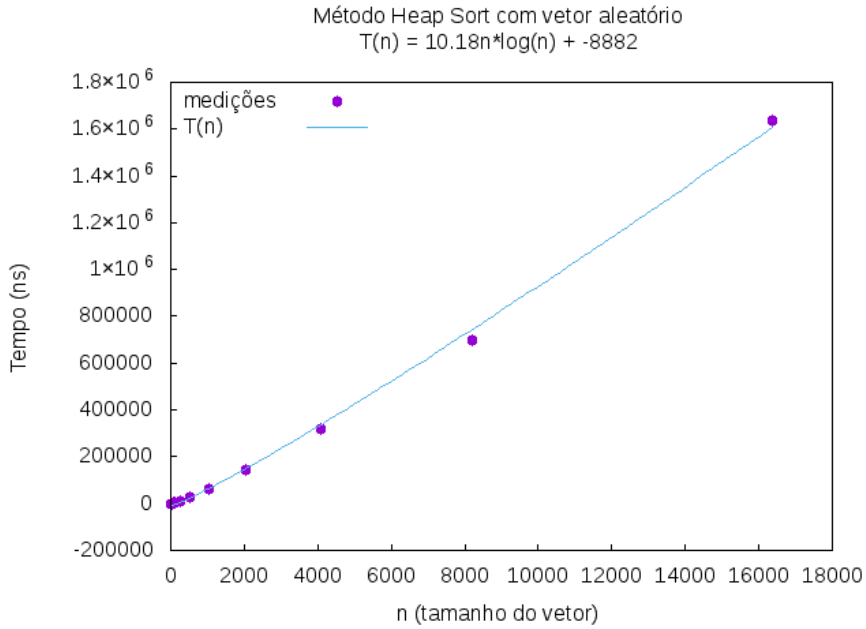

Figura 4.37: *Parentização Bottom Up - Vetor Decrescente P20*

4.5.11 Vetor Decrescente P30

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P30.

Tabela 4.38: *Parentização Bottom Up com vetor Decrescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	8230
32	19874
64	138865
128	1119535
256	7674247
512	70240583
1024	1582607779

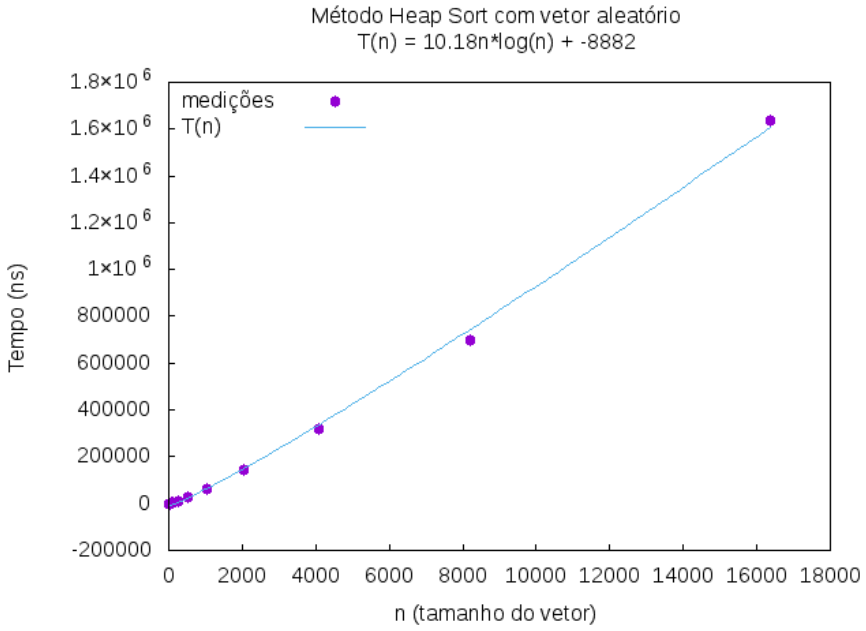

Figura 4.38: *Parentização Bottom Up - Vetor Decrescente P30*

4.5.12 Vetor Decrescente P40

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P40.

Tabela 4.39: *Parentização Bottom Up com vetor Decrescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	8230
32	19874
64	138865
128	1119535
256	7674247
512	70240583
1024	1582607779

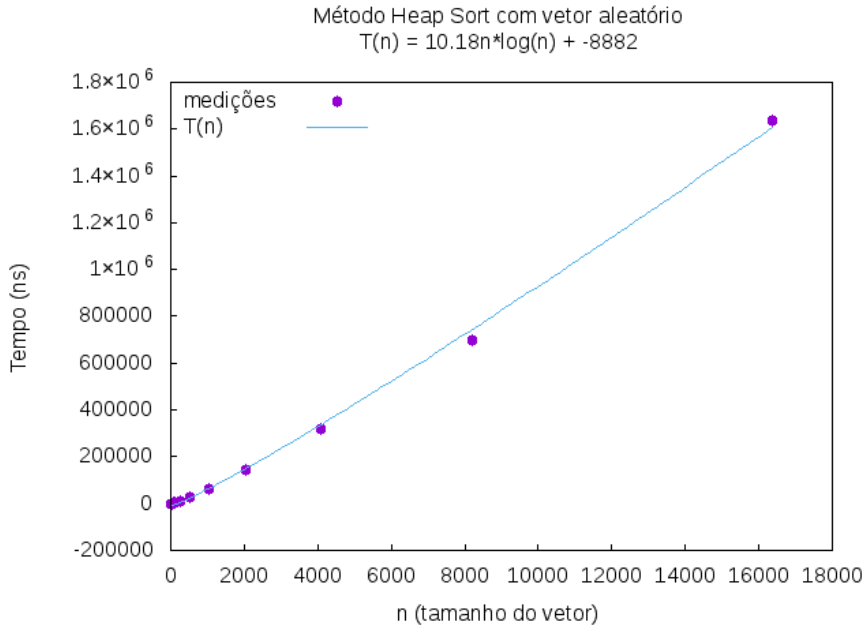

Figura 4.39: *Parentização Bottom Up - Vetor Decrescente P40*

4.5.13 Vetor Decrescente P50

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P50.

Tabela 4.40: *Parentização Bottom Up com vetor Decrescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	8230
32	19874
64	138865
128	1119535
256	7674247
512	70240583
1024	1582607779

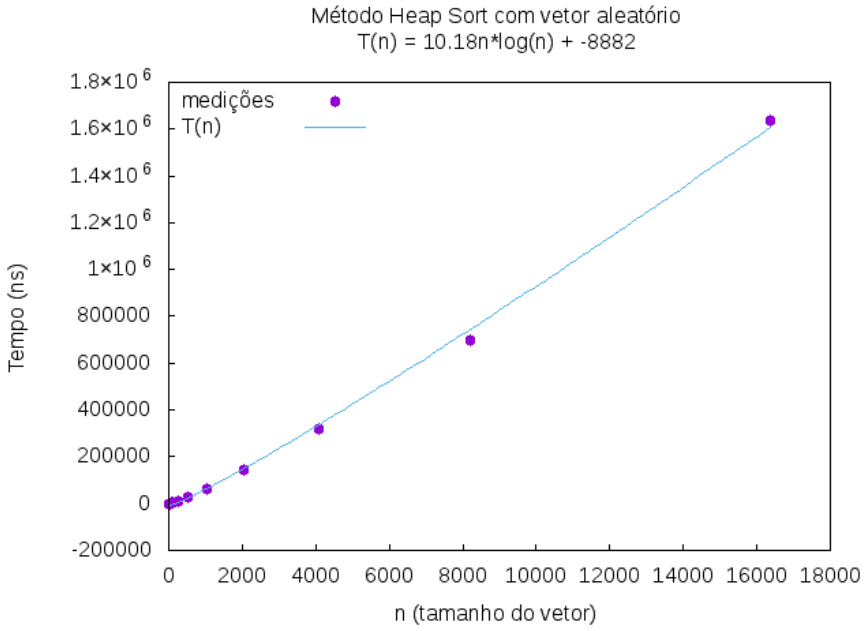

Figura 4.40: *Parentização Bottom Up - Vetor Decrescente P50*

4.6 Parentização Recursiva

bLABLABALBALBAL

Tabela gerada utilizando Parentização Recursiva com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente, crescente, crescente P10, crescente P20, crescente P30, crescente P40, crescente P50, decrescente, decrescente P10, decrescente P20, decrescente P30, decrescente P40, decrescente P50.

