UNIVERSIDADE FEDERAL DO ABC

Heitor Rodrigues Savegnago
Allan de Moraes Navarro

Comparação entre algoritmos de ordenação

Santo André - SP
2017

1 Resultados

Os resultados dos processamentos de cada algoritmo de ordenação é apresentado na figura 1. Além dos métodos indicados na proposta de projeto, também foram analisadas situações onde o `QuickSort` tem pivô central e pivô aleatório.

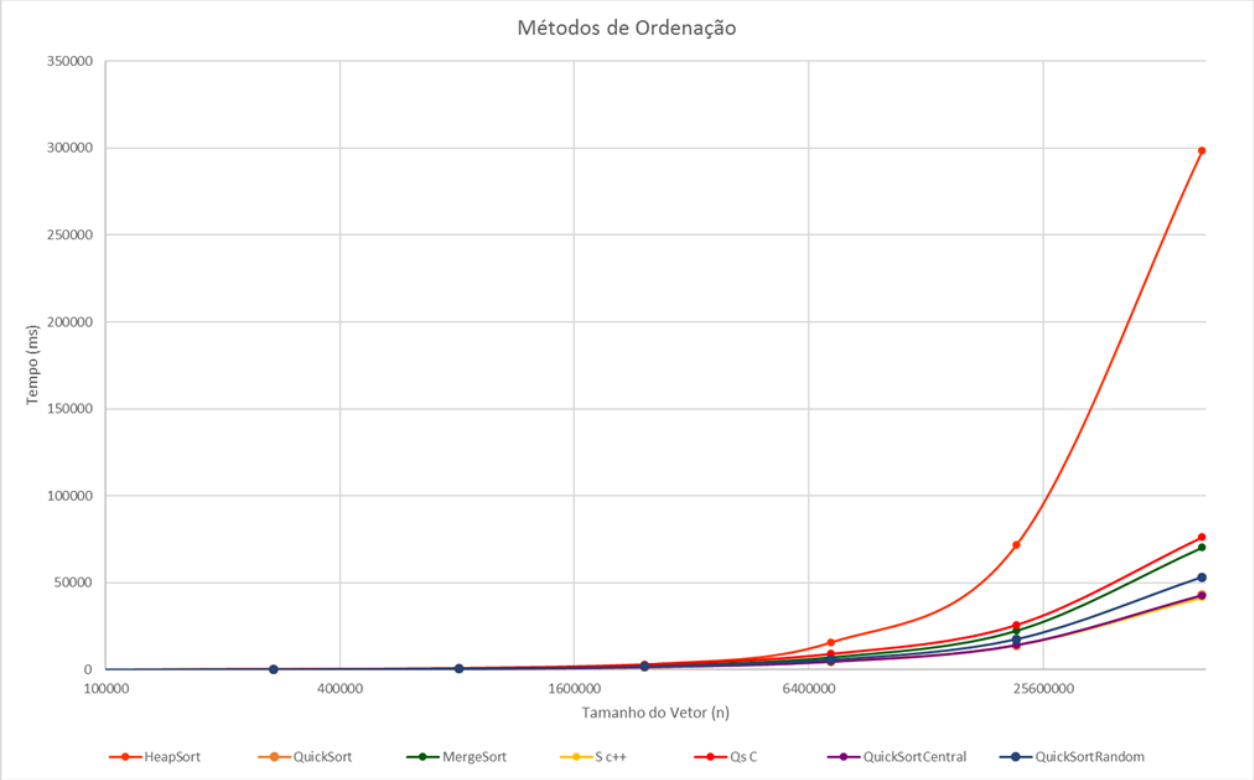

Figura 1: Gráfico comparativo para os algoritmos.

2 O Código

Os métodos selecionados foram:

- `HeapSort`;
- `QuickSort`, com pivô no primeiro item;
- `MergeSort`;
- `Sort`, função da biblioteca `algorithm`;
- `qSort`, função da biblioteca `cstdio`;
- `InsertSort`
- `QuickSort`, com pivô no item centra;
- `QuickSort`, com pivô em item aleatório;
- `SelectSort`;
- `RadixSort`;
- `BubbleSort`;
- `BeadSort`.

Infelizmente nem todos os métodos puderam ser analisados devido ao tempo de processamento.

O código fonte do programa utilizado para a verificação dos métodos é apresentado aqui, mas também está disponível em um diretório do `GitHub`, acessível pelo endereço apresentado na figura 2

Figura 2: Código QR para acesso do link `github.com/HeckRodSav/ProjAED`.

