UNIVERSIDADE FEDERAL DE OURO PRETO

Instituto de Ciências Exatas e Aplicadas - ICEA
Departamento de Computação e Sistemas de Informação - DECSI
Campus João Monlevade

Relatório Do Trabalho Prático

Atividade apresentada como parte das exigências da disciplina de Algorítmos e Estruturas de Dados – CSI488 - da Universidade Federal de Ouro Preto.

Discentes:

Professor: Bruno Hott

JOÃO MONLEVADE – MG

Julho/2018

Sumário

1 - Introdução

Este trabalho tem como objetivo fazer uma análise de alguns dos mais famosos algoritmos de ordenação de dados, são eles: BubbleSort, InsertionSort, SeletcionSort, HeapSort, ShellSort e QuickSort, algoritmos que foram criados com o propósito de organizar os dados crescente ou decrescentemente, e como sempre, se espera que um bom algoritmo execute o mais rápido possivel. Os algoritmos foram implementados, testados para vários tamanhos e configurações do vetor e comparados, como mostraremos a seguir.

2 - O código

O código dos algoritmos de ordenação implementados:

```
 8  void bubblesort(int vetor[],int n){
 9      int k, j, aux;
10      int comp=0,atrib=0;
11      for (k = 1; k < n; k++) {
12          for (j = 0; j < n - k; j++) {
13
14              if (vetor[j] > vetor[j + 1]) {
15                  comp++;
16                  aux           = vetor[j];
17                  vetor[j]      = vetor[j + 1];
18                  vetor[j + 1] = aux;
19                  atrib+=3;
20              }
21              comp++;
22          }
23      }
24      printf("\n||%d comparacoes\n%d atribuicoes||\n",comp,atrib);
25  }
```

BubbleSort

```
26
27  void insertionsort(int vetor[],int n){
28          int i, j, atual;
29          int comp=0,atrib=0;
30      for (i = 1; i < n; i++) {
31
32          atual = vetor[i];
33          atrib++;
34          for (j = i - 1; (j >= 0) && (atual < vetor[j]); j--) {
35                  comp++;
36              vetor[j + 1] = vetor[j];
37              atrib++;
38          }
39          comp++;
40          vetor[j+1] = atual;
41          atrib++;
42      }
43      printf("\n||%d comparacoes\n%d atribuicoes||\n",comp,atrib);
44  }
```

InsertionSort

```
46  void selectionsort(int vetor[],int n){
47      int i, j, min, aux;
48      int comp=0,atrib=0;
49      for (i = 0; i < (n-1); i++) {
50       comp++;
51       min = i;
52       atrib++;
53       for (j = (i+1); j < n; j++) {
54           comp++;
55         if(vetor[j] < vetor[min]){
56          comp++;
57          min = j;
58          atrib++;
59         }
60           comp++;
61       }
62
63       if (vetor[i] != vetor[min]) {
64         comp++;
65         aux = vetor[i];
66         vetor[i] = vetor[min];
67         vetor[min] = aux;
68         atrib=+3;
69       }
70       comp++;
71    }
72    printf("\n||%d comparacoes\n%d atribuicoes||\n",comp,atrib);
73  }
```

SelectionSort

```
124  void shellsort(int vetor[], int n){
125      int i , j , valor;
126      int atrib=0,comp=0;
127      int gap = 1;
128      comp++;
129      while(gap < n) {
130              comp++;
131              atrib++;
132          gap = 3*gap+1;
133      }
134      comp++;
135      while ( gap > 1) {
136          comp++;
137          gap /= 3;
138          atrib++;
139          for(i = gap; i < n; i++) {
140              atrib+=2;
141              valor = vetor[i];
142              j = i;
143              comp++;
144              while (j >= gap && valor < vetor[j - gap]) {
145                  comp++;
146                  vetor[j] = vetor [j - gap];
147                  j = j - gap;
148                  atrib+=2;
149              }
150              comp++;
151              atrib++;
152              vetor [j] = valor;
153          }
154      }
155      comp++;
156      printf("\n||%d comparacoes\n%d atribuicoes||\n",comp,atrib);
157  }
```