Tabela 4.41: *Parentização Recursiva*

Número de Elementos	Tempo de execução em nanosegundos
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175

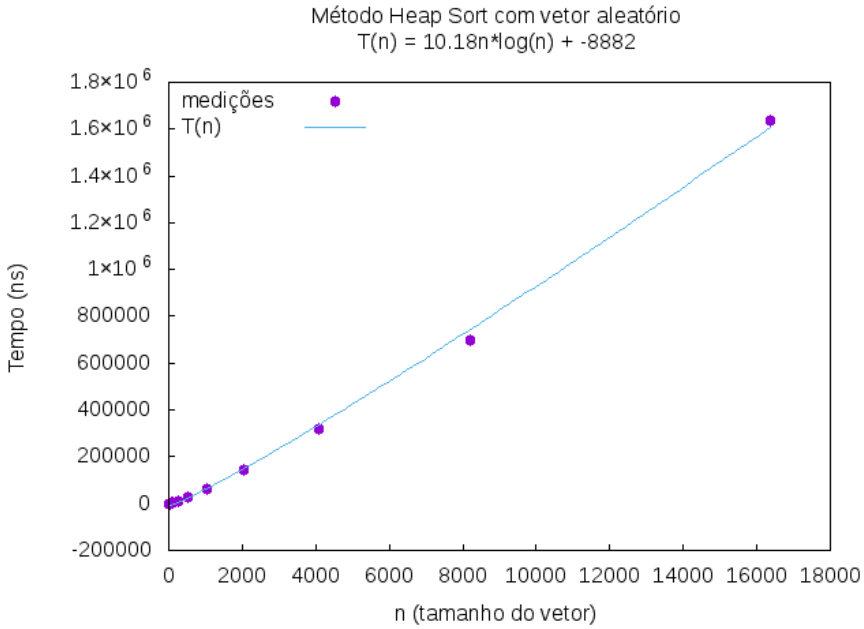

Figura 4.41: *Parentização Recursiva*

Capítulo 5

Estatísticas de Ordem

Programação dinamica blablabla

5.1 Min

falar um pouco sobre

5.1.1 Vetor aleatorio

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente.

Tabela 5.1: *Min com vetor aleatório*

Número de Elementos	**Tempo de execução em nanosegundos**
16	373
32	379
64	417
128	399
256	427
512	477
1024	707
2048	922
4096	1540
8192	3131
16384	7067

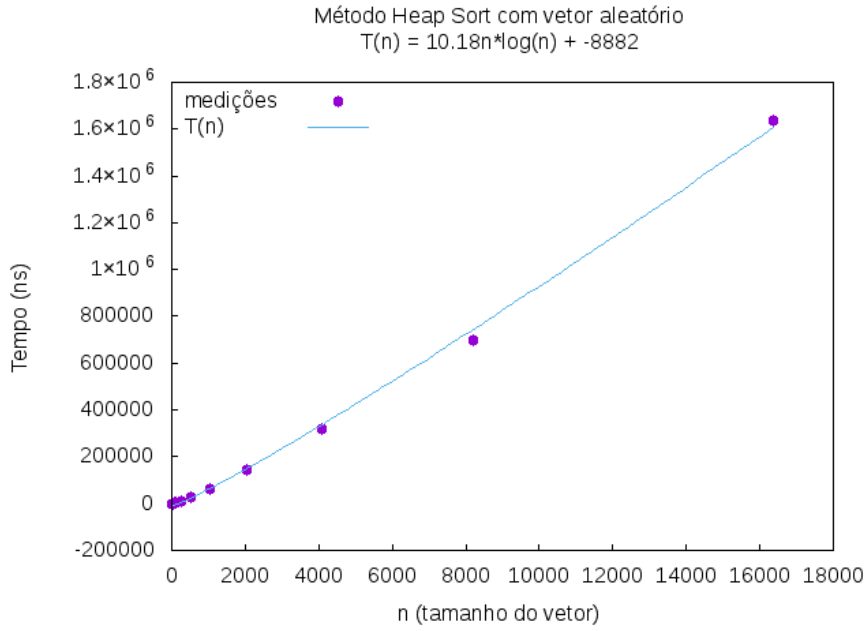

Figura 5.1: *Min - Vetor Aleatório*

5.1.2 Vetor Crescente

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente.

Tabela 5.2: *Min com vetor Crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	373
32	379
64	417
128	399
256	427
512	477
1024	707
2048	922
4096	1540
8192	3131
16384	7067

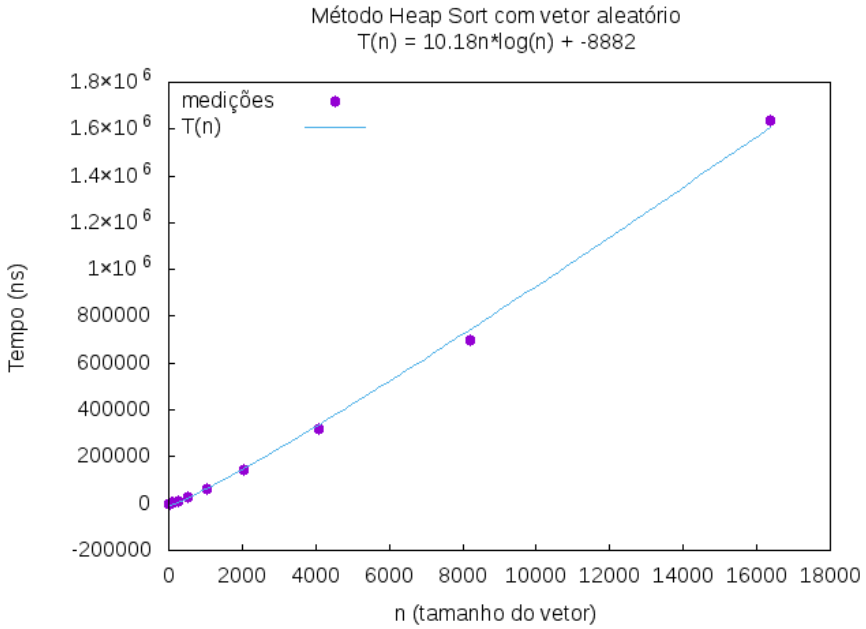

Figura 5.2: *Min - Vetor Crescente*

5.1.3 Vetor Crescente P10

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P10.

Tabela 5.3: *Min com vetor Crescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	373
32	379
64	417
128	399
256	427
512	477
1024	707
2048	922
4096	1540
8192	3131
16384	7067

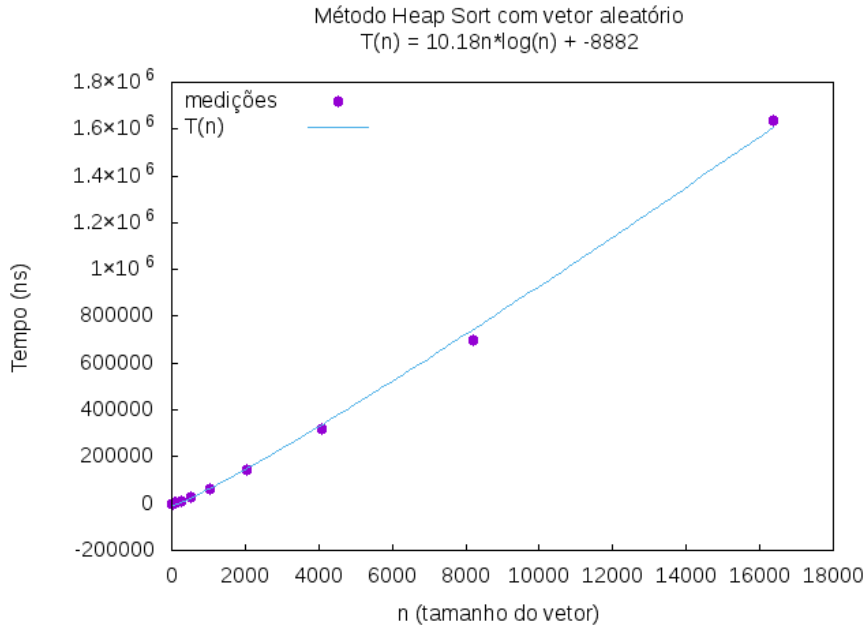

Figura 5.3: *Min - Vetor Crescente P10*

5.1.4 Vetor Crescente P20

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P20.