2.1 Estrutura principal

```cpp
#include <iostream>
#include <iomanip>
#include <random>
#include <time.h>
#include <cstdlib>
#include <string>
#include <cstdio>
#include <algorithm>
#include <fstream>
#include "Ordem.cpp"

//#define DEBUG
#define DEBUGnumber 0

#define sizeofV(X) sizeof(X) / sizeof(X[0])
#define SIZEx 1048576

using namespace std;

typedef short int si;
typedef unsigned short int usi;
typedef unsigned int ui;
typedef long int li;
```

```
24 typedef unsigned long int uli;
25
26 fstream fout;
27
28 ui seeed[] = { 4, 81, 151, 1601, 2307, 4207 };
29 uli sizee[] = { 10000, 30000, 90000, 270000, 810000, 2430000,
      7290000, 21870000, 65610000 };
30 //uli sizee[] = { 10, 30, 90, 270, 810, 2430, 729, 2187, 6561 };
31 //uli sizee[] = { 5, 10, 50, 100, 500, 1000, 5000, 10000, 50000 };
32 //uli sizee[] = { 50, 100, 150, 200, 250, 300, 350, 400, 450, 500 };
33 string algorithm_name[] = {
34   "HeapSort",      //0
35   "QuickSort",     //1
36   "MergeSort",     //2
37   "S c++",       //3
38   "Qs C",        //4
39   "InsertSort",    //5
40   "QuickSortCentral", //6
41   "QuickSortRandom",  //7
42   "SelectSort",    //8
43   "RadixSort",     //9
44   "BubbleSort",    //10
45   "BeadSort" };    //11
46
47 string file_name = "results.txt";
48
49 void pause()
50 {
51   system("pause");
52 }
53
54 int main()
55 {
56   cout << '\a';
57 #ifdef DEBUG
58
59 #if DEBUGnumber == 0
60   /*ui ini, fim;
61   cout << "Inicio do intervalo: ";
62   cin >> ini;
63   cout << endl << "Fim do intervalo: ";
64   cin >> fim;
65   cout << endl;
66   uniform_int_distribution<int> sqn(ini, fim);
67   */
68
69   typedef unsigned int type;
70 #define LEN 20
```

```
71
72    uniform_int_distribution<type> sqn(0, SIZEx);
73    mt19937 rad;
74    rad.seed(clock());
75    type *teste = new type[LEN];
76    //type teste[LEN];
77    for (ui i = 0; i < LEN; i++) teste[i] = sqn(rad);
78    for (ui i = 0; i < LEN; i++) cout << teste[i] << ' ';
79    cout << endl;
80
81
82    ED::Ordem<type, int>::QuickSortCentral(teste, 0, LEN);
83
84    cout << endl;
85
86    for (ui i = 0; i < LEN; i++) cout << teste[i] << ' ';
87    cout << endl;
88
89  #elif DEBUGnumber == 1
90
91    auto n = 10;
92    auto m = 5;
93    for (int i = 0; i < n; i++) cout << ((int)(rand() / (double)(
          RAND_MAX + 1.0) * m) + 1) << ' '; cout << endl;
94
95  #else
96    unsigned int a = 0;
97    cout << "Nothing" << endl << --a << endl;
98  #endif
99
100 #else
101
102   double start = 0, end = 0;
103   ui *V = new ui[sizee[sizeofV(sizee)-1]];
104
105   fout.open(file_name, ios::app);
106   uniform_int_distribution<ui> seq(0,SIZEx);
107   mt19937 rnd;
108
109   for (ui i = 9; i < sizeofV(algorithm_name); i++) //Select
          algorithm
110   {
111     cout << setfill(' ') << setw(15) << left << algorithm_name[i] <<
            "  "; //Show name
112     fout << setfill(' ') << setw(15) << left << algorithm_name[i] <<
