EP 2 - IAA

Pablo Caballero Maciel Nº USP: 14778630

Outubro 2024

Contents

1 Disponibilização dos Materiais

Caso goste do conteúdo e queira olhar o código usado e os materiais gerados, estou disponibilizando-os neste link.

Deixo também meu GitHub.

2 Introdução

O presente documento visa comparar o desempenho de alguns algoritmos de ordenação, medindo o tempo para ordenar vetores de tamanhos variados em ordem crescente, o número de comparações e o número de movimentações. Fique a vontade para navegar pela lista de conteúdos mais acima e ver o teste de seu interesse, basta clicar no tópico desejado para ser levado até ele.

Além disso, cada teste também é subdividido pelo tamanho de um vetor não inicializado presente na estrutura, que assume valores de 1 e 1000.

Para melhor entender a explicação, abaixo está o código:

```
typedef struct {
    int chave;
    int campoDaEstrutura[1];
} Registro1;

typedef struct {
    int chave;
    int campoDaEstrutura[1000];
} Registro1000;
```

Então, cada vetor a ser organizado será uma estrutura que possui um campo para as chaves e outro para um vetor de tamanho 1 ou 1000 não inicializado.

3 Arquitetura da Máquina

O código foi executado em um computador com as seguintes especificações:

Table 1: Especificações do Sistema

Componente	**Especificação**
Processador	AMD Ryzen 5 7530U com Radeon Graphics (2.00 GHz)
Memória RAM	8 GB
Sistema Operacional	Linux Ubuntu (64-bits)

4 Tabelas de Resultados

Nesta seção, você verá 3 tabelas que armazenam os resultados dos testes para chaves em ordem crescente, aleatória e decrescente:

Algoritmo	Tamanho Estrutura	Número Chaves	Tempo (s)	Comparações	Movimentações
BubbleSort	1	100	0.000002	99	0
BubbleSort	1	1000	0.000004	999	0
BubbleSort	1	10000	0.000022	9999	0
BubbleSort	1000	100	0.000002	99	0
BubbleSort	1000	1000	0.000029	999	0
BubbleSort	1000	10000	0.000362	9999	0
HeapSort	1	100	0.000013	1081	1923
HeapSort	1	1000	0.000163	17583	29127
HeapSort	1	10000	0.001912	244460	395871
HeapSort	1000	100	0.000279	1081	1923
HeapSort	1000	1000	0.003589	17583	29127
HeapSort	1000	10000	0.076931	244460	395871
InsertionSort	1	100	0.000003	99	99
InsertionSort	1	1000	0.000007	999	999
InsertionSort	1	10000	0.000038	9999	9999
InsertionSort	1000	100	0.000054	99	99
InsertionSort	1000	1000	0.001097	999	999
InsertionSort	1000	10000	0.013546	9999	9999
MergeSort	1	100	0.000019	672	1344
MergeSort	1	1000	0.000117	9976	19952
MergeSort	1	10000	0.001349	133616	267232
MergeSort	1000	100	0.001007	672	1344
MergeSort	1000	1000	0.013441	9976	19952
MergeSort	1000	10000	0.171872	133616	267232
QuickSort	1	100	0.000013	593	1398
QuickSort	1	1000	0.000074	10182	19653
QuickSort	1	10000	0.000854	156759	303765
QuickSort	1000	100	0.000178	633	1476
QuickSort	1000	1000	0.004300	10945	18939
QuickSort	1000	10000	0.066427	162912	294969
SelectionSort	1	100	0.000014	4950	0
SelectionSort	1	1000	0.001191	499500	0
SelectionSort	1	10000	0.099935	49995000	0
SelectionSort	1000	100	0.000016	4950	0
SelectionSort	1000	1000	0.001124	499500	0
SelectionSort	1000	10000	0.180619	49995000	0
ShellSort	1	100	0.000005	684	342
ShellSort	1	1000	0.000032	10914	5457
ShellSort	1	10000	0.000378	150486	75243
ShellSort	1000	100	0.000125	684	342
ShellSort	1000	1000	0.001856	10914	5457
ShellSort	1000	10000	0.087129	150486	75243

Table 2: Chaves em Ordem Crescente

Algoritmo	TamanhoEstrutura	NumeroChaves	Tempo (s)	Comparacoes	Movimentacoes
BubbleSort	1	100	0.000022	4845	6525
BubbleSort	1	1000	0.001833	497547	744900
BubbleSort	1	10000	0.178199	49984269	75012816
BubbleSort	1000	100	0.000865	4845	6525
BubbleSort	1000	1000	0.081890	497547	744900
BubbleSort	1000	10000	13.025811	49984269	75012816
HeapSort	1	100	0.000011	1041	1788
HeapSort	1	1000	0.000151	16806	27219
HeapSort	1	10000	0.002040	235447	372555
HeapSort	1000	100	0.000286	1041	1788
HeapSort	1000	1000	0.004786	16806	27219
HeapSort	1000	10000	0.051806	235447	372555
InsertionSort	1	100	0.000010	2274	2274
InsertionSort	1	1000	0.000720	249299	249299
InsertionSort	1	10000	0.076228	25014271	25014271
InsertionSort	1000	100	0.000434	2274	2274
InsertionSort	1000	1000	0.040242	249299	249299
InsertionSort	1000	10000	8.556292	25014271	25014271
MergeSort	1	100	0.000014	672	1344
MergeSort	1	1000	0.000163	9976	19952
MergeSort	1	10000	0.001958	133616	267232
MergeSort	1000	100	0.000924	672	1344
MergeSort	1000	1000	0.010885	9976	19952
MergeSort	1000	10000	0.113882	133616	267232
QuickSort	1	100	0.000009	649	1530
QuickSort	1	1000	0.000106	11311	22986
QuickSort	1	10000	0.001243	156649	259809
QuickSort	1000	100	0.000256	618	1434
QuickSort	1000	1000	0.003495	10011	19272
QuickSort	1000	10000	0.048913	158254	277086
SelectionSort	1	100	0.000016	4950	291
SelectionSort	1	1000	0.001051	499500	2970
SelectionSort	1	10000	0.100908	49995000	29964
SelectionSort	1000	100	0.000113	4950	291
SelectionSort	1000	1000	0.003300	499500	2970
SelectionSort	1000	10000	0.181168	49995000	29964
ShellSort	1	100	0.000008	1077	765
ShellSort	1	1000	0.000125	19516	14495
ShellSort	1	10000	0.001904	303572	232615
ShellSort	1000	100	0.000272	1077	765
ShellSort	1000	1000	0.002924	19516	14495
ShellSort	1000	10000	0.078378	303572	232615