Tabela 5.4: *Min com vetor Crescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	373
32	379
64	417
128	399
256	427
512	477
1024	707
2048	922
4096	1540
8192	3131
16384	7067

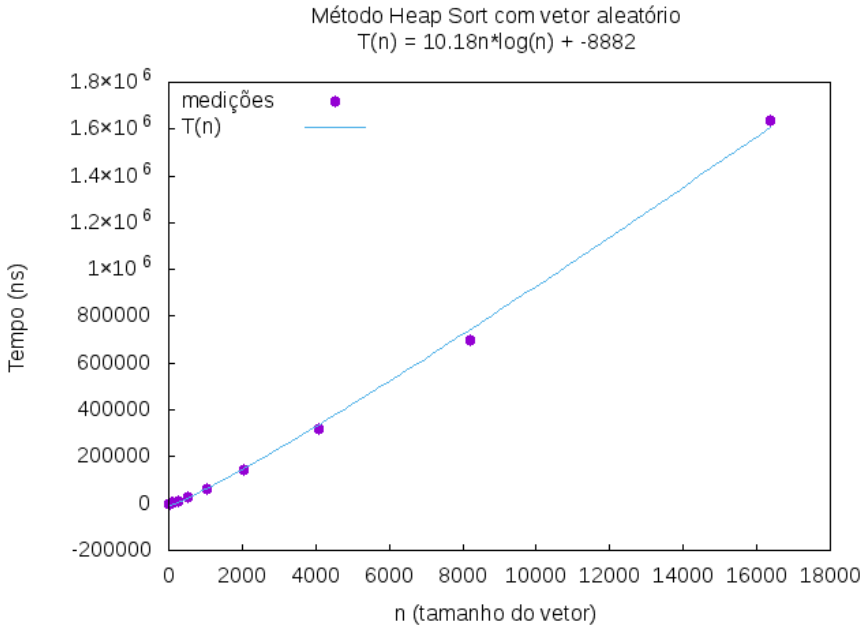

Figura 5.4: *Min - Vetor Crescente P20*

5.1.5 Vetor Crescente P30

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P30.

Tabela 5.5: *Min com vetor Crescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	373
32	379
64	417
128	399
256	427
512	477
1024	707
2048	922
4096	1540
8192	3131
16384	7067

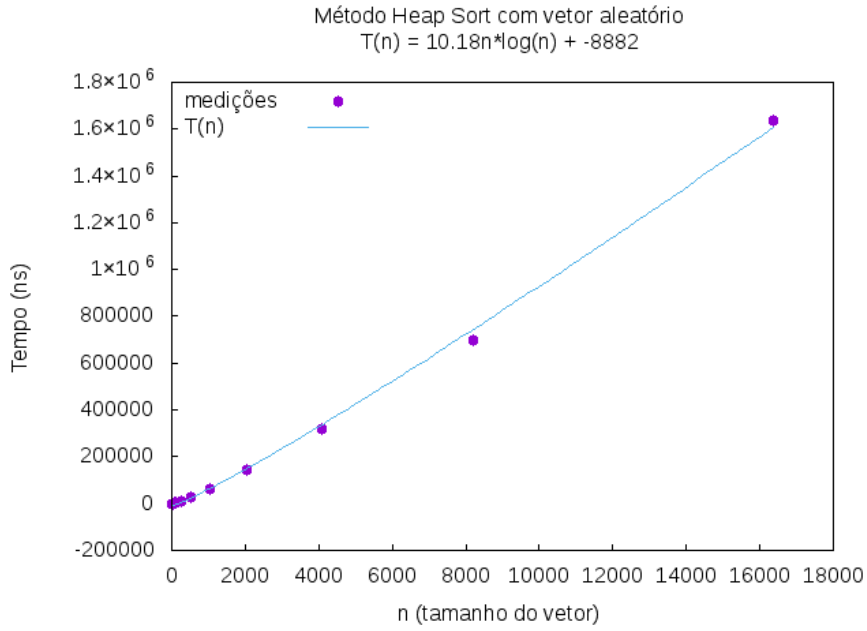

Figura 5.5: *Min - Vetor Crescente P30*

5.1.6 Vetor Crescente P40

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P40.

Tabela 5.6: *Min com vetor Crescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	373
32	379
64	417
128	399
256	427
512	477
1024	707
2048	922
4096	1540
8192	3131
16384	7067

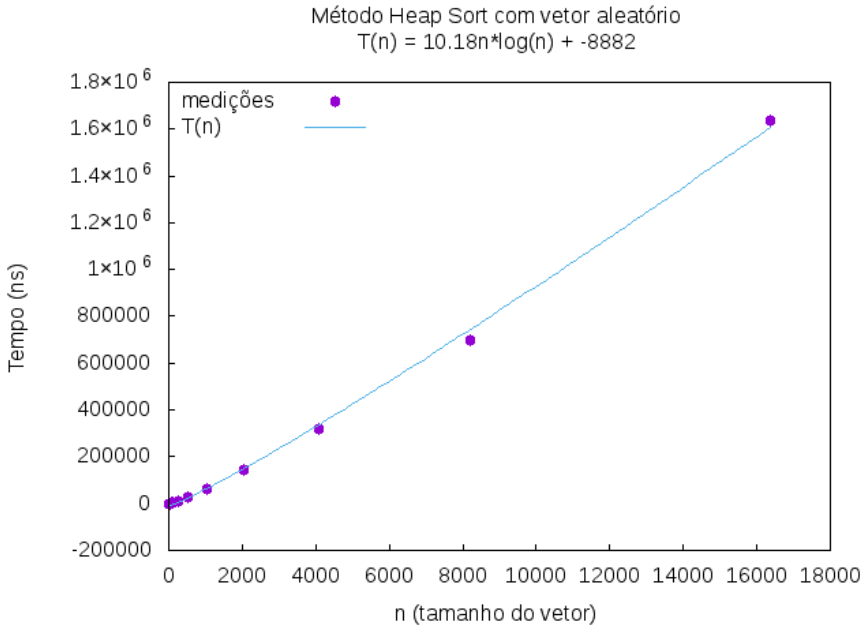

Figura 5.6: *Min - Vetor Crescente P40*

5.1.7 Vetor Crescente P50

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P50.

Tabela 5.7: *Min com vetor Crescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	373
32	379
64	417
128	399
256	427
512	477
1024	707
2048	922
4096	1540
8192	3131
16384	7067

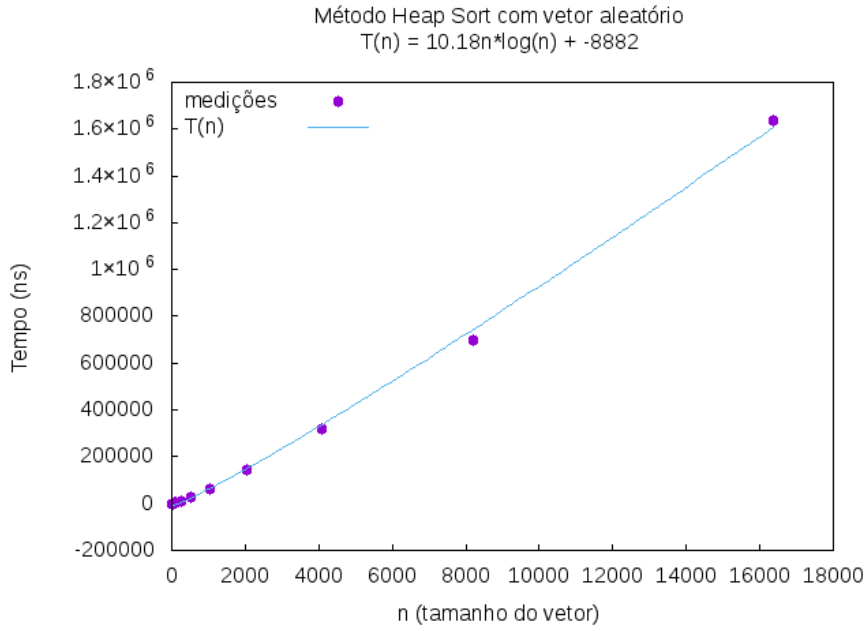

Figura 5.7: *Min - Vetor Crescente P50*

5.1.8 Vetor Decrescente

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente.