            "  ";
113     for (ui k = 0; k < sizeofV(seeed); k++)
114     {
```

```
115         cout << seeed[k] << '\t'; //Show index
116         fout << seeed[k] << '\t';
117       }
118       cout << endl;
119       fout << endl;
120
121       for (ui j = 0; j < sizeofV(sizee); j++) //Select size of vector
122       {
123         for (uli l = 0; l < sizee[j]; l++) V[l] = 0; //Clean the
              vector in the used space
124
125         cout << "size: " << setfill(' ') << setw(9) << left << sizee[j
              ] << ": "; //Show size
126         fout << "size: " << setfill(' ') << setw(9) << left << sizee[j
              ] << ": "; //Show size
127
128         for (ui k = 0; k < sizeofV(seeed); k++) //Select current seed
129         {
130           rnd.seed(seeed[k]); //Set random seed
131           for (uli l = 0; l < sizee[j]; l++) V[l] = seq(rnd); //Set
                random vector
132
133           start = clock();
134           switch (i)
135           {
136           case 0:  ED :: Ordem<ui, li>  :: HeapSort(V, sizee[j]);
                break; /* HeapSort      */
137           case 1:  ED :: Ordem<ui, li>  :: QuickSort(V, 0, sizee[j]);
                  break; /* QuickSort     */
138           case 2:  ED :: Ordem<ui, uli> :: MergeSort(V, sizee[j]);
                break; /* MergeSort     */
139           case 3:  ED :: Ordem<ui, uli> :: Sort(V, sizee[j]);
                break; /* S C++         */
140           case 4:  ED :: Ordem<ui, uli> :: qSort(V, sizee[j]);
                break; /* Qs C          */
141           case 5:  ED :: Ordem<ui, li>  :: InsertSort(V, sizee[j]);
                break; /* InsertSort    */
142           case 6:  ED :: Ordem<ui, li>  :: QuickSortCentral(V, 0,
                sizee[j]);  break; /* QuickSortCentral  */
143           case 7:  ED :: Ordem<ui, li>  :: QuickSortRandom(V, 0, sizee
                [j]); break; /* QuickSortRandom */
144           case 8:  ED :: Ordem<ui, uli> :: SelectSort(V, sizee[j]);
                break; /* SelectSort    */
145           case 9:  ED :: Ordem<ui, uli> :: RadixSort(V, sizee[j]);
                break; /* BubbleSort    */
146           case 10: ED :: Ordem<ui, uli> :: BubbleSort(V, sizee[j]);
                break; /* BubbleSort    */
147           case 11: ED :: Ordem<ui, ui>  :: BeadSort(V, sizee[j], SIZEx
```

```
            );    break; /* BeadtSort     */
148         default: cout << '?'; break;
149         }
150         end = clock();
151         cout << 1000.0 * (end - start) / (CLOCKS_PER_SEC) << '\t';
152         fout << 1000.0 * (end - start) / (CLOCKS_PER_SEC) << '\t';
153       }
154       cout << endl;
155       fout << endl;
156
157     }
158     cout << endl; //Break line after a algorithm
159     fout << endl;
160   }
161   fout.close();
162 #endif
163   cout << '\a';
164   pause();
165   return 0;
166 }
```

