Análise de Algoritmos - Ordenação

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

Faculdade de Computação
Universidade Federal de Uberlândia

28 de junho de 2017

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Sumário

Capítulo 1

Introdução

Este relatŕio tem como objetivo fazer a análise de diversos algoritmos já conhecidos de ordenação. O intúito desde 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 void ordena_intercala(int * v,int p,int r)
49 {
50     int q;
51     if (p < r) {
52         q = floor ((p + r) / 2); // retorna o chão dessa operação
53         ordena_intercala (v, p, q);
54         ordena_intercala(v, q + 1, r);
55         intercala(v, p, q, r);
56     }
57 }
58
59 void intercala(int * v,int p, int q, int r)
60 {
61     int *B = calloc((r+1),sizeof(int));
62     int i,k,j;
63     for (i = p; i<=q; i++)
64         B[i] = v[i];
65     for (j = (q + 1); j<= r;j++) {
66         B[(r + q + 1 - j)] = v[j];
67     }
68     i = p;
69     j = r;
70     for(k = p; k<=r; k++) {
71             if (B[i] <= B[j]) {
72                 v[k] = B[i];
73                 i++;
74             } else {
75                 v[k] = B[j];
76                 j--;
77             }
78     }
79   free(B);
80 }
81
82 void insertion(int *v, int tam)
83 {
84     int chave,i,j;
85     for(j=1;j<tam;j++)
86     {
87         chave = v[j];
88         i = j - 1;
89         while (i >= 0 && v[i] > chave)
90         {
91             v[i+1] = v[i];
92             i = i-1;
93         }
94   v[i+1] = chave;
95     }
96 }
97
98 void heap(int *a, int n) {
```

```
99      int i = n / 2, pai, filho, t;
100     for (;;) {
101         if (i > 0) {
102             i--;
103             t = a[i];
104         } else {
105             n--;
106             if (n == 0) return;
107             t = a[n];
108             a[n] = a[0];
109         }
110         pai = i;
111         filho = i * 2 + 1;
112         while (filho < n) {
113             if ((filho + 1 < n)  &&  (a[filho + 1] > a[filho]))
114                 filho++;
115             if (a[filho] > t) {
116                 a[pai] = a[filho];
117                 pai = filho;
118                 filho = pai * 2 + 1;
119             } else {
120                 break;
121             }
122         }
123         a[pai] = t;
124     }
125 }
126
127 void quick(int *vetor, int inicio, int fim){
128
129     int pivo, aux, i, j, meio;
130
131     i = inicio;
132     j = fim;
133
134     meio = (int) ((i + j) / 2);
135     pivo = vetor[meio];
136
137     do{
138         while (vetor[i] < pivo) i = i + 1;
139         while (vetor[j] > pivo) j = j - 1;
140
141         if(i <= j){
142             aux = vetor[i];
143             vetor[i] = vetor[j];
144             vetor[j] = aux;
145             i = i + 1;
146             j = j - 1;
147         }
148     }while(j > i);
149
150     if(inicio < j) quick(vetor, inicio, j);
151     if(i < fim) quick(vetor, i, fim);
152 }
153
154 void coutingsort(int *A, int tamanho){
155     int k = 10;
156     int aux;
157     int *C = (int*)calloc(k+1,sizeof(int));
```

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

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

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

Listagem 1.4: Automatização dos experimentos

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <time.h>
#include <float.h>
#include <math.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#include "vetor.h"
#include "ordena.h"

#define BILHAO 1000000000L

#define CRONOMETRA(funcao,vetor,n) {                                \
   clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &inicio);                \
   funcao(vetor,n);                                                 \
   clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &fim);                   \
   tempo_de_cpu_aux = BILHAO * (fim.tv_sec - inicio.tv_sec) +       \
                  fim.tv_nsec - inicio.tv_nsec;                     \
   }

int main(int argc, char *argv[]){
  int * v = NULL;
  int n = 0;
  uint64_t tempo_de_cpu_aux = 0;
  int tamanho = 0,count = 0;
  //clock_t inicio, fim;
  struct timespec inicio, fim;
  uint64_t tempo_de_cpu = 0.0;
  char msg[256];
  char nome_do_arquivo[128];
  char **arquivos;
  int k=0,h = 0;
  arquivos = (char**)malloc(200*sizeof(char*));
  for(int i=0;i<200;i++){
    arquivos[i] = (char*)malloc(128*sizeof(char));
  }

  for(int i=0;i<11;i++){
    sprintf(nome_do_arquivo,"vetores/vIntAleatorio_%d.dat",(int)pow(2,i
        +4%15));
    strcpy(arquivos[k], nome_do_arquivo);
    k++;
  }
  for(int i=0;i<11;i++){
    sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d.dat",(int)pow(2,i
        +4%15));
    strcpy(arquivos[k], nome_do_arquivo);
    k++;

  }
  for(int i=0;i<11;i++){
    sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d_P10.dat",(int)pow(2,
        i+4%15));
    strcpy(arquivos[k], nome_do_arquivo);
    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(radixsort, 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 programar 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

Insertion Sort

Insertion Sort é um algoritmo de ordenação que, dado uma estrutura array ou lista ele constrói uma matriz final com um elemento de cada vez, realizando uma inserção por vez. Insertion Sort é um algoritmo de ordenação quadrática, é bastante eficiente para problemas com pequenas entradas.Complexidade pior caso O(n^2) e no melhor caso O(n).

2.1 Insertion Sort - Vetor Aleatório

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

Tabela 2.1: *Insertion Sort com Vetor aleatório*

Número de Elementos	**Tempo de execução em nanosegundos**
16	592
32	623
64	1330
128	3921
256	13475
512	49717
1024	181720
2048	709142
4096	2818906
8192	11332358
16384	44220895

2.1.1 Gráfico Insertion sort - Vetor Aletório

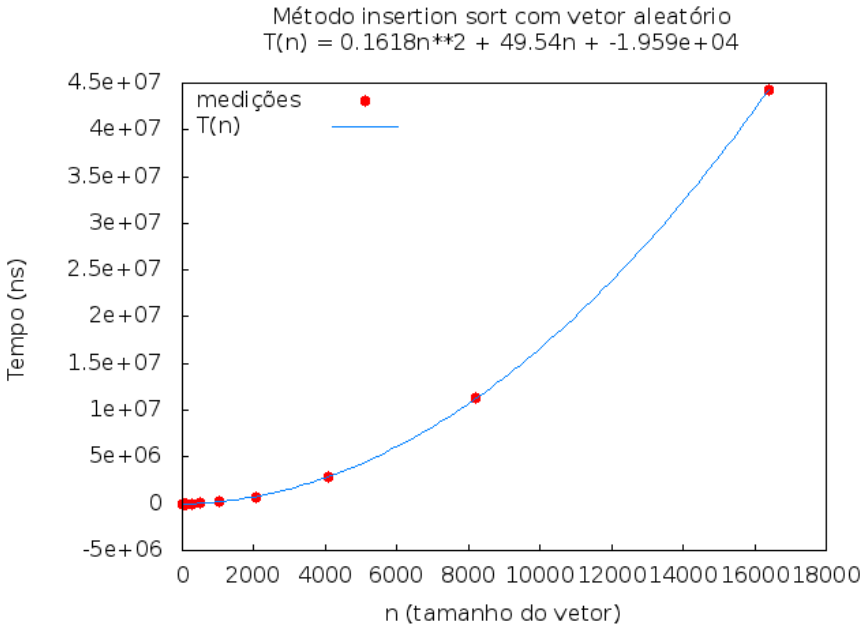

Figura 2.1: *Gráfico Insertion Sort - Vetor Aleatorio*

2.2 Insertion Sort - Vetor Crescente

Tabela gerada utilizando Insertion Sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem crescente.

Tabela 2.2: *Insertion Sort com Vetor ordenado em ordem crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	330
32	359
64	366
128	441
256	653
512	1151
1024	1616
2048	3006
4096	5551
8192	11105
16384	21993

2.2.1 Grafico Insertion Sort - Vetor Crescente

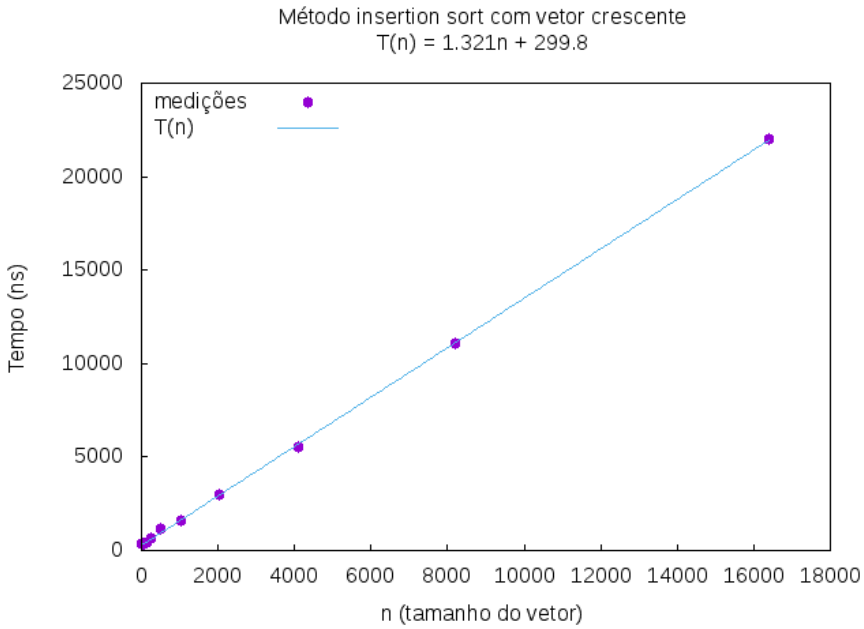

Figura 2.2: *Gráfico Insertion Sort - Vetor Crescente*

2.3 Insertion Sort - Vetor Crescente P10

Tabela gerada utilizando Insertion Sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem crescente estando 10% ordenado.

Tabela 2.3: *Insertion Sort com Vetor ordenado em ordem crescente 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	306
32	401
64	641
128	1477
256	4722
512	17262
1024	73246
2048	304783
4096	1140882
8192	4421923
16384	17366826

2.3.1 Grafico Insertion Sort - Vetor Crescente P10

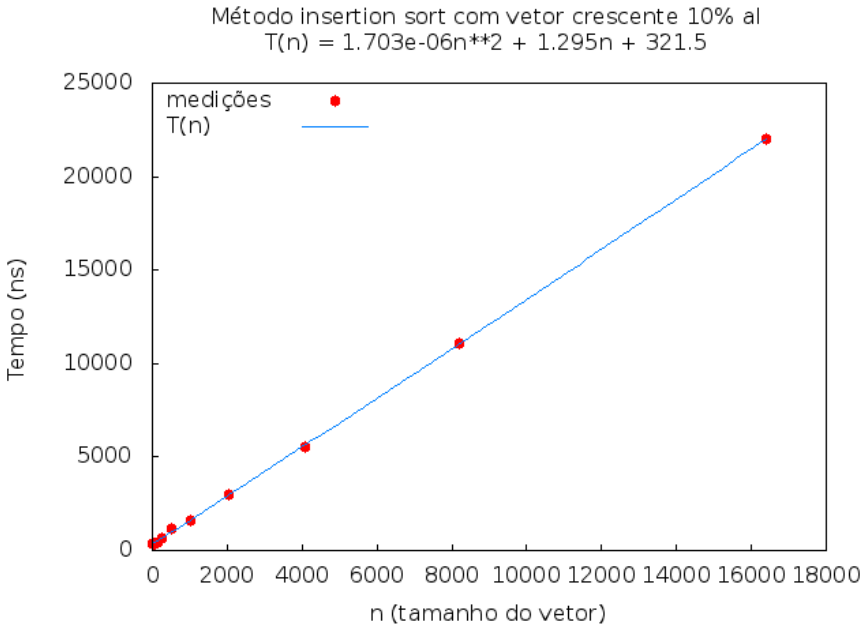

Figura 2.3: *Gráfico Insertion Sort - Vetor Crescente P10*

2.4 Insertion Sort - Vetor Crescente P20

Tabela gerada utilizando Insertion Sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem crescente estando 20% ordenado.

Tabela 2.4: *Insertion Sort com Vetor ordenado em ordem crescente 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	395
32	528
64	942
128	2441
256	8518
512	37766
1024	188395
2048	561783
4096	2024058
8192	8326582
16384	32720219

2.4.1 Grafico Insertion Sort - Vetor Crescente P20

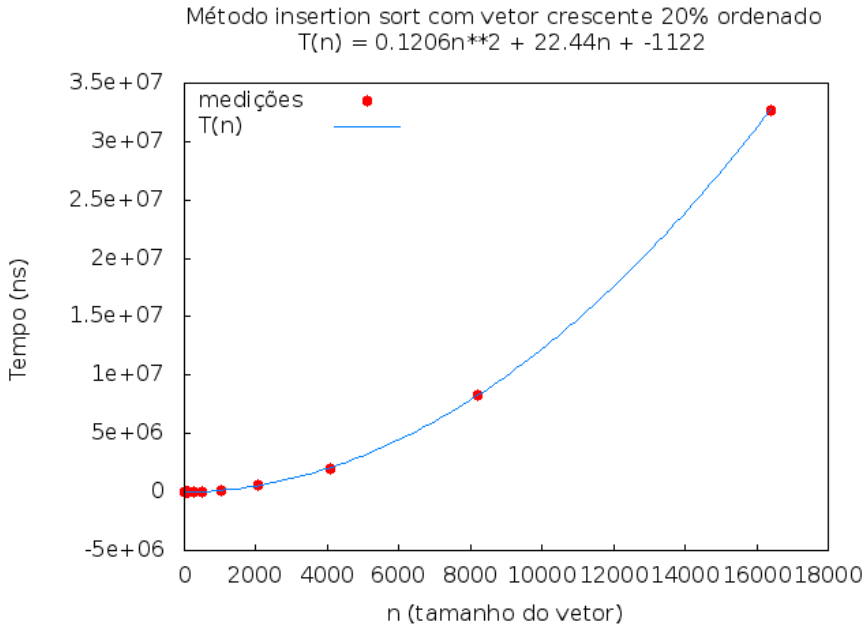

Figura 2.4: *Gráfico Insertion Sort - Vetor Crescente P20*

2.5 Insertion Sort - Vetor Crescente P30

Tabela gerada utilizando Insertion Sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem crescente estando 30% ordenado.

Tabela 2.5: *Insertion Sort com Vetor ordenado em ordem crescente 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	426
32	565
64	1121
128	3639
256	11877
512	47210
1024	188411
2048	747231
4096	2903599
8192	11706837
16384	46407450

2.5.1 Grafico Insertion Sort - Vetor Crescente P30

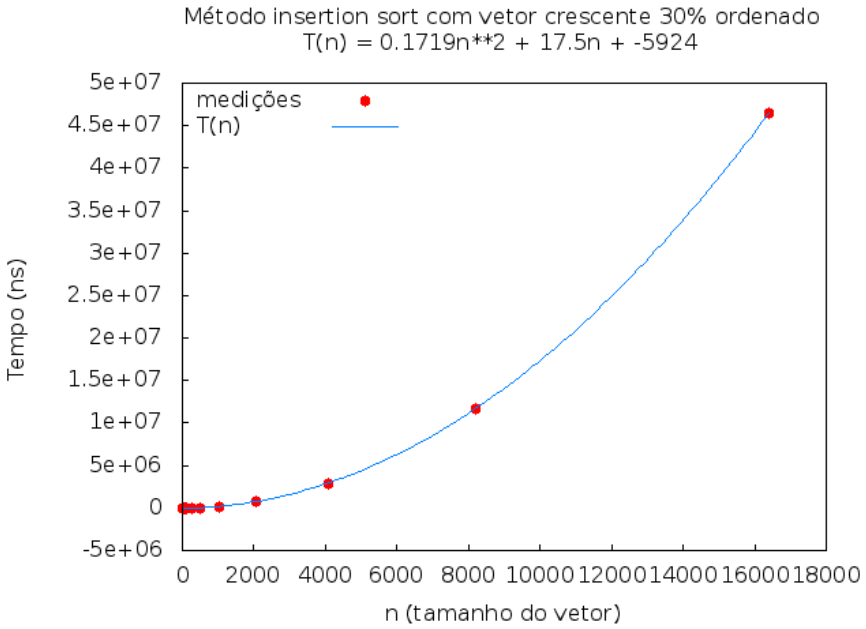

Figura 2.5: *Gráfico Insertion Sort - Vetor Crescente P30*

2.6 Insertion Sort - Vetor Crescente P40

Tabela gerada utilizando Insertion Sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem crescente estando 40% ordenado.

Tabela 2.6: *Insertion Sort com Vetor ordenado em ordem crescente 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	419
32	587
64	1308
128	4090
256	15835
512	65553
1024	224888
2048	916018
4096	3680777
8192	14947212
16384	59251527

2.6.1 Grafico Insertion Sort - Vetor Crescente P40

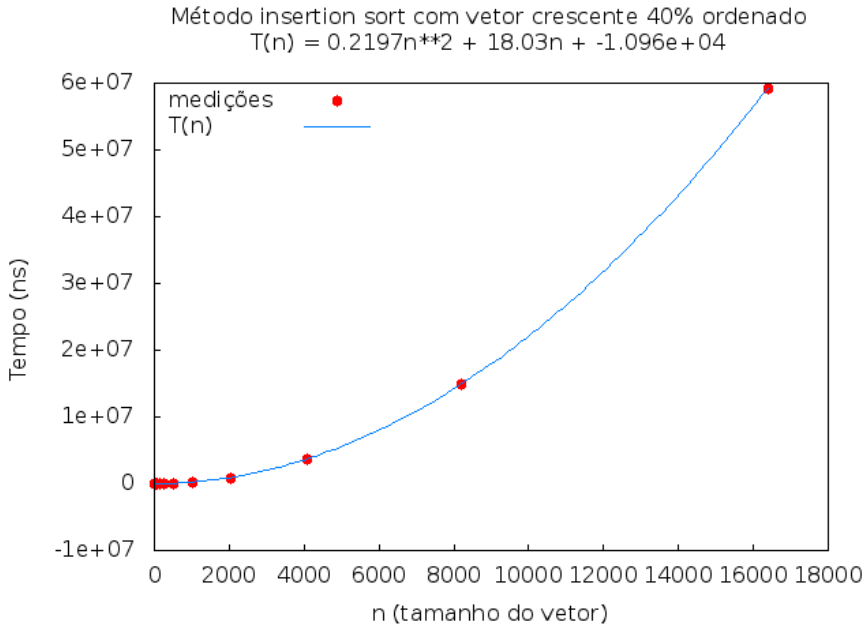

Figura 2.6: *Gráfico Insertion Sort - Vetor Crescente P40*

2.7 Insertion Sort - Vetor Crescente P50

Tabela gerada utilizando Insertion Sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem crescente estando 50% ordenado.

Tabela 2.7: *Insertion Sort com Vetor ordenado em ordem crescente 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	558
32	659
64	1513
128	17337
256	18190
512	67937
1024	262668
2048	1100335
4096	4360290
8192	17429716
16384	67965063

2.7.1 Grafico Insertion Sort - Vetor Crescente P50

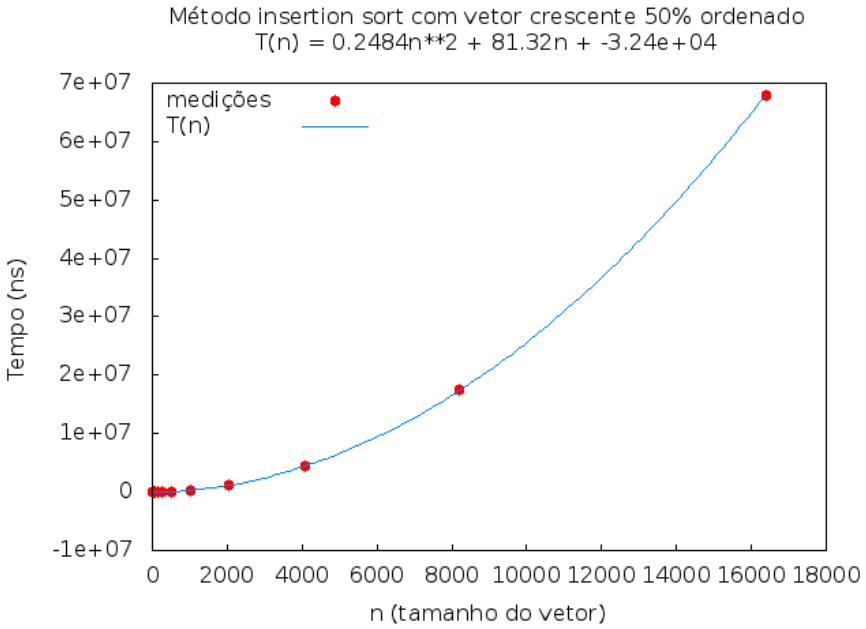

Figura 2.7: *Gráfico Insertion Sort - Vetor Crescente P50*

2.8 Insertion Sort - Vetor Decrescente

Tabela gerada utilizando Insertion Sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem decrescente.

Tabela 2.8: *Insertion Sort com Vetor ordenado em ordem decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	755
32	874
64	2328
128	7374
256	24763
512	93192
1024	348188
2048	1395270
4096	5750603
8192	24268503
16384	95007321

2.8.1 Grafico Insertion Sort - Vetor Decrescente

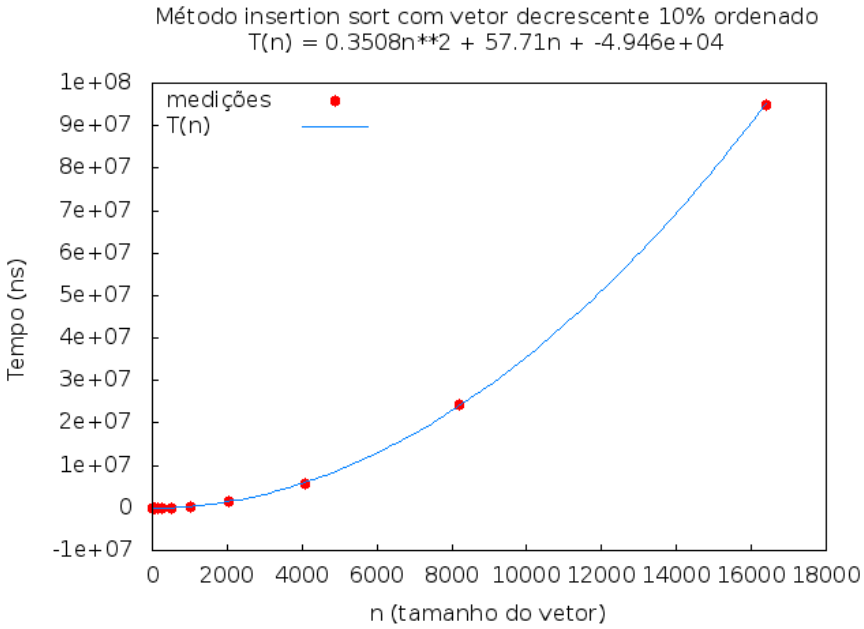

Figura 2.8: *Gráfico Insertion Sort - Vetor Decrescente*

2.9 Insertion Sort - Vetor Decrescente P10

Tabela gerada utilizando Insertion Sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem descrescente estando 10% ordenado.

Tabela 2.9: *Insertion Sort com Vetor ordenado em ordem descrecente 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	458
32	875
64	1835
128	5721
256	19634
512	73075
1024	296395
2048	1193812
4096	4766396
8192	19593295
16384	77500644

2.9.1 Grafico Insertion Sort - Vetor Decrescente P10

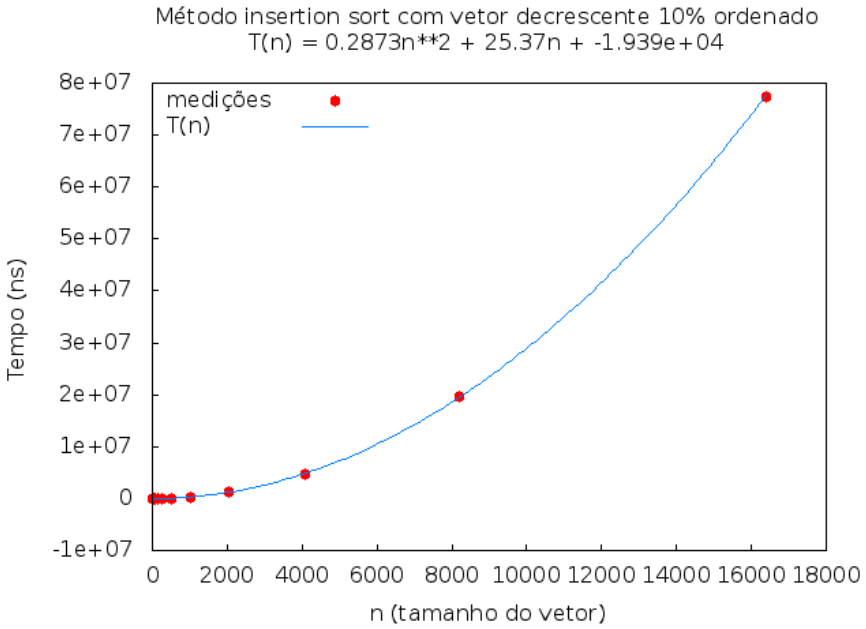

Figura 2.9: *Gráfico Insertion Sort - Vetor Decrescente P10*

2.10 Insertion Sort - Vetor Decrescente P20

Tabela gerada utilizando Insertion Sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem descrescente estando 20% ordenado.