Tabela 5.8: *Min com vetor Decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	373
32	379
64	417
128	399
256	427
512	477
1024	707
2048	922
4096	1540
8192	3131
16384	7067

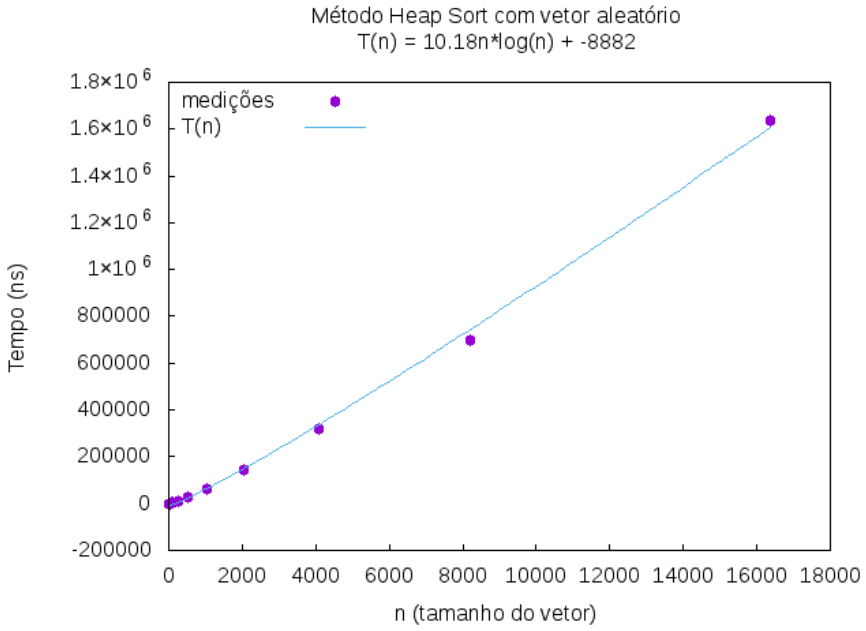

Figura 5.8: *Min - Vetor Decrescente*

5.1.9 Vetor Decrescente P10

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P10.

Tabela 5.9: *Min com vetor Decrescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	373
32	379
64	417
128	399
256	427
512	477
1024	707
2048	922
4096	1540
8192	3131
16384	7067

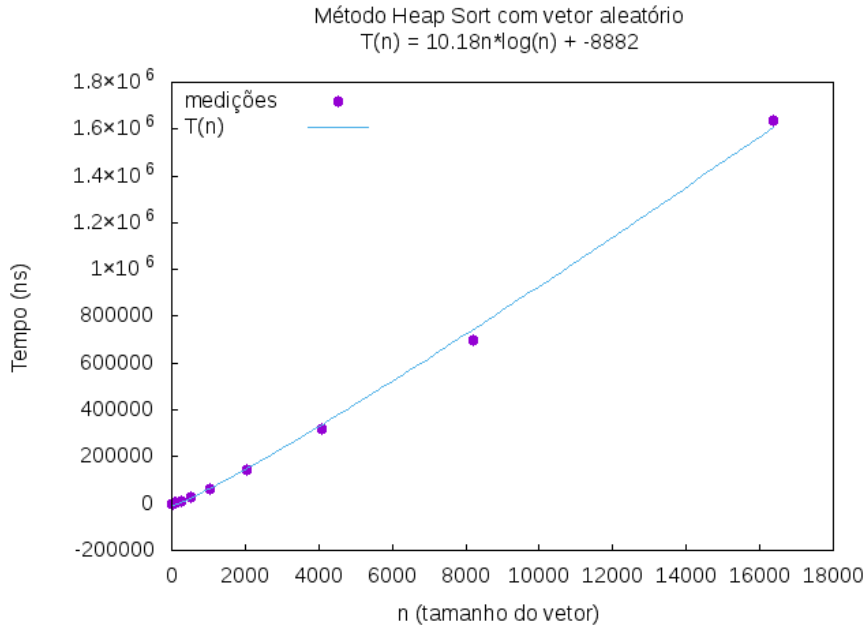

Figura 5.9: *Min - Vetor Decrescente P10*

5.1.10 Vetor Decrescente P20

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P20.

Tabela 5.10: *Min com vetor Decrescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	373
32	379
64	417
128	399
256	427
512	477
1024	707
2048	922
4096	1540
8192	3131
16384	7067

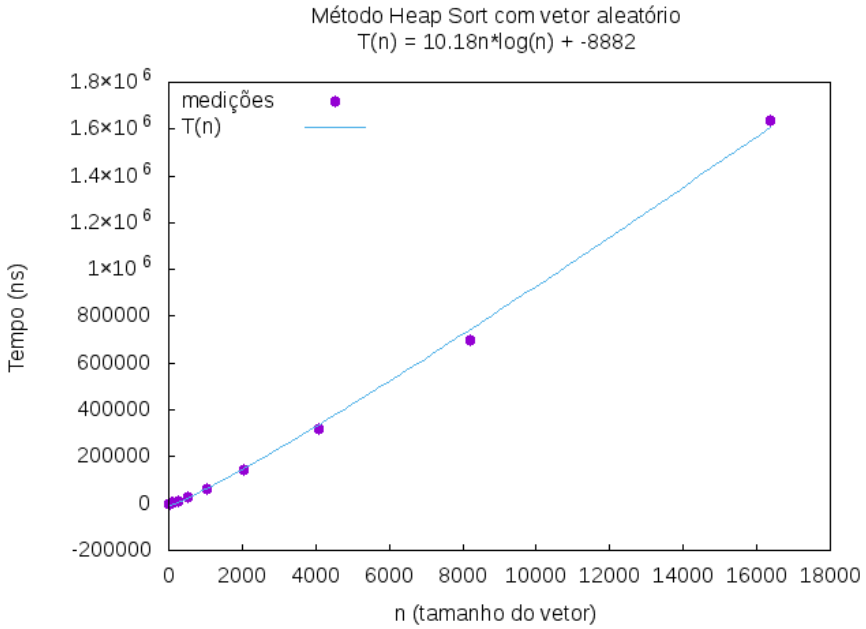

Figura 5.10: *Min - Vetor Decrescente P20*

5.1.11 Vetor Decrescente P30

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P30.

Tabela 5.11: *Min com vetor Decrescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	373
32	379
64	417
128	399
256	427
512	477
1024	707
2048	922
4096	1540
8192	3131
16384	7067

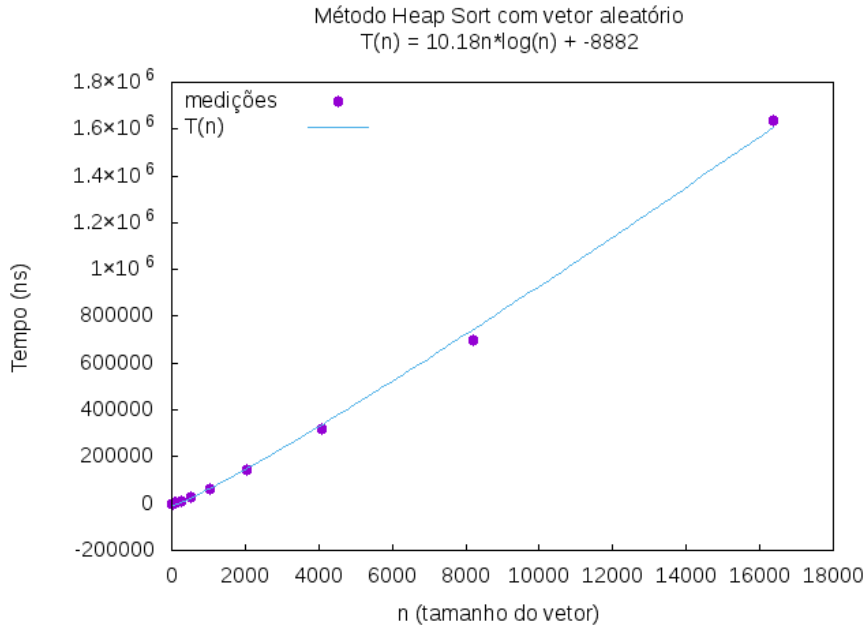

Figura 5.11: *Min - Vetor Decrescente P30*

5.1.12 Vetor Decrescente P40

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P40.