ShellSort

```
75  void heapsort(int vetor[],int n){
76       int comp=0,atrib=0;
77      int i = n / 2, pai, filho, t, n2=n;
78      while(1) {
79            comp++;
80          if (i > 0) {
81              comp++;
82              i--;
83              t = vetor[i];
84                atrib++;
85          } else {
86              comp++;
87              n--;
88              atrib++;
89              comp++;
90              if (n == 0){
91                 printf("\n||%d comparacoes\n%d atribuicoes||\n",comp,atrib);
92                return;
93              }
94              atrib++;
95              atrib++;
96              t = vetor[n];
97              vetor[n] = vetor[0];
98          }
99          atrib+=2;
100         pai = i;
101         filho = i * 2 + 1;
102         comp++;
103         while (filho < n) {
104               comp++;
105             if ((filho + 1 < n)  &&  (vetor[filho + 1] > vetor[filho]))
106                 filho++;
107               comp++;
108             if (vetor[filho] > t) {
109               atrib+=3;
110                vetor[pai] = vetor[filho];
111                pai = filho;
112                filho = pai * 2 + 1;
113             } else {
114                break;
115             }
116             comp++;
117         }
118         atrib++;
119         vetor[pai] = t;
120     }
```

HeapSort

```
174  int particiona(int *vetor, int inicio, int fim ){
175      int esq, dir, pivo, aux;
176      q_atrib+=3;
177      esq = inicio;
178      dir = fim;
179      pivo = vetor[inicio];
180      while(esq < dir){
181              q_comp++;
182          while(vetor[esq] <= pivo){
183              q_comp++;
184              esq++;
185              q_atrib++;
186          }
187          q_comp++;
188          while(vetor[dir] > pivo){
189              q_comp++;
190              dir--;
191              q_atrib;
192          }
193          q_comp++;
194          if(esq < dir){
195              q_comp++;
196              q_atrib+=3;
197              aux = vetor[esq];
198              vetor[esq] = vetor[dir];
199              vetor[dir] = aux;
200          }q_comp++;
201      }
202      q_comp++;
203      q_atrib++;
204      vetor[inicio] = vetor[dir];
205      vetor[dir] = pivo;
206      return dir;
207  }
208  void quicksort(int *vetor, int inicio, int fim) {
209      int pivo;
210
211      if(fim > inicio){
212          q_comp++;
213          q_atrib++;
214          pivo = particiona(vetor, inicio, fim);
215          quicksort(vetor, inicio, pivo-1);
216          quicksort(vetor, pivo+1, fim);
217      }
218      q_comp++;
219  }
```

QuickSort

3 - Testes

Seguem o relatório dos testes efetuados, para o caso de estouro de memoria, preenchemos a tabela com OF(OverFlow).

BubbleSort:

Ordenado					
Tamanho	**10**	**100**	**1000**	**10000**	**100000**
Comparações	45	4950	499500	OF	OF
Atribuições	0	0	0	OF	OF
Tempo (m.s)	1.0	31.0	157.0	541.0	17009.0

Quase Ordenado					
Tamanho	**10**	**100**	**1000**	**10000**	**100000**
Comparações	1145	115049	11509445	OF	OF
Atribuições	3	297	29835	OF	OF
Tempo (m.s)	1.0	1.0	47.0	641.0	17127.0

Desordenado					
Tamanho	**10**	**100**	**1000**	**10000**	**100000**
Comparações	54	115049	11500499	OF	OF
Atribuições	27	297	2997	OF	OF
Tempo (m.s)	1.0	1.0	47.0	531.0	16856.0

Aleatório					
Tamanho	**10**	**100**	**1000**	**10000**	**100000**
Comparações	1160	117161	11752670	OF	OF
Atribuições	45	6633	759510	OF	OF
Tempo (m.s)	1.0	1.0	47.0	783.0	37210.0

Selection Sort

Ordenado					
Tamanho	**10**	**100**	**1000**	**10000**	**100000**
Comparações	108	10098	1000998	100009998	OF
Atribuições	9	99	999	9999	OF
Tempo (m.s)	4.0	8.0	64.0	1969.0	445540.0

Quase Ordenado					
Tamanho	**10**	**100**	**1000**	**10000**	**100000**
Comparações	108	10296	1007463	100522398	OF
Atribuições	9	3	7	52	OF
Tempo (m.s)	4.0	8.0	72.0	886.0	28582.0