Tabela 2.10: *Insertion Sort com Vetor ordenado em ordem descrecente 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	486
32	667
64	1535
128	4529
256	15714
512	58010
1024	230754
2048	906191
4096	4298422
8192	15435783
16384	59268139

2.10.1 Grafico Insertion Sort - Vetor Decrescente P20

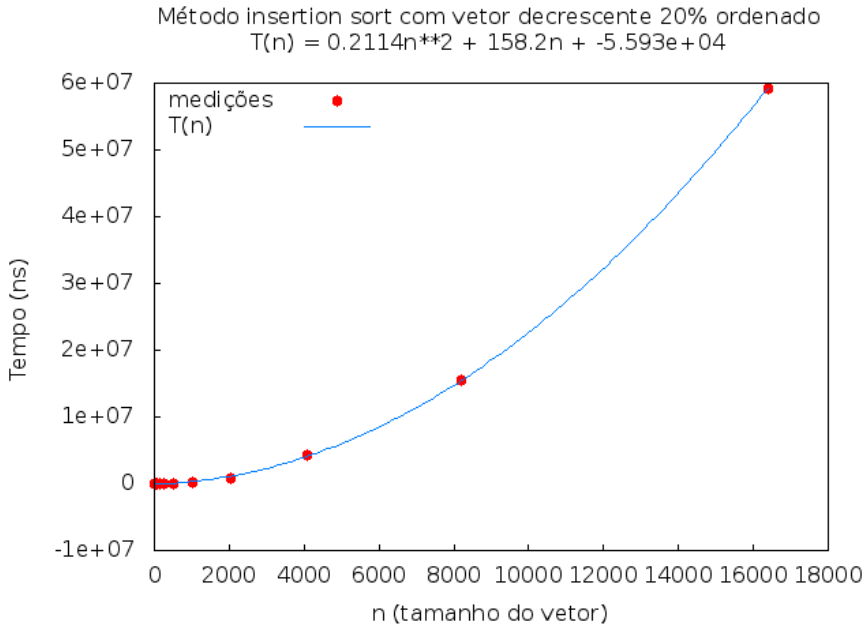

Figura 2.10: *Gráfico Insertion Sort - Vetor Decrescente P20*

2.11 Insertion Sort - Vetor Decrescente P30

Tabela gerada utilizando Insertion Sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem descrescente estando 30% ordenado.

Tabela 2.11: *Insertion Sort com Vetor ordenado em ordem descrecente 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	472
32	618
64	1315
128	3584
256	12212
512	45720
1024	176462
2048	713725
4096	2722869
8192	11098439
16384	43472556

2.11.1 Grafico Insertion Sort - Vetor Decrescente P30

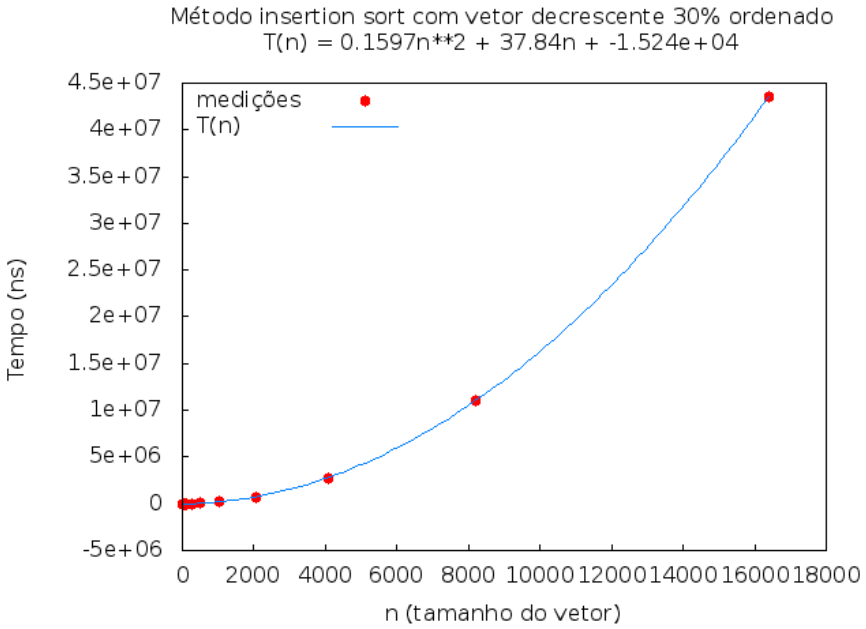

Figura 2.11: *Gráfico Insertion Sort - Vetor Decrescente P30*

2.12 Insertion Sort - Vetor Decrescente P40

Tabela gerada utilizando Insertion Sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem descrescente estando 40% ordenado.

Tabela 2.12: *Insertion Sort com Vetor ordenado em ordem descrecente 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	455
32	617
64	1161
128	5847
256	8949
512	32581
1024	135069
2048	503857
4096	2004770
8192	8252574
16384	31871258

2.12.1 Grafico Insertion Sort - Vetor Decrescente P40

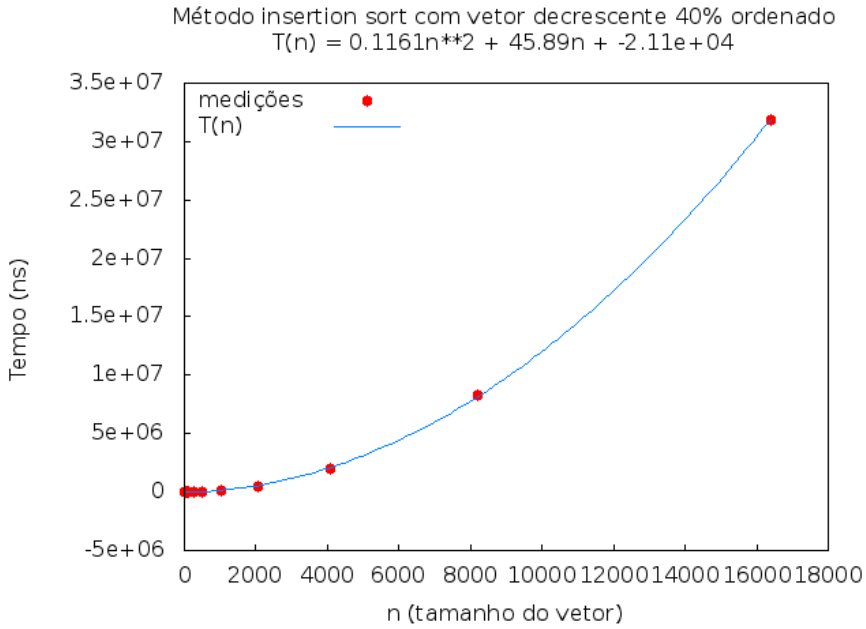

Figura 2.12: *Gráfico Insertion Sort - Vetor Decrescente P40*

2.13 Insertion Sort - Vetor Decrescente P50

Tabela gerada utilizando Insertion Sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem descrescente estando 50% ordenado.

Tabela 2.13: *Insertion Sort com Vetor ordenado em ordem descrecente 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	409
32	461
64	847
128	2144
256	6683
512	23838
1024	89838
2048	352101
4096	1389637
8192	5639428
16384	22348219

2.13.1 Grafico Insertion Sort - Vetor Decrescente P50

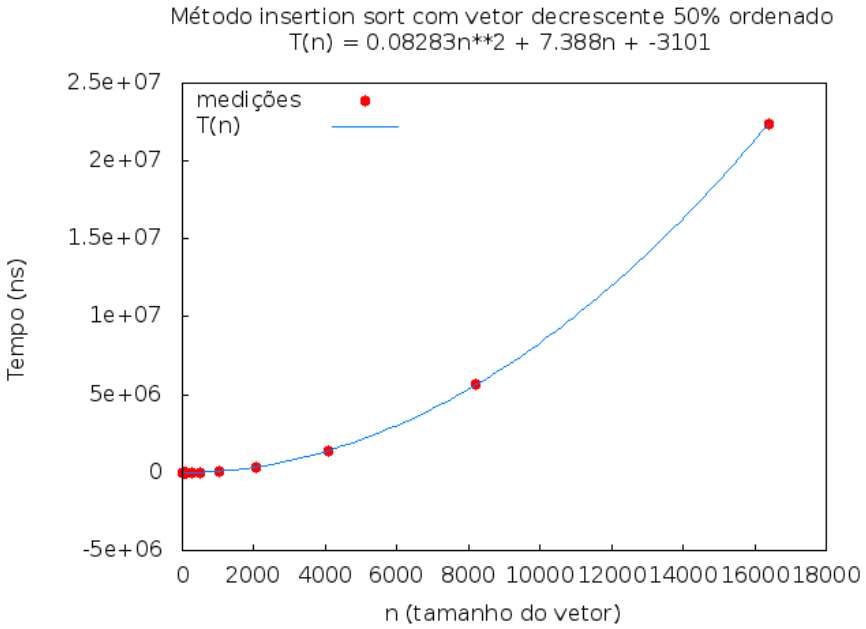

Figura 2.13: *Gráfico Insertion Sort - Vetor Decrescente P50*

2.14 Observações Finais

Insertion em ordem decrescente com 2^k com k = 32 elementos: levaria aproximadamente 205 anos e 22 dias

Capítulo 3

Merge Sort

O algoritmo de ordenação Merge Sort (Intercalação) ordena por comparações utilizando o conceito de dividir-para-conquistar. Basicamente o algoritmo divide o problema em vários sub-problemas e os resolve através de recursividade e conquista pegando o unindo todos os sub-problemas que foram resolvidos. Em qualquer caso, temos que o algoritmo MergeSort tem complexidade de tempo $\theta(nlogn)$.

3.1 Merge Sort - Vetor Aleatório

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

Tabela 3.1: *Merge Sort com vetor aleatório*

Número de Elementos	**Tempo de execução em nanosegundos**
16	2761
32	3438
64	7886
128	16043
256	33294
512	75211
1024	186557
2048	477554
4096	1278453
8192	4144242
16384	18189873

3.1.1 Gráfico Merge Sort - Vetor Aleatório

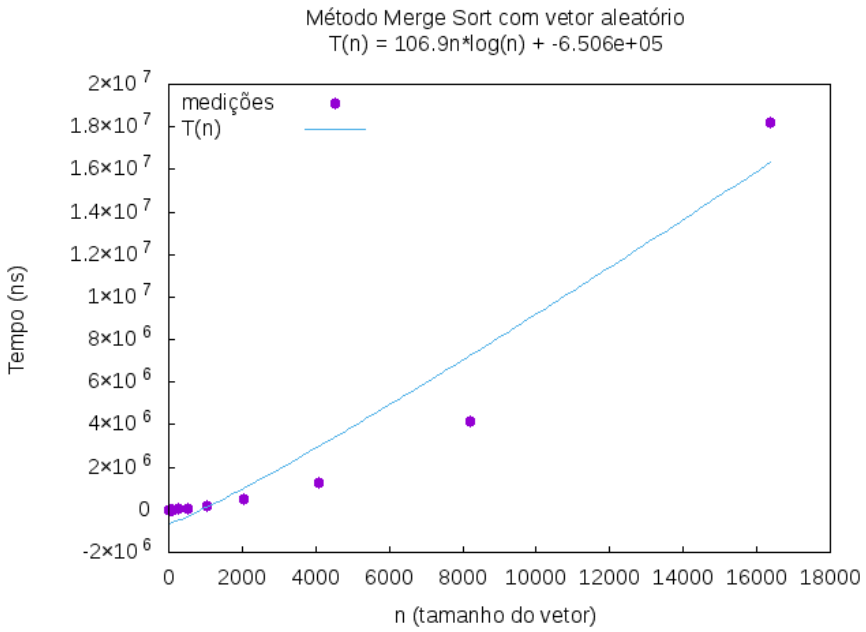

Figura 3.1: *Gráfico Merge Sort - Vetor Aleatório*

3.2 Merge Sort - Vetor Crescente

Tabela gerada utilizando Merge Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente.

Tabela 3.2: *Merge Sort com vetor ordenado em ordem crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	1840
32	3284
64	6768
128	13027
256	26564
512	59916
1024	144530
2048	368955
4096	1095195
8192	3812048
16384	17545434

3.2.1 Gráfico Merge Sort - Vetor Crescente

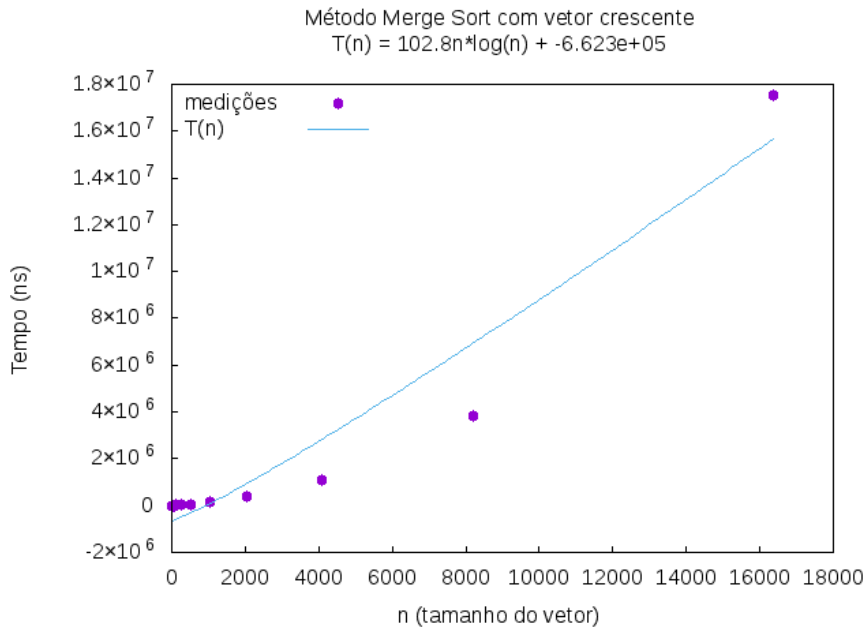

Figura 3.2: *Gráfico Merge Sort - Vetor Crescente*

3.3 Merge Sort - Vetor Crescente P10

Tabela gerada utilizando Merge Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 10% ordenado.

Tabela 3.3: *Merge Sort com vetor ordenado em ordem crescente estando 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1732
32	3328
64	6776
128	12591
256	30350
512	65121
1024	150315
2048	367101
4096	1105073
8192	3896594
16384	17300648

3.3.1 Gráfico Merge Sort - Vetor Crescente P10

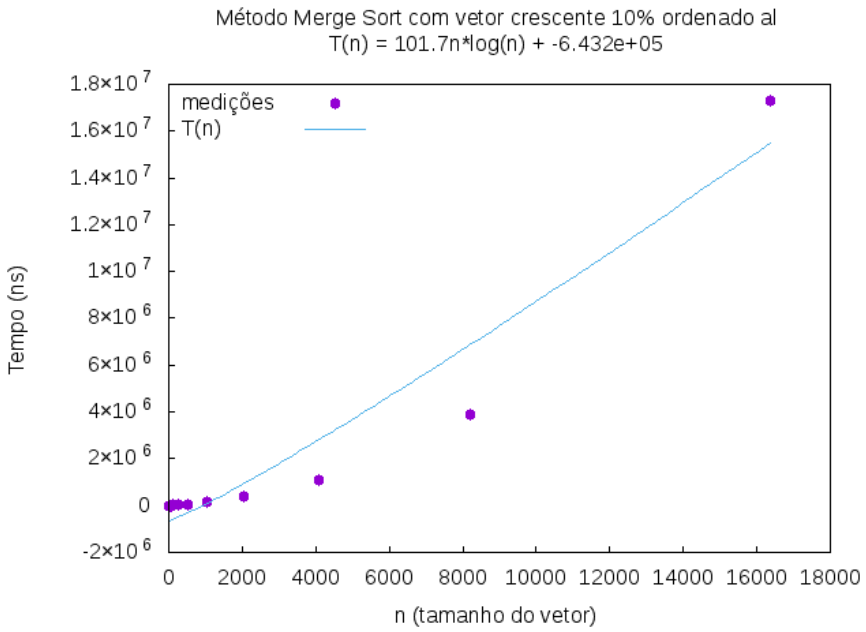

Figura 3.3: *Gráfico Merge Sort - Vetor Crescente P10*

3.4 Merge Sort - Vetor Crescente P20

Tabela gerada utilizando Merge Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 20% ordenado.

Tabela 3.4: *Merge Sort com vetor ordenado em ordem crescente estando 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1789
32	3337
64	6866
128	12755
256	29327
512	63476
1024	148528
2048	367361
4096	1072801
8192	3846028
16384	17413999

3.4.1 Gráfico Merge Sort - Vetor Crescente P20

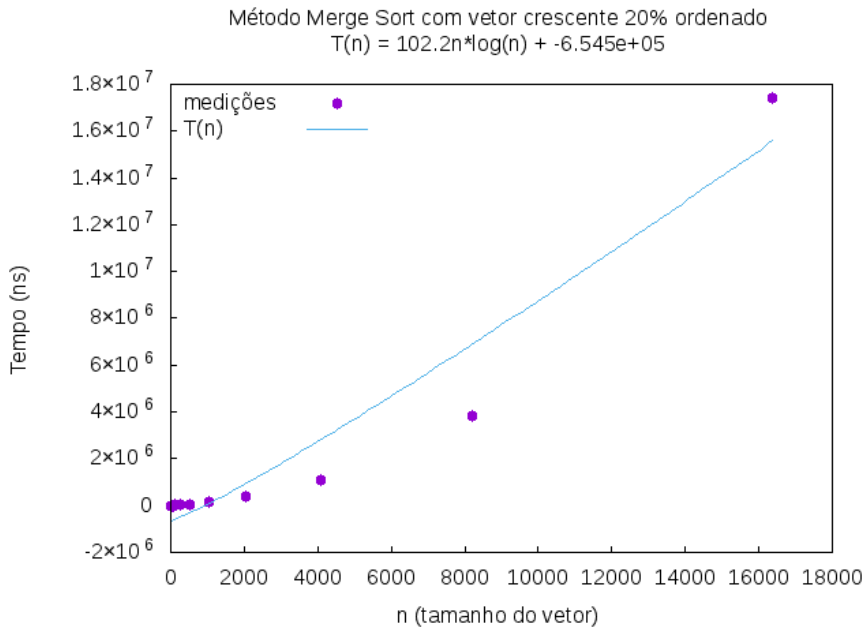

Figura 3.4: *Gráfico Merge Sort - Vetor Crescente P20*

3.5 Merge Sort - Vetor Crescente P30

Tabela gerada utilizando Merge Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 30% ordenado.

Tabela 3.5: *Merge Sort com vetor ordenado em ordem crescente estando 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	2042
32	3304
64	6831
128	14759
256	28846
512	62768
1024	147870
2048	362086
4096	1072534
8192	3822915
16384	17393637

3.5.1 Gráfico Merge Sort - Vetor Crescente P30

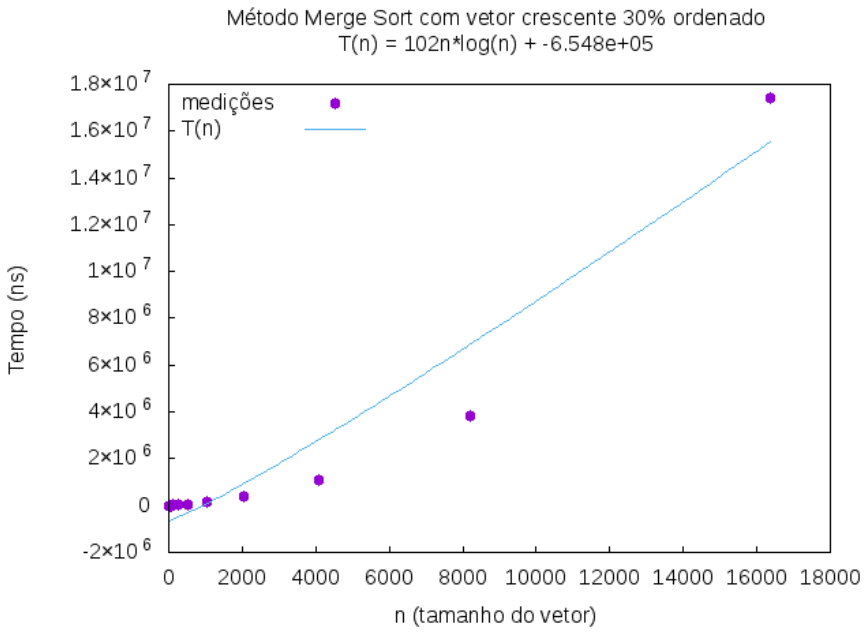

Figura 3.5: *Gráfico Merge Sort - Vetor Crescente P30*

3.6 Merge Sort - Vetor Crescente P40

Tabela gerada utilizando Merge Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 40% ordenado.

Tabela 3.6: *Merge Sort com vetor ordenado em ordem crescente estando 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1951
32	3272
64	6625
128	12620
256	28417
512	66034
1024	151603
2048	366825
4096	1077138
8192	3950780
16384	17461202

3.6.1 Gráfico Merge Sort - Vetor Crescente P40

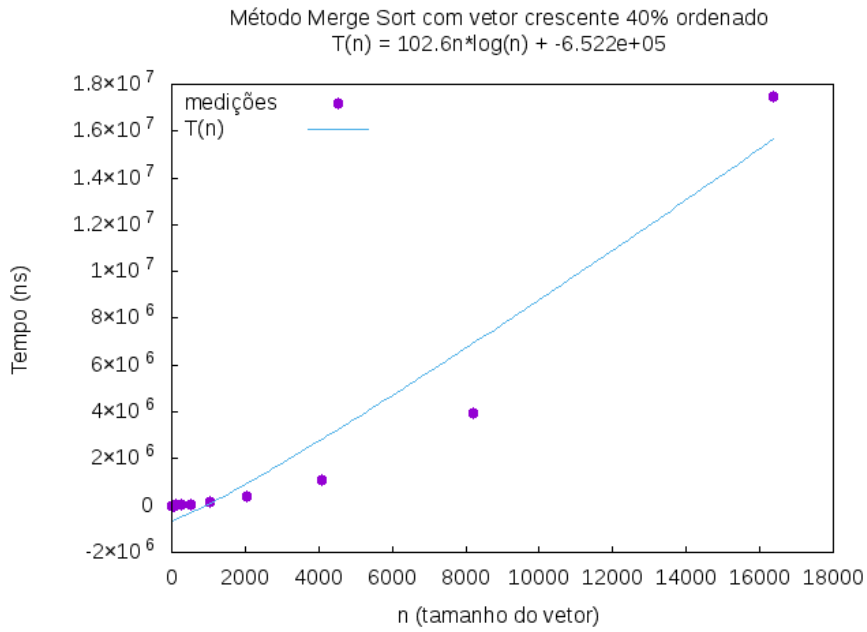

Figura 3.6: *Gráfico Merge Sort - Vetor Crescente P40*

3.7 Merge Sort - Vetor Crescente P50

Tabela gerada utilizando Merge Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 50% ordenado.

Tabela 3.7: *Merge Sort com vetor ordenado em ordem crescente estando 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1850
32	3506
64	6465
128	12959
256	28681
512	62483
1024	146989
2048	365252
4096	1076419
8192	3951343
16384	17351729

3.7.1 Gráfico Merge Sort - Vetor Crescente P50

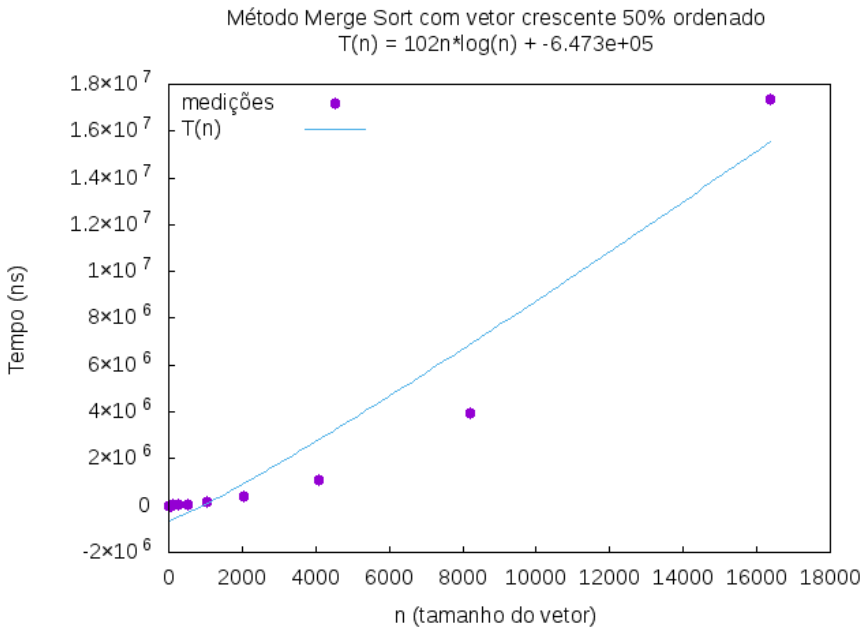

Figura 3.7: *Gráfico Merge Sort - Vetor Crescente P50*

3.8 Merge Sort - Vetor Decrescente

Tabela gerada utilizando Merge Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente.

Tabela 3.8: *Merge Sort com vetor ordenado em ordem decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	1969
32	3171
64	6247
128	12520
256	27287
512	83363
1024	145250
2048	369093
4096	1066356
8192	3968570
16384	17262377

3.8.1 Gráfico Merge Sort - Vetor Decrescente

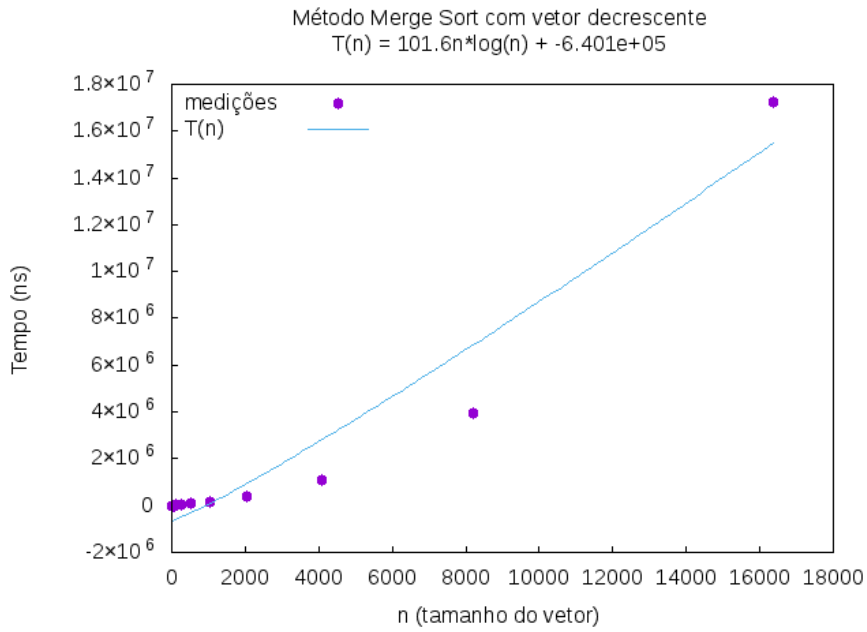

Figura 3.8: *Gráfico Merge Sort - Vetor Decrescente*

3.9 Merge Sort - Vetor Decrescente P10

Tabela gerada utilizando Merge Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 10% ordenado.