Tabela 5.12: *Min com vetor Decrescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	373
32	379
64	417
128	399
256	427
512	477
1024	707
2048	922
4096	1540
8192	3131
16384	7067

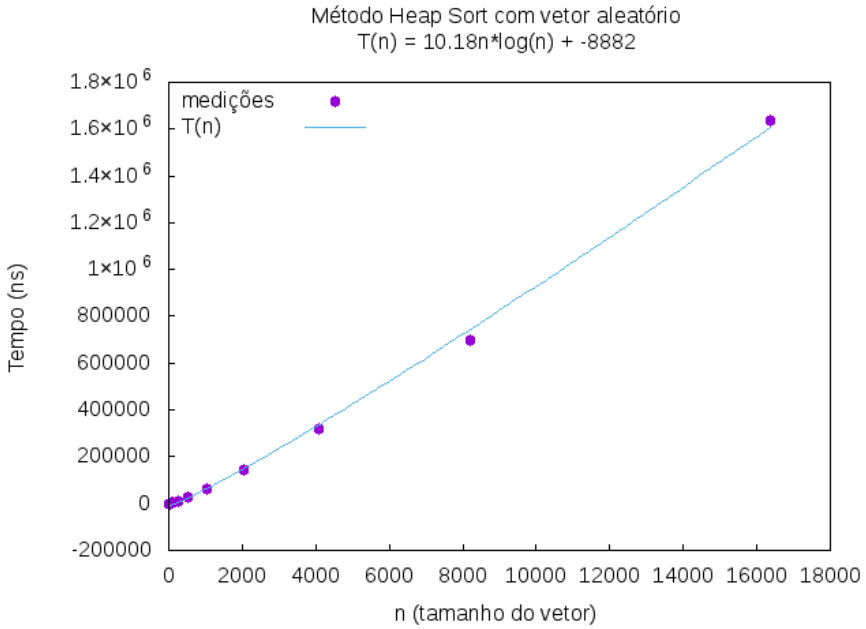

Figura 5.12: *Min - Vetor Decrescente P40*

5.1.13 Vetor Decrescente P50

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P50.

Tabela 5.13: *Min com vetor Decrescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	373
32	379
64	417
128	399
256	427
512	477
1024	707
2048	922
4096	1540
8192	3131
16384	7067

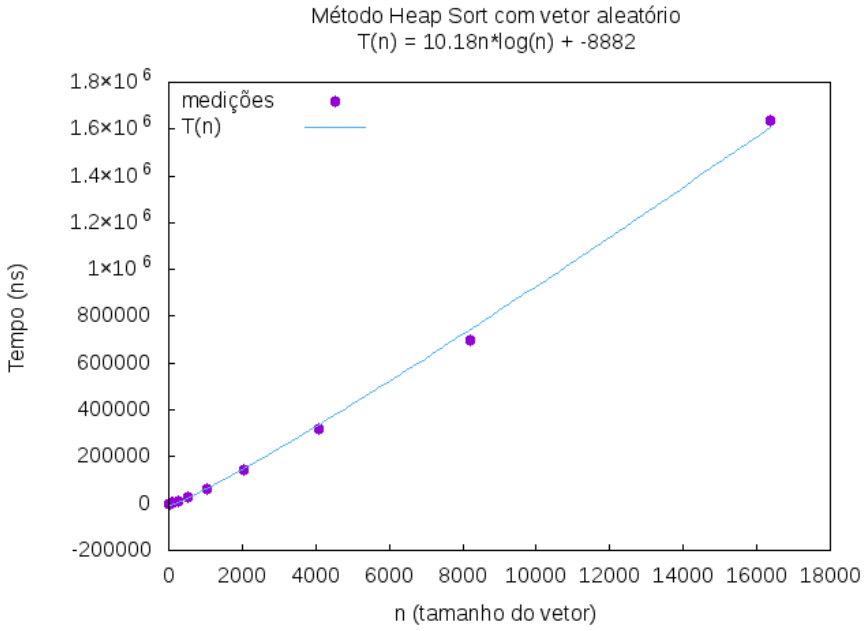

Figura 5.13: *Min - Vetor Decrescente P50*

5.2 MinMax

falar um pouco sobre

5.2.1 Vetor aleatorio

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente.

Tabela 5.14: *MinMax com vetor aleatório*

Número de Elementos	Tempo de execução em nanosegundos
16	628
32	634
64	686
128	492
256	1042
512	806
1024	1087
2048	1250
4096	2139
8192	3898
16384	9144

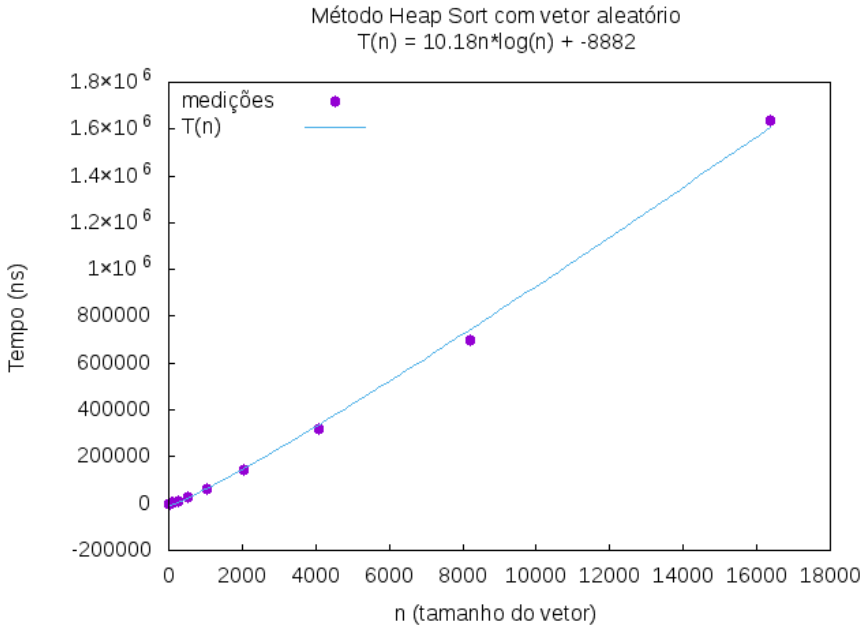

Figura 5.14: *MinMax - Vetor Aleatório*

5.2.2 Vetor Crescente

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente.