Table 3: Chaves em Ordem Aleatória

Algoritmo	TamanhoEstrutura	NumeroChaves	Tempo (s)	Comparacoes	Movimentacoes
BubbleSort	1	100	0.000032	4950	14850
BubbleSort	1	1000	0.002390	499500	1498500
BubbleSort	1	10000	0.239197	49995000	149985000
BubbleSort	1000	100	0.001760	4950	14850
BubbleSort	1000	1000	0.148711	499500	1498500
BubbleSort	1000	10000	19.169552	49995000	149985000
HeapSort	1	100	0.000013	944	1551
HeapSort	1	1000	0.000128	15965	24951
HeapSort	1	10000	0.001698	226682	350091
HeapSort	1000	100	0.000279	944	1551
HeapSort	1000	1000	0.005537	15965	24951
HeapSort	1000	10000	0.058302	226682	350091
InsertionSort	1	100	0.000025	5049	5049
InsertionSort	1	1000	0.001850	500499	500499
InsertionSort	1	10000	0.153152	50004999	50004999
InsertionSort	1000	100	0.001243	5049	5049
InsertionSort	1000	1000	0.084106	500499	500499
InsertionSort	1000	10000	21.026633	50004999	50004999
MergeSort	1	100	0.000019	672	1344
MergeSort	1	1000	0.000120	9976	19952
MergeSort	1	10000	0.001358	133616	267232
MergeSort	1000	100	0.000909	672	1344
MergeSort	1000	1000	0.019726	9976	19952
MergeSort	1000	10000	0.149944	133616	267232
QuickSort	1	100	0.000015	622	1224
QuickSort	1	1000	0.000071	10008	17436
QuickSort	1	10000	0.000920	163578	285930
QuickSort	1000	100	0.000250	718	1557
QuickSort	1000	1000	0.009687	11335	21042
QuickSort	1000	10000	0.069921	156571	276591
SelectionSort	1	100	0.000018	4950	150
SelectionSort	1	1000	0.001155	499500	1500
SelectionSort	1	10000	0.108201	49995000	15000
SelectionSort	1000	100	0.000101	4950	150
SelectionSort	1000	1000	0.002164	499500	1500
SelectionSort	1000	10000	0.131244	49995000	15000
ShellSort	1	100	0.000007	842	572
ShellSort	1	1000	0.000048	14007	9377
ShellSort	1	10000	0.000611	195433	128947
ShellSort	1000	100	0.000232	842	572
ShellSort	1000	1000	0.003951	14007	9377
ShellSort	1000	10000	0.112158	195433	128947

Table 4: Chaves em Ordem Decrescente

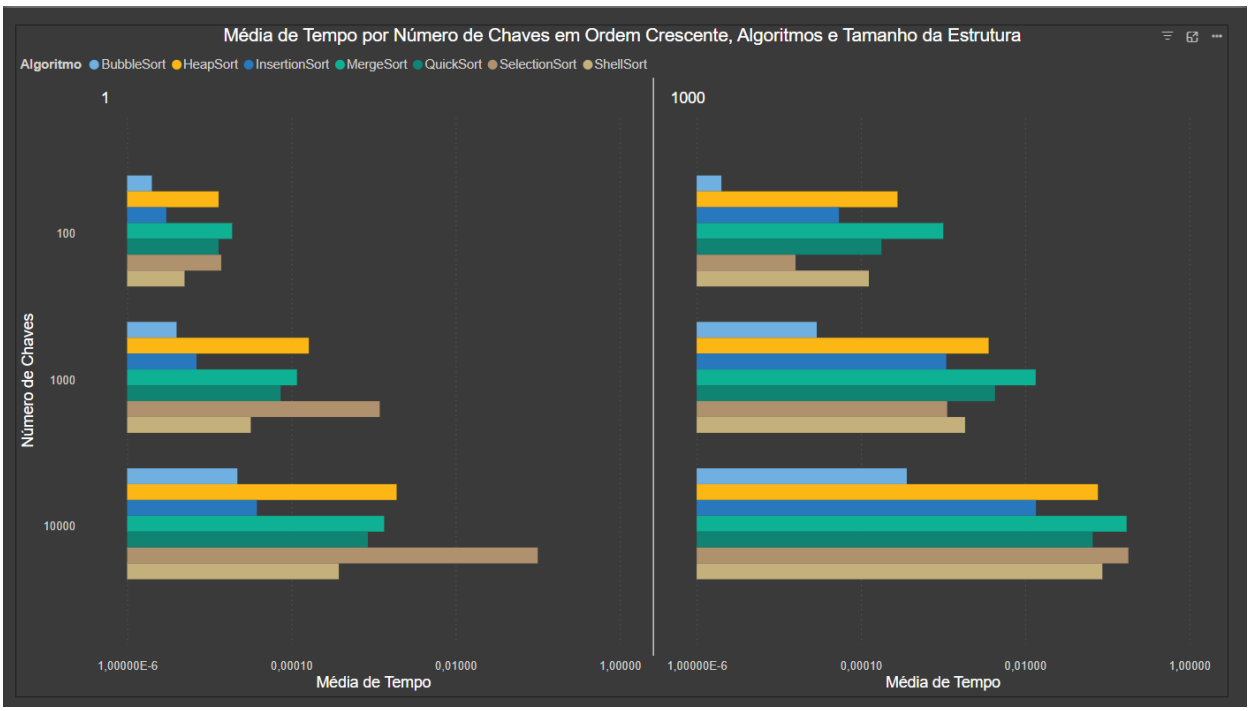
5 Análise dos Gráficos

Vamos analisar os resultados das tabelas por meio de gráficos para facilitar a visualização e entendimento das diferenças

OBS: Foi aplicada uma transformação para escala logarítmica nos dados em função de deixar mais nítidas as diferenças entre os algoritmos.