Desordenado					
Tamanho	**10**	**100**	**1000**	**10000**	**100000**
Comparações	126	100296	1000998	100029996	OF
Atribuições	3	3	999	3	OF
Tempo (m.s)	4.0	6.0	59.0	976.0	26065.0

Aleatórios					
Tamanho	10	100	1000	10000	100000
Comparações	125	10467	1007002	100009998	OF
Atribuições	4	4	5	4	OF
Tempo (m.s)	4.0	8.0	60.0	760.0	27456

InsertionSort

Ordenado					
Tamanho	**10**	**100**	**1000**	**10000**	**100000**
Comparações	9	99	999	9999	OF
Atribuições	18	198	1998	19998	OF
Tempo (m.s)	4.0	8.0	64.0	752.0	6359

Desordenado					
Tamanho	**10**	**100**	**1000**	**10000**	**100000**
Comparações	18	198	1998	19998	OF
Atribuições	27	297	2997	29997	OF
Tempo (m.s)	0.0	8.0	55.0	1760.0	3851.0

Quase Ordenado					
Tamanho	**10**	**100**	**1000**	**10000**	**100000**
Comparações	9	198	10944	1004949	OF
Atribuições	18	297	11943	1014948	OF
Tempo (m.s)	1.0	8.0	56.0	674.0	4783.0

Aleatório					
Tamanho	**10**	**100**	**1000**	**10000**	**100000**
Comparações	33	2443	254200	24817681	OF
Atribuições	42	2542	255199	24827680	OF
Tempo (m.s)	2.0	8.0	58.0	679.0	17385.0

ShellSort

Ordenado					
Tamanho	10	100	1000	10000	100000
Comparações	37	695	10929	150507	1934317
Atribuições	49	1034	16383	225747	2901460
Tempo (m.s)	0.0ms	0.0ms	31.0ms	529.0ms	5701.0ms

Desordenado					
Tamanho	10	100	1000	10000	100000
Comparações	46	794	11928	160506	OF
Atribuições	67	1232	18381	245745	OF
Tempo (m.s)	1.0ms	7.0ms	52.0ms	598.0ms	3792.0ms

Quase ordenado					
Tamanho	10	100	1000	10000	100000
Comparações	37	794	13050	183811	OF
Atribuições	49	1232	20625	292355	OF
Tempo (m.s)	1.0ms	7.0ms	51.0ms	552.0ms	3696.0ms

Aleatório					
Tamanho	10	100	1000	10000	100000
Comparações	52	1151	19242	319651	OF
Atribuições	79	1946	33009	564034	OF
Tempo (m.s)	1.0ms	7.0ms	51.0ms	473.0ms	3725.0ms

HeapSort

Ordenado					
Tamanho	10	100	1000	10000	100000
Comparações	121	2184	31834	421532	OF
Atribuições	138	2418	34122	445866	OF
Tempo (m.s)	1.0ms	7.0ms	58.0ms	631.0ms	3852.0ms

Desordenado					
Tamanho	10	100	1000	10000	100000
Comparações	114	2149	31747	421187	OF
Atribuições	129	2379	34065	445557	OF
Tempo (m.s)	0.0ms	9.0ms	59.0ms	777.0ms	4035.0ms

Quase ordenado					
Tamanho	10	100	1000	10000	100000
Comparações	121	2179	31893	421992	OF
Atribuições	138	2415	34215	445182	OF
Tempo (m.s)	1.0ms	6.0ms	53.0ms	637.0ms	4225.0ms

Aleatório					
Tamanho	10	100	1000	10000	100000
Comparações	108	2076	30406	404818	5044036
Atribuições	117	2262	32190	422898	5224284
Tempo (m.s)	1.0ms	9.0ms	60.0ms	529.0ms	20.0ms