Tabela 5.15: *MinMax com vetor Crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	628
32	634
64	686
128	492
256	1042
512	806
1024	1087
2048	1250
4096	2139
8192	3898
16384	9144

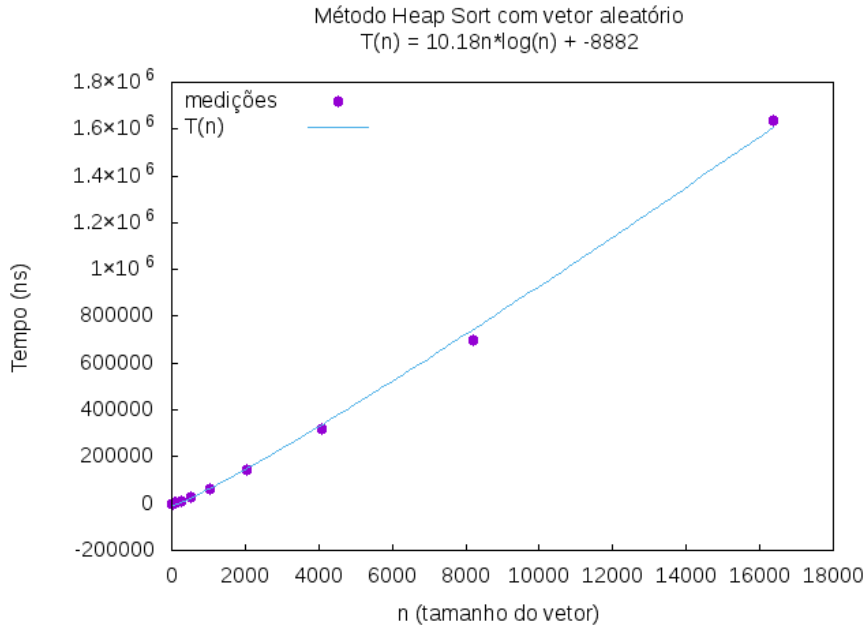

Figura 5.15: *MinMax - Vetor Crescente*

5.2.3 Vetor Crescente P10

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P10.

Tabela 5.16: *MinMax com vetor Crescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	628
32	634
64	686
128	492
256	1042
512	806
1024	1087
2048	1250
4096	2139
8192	3898
16384	9144

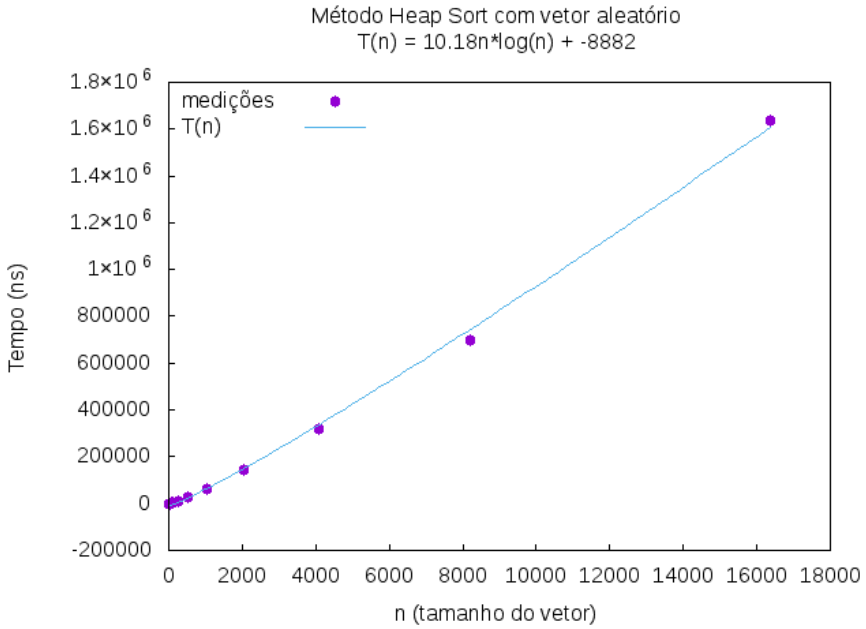

Figura 5.16: *MinMax - Vetor Crescente P10*

5.2.4 Vetor Crescente P20

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P20.

Tabela 5.17: *MinMax com vetor Crescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	628
32	634
64	686
128	492
256	1042
512	806
1024	1087
2048	1250
4096	2139
8192	3898
16384	9144

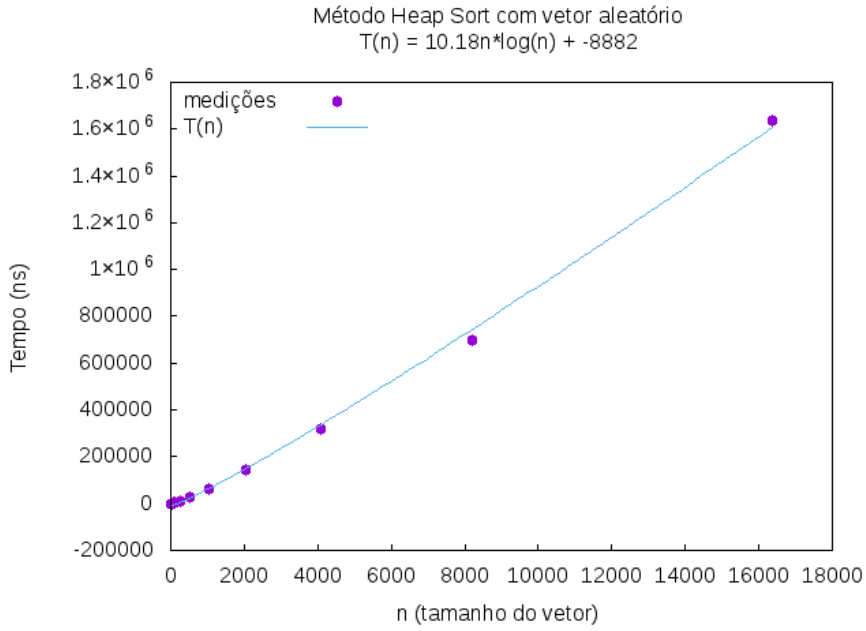

Figura 5.17: *MinMax - Vetor Crescente P20*

5.2.5 Vetor Crescente P30

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P30.

Tabela 5.18: *MinMax com vetor Crescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	628
32	634
64	686
128	492
256	1042
512	806
1024	1087
2048	1250
4096	2139
8192	3898
16384	9144

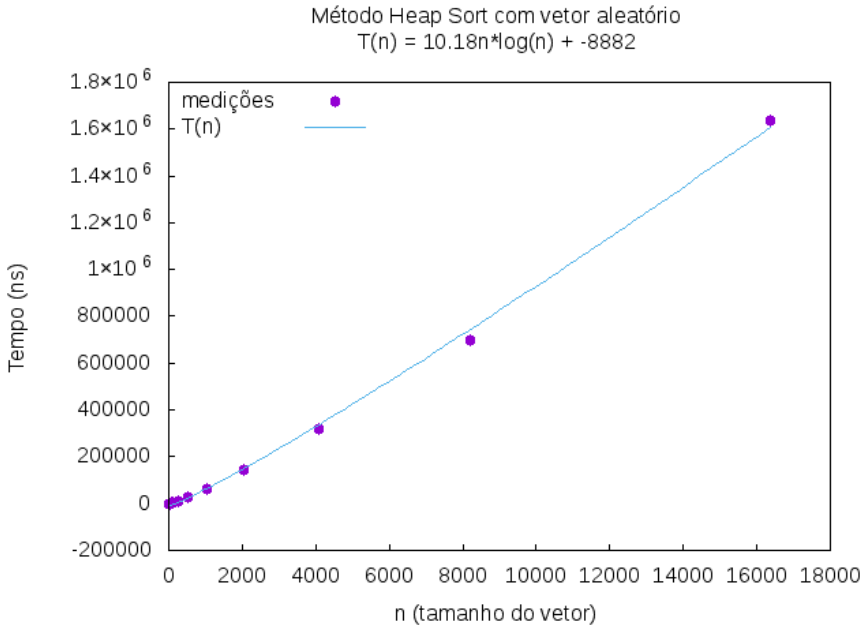

Figura 5.18: *MinMax - Vetor Crescente P30*

5.2.6 Vetor Crescente P40

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P40.