5.1 Tempo X Número de Chaves X Tamanho da Estrutura

5.1.1 Chaves em Ordem Crescente



Olhando o gráfico, a primeira informação que podemos tirar é de que o BubbleSort é disparadamente o mais rápido nesta situação, independente do tamanho da estrutura e do vetor. Isso acontece pois quando o vetor está ordenado, o algoritmo percorre apenas uma vez o array, verificando se há algum elemento fora do lugar e, quando constata que não há, finaliza a execução.

```
void BubbleSort(Registro *V, int N, int *num_comp, int *
   num_mov){
    int i, trocou, fim = N;

    Registro aux;

    do{
        trocou = 0;
        for(i = 0; i < fim-1; i++){

            (*num_comp)++;
            if (V[i].chave > V[i+1].chave){
                aux = V[i];
                V[i] = V[i+1];
                V[i+1] = aux;
                trocou = i;
                (*num_mov) +=3;
            }
        }
        fim--;
    }while(trocou != 0);
}
```

Essa detecção de que há ou não alguém fora do lugar é salva na variável "trocou" do código acima, sendo que trocou = 0 significa que não houve nenhuma troca, ou seja, o vetor está ordenado.

O InsertionSort também apresenta um desempenho satisfatório nessa ocasião, obtendo uma média de 0,00004 segundo para ordenar um vetor de 10 mil elementos.

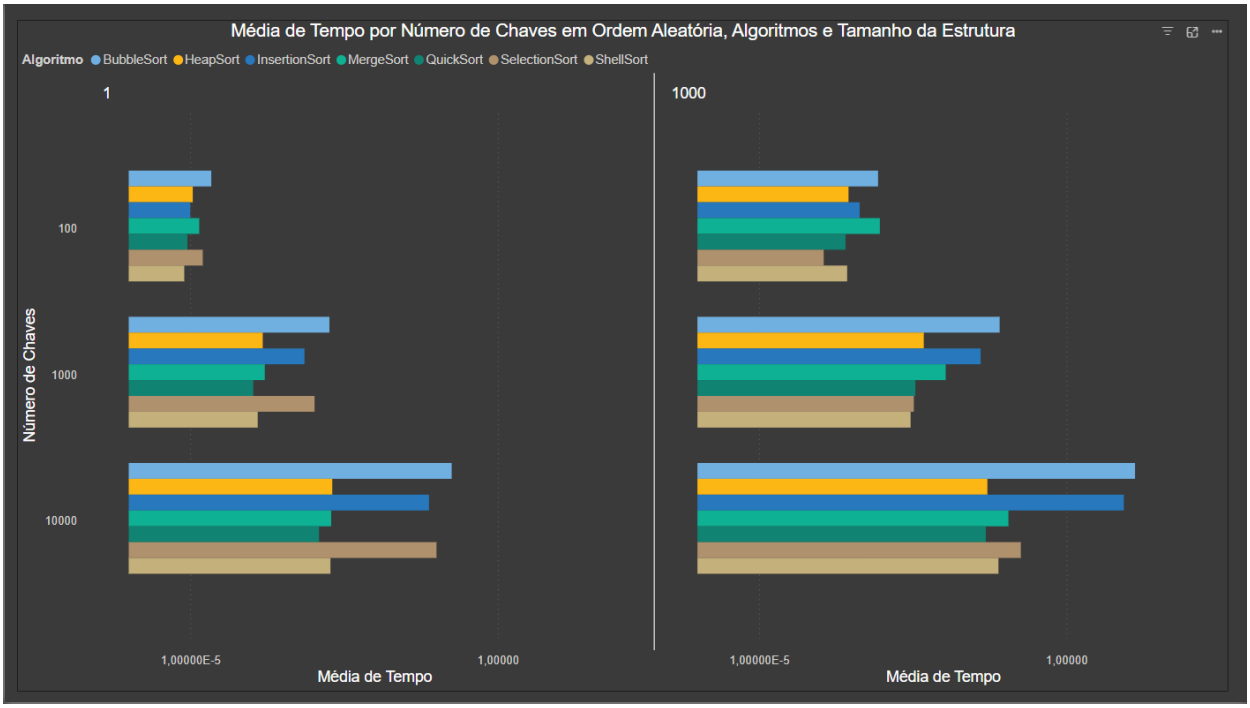
Por outro lado, o SelectionSort é quem obteve o pior resultado no caso de vetores já ordenados, com uma média de 0,1 segundo para ordenar o array de 10 mil elementos. Isso se deve ao fato do número excessivo de comparações para encontrar o menor elemento e colocá-lo no lugar certo, como é possível ver no código abaixo:

```
void SelectionSort(Registro *V, int N, int *num_comp,
   int *num_mov){
    int i, j, min;

    Registro x;

    for (i = 0; i < N-1; i++){
```

```
7           min = i;
8
9           for (j = i+1; j < N; j++){
10
11              (*num_comp)++;
12              if (V[j].chave < V[min].chave){
13                  min = j;
14              }
15          }
16
17          if (min != i){
18              x = V[i];
19              V[i] = V[min];
20              V[min] = x;
21              (*num_mov) += 3;
22          }
23
24      }
25  }
```

Os outros algoritmos apresentaram desempenho razoável, mas que se mostrarão melhores em outros cenários.

5.1.2 Chaves em Ordem Aleatória



Quando passamos a falar de vetores com chaves em ordem aleatória, o cenário inverte. Se antes o BubbleSort liderava em questão de desempenho, agora ele se tornou a pior opção para ordenar o array, levando 0,18 segundo para lidar com 10 mil elementos e estrutura de tamanho 1. Além disso, o algoritmo levou incríveis 13,03 segundos para ordenar os mesmos 10 mil elementos mas com a estrutura de tamanho 1000. Isso se deve ao grande número de cópias que ele faz da estrutura para auxiliar na ordenação (linhas 12 - 14 do código).