Tabela 3.9: *Merge Sort com vetor ordenado em ordem decrescente estando 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1915
32	3234
64	6637
128	12565
256	33715
512	62647
1024	157201
2048	363791
4096	1069486
8192	3941973
16384	17409501

3.9.1 Gráfico Merge Sort - Vetor Decrescente P10

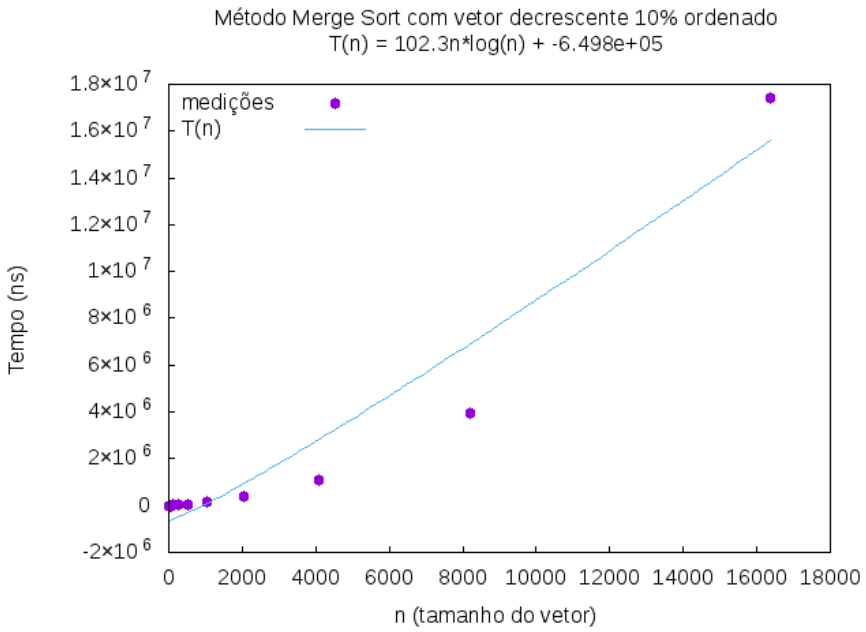

Figura 3.9: *Gráfico Merge Sort - Vetor Decrescente P10*

3.10 Merge Sort - Vetor Decrescente P20

Tabela gerada utilizando Merge Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 20% ordenado.

Tabela 3.10: *Merge Sort com vetor ordenado em ordem decrescente estando 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	2093
32	3162
64	6737
128	12671
256	34914
512	62345
1024	151220
2048	373987
4096	1082704
8192	3849662
16384	17495764

3.10.1 Gráfico Merge Sort - Vetor Decrescente P20

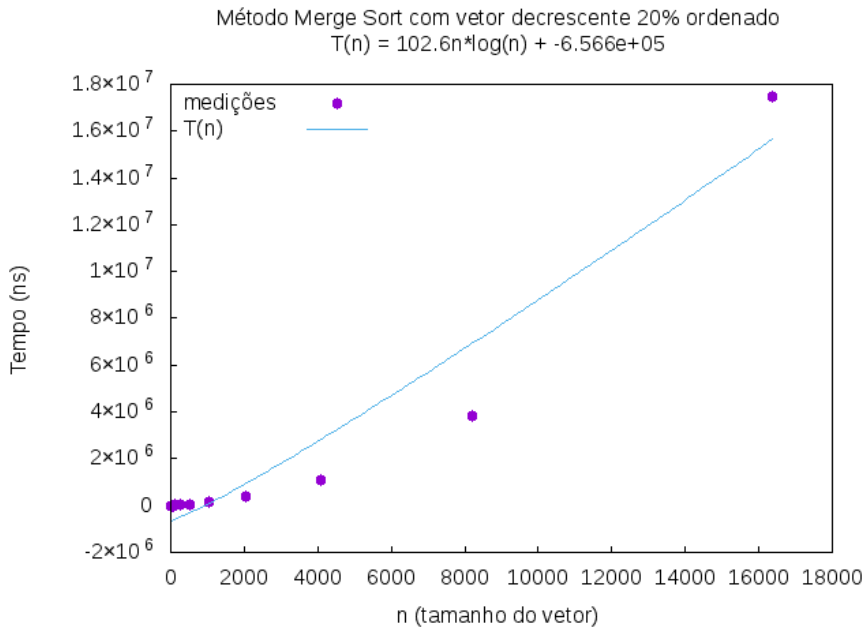

Figura 3.10: *Gráfico Merge Sort - Vetor Decrescente P20*

3.11 Merge Sort - Vetor Decrescente P30

Tabela gerada utilizando Merge Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 30% ordenado.

Tabela 3.11: *Merge Sort com vetor ordenado em ordem decrescente estando 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	2072
32	3361
64	6522
128	12645
256	28527
512	62502
1024	169367
2048	365185
4096	1076239
8192	3899494
16384	17188399

3.11.1 Gráfico Merge Sort - Vetor Decrescente P30

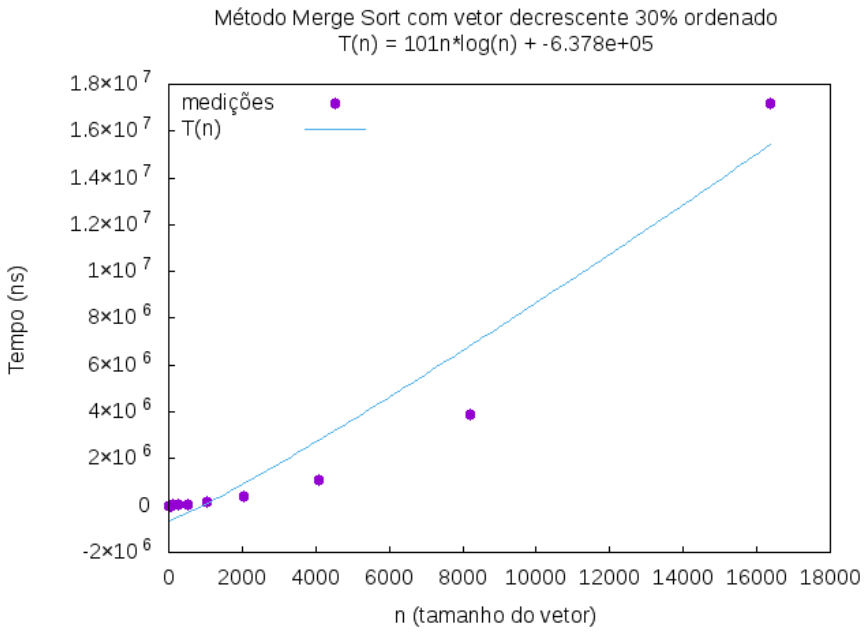

Figura 3.11: *Gráfico Merge Sort - Vetor Decrescente P30*

3.12 Merge Sort - Vetor Decrescente P40

Tabela gerada utilizando Merge Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 40% ordenado.

Tabela 3.12: *Merge Sort com vetor ordenado em ordem decrescente estando 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	2161
32	3295
64	6402
128	13101
256	27715
512	62184
1024	147490
2048	367866
4096	1080496
8192	3878299
16384	17404245

3.12.1 Gráfico Merge Sort - Vetor Decrescente P40

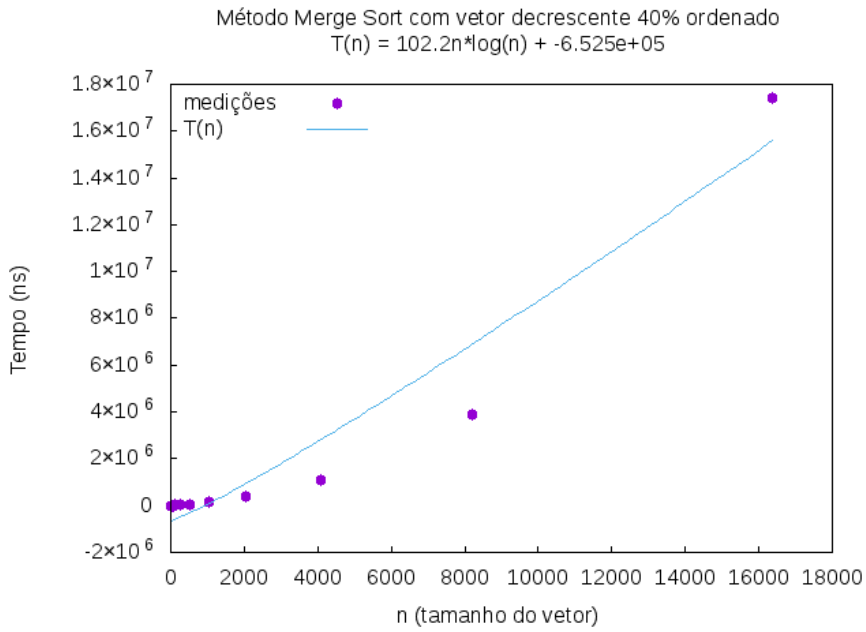

Figura 3.12: *Gráfico Merge Sort - Vetor Decrescente P40*

3.13 Merge Sort - Vetor Decrescente P50

Tabela gerada utilizando Merge Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 50% ordenado.

Tabela 3.13: *Merge Sort com vetor ordenado em ordem decrescente estando 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	2166
32	3258
64	6530
128	12686
256	28095
512	65430
1024	151006
2048	365972
4096	1070690
8192	3858348
16384	17428128

3.13.1 Gráfico Merge Sort - Vetor Decrescente P50

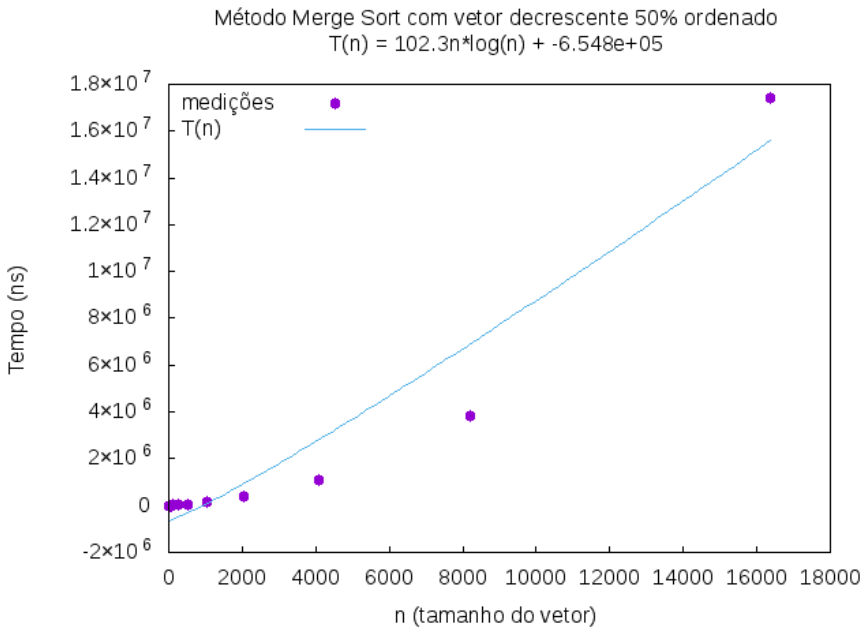

Figura 3.13: *Gráfico Merge Sort - Vetor Decrescente P50*

3.14 Observações Finais

Merge sort com vetor de elementos totalmente aleatório levaria aproximadamente 2 horas e 49 minutos para processar um vetor de 2^k com k = 32 elementos nessas condições.

Capítulo 4

Heap Sort

O algoritmo de ordenação Heap Sort é um algoritmo não estável. Ele faz uso de uma estrutura chamadas heap e ordena os elementos a medida que insere na estrutura, assim ao terminar as inserções, os elementos podem ser removidos da raíz sucessivamente na ordem desejada, mantendo a propriedade de max-heap(ou min-heap). Ele basicamente funciona montando uma árvore binária. Em qualquer caso, temos que o algoritmo HeapSort tem complexidade de tempo $\theta(nlogn)$.

4.1 Heap Sort - Vetor Aleatório

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

Tabela 4.1: *Heap Sort com vetor aleatório*

Número de Elementos	**Tempo de execução em nanosegundos**
16	725
32	971
64	2123
128	5011
256	10633
512	27914
1024	65078
2048	145462
4096	318047
8192	697040
16384	1635985

4.1.1 Gráfico Heap Sort - Vetor Aleatório

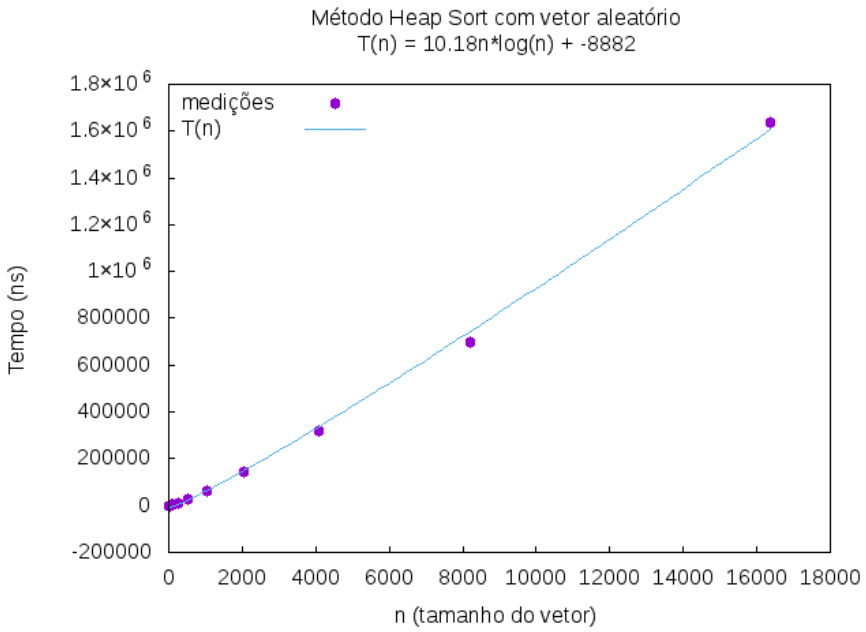

Figura 4.1: *Gráfico Heap Sort - Vetor Aleatório*

4.2 Heap Sort - Vetor Crescente

Tabela gerada utilizando Heap Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente.

Tabela 4.2: *Heap Sort com vetor ordenado em ordem crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	842
32	1075
64	1668
128	9925
256	10534
512	26131
1024	57088
2048	124606
4096	218104
8192	499002
16384	959418

4.2.1 Gráfico Heap Sort - Vetor Crescente

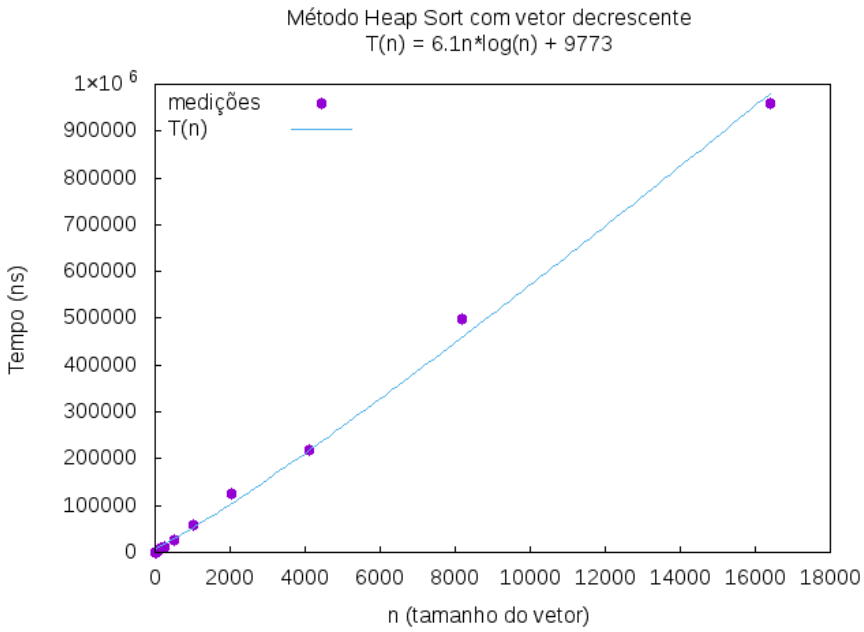

Figura 4.2: *Gráfico Heap Sort - Vetor Crescente*

4.3 Heap Sort - Vetor Crescente P10

Tabela gerada utilizando Heap Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 10% ordenado.

Tabela 4.3: *Heap Sort com vetor ordenado em ordem crescente estando 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	573
32	792
64	1669
128	3719
256	9700
512	25058
1024	55952
2048	121179
4096	238053
8192	451335
16384	946165

4.3.1 Gráfico Heap Sort - Vetor Crescente P10

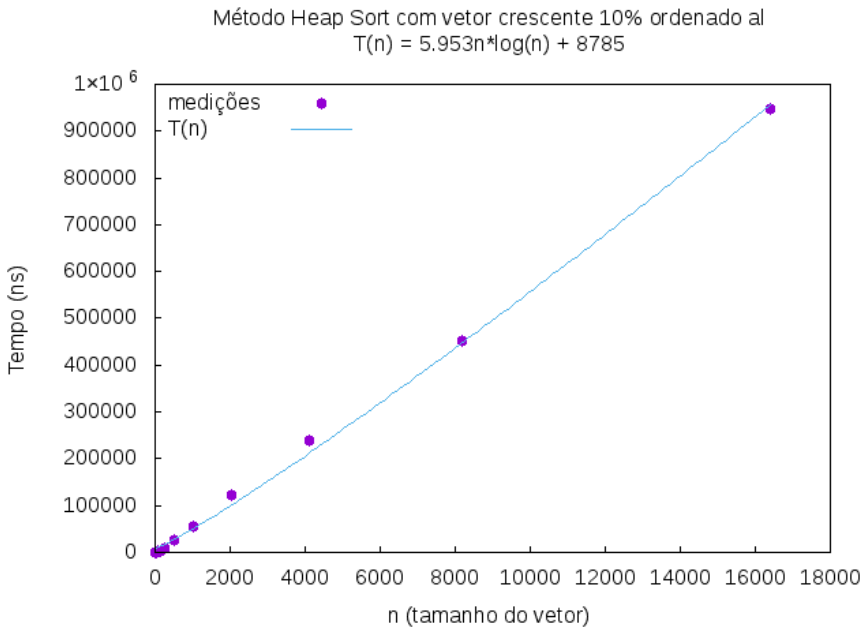

Figura 4.3: *Gráfico Heap Sort - Vetor Crescente P10*

4.4 Heap Sort - Vetor Crescente P20

Tabela gerada utilizando Heap Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 20% ordenado.

Tabela 4.4: *Heap Sort com vetor ordenado em ordem crescente estando 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	534
32	875
64	1628
128	4073
256	9612
512	25183
1024	60872
2048	127670
4096	228395
8192	472599
16384	957241

4.4.1 Gráfico Heap Sort - Vetor Crescente P20

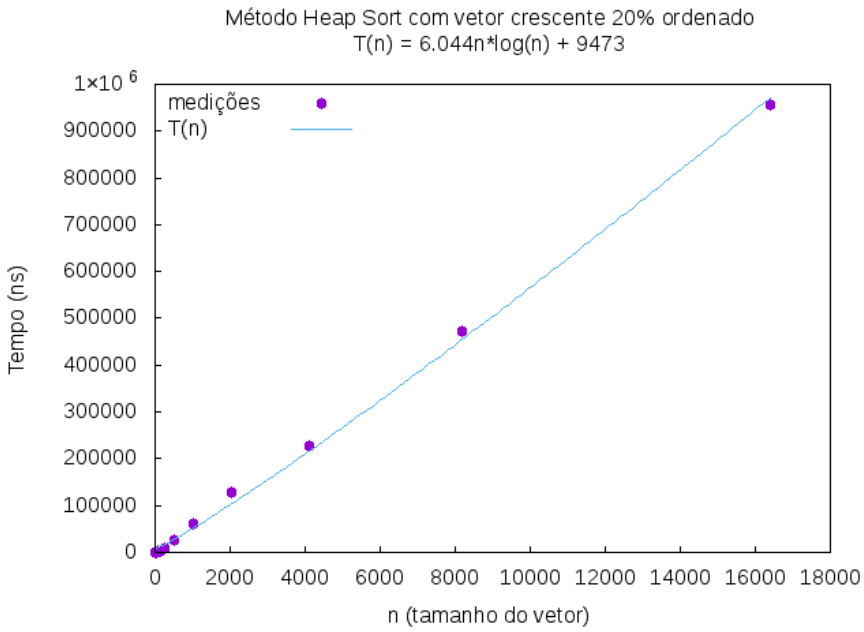

Figura 4.4: *Gráfico Heap Sort - Vetor Crescente P20*

4.5 Heap Sort - Vetor Crescente P30

Tabela gerada utilizando Heap Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 30% ordenado.

Tabela 4.5: *Heap Sort com vetor ordenado em ordem crescente estando 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	510
32	862
64	1630
128	3750
256	9836
512	24383
1024	65255
2048	122553
4096	241699
8192	493026
16384	954964

4.5.1 Gráfico Heap Sort - Vetor Crescente P30

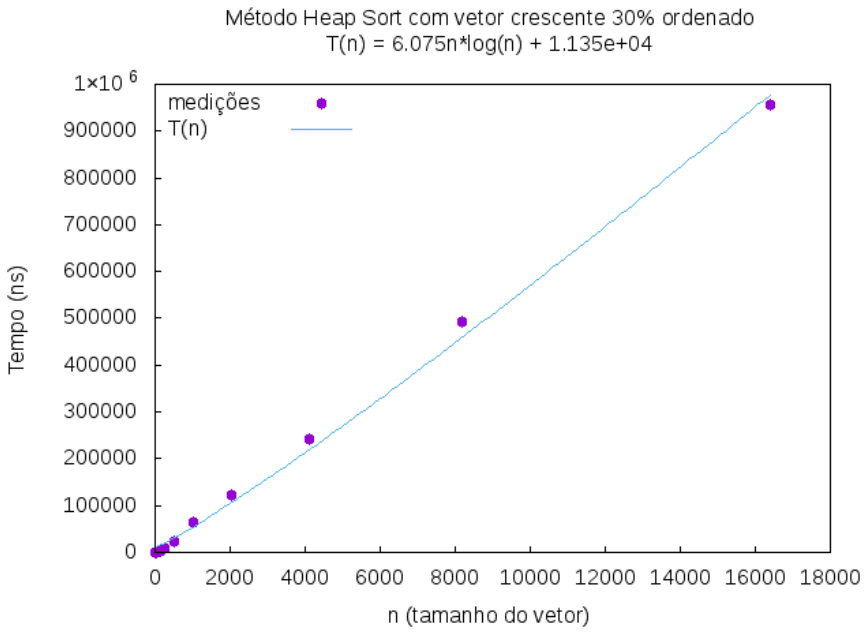

Figura 4.5: *Gráfico Heap Sort - Vetor Crescente P30*

4.6 Heap Sort - Vetor Crescente P40

Tabela gerada utilizando Heap Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 40% ordenado.

Tabela 4.6: *Heap Sort com vetor ordenado em ordem crescente estando 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	501
32	786
64	1799
128	9422
256	10202
512	25220
1024	55734
2048	112945
4096	233312
8192	479594
16384	947705

4.6.1 Gráfico Heap Sort - Vetor Crescente P40

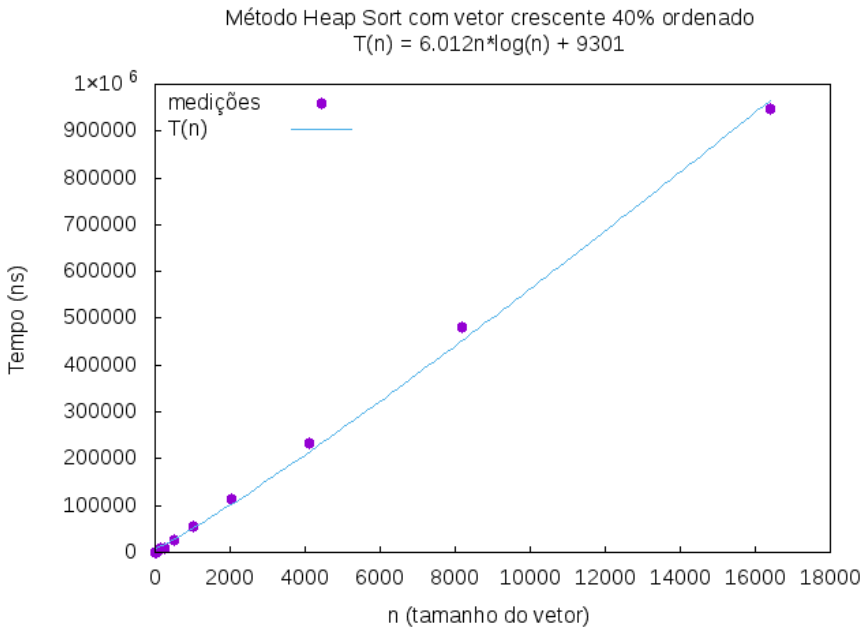

Figura 4.6: *Gráfico Heap Sort - Vetor Crescente P40*

4.7 Heap Sort - Vetor Crescente P50

Tabela gerada utilizando Heap Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 50% ordenado.

Tabela 4.7: *Heap Sort com vetor ordenado em ordem crescente estando 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	546
32	782
64	1652
128	3627
256	16960
512	27534
1024	67128
2048	114637
4096	266050
8192	477692
16384	960498

4.7.1 Gráfico Heap Sort - Vetor Crescente P50

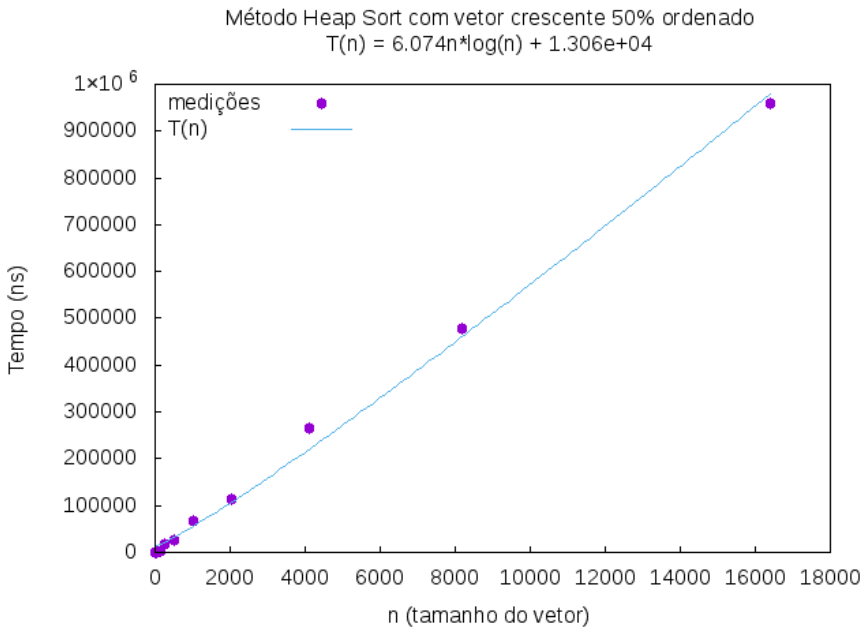

Figura 4.7: *Gráfico Heap Sort - Vetor Crescente P50*

4.8 Heap Sort - Vetor Decrescente

Tabela gerada utilizando Heap Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente.