Tabela 5.19: *MinMax com vetor Crescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	628
32	634
64	686
128	492
256	1042
512	806
1024	1087
2048	1250
4096	2139
8192	3898
16384	9144

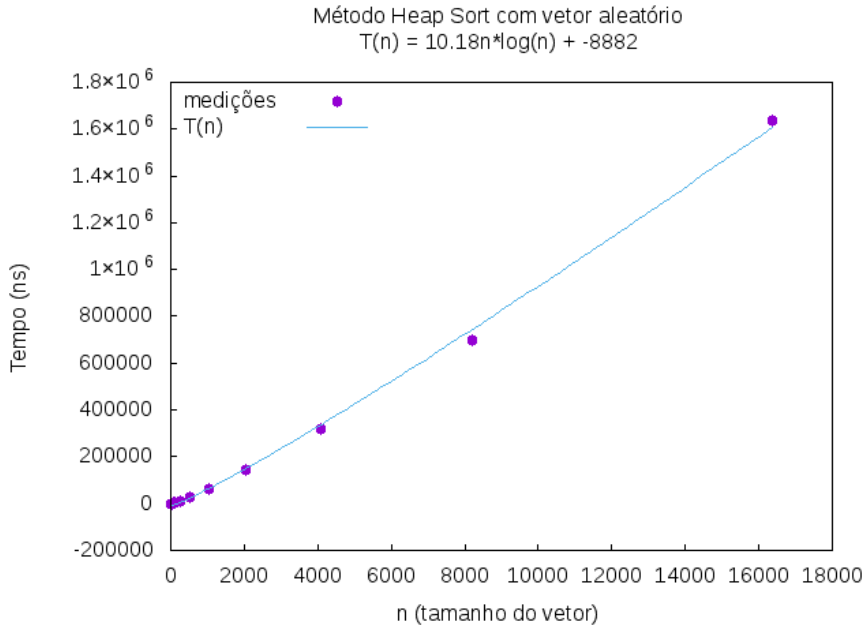

Figura 5.19: *MinMax - Vetor Crescente P40*

5.2.7 Vetor Crescente P50

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P50.

Tabela 5.20: *MinMax com vetor Crescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	628
32	634
64	686
128	492
256	1042
512	806
1024	1087
2048	1250
4096	2139
8192	3898
16384	9144

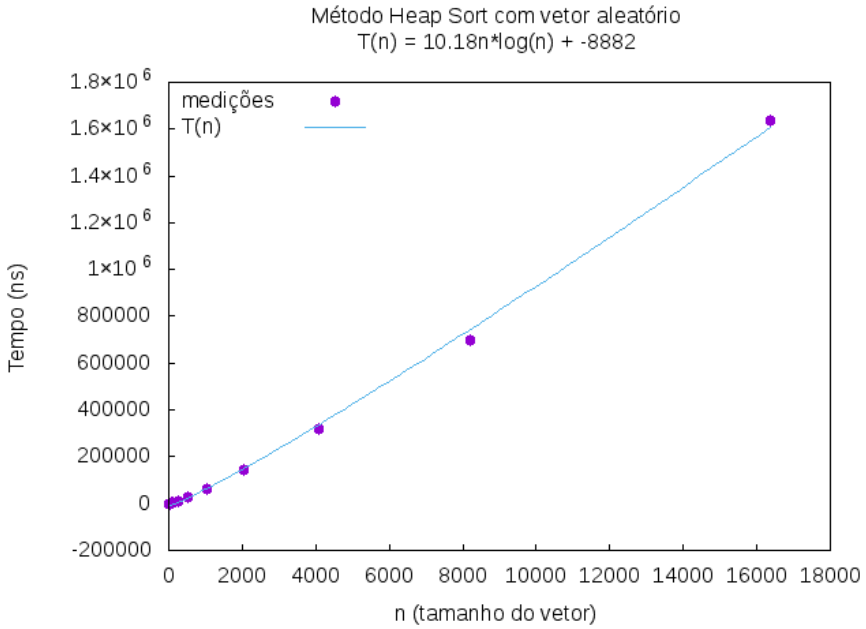

Figura 5.20: *MinMax - Vetor Crescente P50*

5.2.8 Vetor Decrescente

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente.

Tabela 5.21: *MinMax com vetor Decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	628
32	634
64	686
128	492
256	1042
512	806
1024	1087
2048	1250
4096	2139
8192	3898
16384	9144

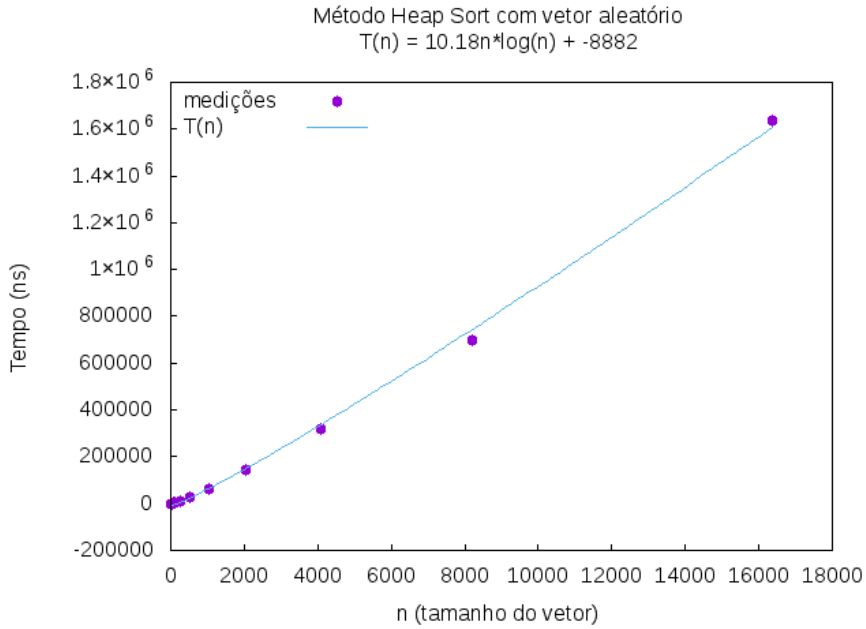

Figura 5.21: *MinMax - Vetor Decrescente*

5.2.9 Vetor Decrescente P10

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P10.

Tabela 5.22: *MinMax com vetor Decrescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	628
32	634
64	686
128	492
256	1042
512	806
1024	1087
2048	1250
4096	2139
8192	3898
16384	9144

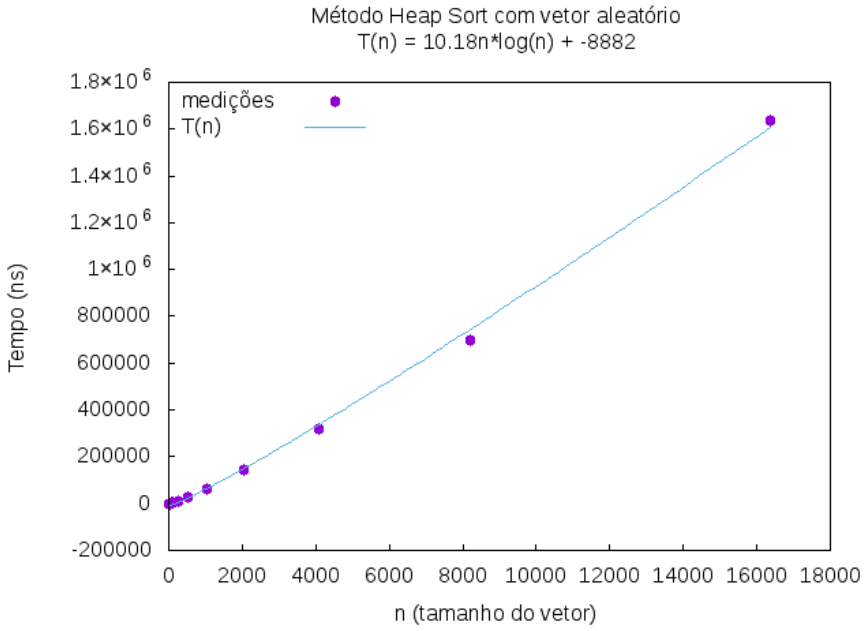

Figura 5.22: *MinMax - Vetor Decrescente P10*

5.2.10 Vetor Decrescente P20

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P20.