2.2 Classe com ordenadores

```
1  #pragma once
2  #include <cstdlib>
3  #include <algorithm>
4  #include <iostream>
5
6  namespace ED
7  {
8    template <typename Tipo, typename Size> class Ordem
9    {
10     static void heapify(Tipo *vet, Size pai, Size heapsize);
11     static void merge(Tipo *vet, Tipo *aux, Size esq, Size meio,
          Size dir);
12     static void m_sort(Tipo *vet, Tipo *aux, Size esq, Size dir);
13   public:
14     static void InsertSort(Tipo* vet, Size length);
15     static void HeapSort(Tipo *vet, Size length);
16     static void MergeSort(Tipo *vet, Size length);
17     static void BubbleSort(Tipo *vet, Size length);
18     static void QuickSort(Tipo *vet, Size start, Size end);
19     static void QuickSortCentral(Tipo *vet, Size start, Size end);
20     static void QuickSortRandom(Tipo *vet, Size start, Size end);
21     static void SelectSort(Tipo *vet, Size lenght);
22     static void RadixSort(Tipo* vet, Size length);
23     static void BeadSort(Tipo * vet, Size length, Tipo Max);
```

```
24      static void qSort(Tipo * vet, Size length);
25      static void Sort(Tipo * vet, Size length);
26
27    };
28
29
30    template<typename Tipo, typename Size> inline void Ordem<Tipo,
          Size>::heapify(Tipo * vet, Size pai, Size heapsize)
31    {
32      Size fl, fr, imaior;
33      fl = (pai << 1) + 1;
34      fr = fl + 1;
35      while (true)
36      {
37        if ((fl < heapsize) && (vet[fl] > vet[pai])) imaior = fl;
38        else imaior = pai;
39        if ((fr < heapsize) && (vet[fr] > vet[imaior])) imaior = fr;
40        if (imaior != pai)
41        {
42          Tipo aux = vet[pai];
43          vet[pai] = vet[imaior];
44          vet[imaior] = aux;
45          pai = imaior;
46          fl = (pai << 1) + 1;
47          fr = fl + 1;
48        }
49        else break;
50      }
51    }
52
53    template<typename Tipo, typename Size> inline void Ordem<Tipo,
          Size>::merge(Tipo * vet, Tipo * aux, Size esq, Size meio, Size
          dir) {
54      Size i, j, k;
55      i = k = esq; j = meio + 1;
56      while ((i <= meio) && (j <= dir)) {
57        if (vet[i] < vet[j]) aux[k++] = vet[i++];
58        else aux[k++] = vet[j++];
59      }
60      while (i <= meio) aux[k++] = vet[i++];
61      while (j <= dir) aux[k++] = vet[j++];
62      while (esq <= dir) vet[esq] = aux[esq++];
63    }
64
65    template<typename Tipo, typename Size> inline void Ordem<Tipo,
          Size>::m_sort(Tipo * vet, Tipo * aux, Size esq, Size dir)
66    {
67      if (dir <= esq) return;
```

```
68      Size meio = (dir + esq) >> 1;
69      m_sort(vet, aux, esq, meio); //First Call
70      m_sort(vet, aux, meio + 1, dir); // Second Call
71      if (vet[meio] <= vet[meio + 1]) return;
72      merge(vet, aux, esq, meio, dir); // Merge
73    }
74
75    template<typename Tipo, typename Size> inline void Ordem<Tipo,
          Size>::InsertSort(Tipo * vet, Size length)
76    {
77      Tipo aux;
78      Size i = 0, j = 0;
79      for (j = 1; j < length; j++)
80      {
81        aux = vet[j];
82        i = j - 1;
83        while ((i >= 0) && (vet[i] > aux))
84        {
85          vet[i + 1] = vet[i];
86          i--;
87        }
88        vet[i + 1] = aux;
89      }
90    }
91
92    template<typename Tipo, typename Size> inline void Ordem<Tipo,
          Size>::HeapSort(Tipo * vet, Size length)
93    {
94      Size i;
95      for (i = (length >> 1) - 1; i >= 0; i--) heapify(vet, i, length)
            ;
96      for (i = length - 1; i > 0; i--)
97      {
98        Tipo aux = vet[0];
99        vet[0] = vet[i];
100       vet[i] = aux;
101       heapify(vet, 0, i);
102     }
103   }
104
105   template<typename Tipo, typename Size> inline void Ordem<Tipo,
          Size>::MergeSort(Tipo * vet, Size length)
106   {
107     Tipo *aux = new Tipo[length]; // Alocacao do vetor auxiliar
108     m_sort(vet, aux, 0, length - 1);
109     delete aux;
110   }
111
```

```
112  template<typename Tipo, typename Size> inline void Ordem<Tipo,
       Size>::BubbleSort(Tipo * vet, Size length)
113  {
114    for (Size i = length - 1; i > 0; i--)
115    {
116      for (Size j = 0; j < i; j++)
117      {
118        if (vet[j] > vet[j + 1])
119        {
120          Tipo a = vet[j];
121          vet[j] = vet[j + 1];
122          vet[j + 1] = a;
123        }
124      }
125    }
126  }
127
128  template<typename Tipo, typename Size> inline void Ordem<Tipo,
       Size>::QuickSort(Tipo * vet, Size start, Size end)
129  {
130    if (end <= start) return;
131