Tabela 4.8: *Heap Sort com vetor ordenado em ordem decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	497
32	723
64	1485
128	3515
256	9684
512	23405
1024	55215
2048	121471
4096	238966
8192	492005
16384	1021937

4.8.1 Gráfico Heap Sort - Vetor Decrescente

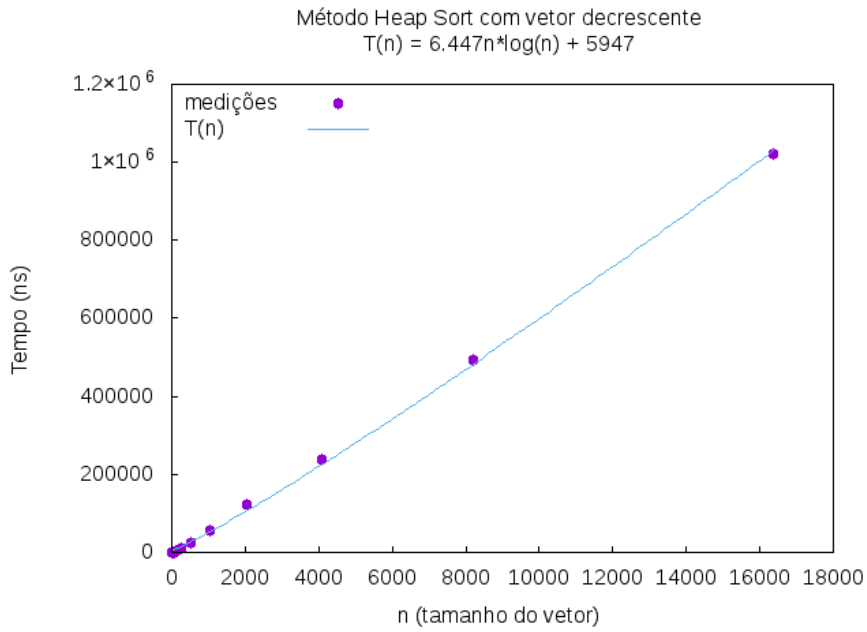

Figura 4.8: *Gráfico Heap Sort - Vetor Decrescente*

4.9 Heap Sort - Vetor Decrescente P10

Tabela gerada utilizando Heap Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 10% ordenado.

Tabela 4.9: *Heap Sort com vetor ordenado em ordem decrescente estando 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	533
32	885
64	1825
128	3787
256	9940
512	24112
1024	55700
2048	131589
4096	239256
8192	497985
16384	1019311

4.9.1 Gráfico Heap Sort - Vetor Decrescente P10

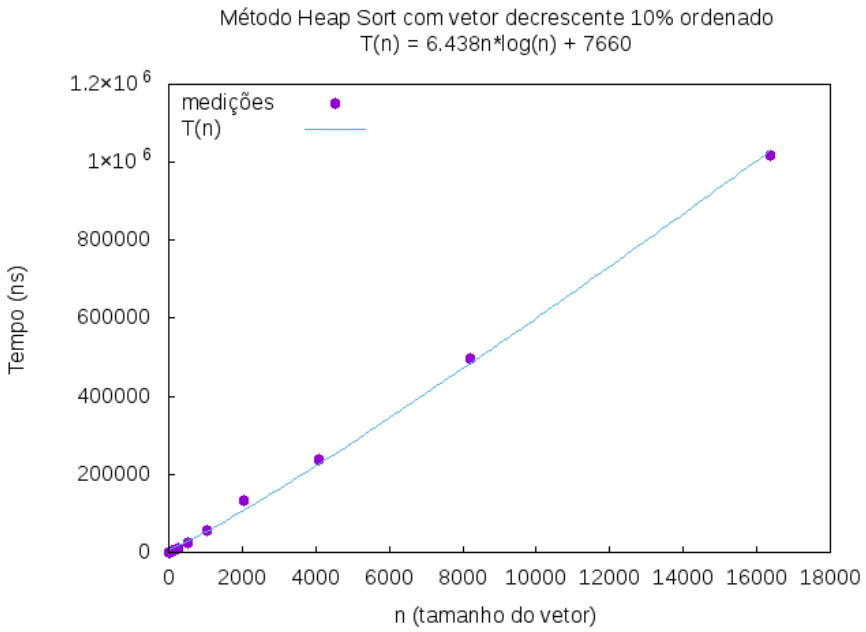

Figura 4.9: *Gráfico Heap Sort - Vetor Decrescente P10*

4.10 Heap Sort - Vetor Decrescente P20

Tabela gerada utilizando Heap Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 20% ordenado.

Tabela 4.10: *Heap Sort com vetor ordenado em ordem decrescente estando 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	552
32	851
64	1775
128	4138
256	10415
512	34619
1024	56092
2048	120560
4096	236652
8192	505112
16384	991519

4.10.1 Gráfico Heap Sort - Vetor Decrescente P20

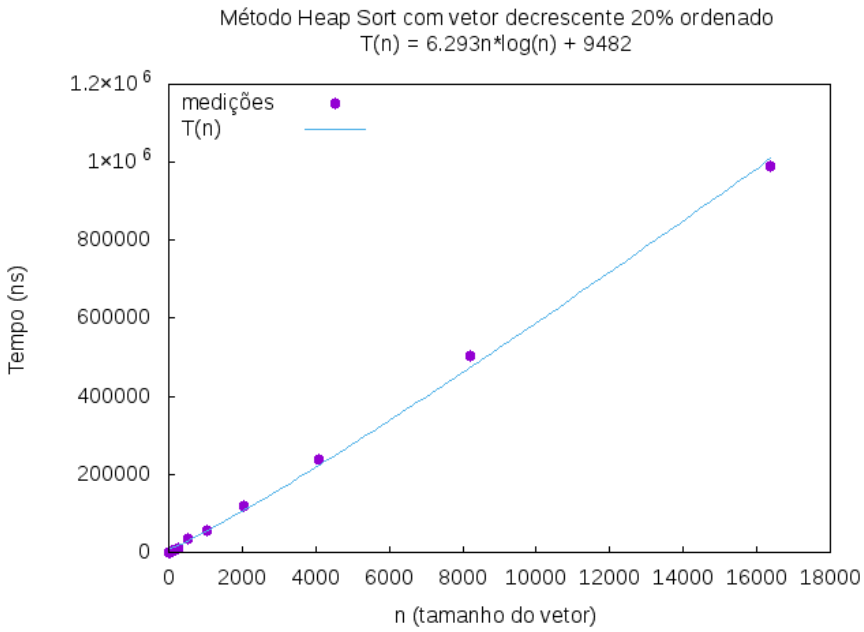

Figura 4.10: *Gráfico Heap Sort - Vetor Decrescente P20*

4.11 Heap Sort - Vetor Decrescente P30

Tabela gerada utilizando Heap Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 30% ordenado.

Tabela 4.11: *Heap Sort com vetor ordenado em ordem decrescente estando 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	574
32	901
64	1805
128	4445
256	9851
512	25314
1024	58946
2048	115970
4096	236945
8192	517471
16384	1013956

4.11.1 Gráfico Heap Sort - Vetor Decrescente P30

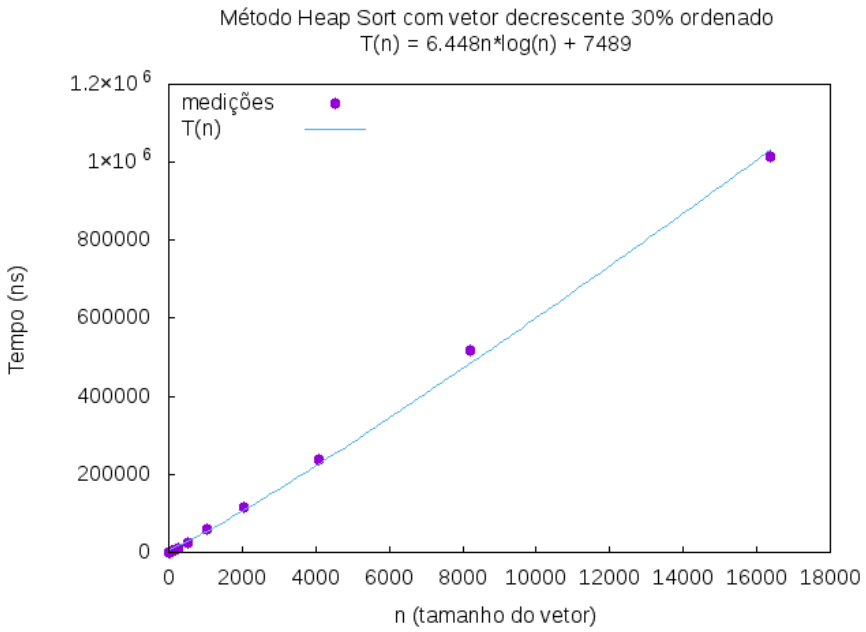

Figura 4.11: *Gráfico Heap Sort - Vetor Decrescente P30*

4.12 Heap Sort - Vetor Decrescente P40

Tabela gerada utilizando Heap Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 40% ordenado.

Tabela 4.12: *Heap Sort com vetor ordenado em ordem decrescente estando 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	566
32	878
64	1773
128	3932
256	9781
512	24638
1024	55357
2048	117612
4096	247034
8192	479714
16384	966254

4.12.1 Gráfico Heap Sort - Vetor Decrescente P40

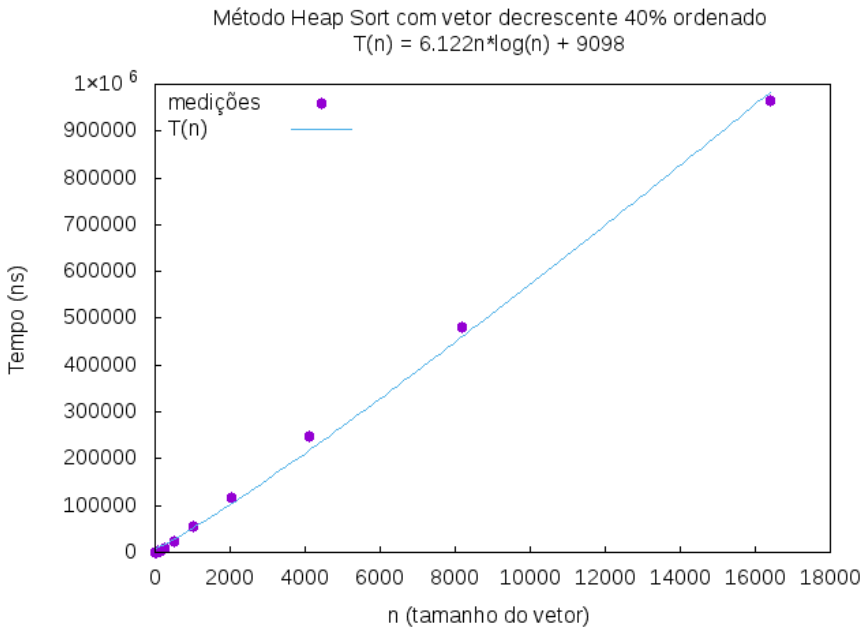

Figura 4.12: *Gráfico Heap Sort - Vetor Decrescente P40*

4.13 Heap Sort - Vetor Decrescente P50

Tabela gerada utilizando Heap Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 50% ordenado.

Tabela 4.13: *Heap Sort com vetor ordenado em ordem decrescente estando 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	676
32	949
64	1959
128	4289
256	10597
512	24941
1024	54696
2048	138827
4096	236799
8192	531128
16384	947050

4.13.1 Gráfico Heap Sort - Vetor Decrescente P50

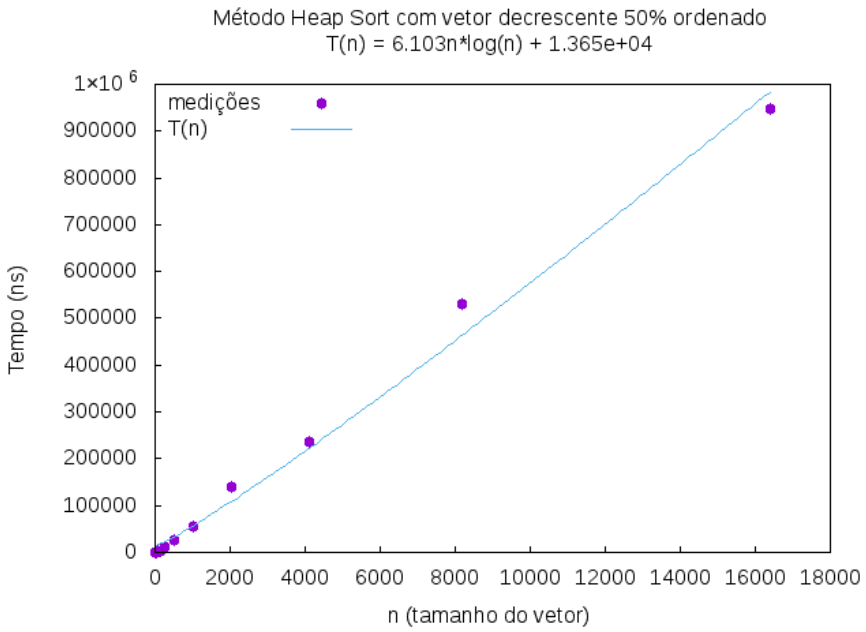

Figura 4.13: *Gráfico Heap Sort - Vetor Decrescente P50*

4.14 Observações Finais

heap sort com vetor em ordem decrescente com 10% dos elementos ordenados levaria aproximadamente 10 minutos para processar um vetor de 2^k com k = 32 elementos.

Capítulo 5

Quick Sort

QuickSort adota a estratégia de divisão e conquista, basicamente ele trabalha escolhendo um pivo, ele rearranja a lista da forma que os maiores que pivo ficam a direita e menores a esquerda, ele faz isso recursivamente até que tenha uma lista ordenada. Em qualquer caso, temos que o algoritmo QuickSort tem complexidade de tempo O(n^2) no pior caso e tanto no médio quanto no melhor caso é O(nlogn).

5.1 Quick Sort - Vetor Aleatório

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

Tabela 5.1: *Quick Sort com vetor aleatório*

Número de Elementos	**Tempo de execução em nanosegundos**
16	925
32	1077
64	2100
128	4491
256	10525
512	23829
1024	62993
2048	125500
4096	278666
8192	576399
16384	1261602

5.1.1 Gráfico Quick Sort - Vetor Aleatório

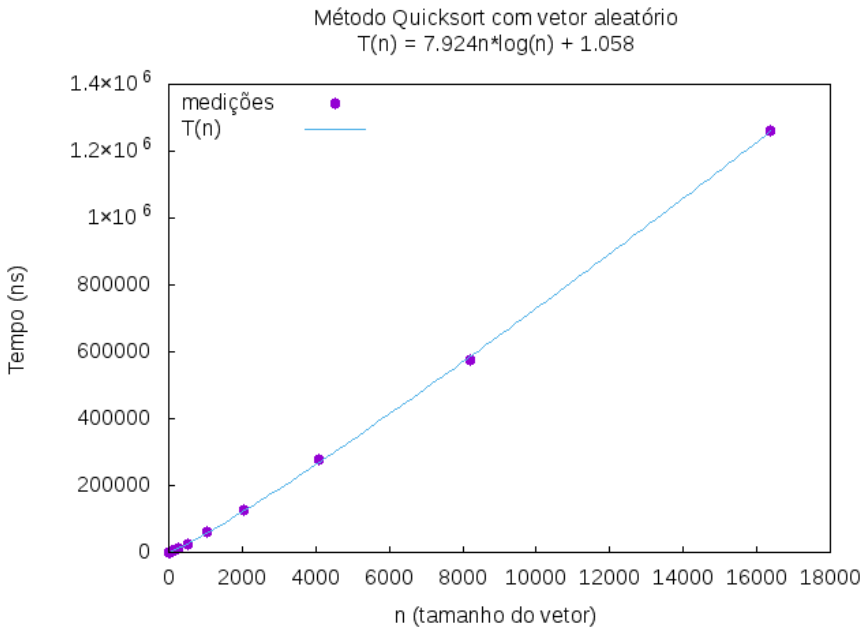

Figura 5.1: *Gráfico Quick Sort - Vetor Aleatório*

5.2 Quick Sort - Vetor Crescente

Tabela gerada utilizando Quick Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente.

Tabela 5.2: *Quick Sort com vetor ordenado em ordem crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	706
32	954
64	1363
128	2163
256	3711
512	7086
1024	13945
2048	28288
4096	61831
8192	127886
16384	299130

5.2.1 Gráfico Quick Sort - Vetor Crescente

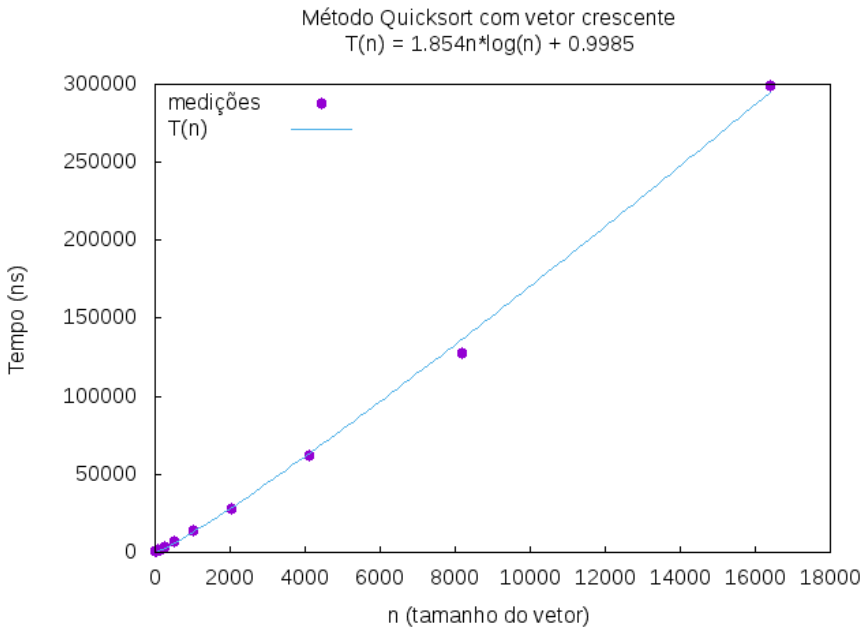

Figura 5.2: *Gráfico Quick Sort - Vetor Crescente*

5.3 Quick Sort - Vetor Crescente P10

Tabela gerada utilizando Quick Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 10% ordenado.

Tabela 5.3: *Quick Sort com vetor ordenado em ordem crescente estando 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	574
32	829
64	1158
128	2138
256	3617
512	6888
1024	13829
2048	28293
4096	60946
8192	131339
16384	269900

5.3.1 Gráfico Quick Sort - Vetor Crescente P10

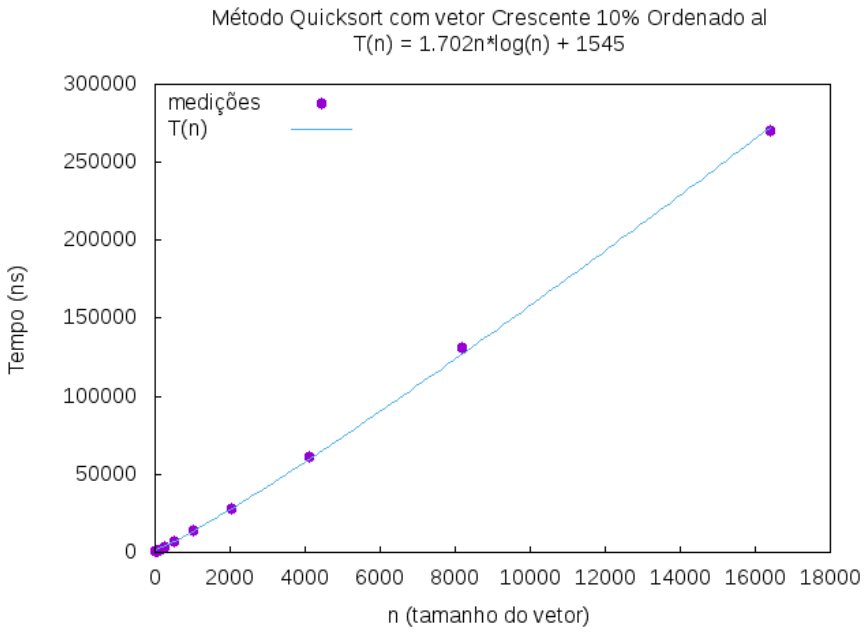

Figura 5.3: *Gráfico Quick Sort - Vetor Crescente P10*

5.4 Quick Sort - Vetor Crescente P20

Tabela gerada utilizando Quick Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 20% ordenado.

Tabela 5.4: *Quick Sort com vetor ordenado em ordem crescente estando 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	685
32	807
64	1334
128	2136
256	3568
512	6805
1024	13583
2048	27666
4096	56503
8192	121141
16384	264576

5.4.1 Gráfico Quick Sort - Vetor Crescente P20

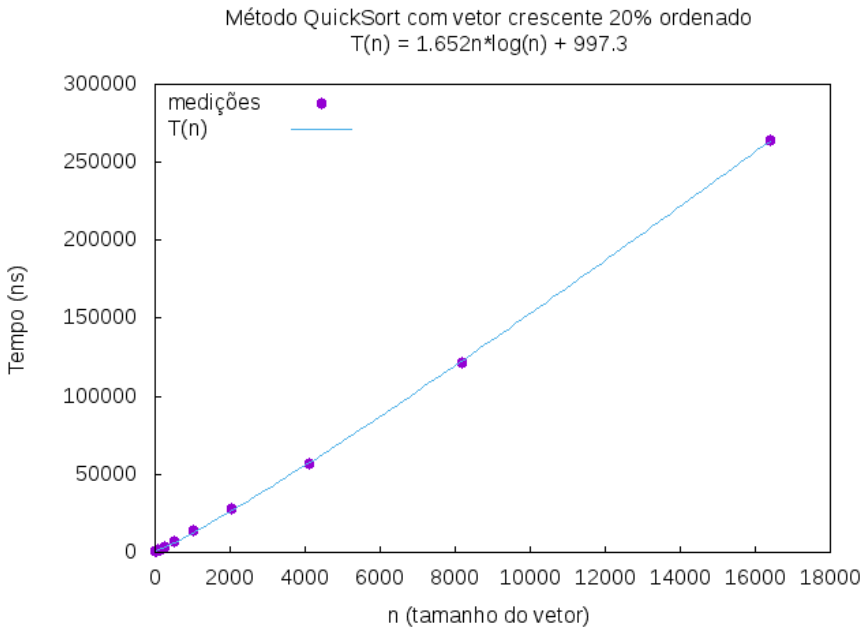

Figura 5.4: *Gráfico Quick Sort - Vetor Crescente P20*

5.5 Quick Sort - Vetor Crescente P30

Tabela gerada utilizando Quick Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 30% ordenado.

Tabela 5.5: *Quick Sort com vetor ordenado em ordem crescente estando 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	651
32	919
64	1282
128	2141
256	3705
512	6656
1024	13097
2048	27936
4096	66555
8192	121364
16384	248146

5.5.1 Gráfico Quick Sort - Vetor Crescente P30

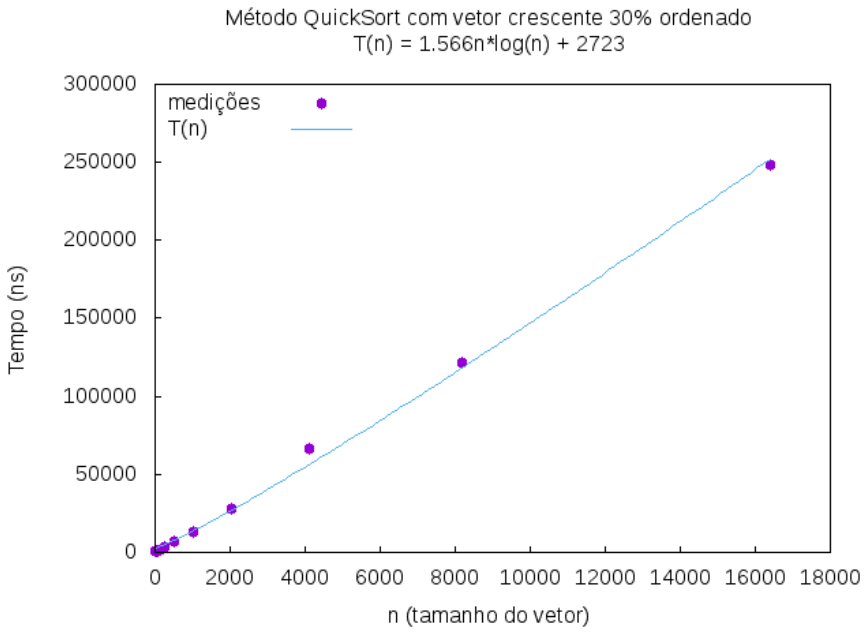

Figura 5.5: *Gráfico Quick Sort - Vetor Crescente P30*

5.6 Quick Sort - Vetor Crescente P40

Tabela gerada utilizando Quick Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 40% ordenado.