Tabela 5.23: *MinMax com vetor Decrescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	628
32	634
64	686
128	492
256	1042
512	806
1024	1087
2048	1250
4096	2139
8192	3898
16384	9144

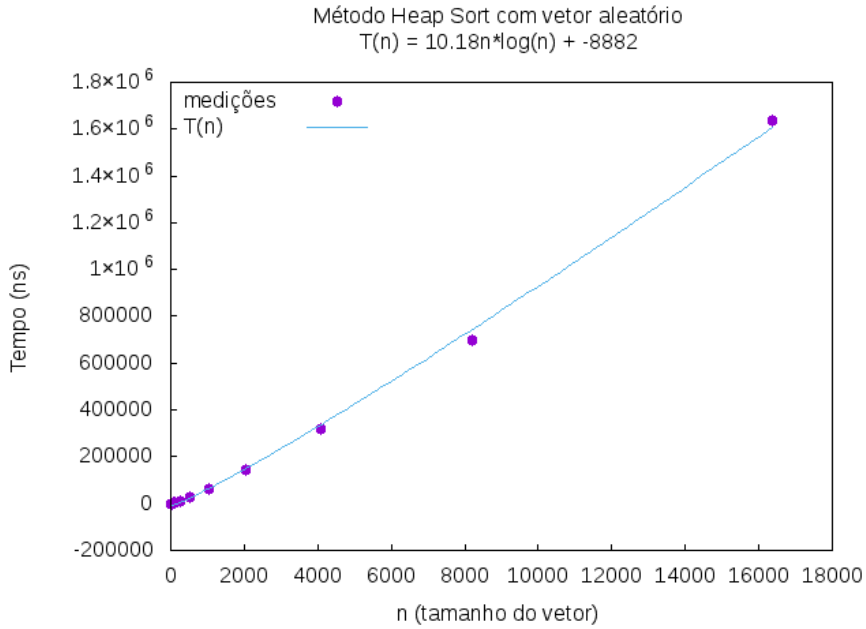

Figura 5.23: *MinMax - Vetor Decrescente P20*

5.2.11 Vetor Decrescente P30

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P30.

Tabela 5.24: *MinMax com vetor Decrescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	628
32	634
64	686
128	492
256	1042
512	806
1024	1087
2048	1250
4096	2139
8192	3898
16384	9144

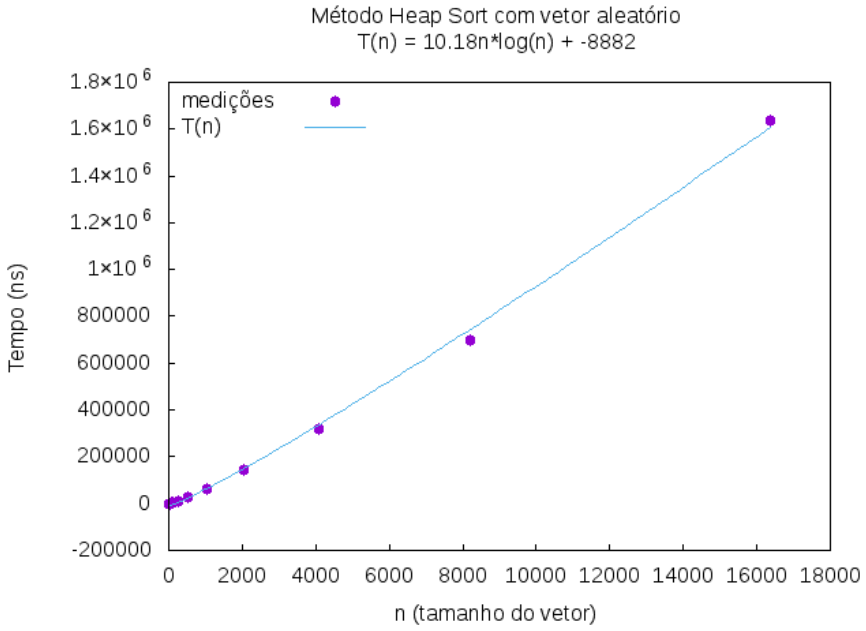

Figura 5.24: *MinMax - Vetor Decrescente P30*

5.2.12 Vetor Decrescente P40

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P40.

Tabela 5.25: *MinMax com vetor Decrescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	628
32	634
64	686
128	492
256	1042
512	806
1024	1087
2048	1250
4096	2139
8192	3898
16384	9144

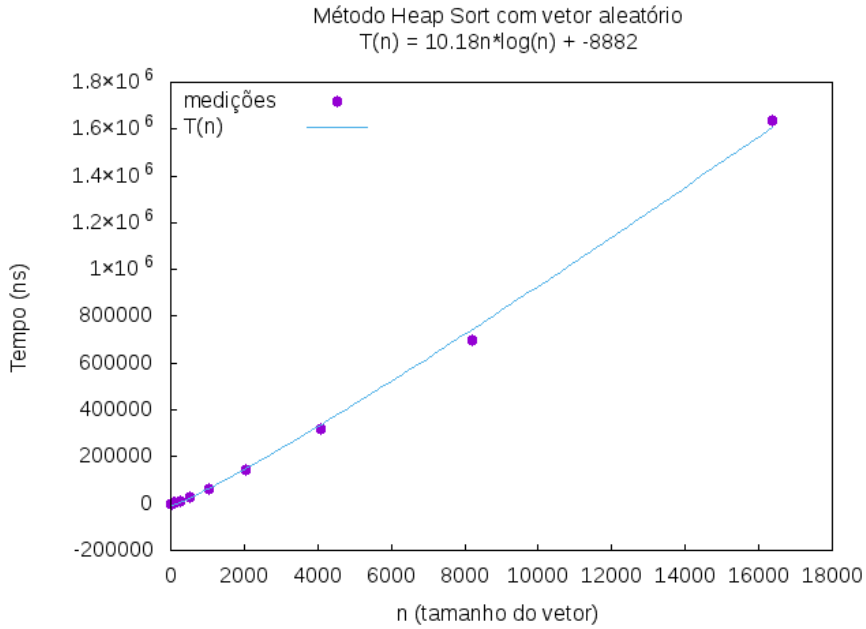

Figura 5.25: *MinMax - Vetor Decrescente P40*

5.2.13 Vetor Decrescente P50

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P50.

Tabela 5.26: *MinMax com vetor Decrescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	628
32	634
64	686
128	492
256	1042
512	806
1024	1087
2048	1250
4096	2139
8192	3898
16384	9144

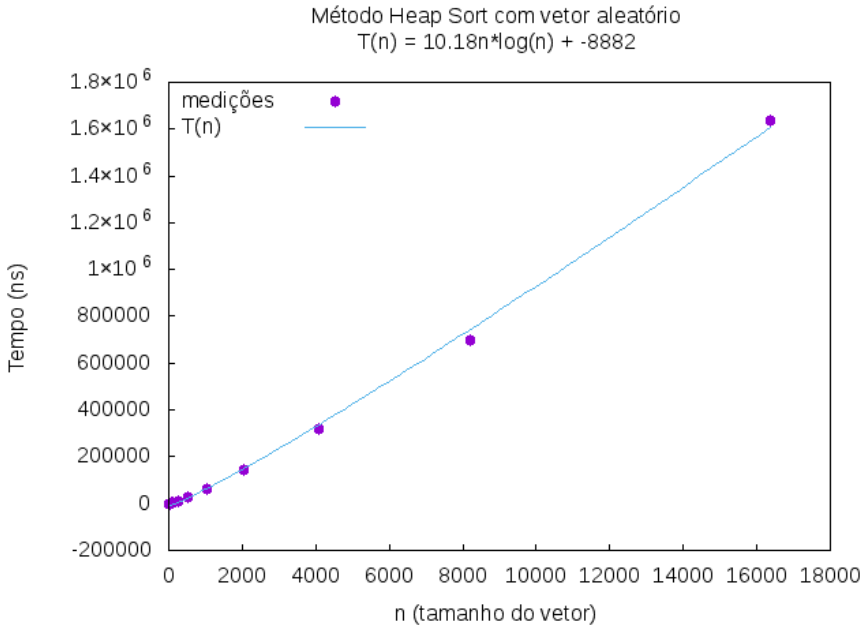

Figura 5.26: *MinMax - Vetor Decrescente P50*

Capítulo 6

Referências

Insertion Sort
Merge Sort
Heap Sort
Quick Sort
Counting Sort
Radix Sort
Bucket Sort
Introduction to algorithms 3rd Edition,Cormen, Thomas H,2009