Já o InsertionSort, que era uma boa opção no cenário passado, possui um desempenho razoável para vetores pequenos, de 100 a 1000 elementos, mas que piora drásticamente para arrays maiores, de 10 mil+ elementos. Isso é explicado por ele ter complexidade de $O(n^2)$, levando 8,56 segundos para ordenar os 10 mil elementos e estrutura de tamanho 1000.

Partindo para os "vencedores" do teste. vemos que o Quick, Heap e MergeSort apresentaram os melhores desempenhos, respectivamente. Para ordenar o array de 10 mil elementos e tamanho da estrutura 1000, o QuickSort levou 0,048 segundo, O HeapSort 0,051 segundo e o MergeSort 0,11 segundo. Isso parece razóavel, já que a complexidade

deles é de $O(n \log n)$ e são insensíveis à ordem inicial do vetor. Abaixo está o código dos três:

```
void Merge(Registro *V, int esq, int meio, int dir, int
   *num_comp, int *num_mov){
    int n1 = (meio - esq) + 1;
    int n2 = dir - meio;

    Registro *L = (Registro *)malloc(n1 * sizeof(
       Registro));
    Registro *R = (Registro *)malloc(n2 * sizeof(
       Registro));

    for (int i = 0; i < n1; i++) {
        L[i] = V[esq + i];
        (*num_mov)++;
    }

    for (int j = 0; j < n2; j++) {
        R[j] = V[meio + 1 + j];
        (*num_mov)++;
    }

    int k1 = 0;
    int k2 = 0;

    for (int k = esq; k <= dir; k++){

        (*num_comp)++;
        if (k1 < n1 && (k2 >= n2 || L[k1].chave <= R[k2
           ].chave)){
            V[k] = L[k1];
            k1++;
            (*num_mov)++;
        }
        else if (k2 < n2){
            V[k] = R[k2];
            k2++;
            (*num_mov)++;
        }
    }
```

```
35
36      free(L);
37      free(R);
38  }
39
40  void MergeSort(Registro *V, int inicio, int fim, int *
        num_comp, int *num_mov){
41      int meio;
42
43      if (inicio < fim){
44          meio = (inicio + fim) / 2;
45          MergeSort(V, inicio, meio, num_comp, num_mov);
46          MergeSort(V, meio+1, fim, num_comp, num_mov);
47          Merge(V, inicio, meio, fim, num_comp, num_mov);
48      }
49  }
```

```
1       void ConstroiHeap(Registro *V, int N, int i, int *
            num_comp, int *num_mov){
2       int maior = i;
3       int filho_esq = (2 * i) + 1;
4       int filho_dir = (2 * i) + 2;
5
6       if (filho_esq < N){
7
8           (*num_comp)++;
9           if (V[filho_esq].chave > V[maior].chave){
10              maior = filho_esq;
11          }
12      }
13
14      if (filho_dir < N){
15
16          (*num_comp)++;
17          if (V[filho_dir].chave > V[maior].chave){
18              maior = filho_dir;
19          }
20      }
21
22      if (maior != i){
```

```
        Registro aux = V[i];
        V[i] = V[maior];
        V[maior] = aux;

        (*num_mov) += 3;

        ConstroiHeap(V, N, maior, num_comp, num_mov);
    }
}

void HeapSort(Registro *V, int N, int *num_comp, int *
   num_mov){
    for (int i = (N / 2) - 1; i >= 0; i--){
        ConstroiHeap(V, N, i, num_comp, num_mov);
    }

    for (int i = N-1; i >= 0 ; i--){
        Registro aux = V[0];
        V[0] = V[i];
        V[i] = aux;

        (*num_mov) += 3;

        ConstroiHeap(V, i, 0, num_comp, num_mov);
    }
}
```

```
    int particao(Registro *V, int p, int r, int *
       num_comp, int *num_mov) {
    Registro x = V[r];
    int i = p - 1;

    for (int j = p; j <= r - 1; j++) {

        (*num_comp)++;
        if (V[j].chave <= x.chave) {
            i++;
            Registro aux = V[i];
            V[i] = V[j];
            V[j] = aux;
```

```
13              (*num_mov) += 3;
14          }
15      }
16
17      Registro aux = V[i + 1];
18      V[i + 1] = V[r];
19      V[r] = aux;
20      (*num_mov) += 3;
21
22      return i + 1;
23  }
24
25  int particaoAleatoria(Registro *V, int p, int r, int *
        num_comp, int *num_mov){
26      int deslocamento, i;
27
28      Registro aux;
29
30      deslocamento = rand() % (r-p+1);
31      i = p + deslocamento;
32
33      aux = V[r];
34      V[r] = V[i];
35      V[i] = aux;
36      (*num_mov) += 3;
37
38      return particao(V, p, r, num_comp, num_mov);
39  }
40
41  void quickSortAleatorio(Registro *V, int p, int r, int *
        num_comp, int *num_mov){
42      if (p < r){
43          int q = particaoAleatoria(V, p, r, num_comp,
                num_mov);
44          quickSortAleatorio(V, p, q - 1, num_comp,
                num_mov);
45          quickSortAleatorio(V, q + 1, r, num_comp,
                num_mov);
46      }
47  }
```

Vemos que os códigos e a lógica são consideravelmente mais elaboradas, refletindo na otimização do algoritmo.