Tabela 5.6: *Quick Sort com vetor ordenado em ordem crescente estando 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	549
32	879
64	1148
128	1817
256	3197
512	6362
1024	12558
2048	31216
4096	53019
8192	120388
16384	238560

5.6.1 Gráfico Quick Sort - Vetor Crescente P40

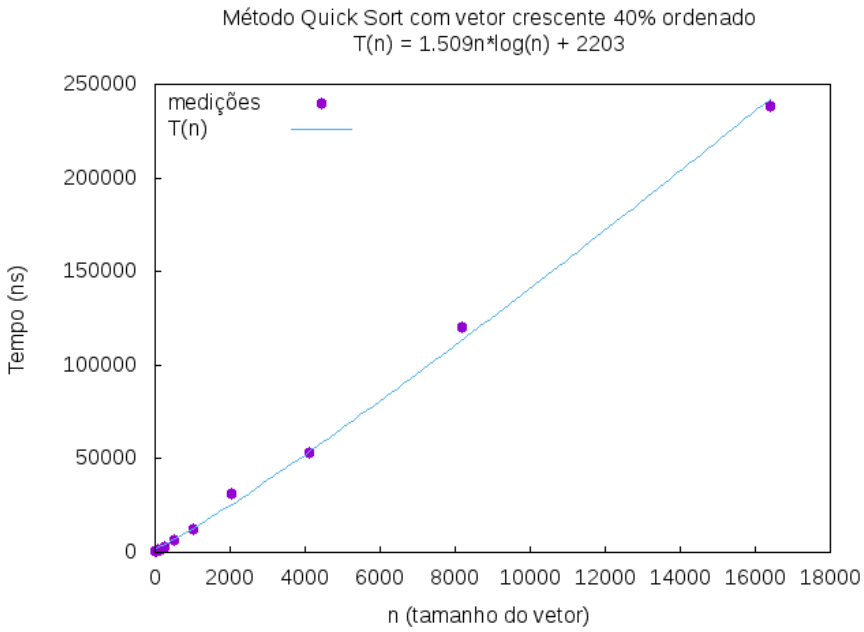

Figura 5.6: *Gráfico Quick Sort - Vetor Crescente P40*

5.7 Quick Sort - Vetor Crescente P50

Tabela gerada utilizando Quick Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 50% ordenado.

Tabela 5.7: *Quick Sort com vetor ordenado em ordem crescente estando 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	560
32	718
64	1083
128	1784
256	3276
512	6081
1024	12595
2048	26004
4096	54648
8192	145564
16384	241835

5.7.1 Gráfico Quick Sort - Vetor Crescente P50

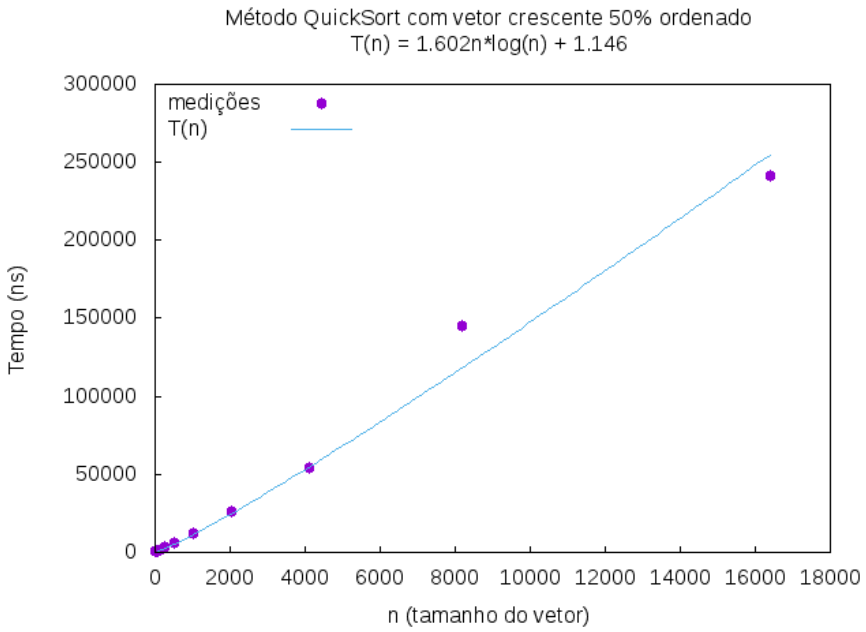

Figura 5.7: *Gráfico Quick Sort - Vetor Crescente P50*

5.8 Quick Sort - Vetor Decrescente

Tabela gerada utilizando Quick Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente.

Tabela 5.8: *Quick Sort com vetor ordenado em ordem decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	589
32	749
64	883
128	1476
256	2584
512	4991
1024	9893
2048	20693
4096	50409
8192	93702
16384	201092

5.8.1 Gráfico Quick Sort - Vetor Decrescente

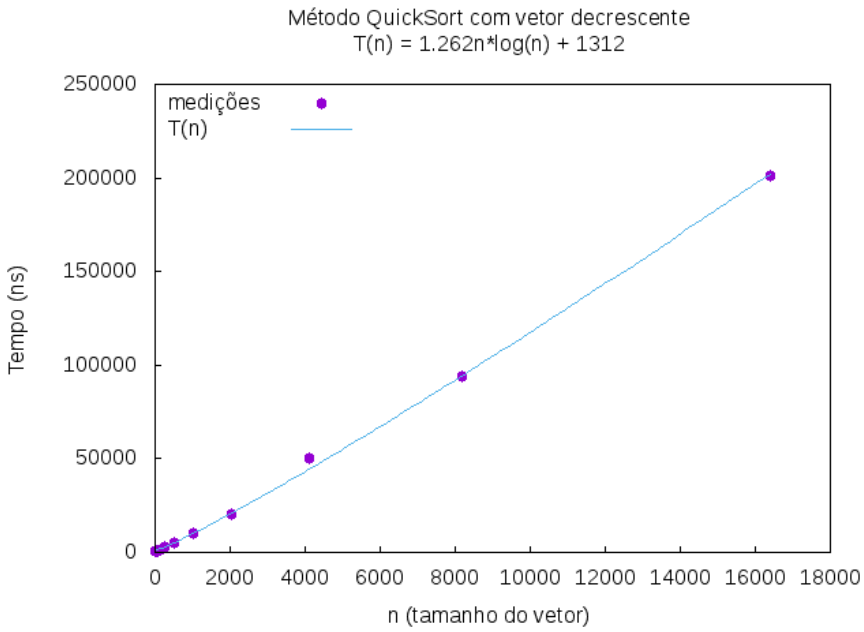

Figura 5.8: *Gráfico Quick Sort - Vetor Decrescente*

5.9 Quick Sort - Vetor Decrescente P10

Tabela gerada utilizando Quick Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 10% ordenado.

Tabela 5.9: *Quick Sort com vetor ordenado em ordem decrescente estando 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	698
32	701
64	1162
128	1805
256	3019
512	5849
1024	11126
2048	22521
4096	49710
8192	102070
16384	222690

5.9.1 Gráfico Quick Sort - Vetor Decrescente P10

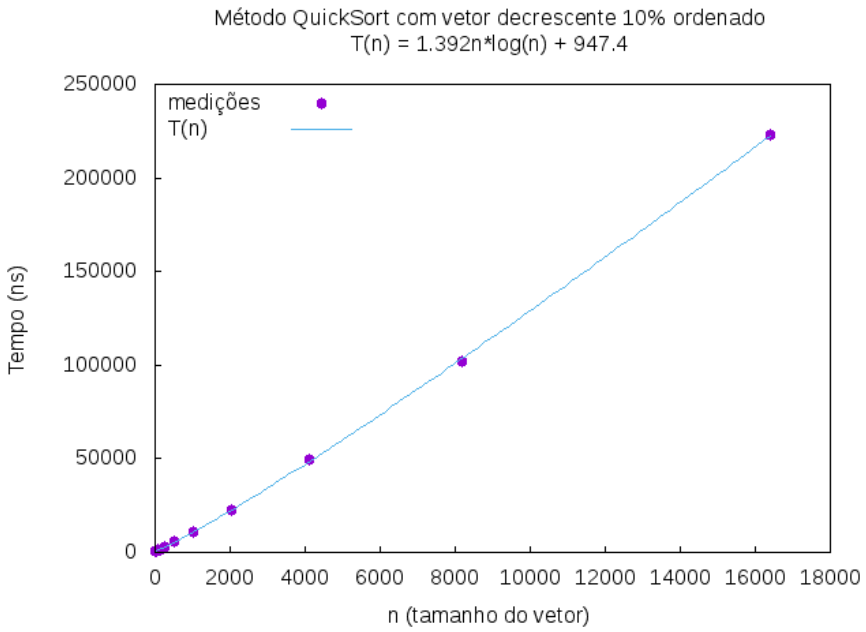

Figura 5.9: *Gráfico Quick Sort - Vetor Decrescente P10*

5.10 Quick Sort - Vetor Decrescente P20

Tabela gerada utilizando Quick Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 20% ordenado.

Tabela 5.10: *Quick Sort com vetor ordenado em ordem decrescente estando 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	835
32	868
64	1218
128	1905
256	3398
512	6020
1024	11484
2048	23470
4096	48791
8192	110327
16384	218467

5.10.1 Gráfico Quick Sort - Vetor Decrescente P20

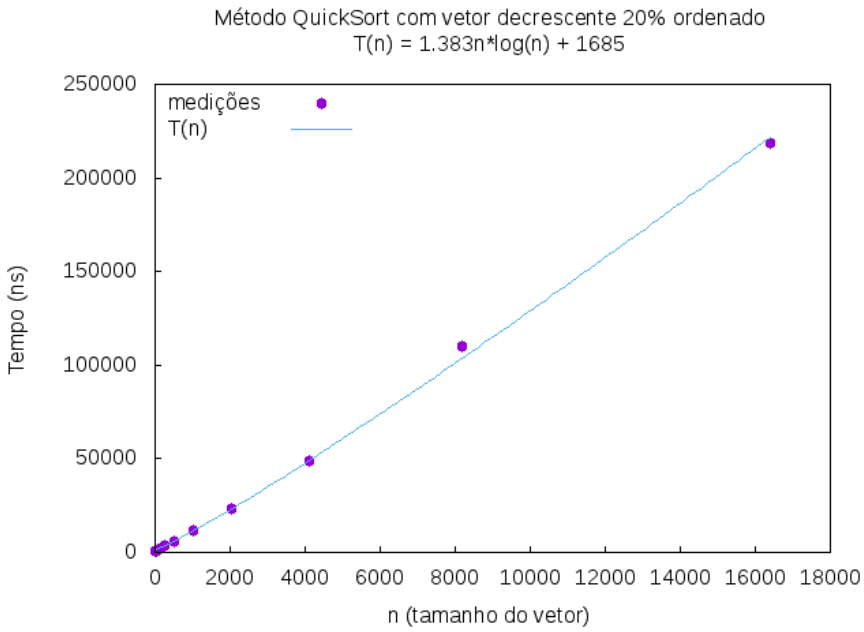

Figura 5.10: *Gráfico Quick Sort - Vetor Decrescente P20*

5.11 Quick Sort - Vetor Decrescente P30

Tabela gerada utilizando Quick Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 30% ordenado.

Tabela 5.11: *Quick Sort com vetor ordenado em ordem decrescente estando 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	585
32	785
64	1168
128	1850
256	3089
512	5738
1024	11419
2048	23348
4096	49489
8192	106600
16384	222547

5.11.1 Gráfico Quick Sort - Vetor Decrescente P30

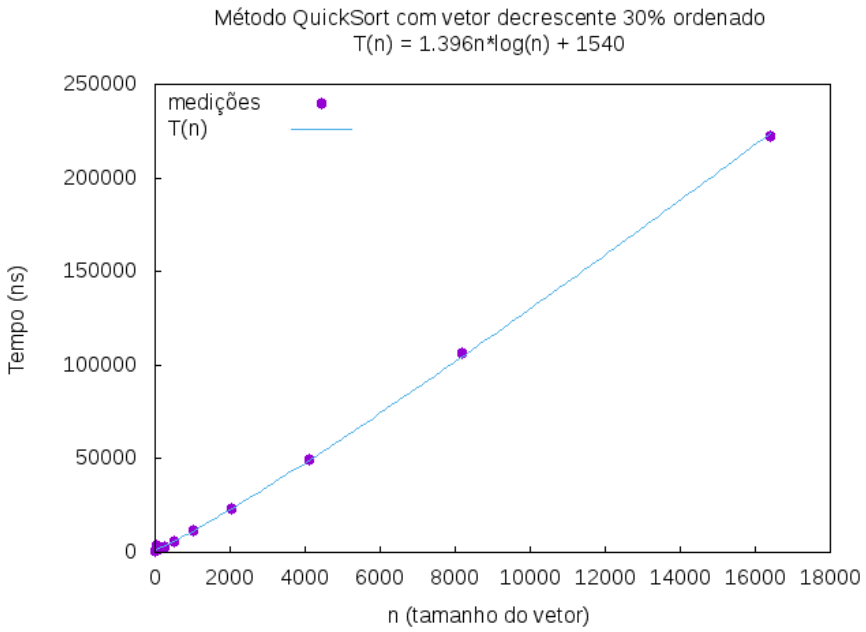

Figura 5.11: *Gráfico Quick Sort - Vetor Decrescente P30*

5.12 Quick Sort - Vetor Decrescente P40

Tabela gerada utilizando Quick Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 40% ordenado.

Tabela 5.12: *Quick Sort com vetor ordenado em ordem decrescente estando 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	677
32	905
64	1706
128	2646
256	3768
512	6826
1024	12542
2048	25469
4096	62872
8192	114528
16384	229091

5.12.1 Gráfico Quick Sort - Vetor Decrescente P40

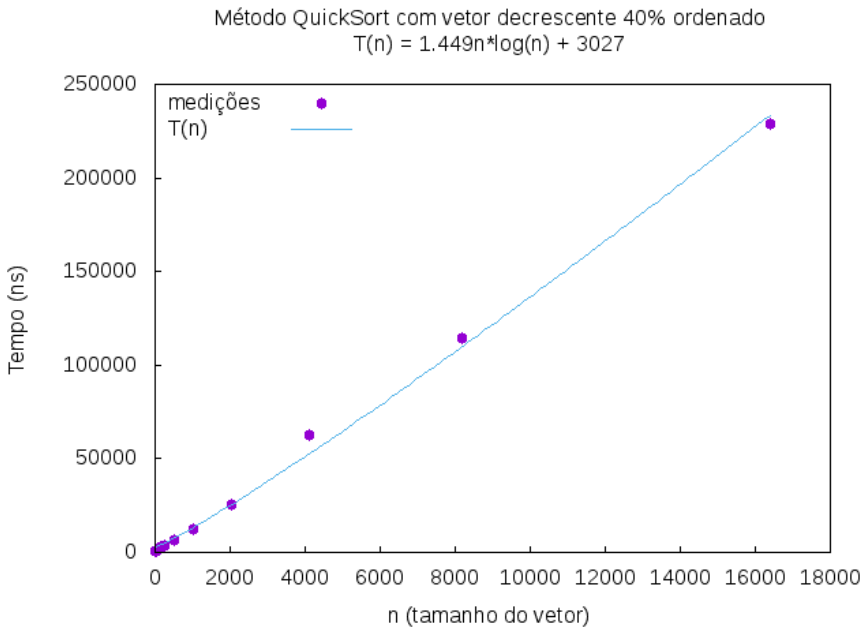

Figura 5.12: *Gráfico Quick Sort - Vetor Decrescente P40*

5.13 Quick Sort - Vetor Decrescente P50

Tabela gerada utilizando Quick Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 50% ordenado.

Tabela 5.13: *Quick Sort com vetor ordenado em ordem decrescente estando 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	638
32	803
64	1209
128	1954
256	3424
512	6323
1024	12541
2048	26206
4096	54582
8192	121376
16384	239823

5.13.1 Gráfico Quick Sort - Vetor Decrescente P50

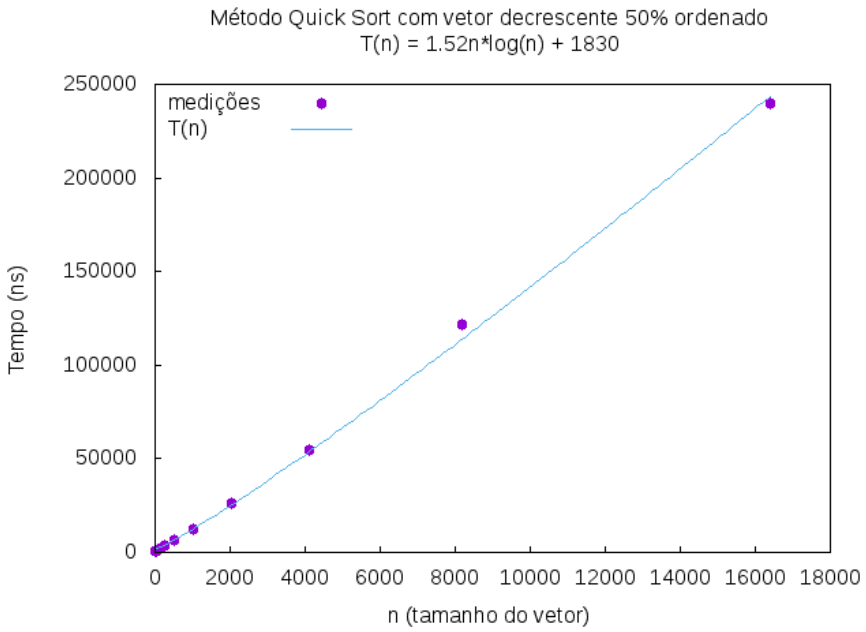

Figura 5.13: *Gráfico Quick Sort - Vetor Decrescente P50*

5.14 Observações Finais

quick sort com vetor de elementos totalmente aleatório levaria aproximadamente 12 minutos para processar um vetor de 2^k com k = 32 elementos nessas condições.

Capítulo 6

Counting Sort

O algoritmo de ordenação Counting Sort é um algoritmo estável. Basicamente ele determina para cada entrada X, o número de elementos menor que ele, e utilizando essa informação, ele coloca o elemento x diretamente na posição correta no vetor de saída. Em qualquer caso, temos que o algoritmo CountingSort tem complexidade de tempo O(n).

6.1 Counting Sort - Vetor Aleatório

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

Tabela 6.1: *Counting Sort com vetor aleatório*

Número de Elementos	**Tempo de execução em nanosegundos**
16	1180
32	763
64	753
128	980
256	1277
512	3643
1024	5315
2048	12623
4096	23526
8192	64405
16384	163553

6.1.1 Gráfico Counting Sort - Vetor Aleatório

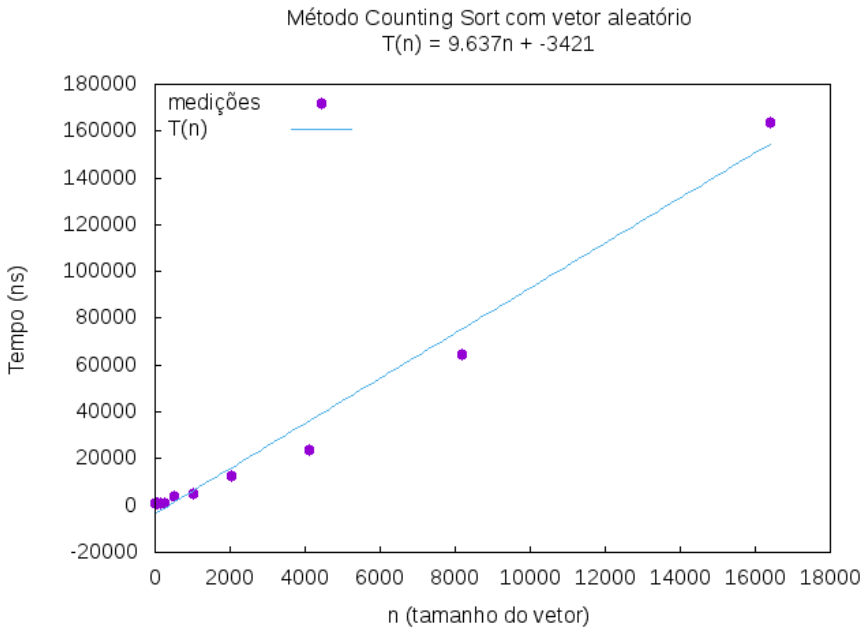

Figura 6.1: *Gráfico Counting Sort - Vetor Aleatório*

6.2 Counting Sort - Vetor Crescente

Tabela gerada utilizando Counting Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente.

Tabela 6.2: *Counting Sort com vetor ordenado em ordem crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	942
32	665
64	790
128	1183
256	1396
512	2115
1024	8305
2048	17296
4096	41228
8192	59797
16384	107430

6.2.1 Gráfico Counting Sort - Vetor Crescente

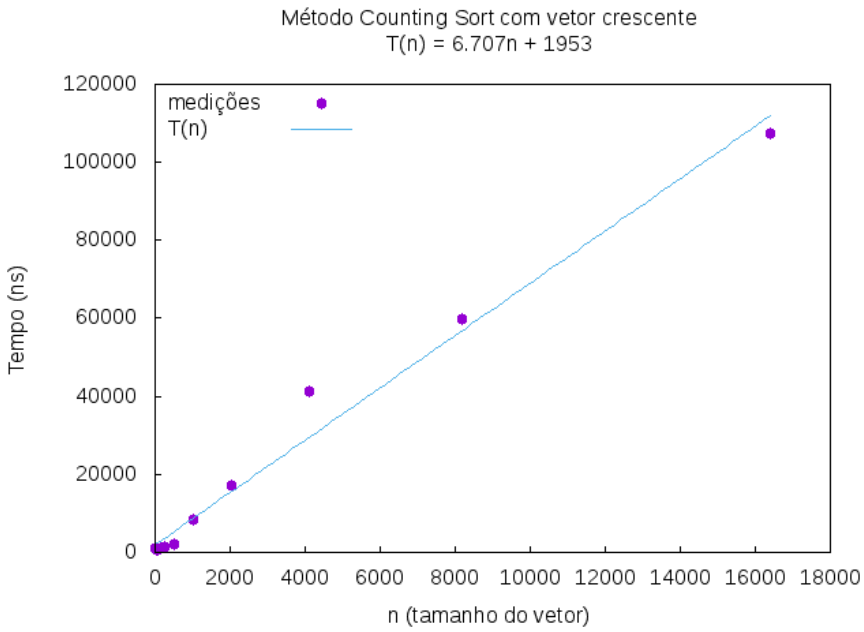

Figura 6.2: *Gráfico Counting Sort - Vetor Crescente*

6.3 Counting Sort - Vetor Crescente P10

Tabela gerada utilizando Counting Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 10% ordenado.

Tabela 6.3: *Counting Sort com vetor ordenado em ordem crescente estando 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	986
32	703
64	935
128	1027
256	1433
512	2054
1024	5931
2048	16341
4096	31788
8192	73657
16384	151634

6.3.1 Gráfico Counting Sort - Vetor Crescente P10

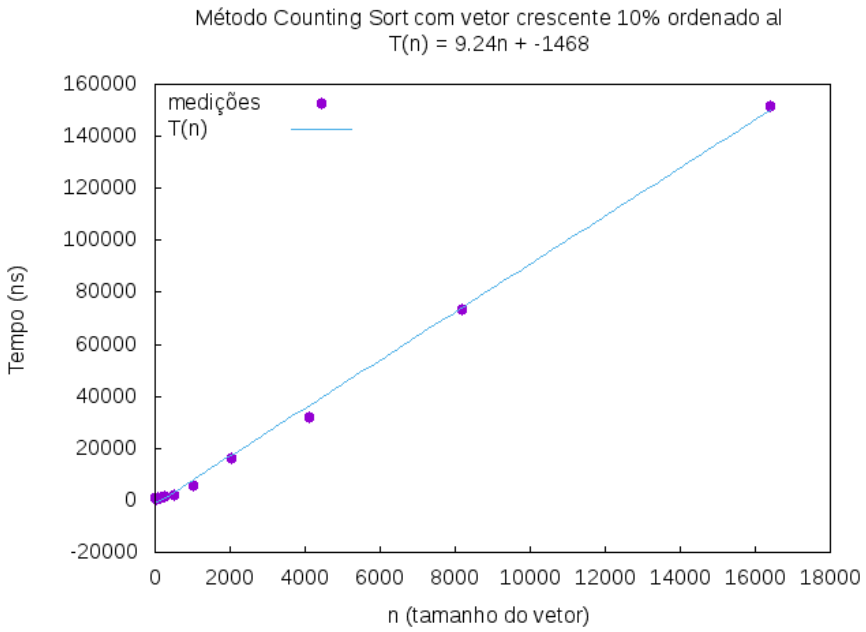

Figura 6.3: *Gráfico Counting Sort - Vetor Crescente P10*

6.4 Counting Sort - Vetor Crescente P20

Tabela gerada utilizando Counting Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 20% ordenado.

Tabela 6.4: *Counting Sort com vetor ordenado em ordem crescente estando 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	931
32	698
64	739
128	983
256	1409
512	4075
1024	5644
2048	17222
4096	28122
8192	66564
16384	102014

6.4.1 Gráfico Counting Sort - Vetor Crescente P20

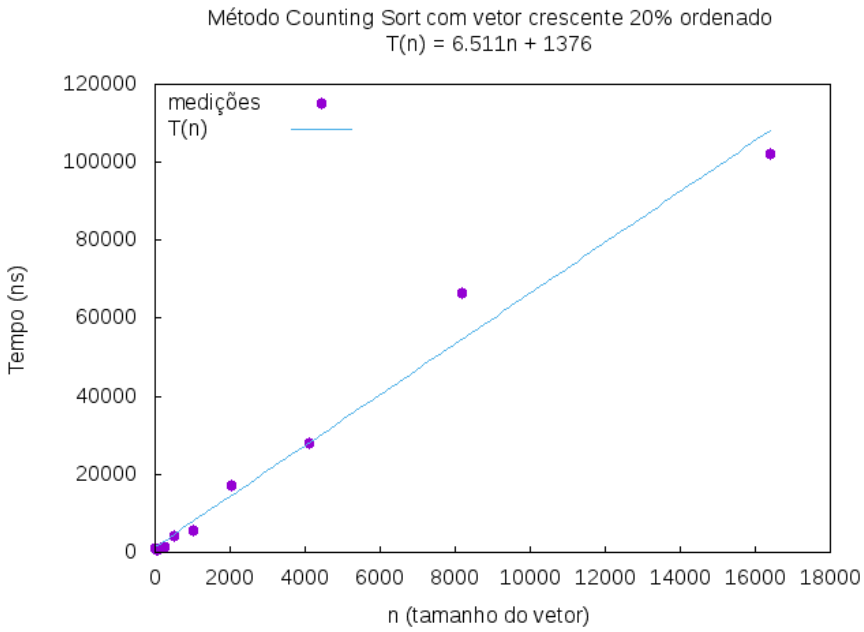

Figura 6.4: *Gráfico Counting Sort - Vetor Crescente P20*

6.5 Counting Sort - Vetor Crescente P30

Tabela gerada utilizando Counting Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 30% ordenado.