QuickSort

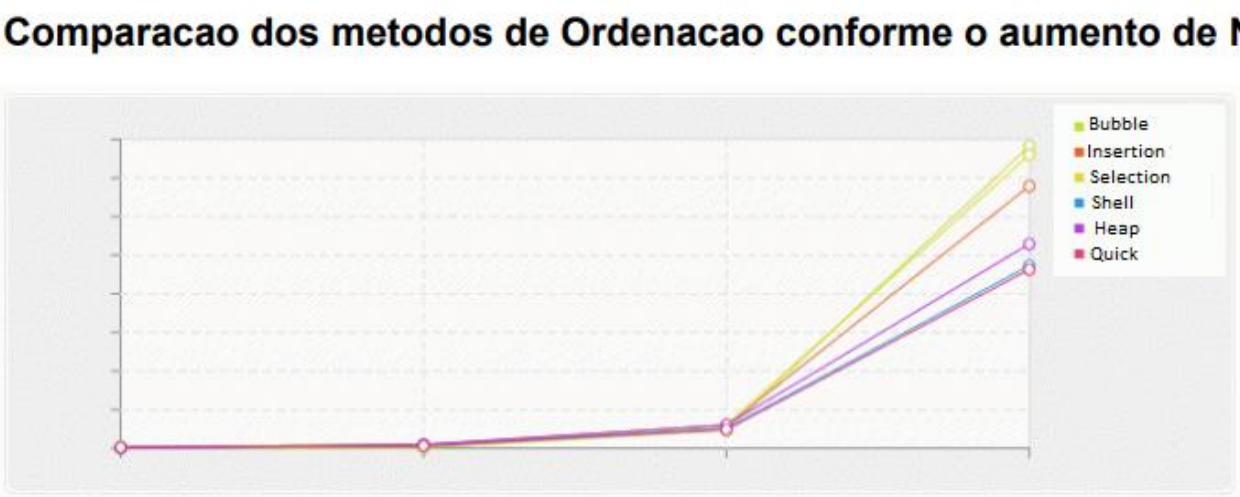
Desordenado					
Tamanho	10	100	1000	10000	100000
Comparações	92	5842	508492	50084992	OF
Atribuições	48	594	5994	59994	OF
Tempo (m.s)	0.0ms	6.0ms	48.0ms	634.0ms	24.261s

Desordenado					
Tamanho	10	100	1000	10000	100000
Comparações	127	5842	508492	50084992	OF
Atribuições	54	594	5994	59994	OF
Tempo (m.s)	0.0ms	7.0ms	48.0ms	717.0ms	OF

Quase ordenado					
Tamanho	10	100	1000	10000	100000
Comparações	127	3196	62632	1127999	56214410
Atribuições	54	497	13129	586199	50566910
Tempo (m.s)	1.0ms	6.0ms	46.0ms	562.0ms	3780.0ms

Aleatório					
Tamanho	10	100	1000	10000	100000
Comparações	118	1878	26939	357722	4610797
Atribuições	83	1138	16463	196152	2652757
Tempo (m.s)	2.0ms	6.0ms	48.0ms	463.0ms	3515.0ms

O seguinte Gráfico foi gerado com base nos dados de teste:



	10	100	1000	10000
Bubble	1	1	47	783
Insertion	2	8	58	679
Selection	4	8	60	760
Shell	1	7	51	473
Heap	1	9	60	529
Quick	2	6	48	463

Não incluímos o valor 100000 pois causou estouro de memória na maioria dos métodos, mas a tendência, com o aumento do numero de dados é que os métodos que tem complexidade $O(n^2)$ subam seu tempo de execução muito rápido, enquanto os demais métodos teriam um aumento mais moderado. O ultimo valor do bubble sort não é real, foi escolhido arbitrariamente para que fosse possível a confecção do gráfico.

Conclusão

Através deste trabalho prático pudemos aplicar o que aprendemos em teoria durante o semestre na disciplina de Algorítmos e Estrutura de Dados I, através da elaboração da aplicação relembramos muitas coisas vistas em sala e não sabíamos bem o que significava na prática, não foi uma tarefa tão fácil quanto pensamos que seria de início, mas hoje podemos dizer que implementamos com sucesso o que foi pedido, pudemos concluir que realmente os algoritmos de complexidade logarítmica tem uma diferença real no tempo de execução.

Referencias Bibliográficas

Notas de Aula;

Canal do Professor André Backes
(https://www.youtube.com/user/progdescomplicada)

https://www.cprogressivo.net/