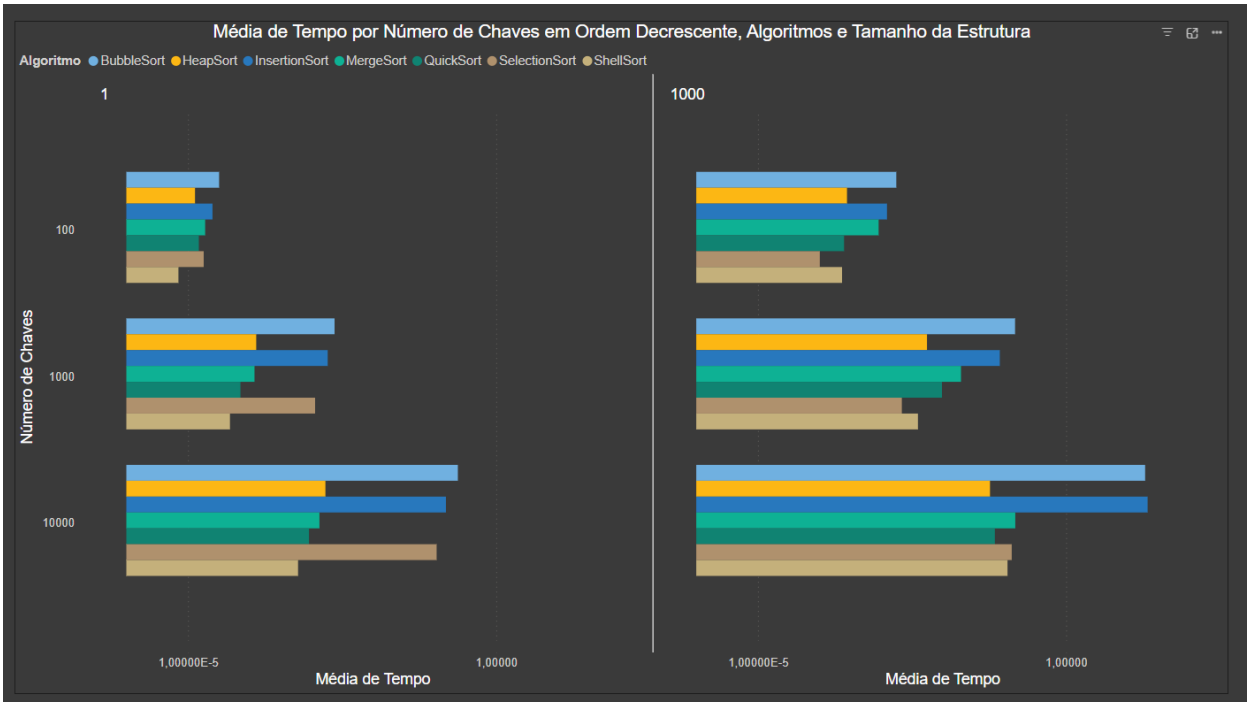
Antes de passar para o próximo teste, vale a pena comentar sobre o ShellSort. Este algoritmo possui uma complexidade de $O(n \log^2 n)$ ou de $O(n^{1,5})$, a depender da sequência de Gaps usada. Essas sequências fazem com que o algoritmo compare elementos mais distantes e diminui, assim, o número de comparações totais. Ele é uma boa opção para competir com os algoritmos $O(n \log n)$, se saindo melhor que alguns deles em certos casos.

Nesse exemplo do vetor de 10 mil elementos e tamanho de estrutura 1000, o ShellSort levou 0,078 segundo enquanto o MergeSort levou 0,11 segundo, apresentando um desempenho de tempo maior nesse caso. Abaixo está seu código:

```
1   void ShellSort(Registro *V, int N, int *num_comp,
        int *num_mov) {
2   int i, j, h;
3
4   Registro chave;
5
6   for (h = 1; h < N; h = 3*h+1);
7
8   h = (h-1)/3;
9   while (h > 0) {
10
11      for(j = h; j < N; j++) {
12          chave = V[j];
13          i = j;
14
15          (*num_comp)++;
16          while (i >= h && V[i - h].chave > chave.
                chave) {
17              (*num_comp)++;
18              V[i] = V[i - h];
19              i -= h;
20              (*num_mov)++;
21          }
22
23          if (i >= h) (*num_comp)++;
24
25          V[i] = chave;
26          (*num_mov)++;
27      }
```

```
28              h = (h-1)/3;
29          }
30      }
```

5.1.3 Chaves em Ordem Decrescente



A principal diferença entre o que vimos com os elementos em ordem aleatória e o que veremos agora é simplesmente que os algoritmos $O(n \log n)$ disparam ainda mais na velocidade de ordenação.

Bubble e InsertionSort continuam com os mesmos problemas citados anteriormente e sua complexidade de $O(n^2)$. Eles chegaram a levar 19,17 segundos e 21,03 segundos para ordenar o vetor de 10 mil elementos em ordem decrescente e tamanho da estrutura 1000, respectivamente.

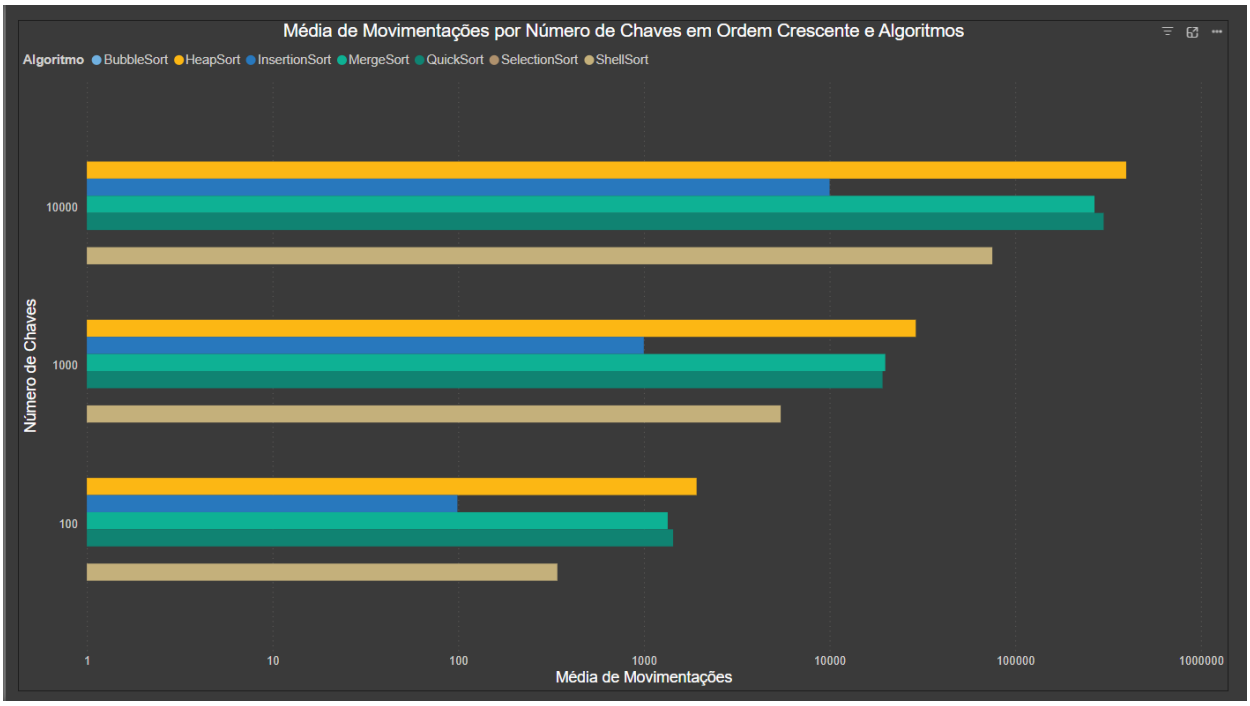
Enquanto isso, o Quick, Heap, Shell e MergeSort levaram 0,07 segundo, 0,06 segundo, 0,11 segundo e 0,15 segundo, respectivamente. Isso mostra como eles se mostram mais eficientes em cenários em que os vetores estão ordenado aleatoriamente ou decrescentemente, como