Tabela 6.5: *Counting Sort com vetor ordenado em ordem crescente estando 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	884
32	607
64	793
128	4111
256	1340
512	3869
1024	6533
2048	14310
4096	33867
8192	56900
16384	117884

6.5.1 Gráfico Counting Sort - Vetor Crescente P30

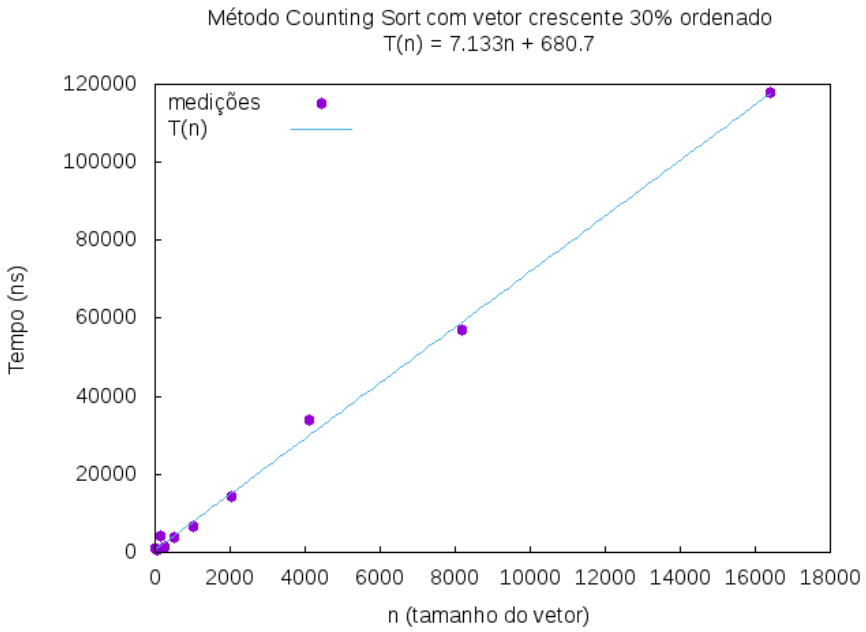

Figura 6.5: *Gráfico Counting Sort - Vetor Crescente P30*

6.6 Counting Sort - Vetor Crescente P40

Tabela gerada utilizando Counting Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 40% ordenado.

Tabela 6.6: *Counting Sort com vetor ordenado em ordem crescente estando 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1114
32	644
64	764
128	1036
256	1327
512	1919
1024	10465
2048	18442
4096	51975
8192	56684
16384	105513

6.6.1 Gráfico Counting Sort - Vetor Crescente P40

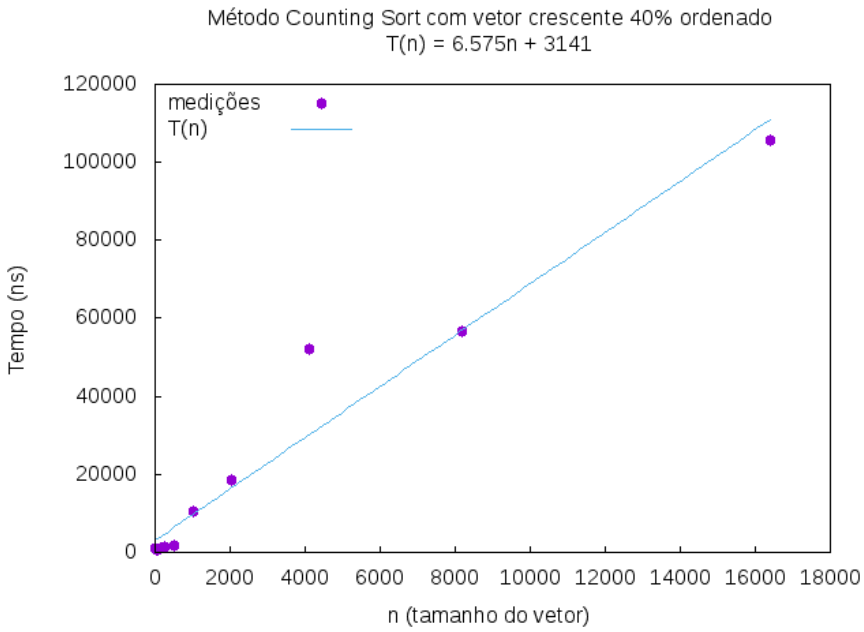

Figura 6.6: *Gráfico Counting Sort - Vetor Crescente P40*

6.7 Counting Sort - Vetor Crescente P50

Tabela gerada utilizando Counting Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 50% ordenado.

Tabela 6.7: *Counting Sort com vetor ordenado em ordem crescente estando 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	3878
32	655
64	771
128	1086
256	1391
512	4088
1024	5410
2048	15451
4096	29168
8192	51542
16384	114574

6.7.1 Gráfico Counting Sort - Vetor Crescente P50

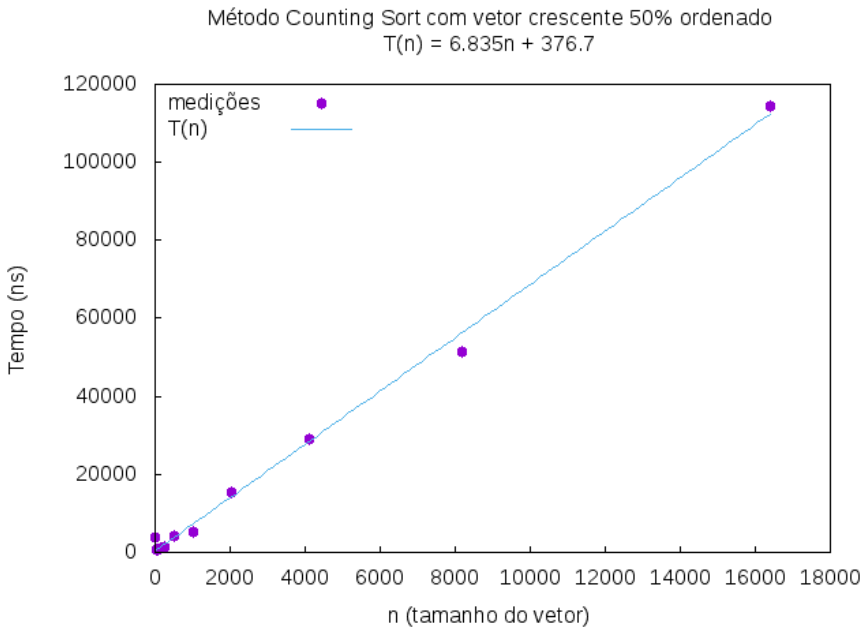

Figura 6.7: *Gráfico Counting Sort - Vetor Crescente P50*

6.8 Counting Sort - Vetor Decrescente

Tabela gerada utilizando Counting Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente.

Tabela 6.8: *Counting Sort com vetor ordenado em ordem decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	1111
32	669
64	807
128	1038
256	1308
512	7100
1024	5235
2048	13556
4096	31042
8192	55271
16384	121497

6.8.1 Gráfico Counting Sort - Vetor Decrescente

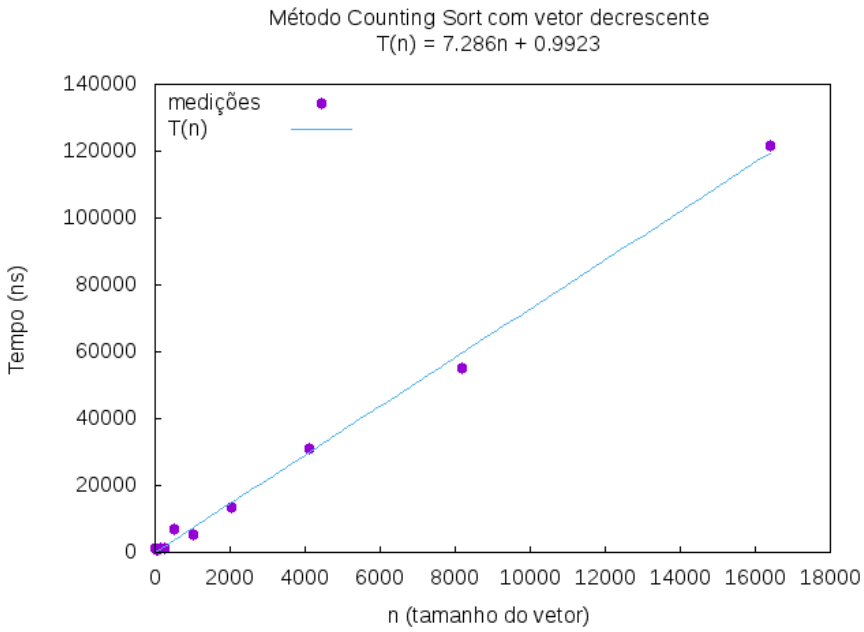

Figura 6.8: *Gráfico Counting Sort - Vetor Decrescente*

6.9 Counting Sort - Vetor Decrescente P10

Tabela gerada utilizando Counting Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 10% ordenado.

Tabela 6.9: *Counting Sort com vetor ordenado em ordem decrescente estando 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1035
32	609
64	784
128	1082
256	1326
512	2019
1024	5279
2048	12392
4096	25377
8192	46191
16384	93057

6.9.1 Gráfico Counting Sort - Vetor Decrescente P10

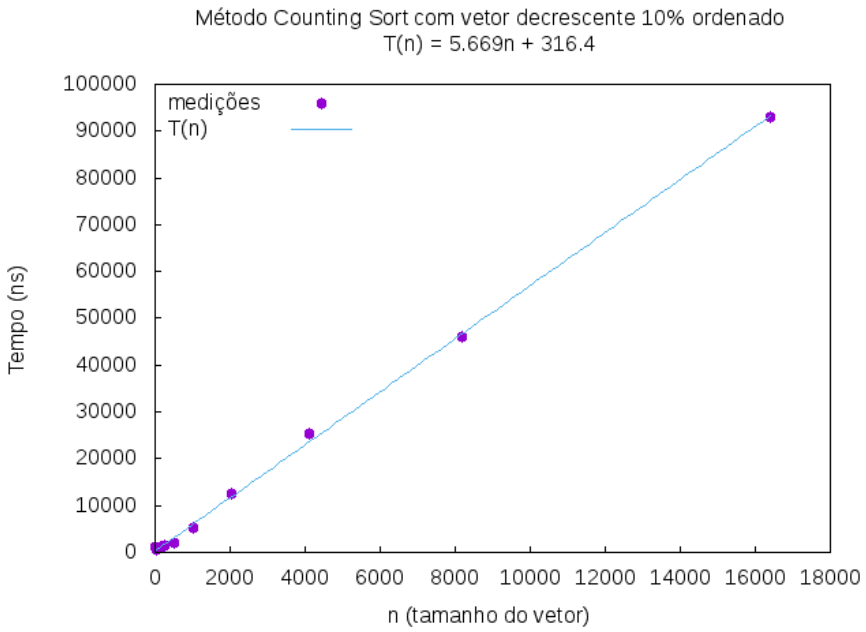

Figura 6.9: *Gráfico Counting Sort - Vetor Decrescente P10*

6.10 Counting Sort - Vetor Decrescente P20

Tabela gerada utilizando Counting Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 20% ordenado.

Tabela 6.10: *Counting Sort com vetor ordenado em ordem decrescente estando 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1251
32	1486
64	764
128	1023
256	1311
512	3741
1024	5552
2048	13647
4096	39382
8192	48979
16384	99118

6.10.1 Gráfico Counting Sort - Vetor Decrescente P20

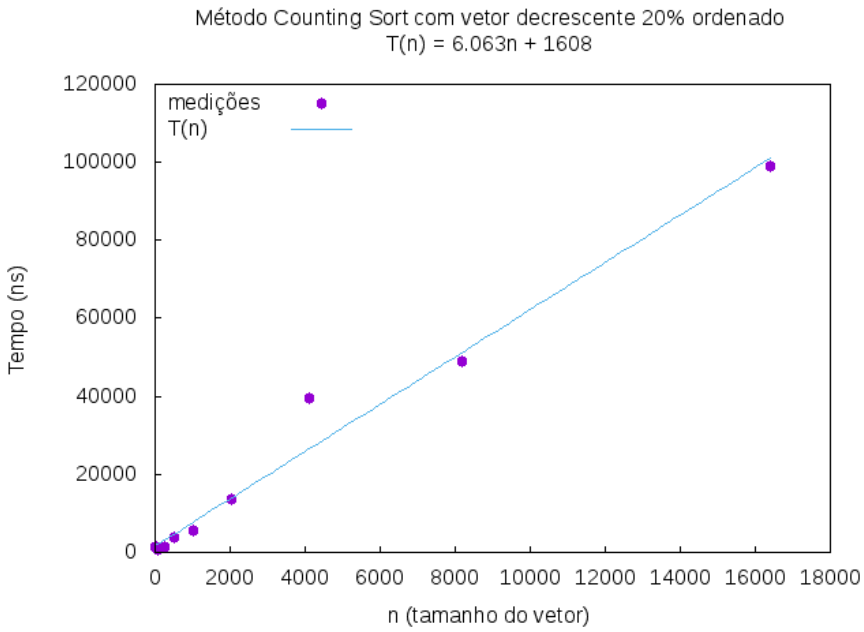

Figura 6.10: *Gráfico Counting Sort - Vetor Decrescente P20*

6.11 Counting Sort - Vetor Decrescente P30

Tabela gerada utilizando Counting Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 30% ordenado.

Tabela 6.11: *Counting Sort com vetor ordenado em ordem decrescente estando 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1014
32	676
64	756
128	1140
256	1462
512	4058
1024	5755
2048	16655
4096	35422
8192	64506
16384	103663

6.11.1 Gráfico Counting Sort - Vetor Decrescente P30

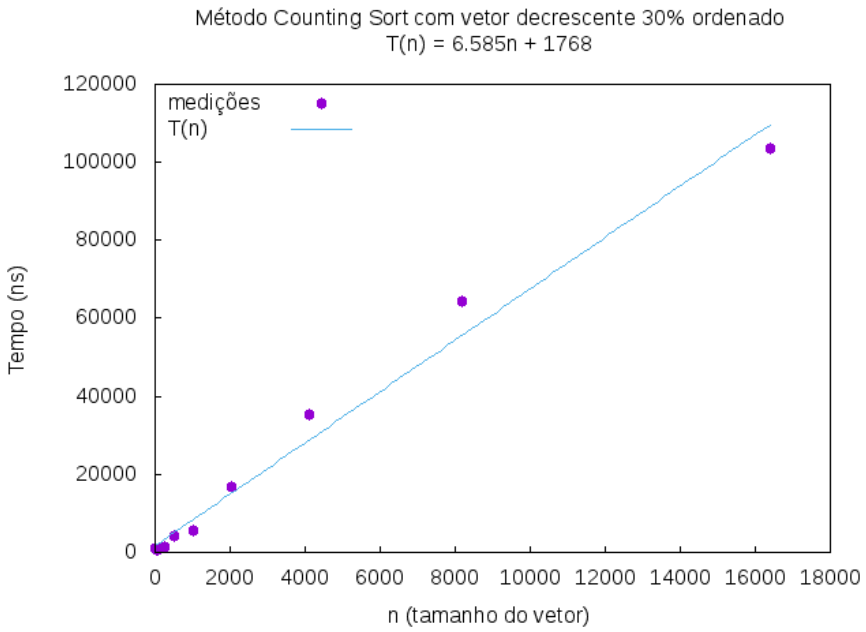

Figura 6.11: *Gráfico Counting Sort - Vetor Decrescente P30*

6.12 Counting Sort - Vetor Decrescente P40

Tabela gerada utilizando Counting Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 40% ordenado.

Tabela 6.12: *Counting Sort com vetor ordenado em ordem decrescente estando 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1129
32	705
64	754
128	1004
256	1518
512	1925
1024	5353
2048	12747
4096	24478
8192	49552
16384	120622

6.12.1 Gráfico Counting Sort - Vetor Decrescente P40

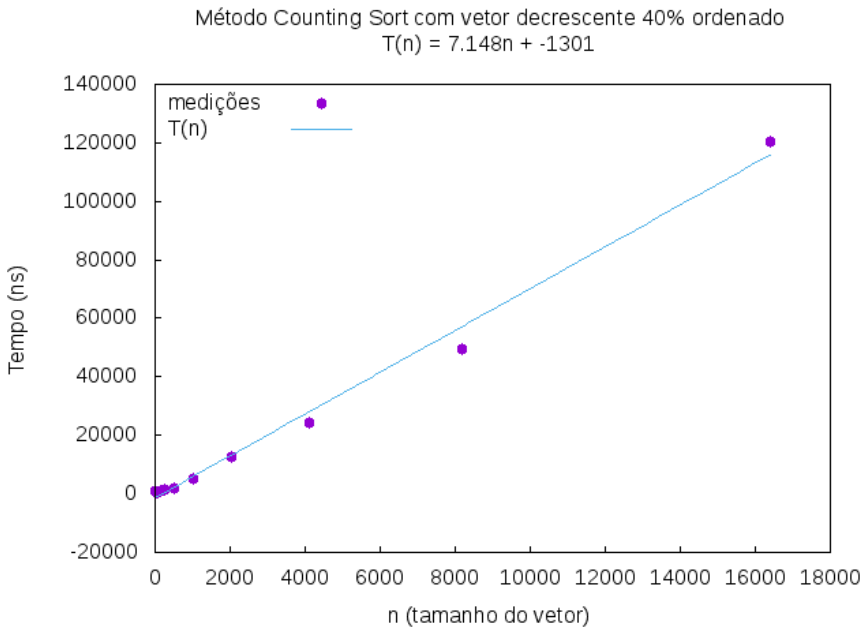

Figura 6.12: *Gráfico Counting Sort - Vetor Decrescente P40*

6.13 Counting Sort - Vetor Decrescente P50

Tabela gerada utilizando Counting Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 50% ordenado.

Tabela 6.13: *Counting Sort com vetor ordenado em ordem decrescente estando 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	960
32	586
64	792
128	1090
256	1344
512	2043
1024	5051
2048	14108
4096	26611
8192	50560
16384	140356

6.13.1 Gráfico Counting Sort - Vetor Decrescente P50

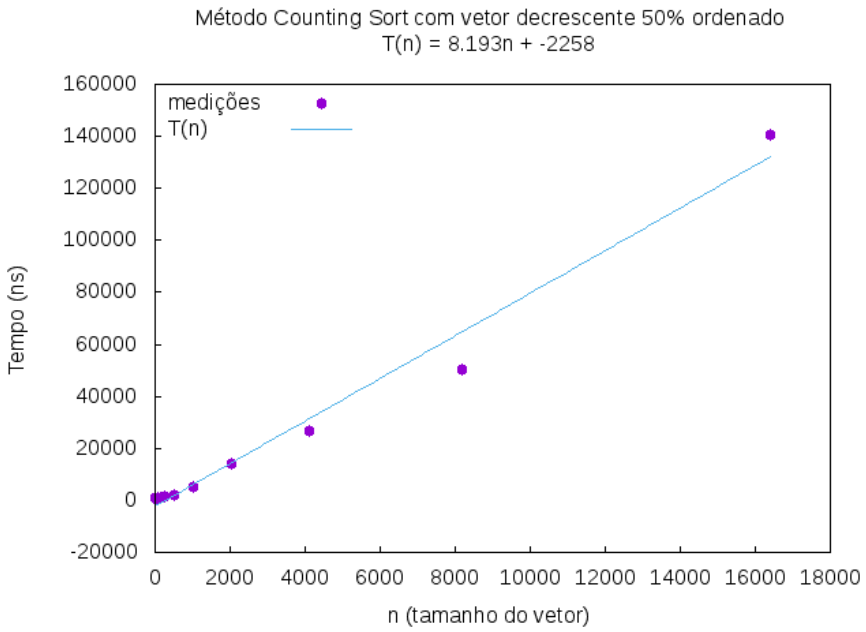

Figura 6.13: *Gráfico Counting Sort - Vetor Decrescente P50*

6.14 Observações Finais

couting sort com vetor de elementos totalmente aleatório levaria aproximadamente 49 segundos para processar um vetor de 2^k com k = 32 elementos nessas condições.

Capítulo 7

Radix Sort

O algoritmo de ordenação Radix Sort é um algoritmo estável. Ele ordena itens que são identificados por chaves únicas. Cada chave é uma cadeia de caracteres ou número, o radix ordena as chaves. Ele ordena inteiros processando digitos indivíduais. Em qualquer caso, temos que o algoritmo RadixSort tem complexidade de tempo O(n).

7.1 Radix Sort - Vetor Aleatório

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

Tabela 7.1: *Radix Sort com vetor aleatório*

Número de Elementos	**Tempo de execução em nanosegundos**
16	1773
32	1345
64	2116
128	4744
256	10074
512	19320
1024	50121
2048	101591
4096	224802
8192	478058
16384	1253961

7.1.1 Gráfico Radix Sort - Vetor Aleatório

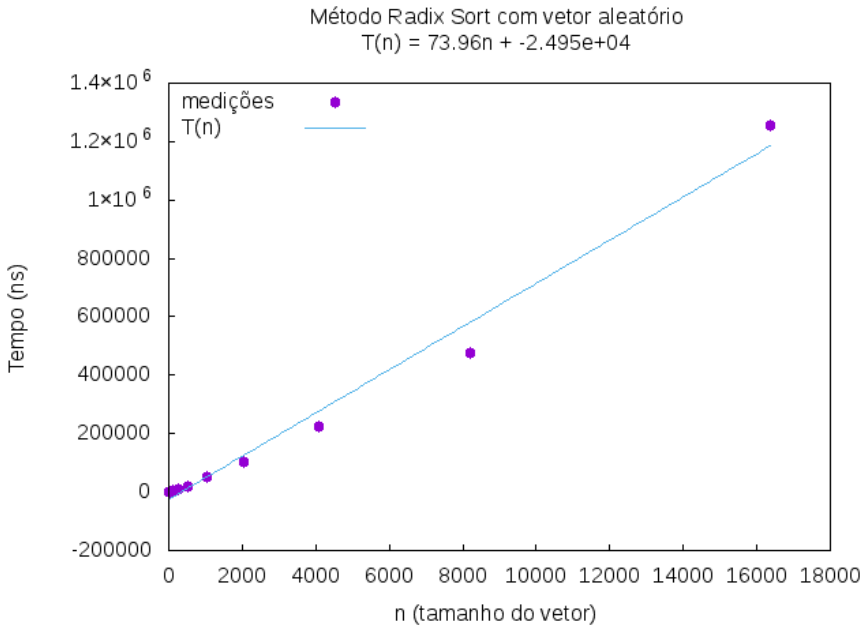

Figura 7.1: *Gráfico Radix Sort - Vetor Aleatório*

7.2 Radix Sort - Vetor Crescente

Tabela gerada utilizando Radix Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente.

Tabela 7.2: *Radix Sort com vetor ordenado em ordem crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	1257
32	1550
64	2657
128	4851
256	9156
512	18680
1024	51585
2048	105519
4096	208193
8192	414850
16384	1078854

7.2.1 Gráfico Radix Sort - Vetor Crescente

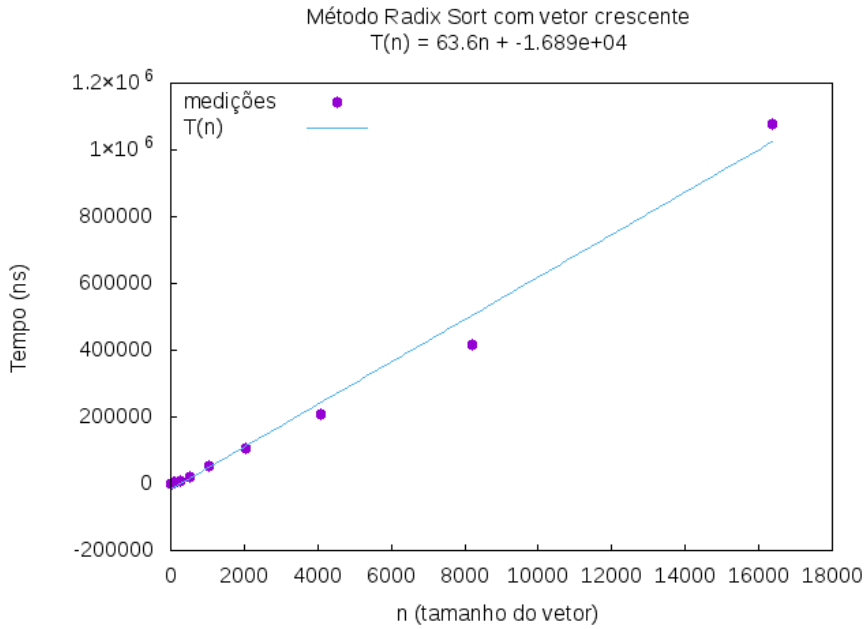

Figura 7.2: *Gráfico Radix Sort - Vetor Crescente*

7.3 Radix Sort - Vetor Crescente P10

Tabela gerada utilizando Radix Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 10% ordenado.

Tabela 7.3: *Radix Sort com vetor ordenado em ordem crescente estando 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1449
32	1470
64	2184
128	4827
256	8697
512	19543
1024	58018
2048	114081
4096	266876
8192	558293
16384	1079561

7.3.1 Gráfico Radix Sort - Vetor Crescente P10

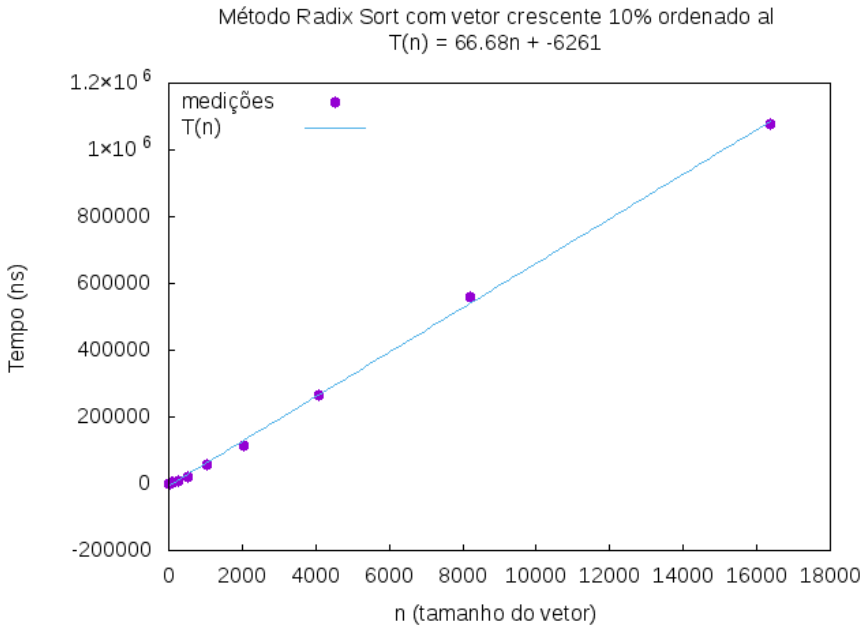

Figura 7.3: *Gráfico Radix Sort - Vetor Crescente P10*

7.4 Radix Sort - Vetor Crescente P20

Tabela gerada utilizando Radix Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 20% ordenado.