132    Size i = start, j = end;
133    Tipo pivo = vet[start];
134    while (true) {
135      while ((j > i) && (vet[j] > pivo)) j--;
136      if (i == j) break;
137      vet[i] = vet[j]; i++;
138      while ((i < j) && (vet[i] < pivo)) i++;
139      if (i == j) break;
140      vet[j] = vet[i]; j--;
141    }
142    vet[i] = pivo;
143
144    QuickSort(vet, start, i - 1);
145    QuickSort(vet, i + 1, end);
146  }
147
148  template<typename Tipo, typename Size> inline void Ordem<Tipo,
       Size>::QuickSortCentral(Tipo * vet, Size start, Size end)
149  {
150    if (end <= start) return;
151    Size i, j;
152
153    i = (start + end) / 2;
154    Tipo pivo = vet[i];
155    vet[i] = vet[start];
156    vet[start] = pivo;
```

```
157
158       i = start;
159       j = end;
160
161       while (true) {
162         while ((j > i) && (vet[j] > pivo)) j--;
163         if (i == j) break;
164         vet[i] = vet[j]; i++;
165         while ((i < j) && (vet[i] < pivo)) i++;
166         if (i == j) break;
167         vet[j] = vet[i]; j--;
168       }
169       vet[i] = pivo;
170       QuickSortCentral(vet, start, i - 1);
171       QuickSortCentral(vet, i + 1, end);
172     }
173
174     template<typename Tipo, typename Size> inline void Ordem<Tipo,
          Size>::QuickSortRandom(Tipo * vet, Size start, Size end)
175     {
176       if (end <= start) return;
177       Size i, j;
178       i = rand() % (end - start) + start + 1;
179       Tipo pivo = vet[i];
180       vet[i] = vet[start];
181       vet[start] = pivo;
182
183       i = start;
184       j = end;
185
186       while (true) {
187         while ((j > i) && (vet[j] > pivo)) j--;
188         if (i == j) break;
189         vet[i] = vet[j]; i++;
190         while ((i < j) && (vet[i] < pivo)) i++;
191         if (i == j) break;
192         vet[j] = vet[i]; j--;
193       }
194       vet[i] = pivo;
195       QuickSortRandom(vet, start, i - 1);
196       QuickSortRandom(vet, i + 1, end);
197     }
198
199     template<typename Tipo, typename Size> inline void Ordem<Tipo,
          Size>::SelectSort(Tipo * vet, Size length)
200     {
201       Tipo aux;
202       Size imenor, i, j;
```

```
203       for (i = 0; i < length - 1; i++) {
204         imenor = i;
205         for (j = i + 1; j < length; j++)
206           if (vet[j] < vet[imenor]) imenor = j;
207         aux = vet[i];
208         vet[i] = vet[imenor];
209         vet[imenor] = aux;
210       }
211     }
212
213     template<typename Tipo, typename Size> inline void Ordem<Tipo,
          Size>::RadixSort(Tipo * vet, Size length) {
214       Size i, j;
215       Tipo* temp = new Tipo[length];
216
217       for (Size shift = sizeof(Tipo) * 8 - 1; shift > -1; --shift)
218       {
219         j = 0;
220
221         for (i = 0; i < length; ++i)
222         {
223           bool move = (vet[i] << shift) >= 0;
224
225           if (shift == 0 ? !move : move)
226             vet[i - j] = vet[i];
227           else
228             temp[j++] = vet[i];
229         }
230
231         for (i = 0; i < j; i++)
232         {
233           vet[(length - j) + i] = temp[i];
234         }
235       }
236
237       delete temp;
238     }
239
240     template<typename Tipo, typename Size> inline void Ordem<Tipo,
          Size>::BeadSort(Tipo * vet, Size length, Tipo Max)
241     {
242       // initialize
243       Tipo *level_count = new Tipo[length + 1];
244       Tipo *rod_count = new Tipo[Max + 1];
245       for (Size i = 1; i <= length; i++) level_count[i] = 0;
246       for (Tipo i = 1; i < Max; i++) rod_count[i] = 0;
247       // sort
248       for (Size i = 0; i < length; i++)
```

```
249     {
250       for (Tipo j = 1; j <= vet[i]; j++) {
251         ++level_count[++rod_count[j]];
252       }
253     }
254 
255     //for (int i = 0; i <= length; i++) std::cout << level_count[i]
          << ' '; std::couc << endl;
256 
257     for (Size i = 0; i < length; i++) vet[i] = level_count[length -
          i];
258     delete level_count;
259     delete rod_count;
260   }
261 
262   template<typename Tipo, typename Size> inline void Ordem<Tipo,
       Size>::qSort(Tipo * vet, Size length)
263   {
264     qsort(vet, length, sizeof(Tipo), [](const void *A, const void *B
          ) {return *(Tipo*)A > *(Tipo*)B ? 1 : *(Tipo*)A < *(Tipo*)B ?
           -1 : 0;});
265   }
266 
267   template<typename Tipo, typename Size> inline void Ordem<Tipo,
       Size>::Sort(Tipo * vet, Size length)
268   {
269     sort(vet, vet + length, [](Tipo A, Tipo B) { return A < B;});
270   }
271 }
```