é comum vermos do dia a dia.

Por fim, um detalhe que pode estar gerando algumas dúvidas é o fato de não abordarmos o SelectionSort nesses dois últimos cenários. Se olharmos mais atentamente aos gráficos da direita (tamanho da estrutura 1000), vemos que ele acaba apresentando um tempo de ordenação não tão longe dos algoritmos $O(n \log n)$, mesmo sendo $O(n^2)$. Veremos a explicação para este detalhe na seção seguinte.

5.2 Número de Movimentações X Número de Chaves

5.2.1 Chaves em Ordem Crescente



Quanto ao número de movimentações, a primeira informação que ressalta aos olhos é de que tanto o BubbleSort quanto o SelectionSort estão ausentes do gráfico. Isso se deve ao fato de seus códigos não realizarem nenhuma movimentação quando os valores já estão ordenados, convergindo para o fato de serem as melhores escolhas no caso de vetores já ordenados.

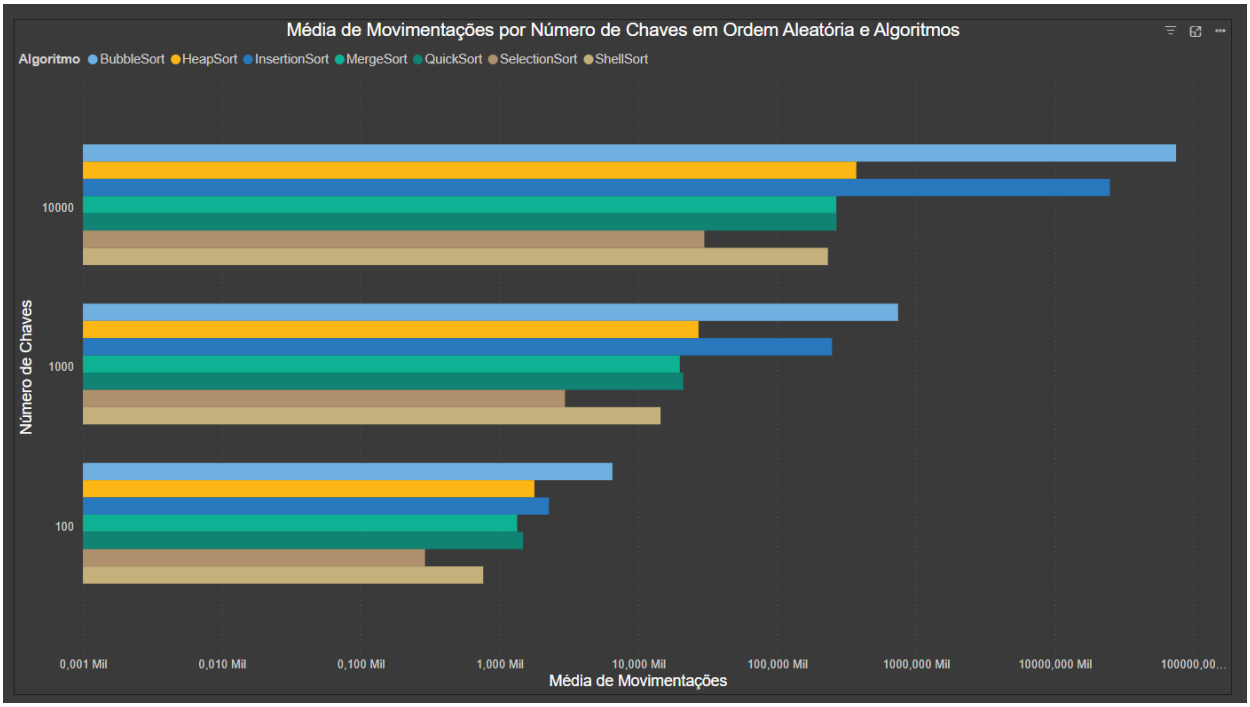
Agora vamos falar sobre o HeapSort, detentor do maior número de movimentações nos teste (395871 no caso do vetor de 10 mil chaves). Devido à sua lógica de construção ser

baseada na propriedade do heap, na qual os elementos são posicionados de forma semelhante a uma árvore binária em que aqueles a esquerda são menores que os da direita, seu número de movimentações cresce muito para atender tal propriedade. Além disso, o heap é contruído diversas vezes durante a execução do algoritmo, reforçando o motivo dele liderar o teste.

O QuickSort, MergeSort e o ShellSort também não ficam para trás, apresentando altos números de movimentações devido à manejos no vetor para facilitar a ordenação. O Quick precisa escolher seu pivô e movimentar os elementos para deixar a lógica do algoritmo correta. O Merge usa a estratégia de divisão e conquista, dividindo e mesclando o array diversas vezes. E o Shell movimenta os elementos obedecendo a regra de gaps adotada.