Tabela 7.4: *Radix Sort com vetor ordenado em ordem crescente estando 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1569
32	1443
64	2196
128	4695
256	8532
512	20225
1024	50143
2048	108896
4096	215786
8192	434859
16384	1083653

7.4.1 Gráfico Radix Sort - Vetor Crescente P20

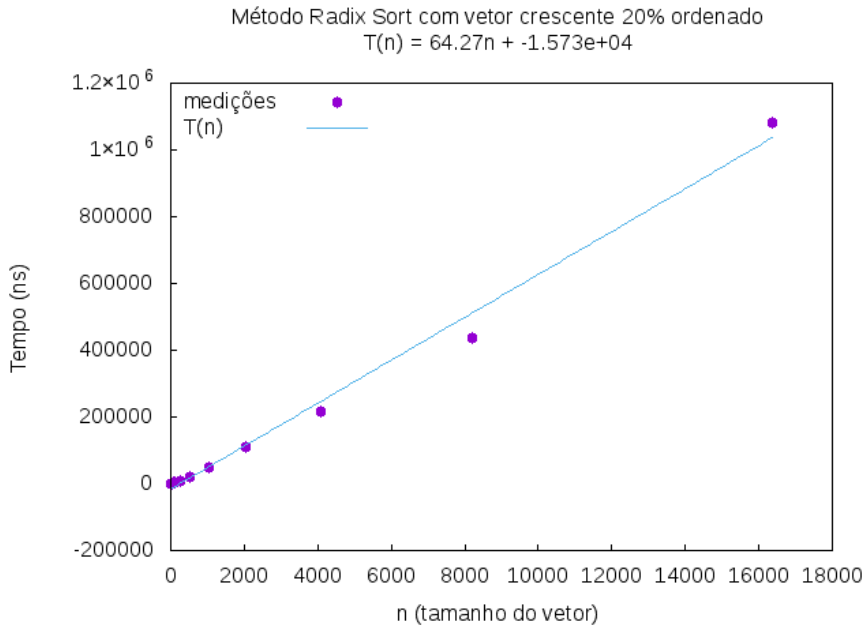

Figura 7.4: *Gráfico Radix Sort - Vetor Crescente P20*

7.5 Radix Sort - Vetor Crescente P30

Tabela gerada utilizando Radix Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 30% ordenado.

Tabela 7.5: *Radix Sort com vetor ordenado em ordem crescente estando 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1566
32	1507
64	2394
128	8107
256	10217
512	20026
1024	52093
2048	125015
4096	315646
8192	414589
16384	1056987

7.5.1 Gráfico Radix Sort - Vetor Crescente P30

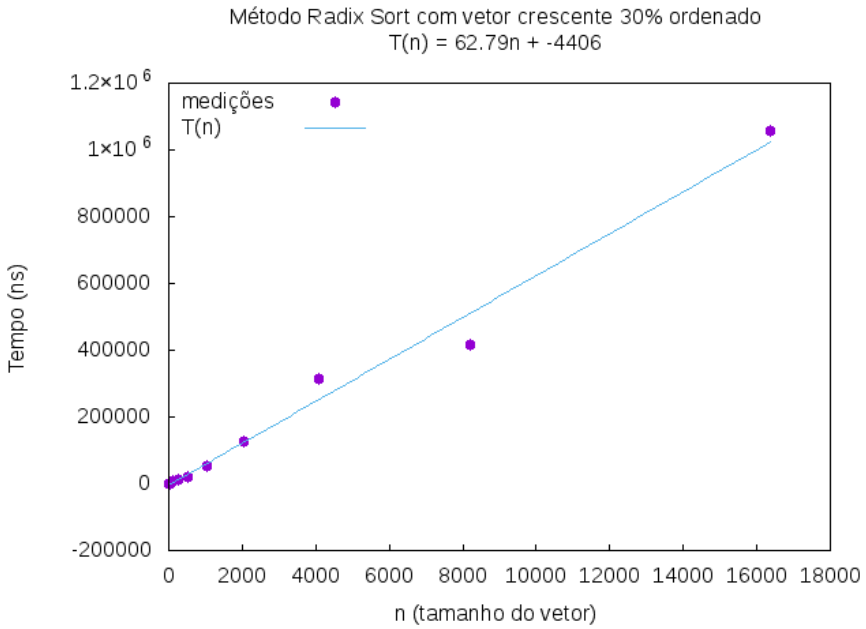

Figura 7.5: *Gráfico Radix Sort - Vetor Crescente P30*

7.6 Radix Sort - Vetor Crescente P40

Tabela gerada utilizando Radix Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 40% ordenado.

Tabela 7.6: *Radix Sort com vetor ordenado em ordem crescente estando 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1511
32	1449
64	2159
128	4897
256	10195
512	19773
1024	50826
2048	105839
4096	228973
8192	412796
16384	1213010

7.6.1 Gráfico Radix Sort - Vetor Crescente P40

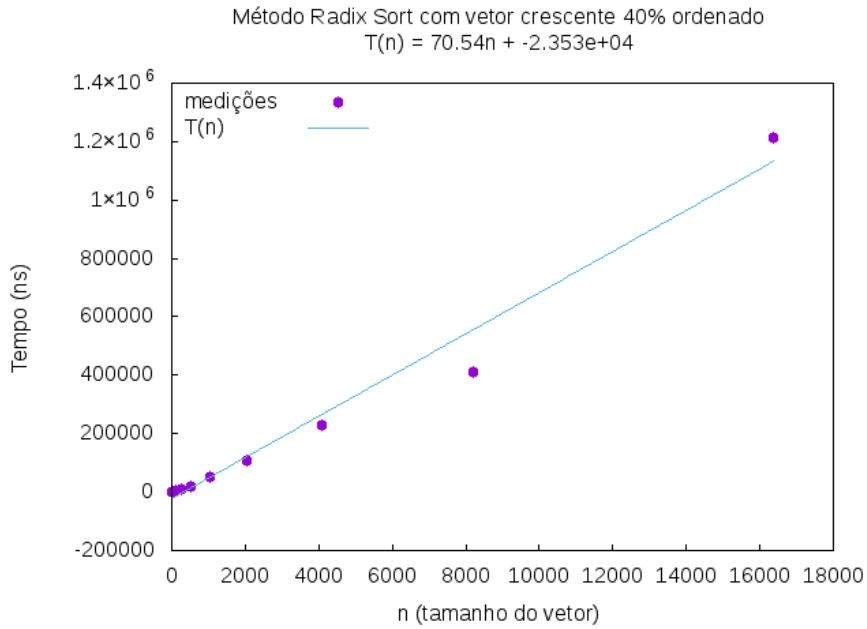

Figura 7.6: *Gráfico Radix Sort - Vetor Crescente P40*

7.7 Radix Sort - Vetor Crescente P50

Tabela gerada utilizando Radix Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem crescente estando 50% ordenado.

Tabela 7.7: *Radix Sort com vetor ordenado em ordem crescente estando 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	2771
32	1584
64	2927
128	6718
256	14551
512	28748
1024	83909
2048	134313
4096	205712
8192	440127
16384	1104181

7.7.1 Gráfico Radix Sort - Vetor Crescente P50

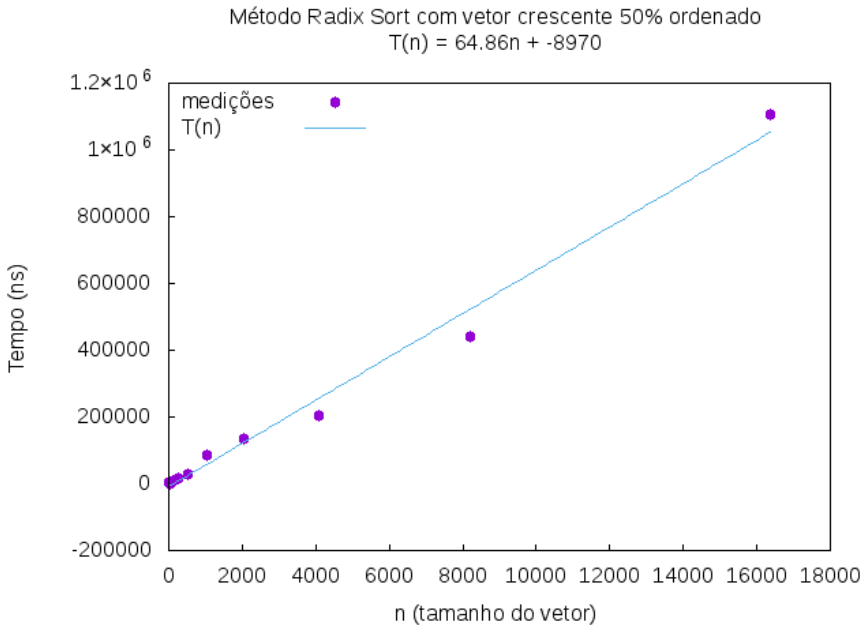

Figura 7.7: *Gráfico Radix Sort - Vetor Crescente P50*

7.8 Radix Sort - Vetor Decrescente

Tabela gerada utilizando Radix Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente.

Tabela 7.8: *Radix Sort com vetor ordenado em ordem decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	1509
32	1380
64	2209
128	4755
256	10156
512	21064
1024	50785
2048	105126
4096	204012
8192	421833
16384	1300937

7.8.1 Gráfico Radix Sort - Vetor Decrescente

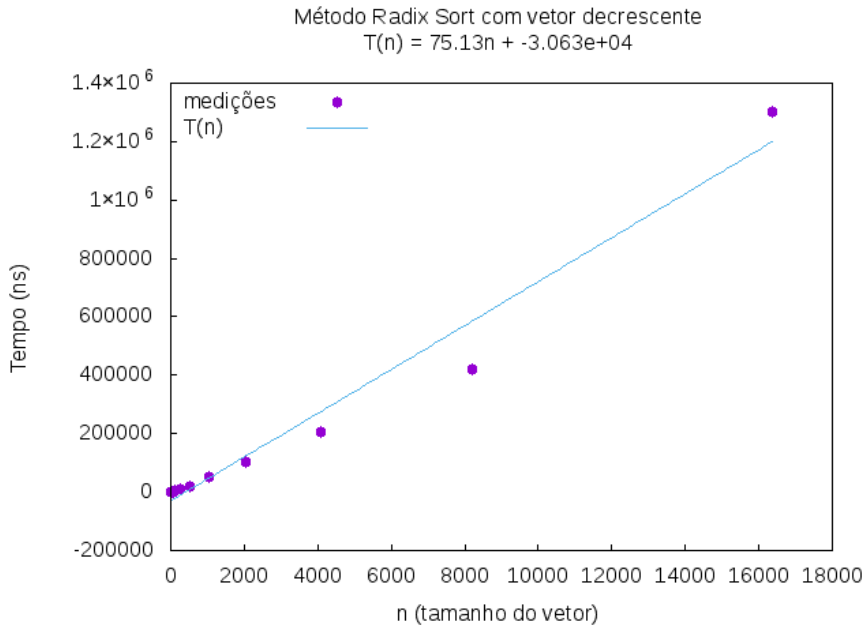

Figura 7.8: *Gráfico Radix Sort - Vetor Decrescente*

7.9 Radix Sort - Vetor Decrescente P10

Tabela gerada utilizando Radix Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 10% ordenado.

Tabela 7.9: *Radix Sort com vetor ordenado em ordem decrescente estando 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1575
32	1407
64	2223
128	4714
256	10106
512	19246
1024	49723
2048	99033
4096	196898
8192	432726
16384	1075879

7.9.1 Gráfico Radix Sort - Vetor Decrescente P10

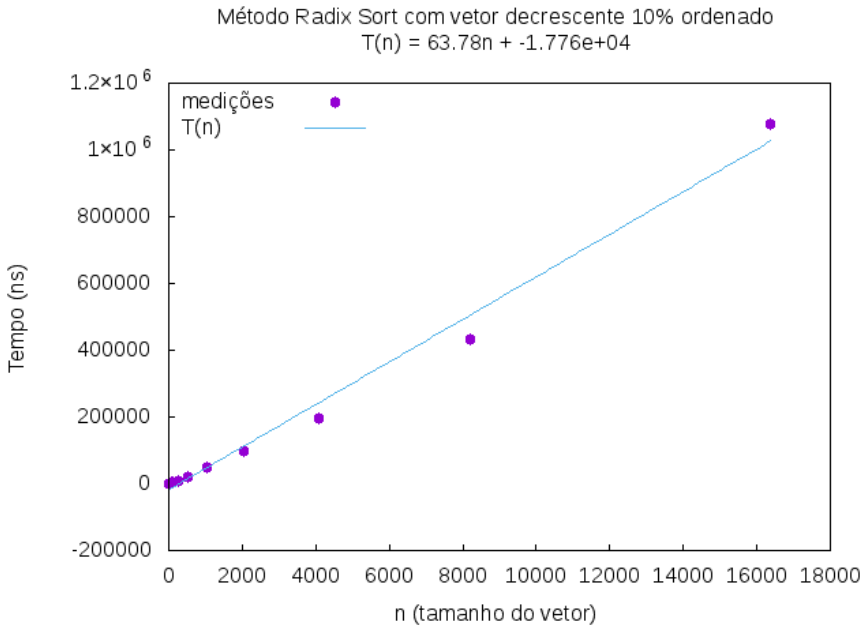

Figura 7.9: *Gráfico Radix Sort - Vetor Decrescente P10*

7.10 Radix Sort - Vetor Decrescente P20

Tabela gerada utilizando Radix Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 20% ordenado.

Tabela 7.10: *Radix Sort com vetor ordenado em ordem decrescente estando 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1462
32	1415
64	2168
128	4773
256	12913
512	18982
1024	50323
2048	102306
4096	213200
8192	425845
16384	1280537

7.10.1 Gráfico Radix Sort - Vetor Decrescente P20

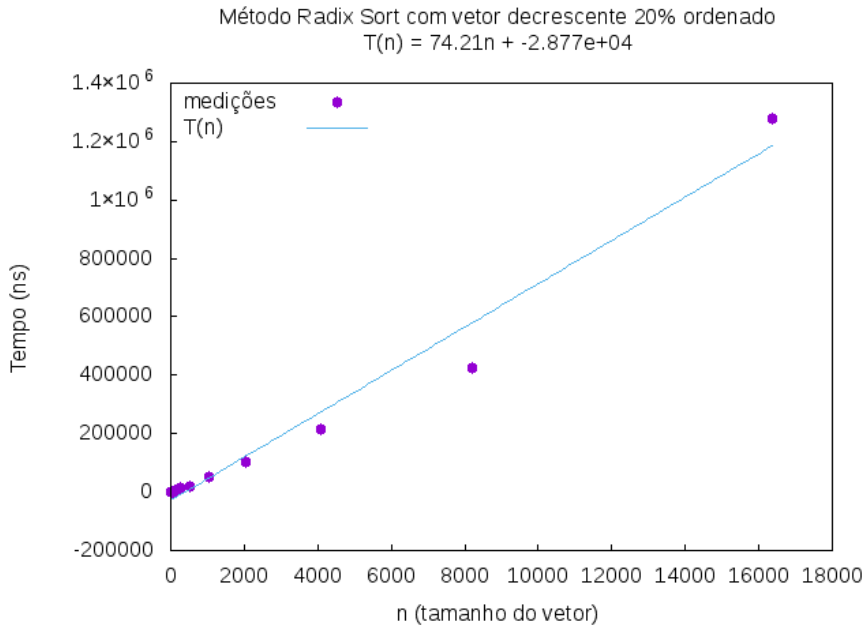

Figura 7.10: *Gráfico Radix Sort - Vetor Decrescente P20*

7.11 Radix Sort - Vetor Decrescente P30

Tabela gerada utilizando Radix Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 30% ordenado.

Tabela 7.11: *Radix Sort com vetor ordenado em ordem decrescente estando 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1642
32	1432
64	2173
128	4948
256	9691
512	18212
1024	49862
2048	101564
4096	206893
8192	430726
16384	1055471

7.11.1 Gráfico Radix Sort - Vetor Decrescente P30

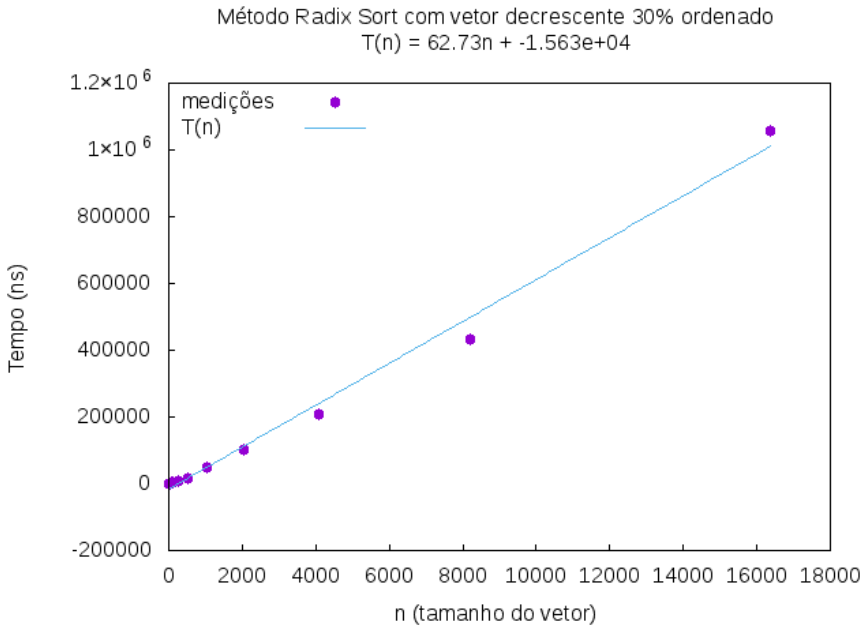

Figura 7.11: *Gráfico Radix Sort - Vetor Decrescente P30*

7.12 Radix Sort - Vetor Decrescente P40

Tabela gerada utilizando Radix Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 40% ordenado.

Tabela 7.12: *Radix Sort com vetor ordenado em ordem decrescente estando 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1306
32	1415
64	2110
128	5098
256	9740
512	19400
1024	50306
2048	107601
4096	217893
8192	469702
16384	1141214

7.12.1 Gráfico Radix Sort - Vetor Decrescente P40

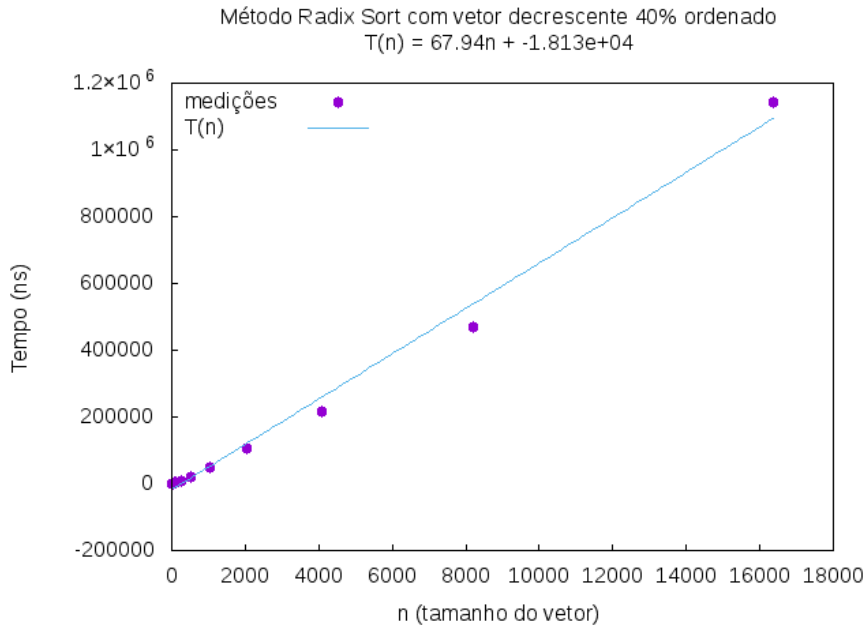

Figura 7.12: *Gráfico Radix Sort - Vetor Decrescente P40*

7.13 Radix Sort - Vetor Decrescente P50

Tabela gerada utilizando Radix Sort com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos em ordem decrescente estando 50% ordenado.

Tabela 7.13: *Radix Sort com vetor ordenado em ordem decrescente estando 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1511
32	1456
64	2208
128	4801
256	8646
512	18827
1024	50937
2048	110078
4096	226903
8192	411363
16384	1057336

7.13.1 Gráfico Radix Sort - Vetor Decrescente P50

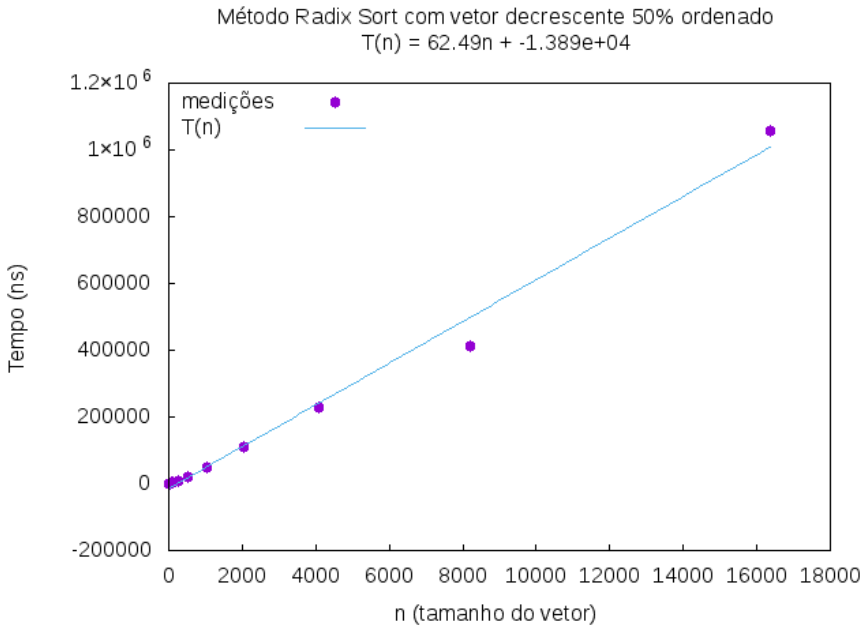

Figura 7.13: *Gráfico Radix Sort - Vetor Decrescente P50*

7.14 Observações Finais

hadix sort com vetor totalmente decrescente levaria aproximadamente 4 minutos e 34 segundos para processar um vetor de 2^k com k = 32 elementos.

Capítulo 8

Bucket sort

Bucket sort é um algoritmo de ordenação que funciona dividindo um vetor em um número finito de recipientes. Cada recipiente é então ordenado individualmente, seja usando um algoritmo de ordenação diferente, ou usando o algoritmo bucket sort recursivamente. Complexidade no pior caso O(n^2) e no melhor caso caso O(n+k).

8.1 Bucket sort - Vetor Aleatório

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

Tabela 8.1: *Bucket sort com Vetor aleatório*

Número de Elementos	**Tempo de execução em nanosegundos**
16	1973
32	1577
64	3014
128	8748
256	23934
512	89199
1024	284571
2048	1087923
4096	4294051
8192	16840640
16384	66475936

8.1.1 Gráfico Bucket sort - Vetor Aletório

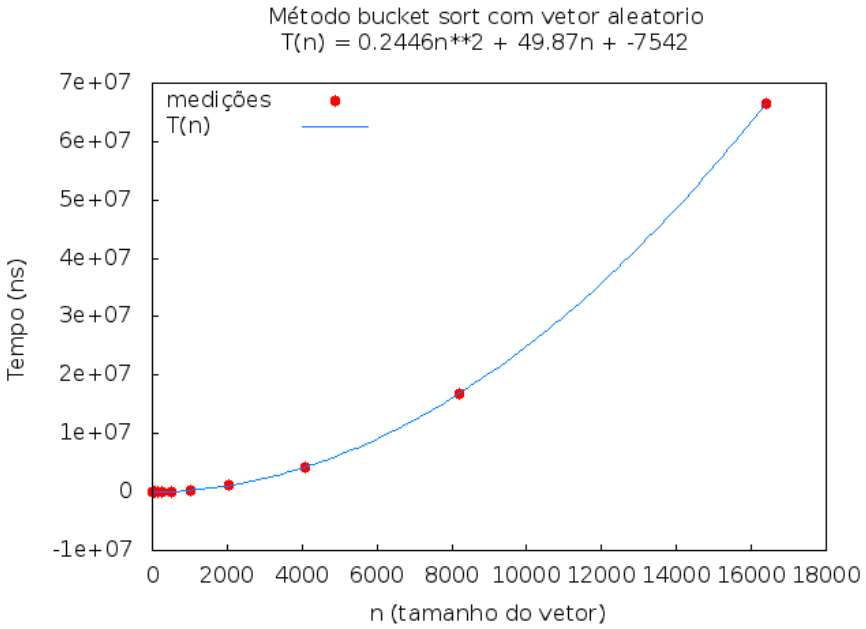

Figura 8.1: *Gráfico Bucket sort - Vetor Aleatorio*

8.2 Bucket sort - Vetor Crescente

Tabela gerada utilizando Bucket sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem crescente.