Por fim, o InsertionSort se saiu melhor neste caso, sendo o terceiro melhor dos algoritmos em número de movimentações em um vetor ordenado.

5.2.2 Chaves em Ordem Aleatória



Com o BubbleSort de volta no jogo (agora que o array não está ordenado), percebemos que ele lidera os algoritmos com mais movimentações junto com o InsertionSort, realizando

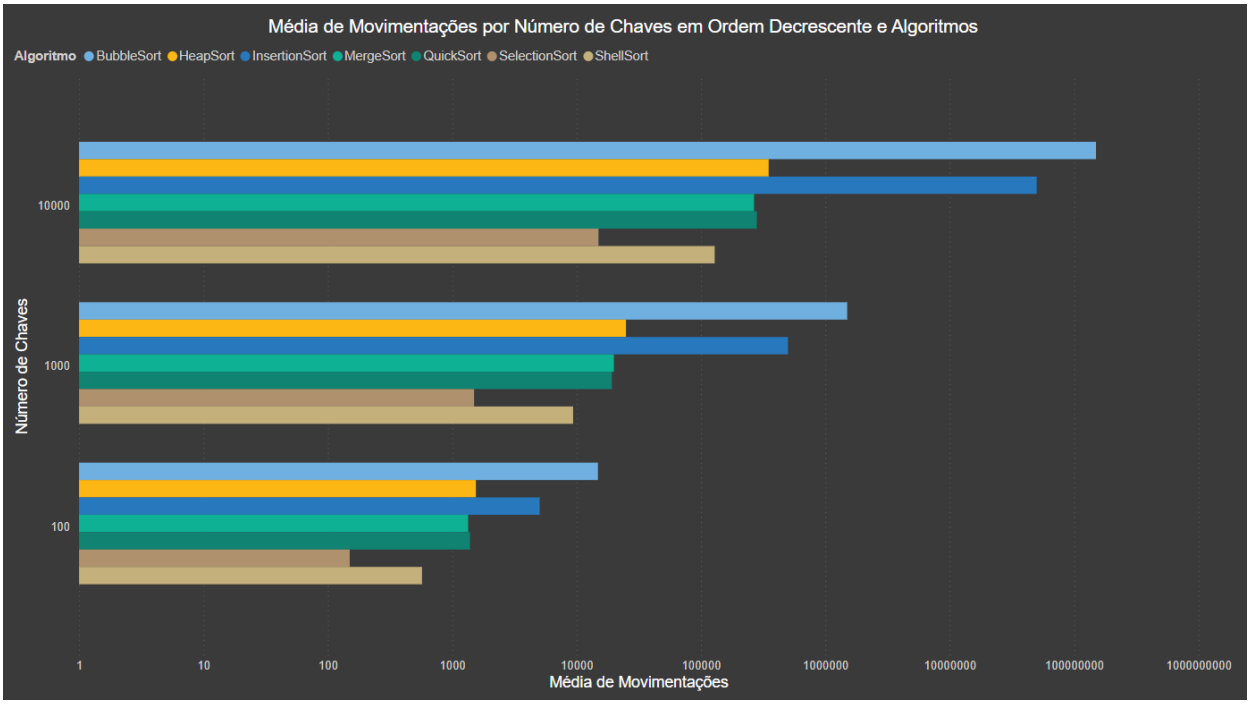
75012816 e 25014271 movimentações em um vetor com 10 mil chaves, respectivamente.

O Quick, Heap, Merge e ShellSort reduziram significativamente o seu número de movimentações, mas continuam apresentando o mesmo desempenho relativamente.

A estrela da vez é o SelectionSort, que se torna o algoritmo com menos movimentações dentre os testes feitos (29964 no vetor de 10 mil chaves). Seu baixo número de movimentações se deve ao fato de ele só trocar os elementos de lugar ao final de uma iteração, quando encontra o menor elemento no array. Entretanto, ele compensa esse fato com o alto número de comparações, como veremos na seção seguinte.

Então, agora que descobrimos que o SelectionSort possui um número baixo de movimentações, nós podemos explicar o por quê de ele ficar tão próximo de algoritmos $O(n \log n)$ em questão de tempo mesmo sendo $O(n^2)$, no caso em que o tamanho da estrutura é 1000. O que acontece é que como os outros algoritmos fazem um número de movimentações maior, ou seja, copiam mais vezes o vetor, eles gastam um tempo maior que é economizado pelo Selection. Isso faz com que ele consiga diminuir a diferença de tempo entre eles, ainda que seja mais lento.