Tabela 8.2: *Bucket sort com Vetor ordenado em ordem crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	1462
32	1276
64	1598
128	2201
256	5808
512	5298
1024	8476
2048	23017
4096	57903
8192	98423
16384	218516

8.2.1 Grafico Bucket sort - Vetor Crescente

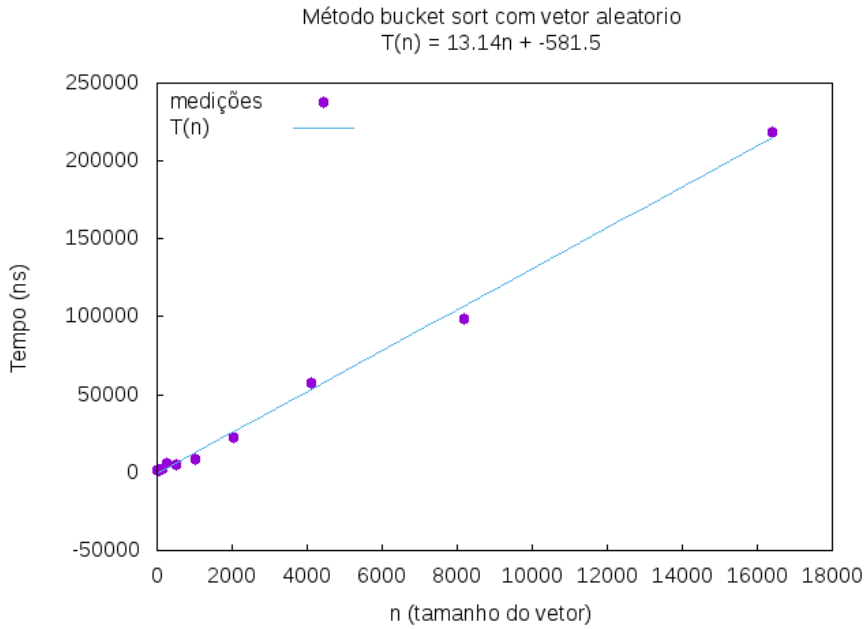

Figura 8.2: *Gráfico Bucket sort - Vetor Crescente*

8.3 Bucket sort - Vetor Crescente P10

Tabela gerada utilizando Bucket sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem crescente estando 10% ordenado.

Tabela 8.3: *Bucket sort com Vetor ordenado em ordem crescente 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1225
32	1321
64	1898
128	3800
256	8864
512	29263
1024	112675
2048	426087
4096	1724471
8192	6498292
16384	25599240

8.3.1 Grafico Bucket sort - Vetor Crescente P10

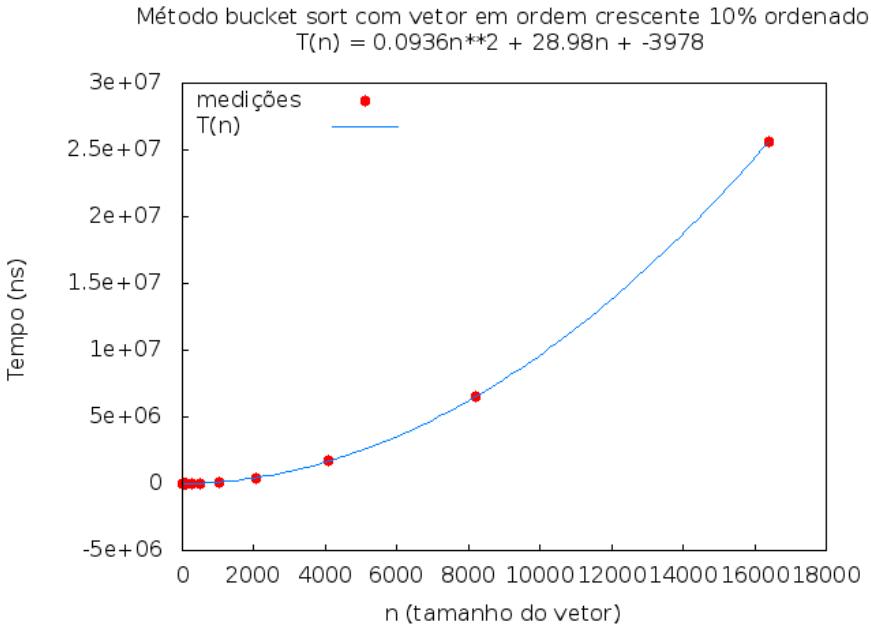

Figura 8.3: *Gráfico Bucket sort - Vetor Crescente P10*

8.4 Bucket sort - Vetor Crescente P20

Tabela gerada utilizando Bucket sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem crescente estando 20% ordenado.

Tabela 8.4: *Bucket sort com Vetor ordenado em ordem crescente 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1394
32	1396
64	2319
128	5109
256	14409
512	54023
1024	206030
2048	782374
4096	3106445
8192	12244799
16384	48165325

8.4.1 Grafico Bucket sort - Vetor Crescente P20

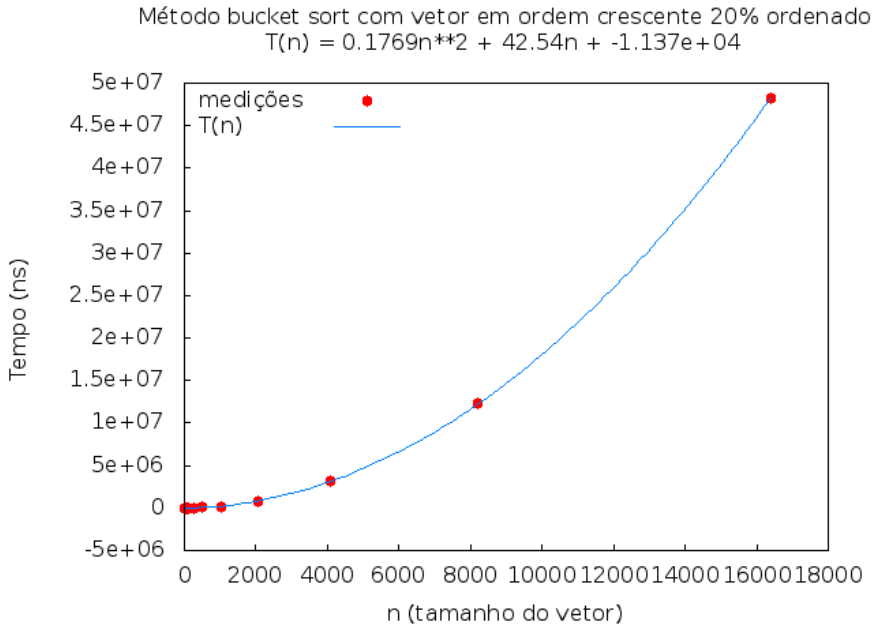

Figura 8.4: *Gráfico Bucket sort - Vetor Crescente P20*

8.5 Bucket sort - Vetor Crescente P30

Tabela gerada utilizando Bucket sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem crescente estando 30% ordenado.

Tabela 8.5: *Bucket sort com Vetor ordenado em ordem crescente 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1420
32	1477
64	3137
128	6966
256	20704
512	72550
1024	276812
2048	1088868
4096	4363382
8192	17125393
16384	67952052

8.5.1 Grafico Bucket sort - Vetor Crescente P30

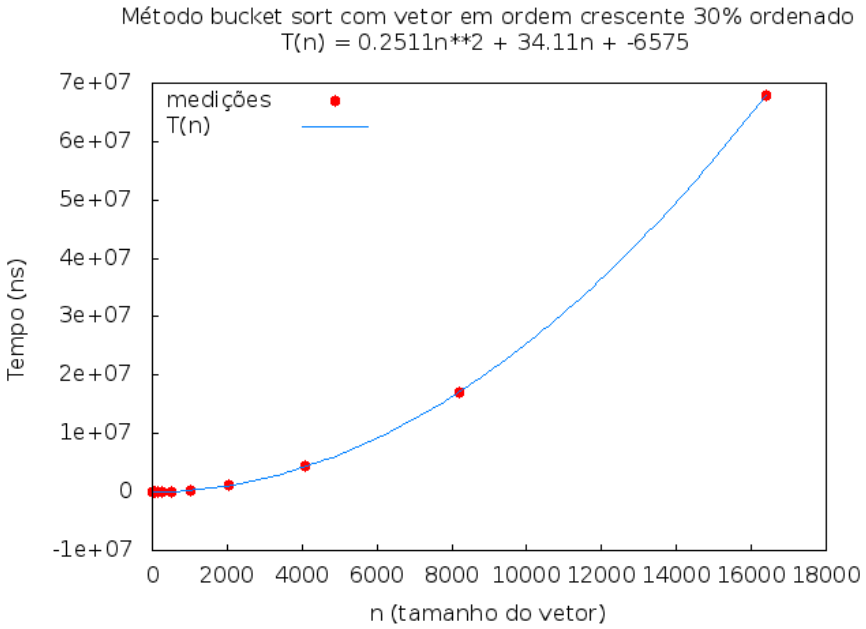

Figura 8.5: *Gráfico Bucket sort - Vetor Crescente P30*

8.6 Bucket sort - Vetor Crescente P40

Tabela gerada utilizando Bucket sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem crescente estando 40% ordenado.

Tabela 8.6: *Bucket sort com Vetor ordenado em ordem crescente 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1535
32	1722
64	2884
128	7462
256	24820
512	89196
1024	346035
2048	1363420
4096	5462719
8192	21522353
16384	85206807

8.6.1 Grafico Bucket sort - Vetor Crescente P40

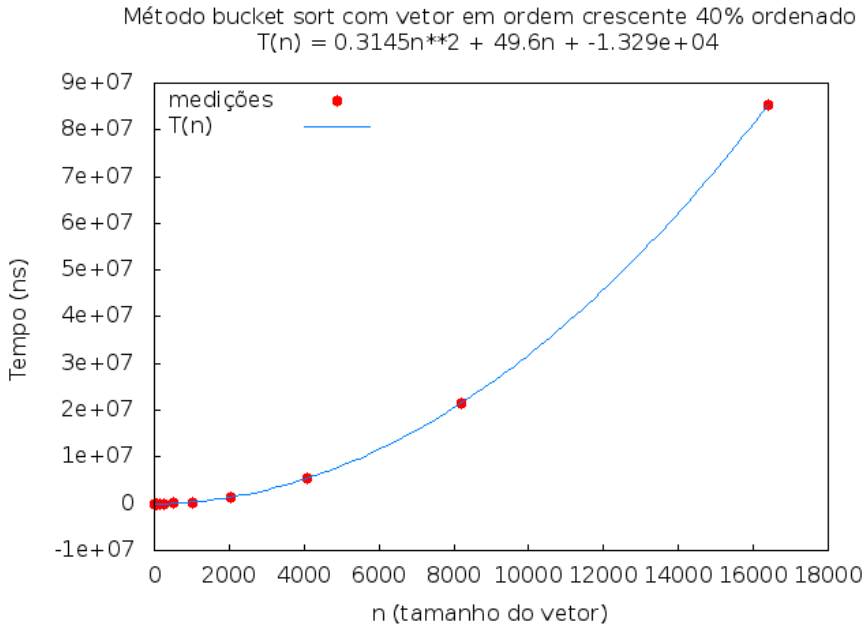

Figura 8.6: *Gráfico Bucket sort - Vetor Crescente P40*

8.7 Bucket sort - Vetor Crescente P50

Tabela gerada utilizando Bucket sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem crescente estando 50% ordenado.

Tabela 8.7: *Bucket sort com Vetor ordenado em ordem crescente 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1495
32	1722
64	3257
128	9138
256	34405
512	103202
1024	427482
2048	1597461
4096	6416766
8192	25148241
16384	99797143

8.7.1 Grafico Bucket sort - Vetor Crescente P50

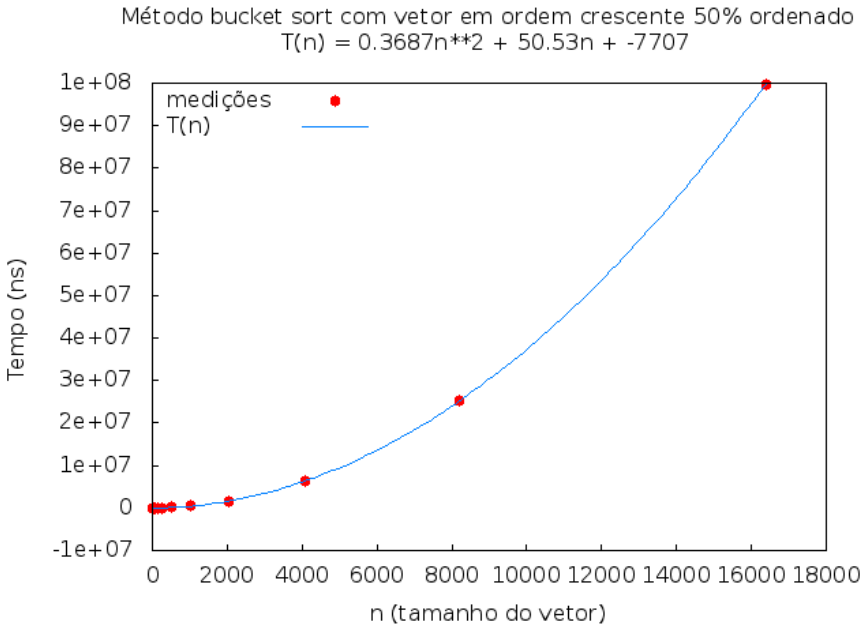

Figura 8.7: *Gráfico Bucket sort - Vetor Crescente P50*

8.8 Bucket sort - Vetor Decrescente

Tabela gerada utilizando Bucket sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem decrescente.

Tabela 8.8: *Bucket sort com Vetor ordenado em ordem decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	1733
32	1935
64	3800
128	10621
256	35688
512	134053
1024	535638
2048	2105018
4096	8465724
8192	33308407
16384	132619442

8.8.1 Grafico Bucket sort - Vetor Decrescente

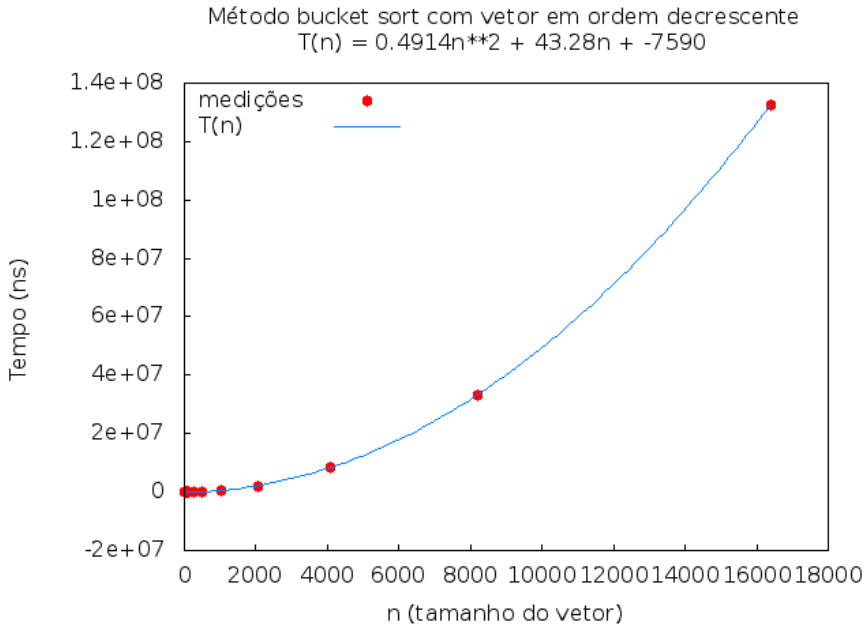

Figura 8.8: *Gráfico Bucket sort - Vetor Decrescente*

8.9 Bucket sort - Vetor Decrescente P10

Tabela gerada utilizando Bucket sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem descrescente estando 10% ordenado.

Tabela 8.9: *Bucket sort com Vetor ordenado em ordem descrecente 10% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1652
32	1926
64	3661
128	9606
256	30709
512	111680
1024	444689
2048	1735199
4096	6904035
8192	27147001
16384	108381572

8.9.1 Grafico Bucket sort - Vetor Decrescente P10

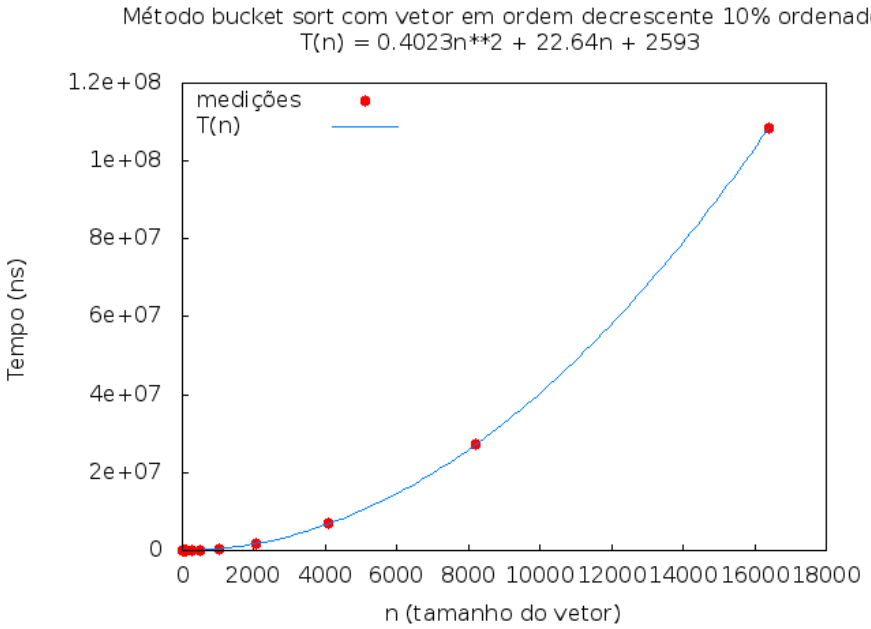

Figura 8.9: *Gráfico Bucket sort - Vetor Decrescente P10*

8.10 Bucket sort - Vetor Decrescente P20

Tabela gerada utilizando Bucket sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem descrescente estando 20% ordenado.

Tabela 8.10: *Bucket sort com Vetor ordenado em ordem descrecente 20% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1594
32	1710
64	3260
128	8207
256	25049
512	89341
1024	343952
2048	1362340
4096	5533807
8192	21616313
16384	86167002

8.10.1 Grafico Bucket sort - Vetor Decrescente P20

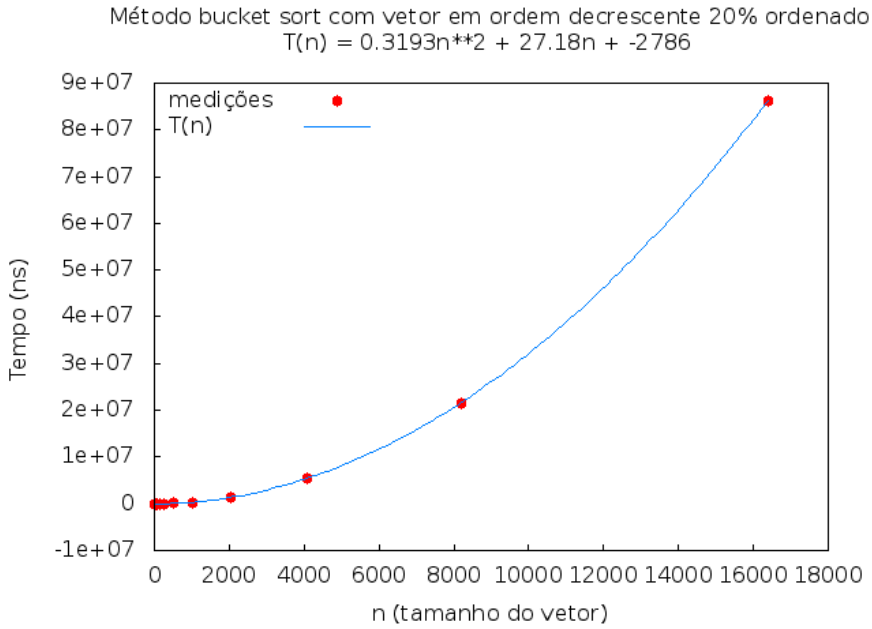

Figura 8.10: *Gráfico Bucket sort - Vetor Decrescente P20*

8.11 Bucket sort - Vetor Decrescente P30

Tabela gerada utilizando Bucket sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem descrescente estando 30% ordenado.

Tabela 8.11: *Bucket sort com Vetor ordenado em ordem descrecente 30% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1615
32	1701
64	2935
128	6735
256	19846
512	70981
1024	267001
2048	1048944
4096	4196396
8192	16480503
16384	65337027

8.11.1 Grafico Bucket sort - Vetor Decrescente P30

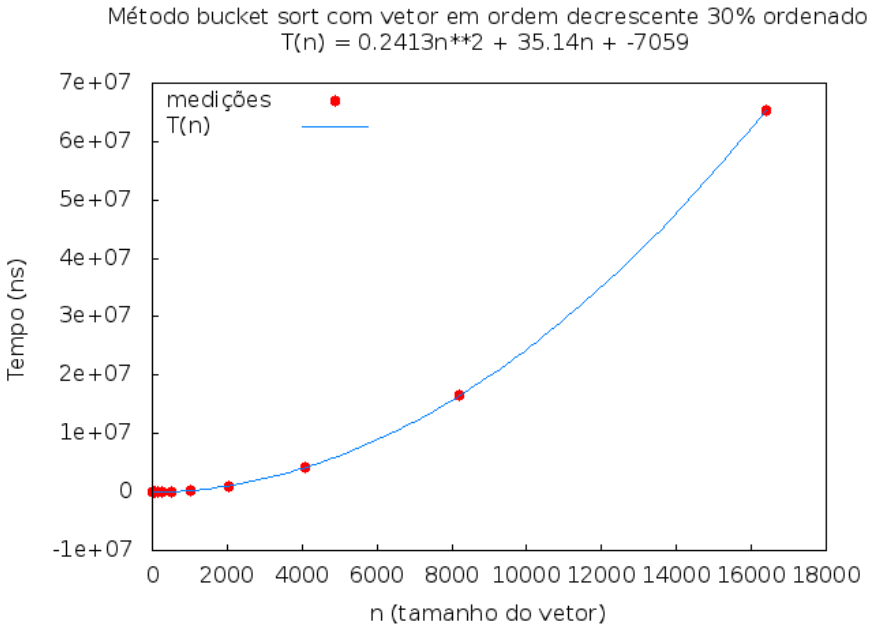

Figura 8.11: *Gráfico Bucket sort - Vetor Decrescente P30*

8.12 Bucket sort - Vetor Decrescente P40

Tabela gerada utilizando Bucket sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem descrescente estando 40% ordenado.

Tabela 8.12: *Bucket sort com Vetor ordenado em ordem descrecente 40% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1540
32	1635
64	2675
128	5730
256	15468
512	53749
1024	197174
2048	783752
4096	3138458
8192	12246283
16384	48159817

8.12.1 Grafico Bucket sort - Vetor Decrescente P40

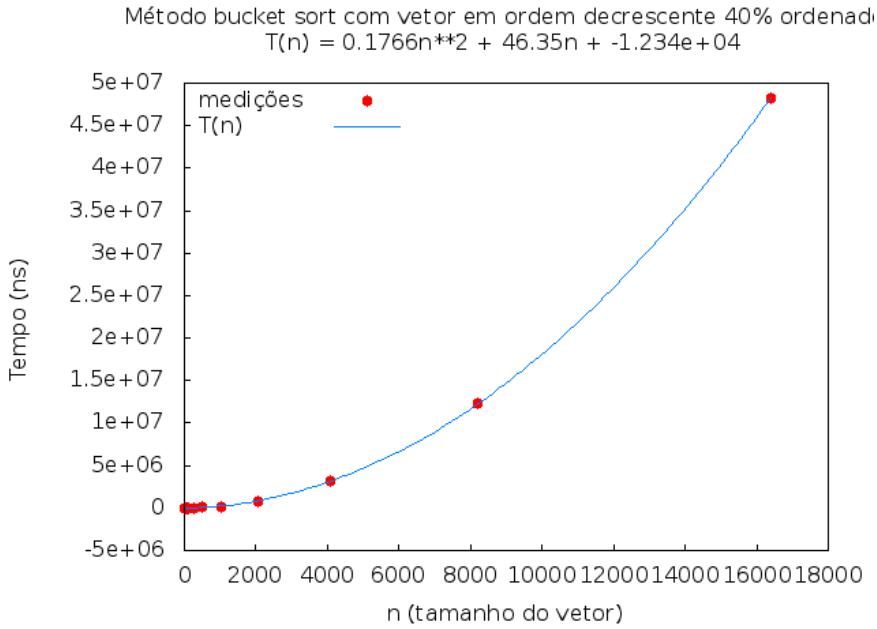

Figura 8.12: *Gráfico Bucket sort - Vetor Decrescente P40*

8.13 Bucket sort - Vetor Decrescente P50

Tabela gerada utilizando Bucket sort com vetores de tamanho n, sendo = (2^k), k = 4...14 e inseridos em ordem descrescente estando 50% ordenado.

Tabela 8.13: *Bucket sort com Vetor ordenado em ordem descrecente 50% ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	1541
32	1490
64	2264
128	4583
256	11730
512	38408
1024	144989
2048	539921
4096	2153095
8192	8579882
16384	33392472s

8.13.1 Grafico Bucket sort - Vetor Decrescente P50

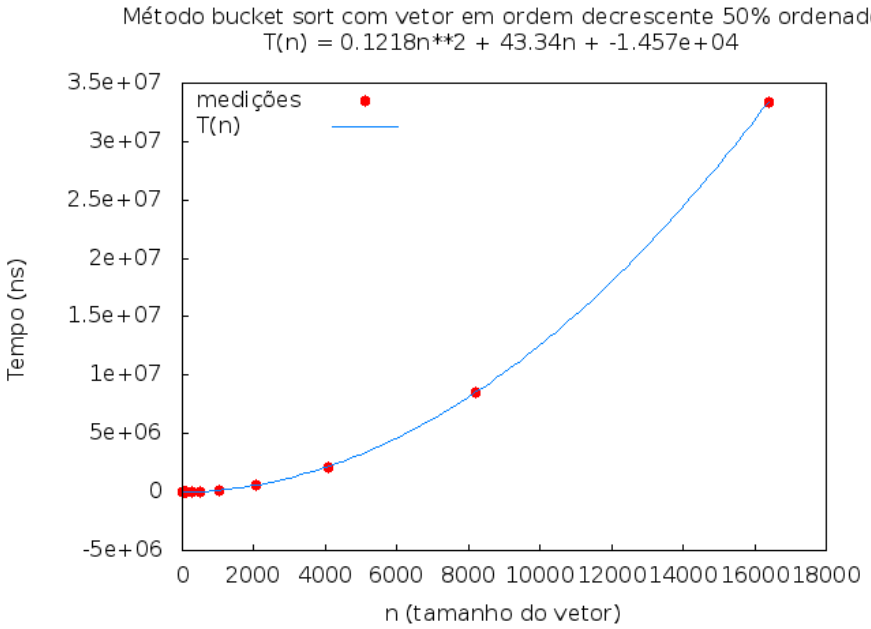

Figura 8.13: *Gráfico Bucket sort - Vetor Decrescente P50*

8.14 Observações Finais

Bucket sort com vetor em ordem aleatória com 2^k com k = 32 elementos levaria aproximadamente 287 anos 2 meses e 30 dias para processar nessas condições.

Capítulo 9

Referências

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