5.2.3 Chaves em Ordem Decrescente

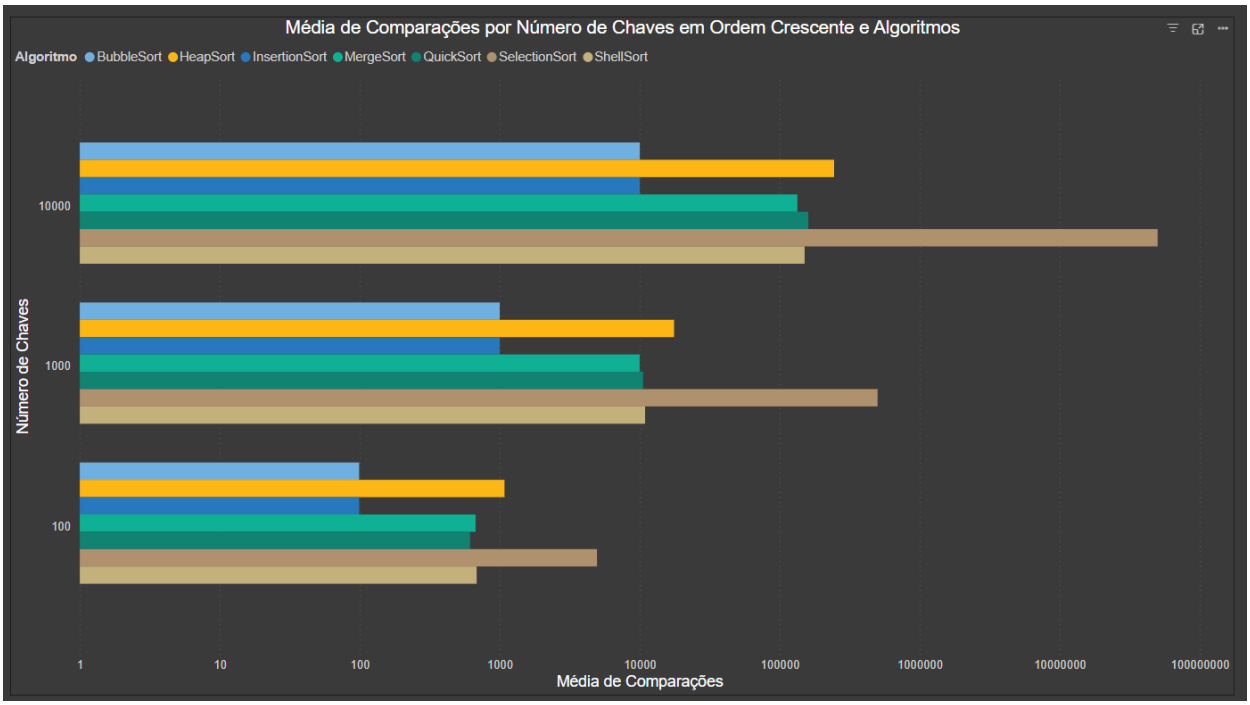


Este cenário não apresenta maiores novidades que o anterior, ainda temos o SelectionSort como o com menor número de movimentações e um destaque para o ShellSort como o segundo menor.

Vamos explorar então o último teste feito: Número de Comparações.

5.3 Número de Comparações X Número de Chaves

5.3.1 Chaves em Ordem Crescente



Como dito anteriormente, o SelectionSort paga o preço pelo seu baixo número de movimentações aqui. Com 49995000 comparações em um vetor de 10 mil chaves, ele assume disparadamente o posto de algoritmo com maior número de comparações em vetores já ordenados. Isso ocorre pois ele começa no primeiro elemento do vetor e o compara com todos os outros, procurando o menor. O cálculo do número de comparações assume então uma P.A de razão -1, vamos calcular ela no caso do vetor de 10 mil elementos para ver se bate com o nosso resultado:

$$S_{9999} = \frac{n}{2} \cdot (a_1 + a_n)$$

onde $n = 9999$ é o número de termos, $a_1 = 9999$ é o primeiro termo e $a_n = 1$ é o último termo.

Substituindo:

$$S_{9999} = \frac{9999}{2} \cdot (9999 + 1)$$

$$S_{9999} = \frac{9999}{2} \cdot 10000$$

$$S_{9999} = 4999.5 \cdot 10000$$

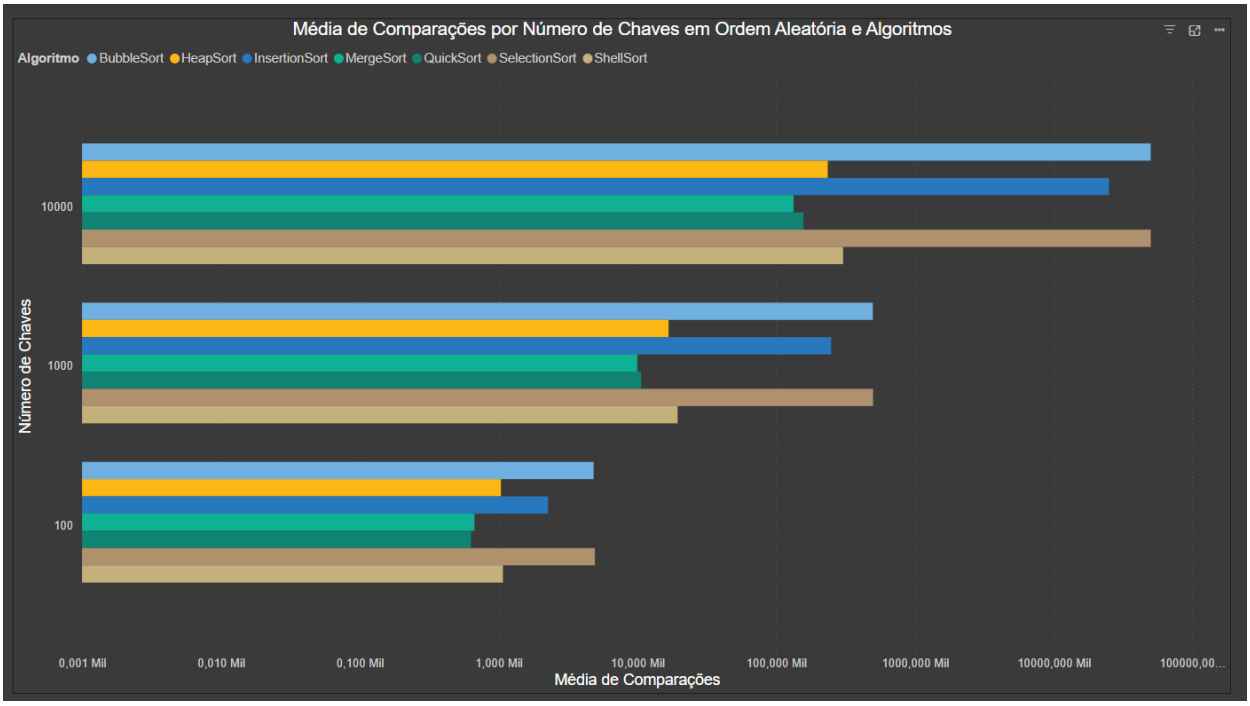
$$S_{9999} = 49995000$$

Batendo com o nosso resultado.

O BubbleSort e o InsertionSort apresentam os menores números, reforçando sua invencibilidade em vetores já ordenados.

Já os algorimtos $O(n \log n)$ e o ShellSort mostraram resultados medianos, assim como em praticamente todos os outros testes, justificando sua eficiência constante em todos os cenários.

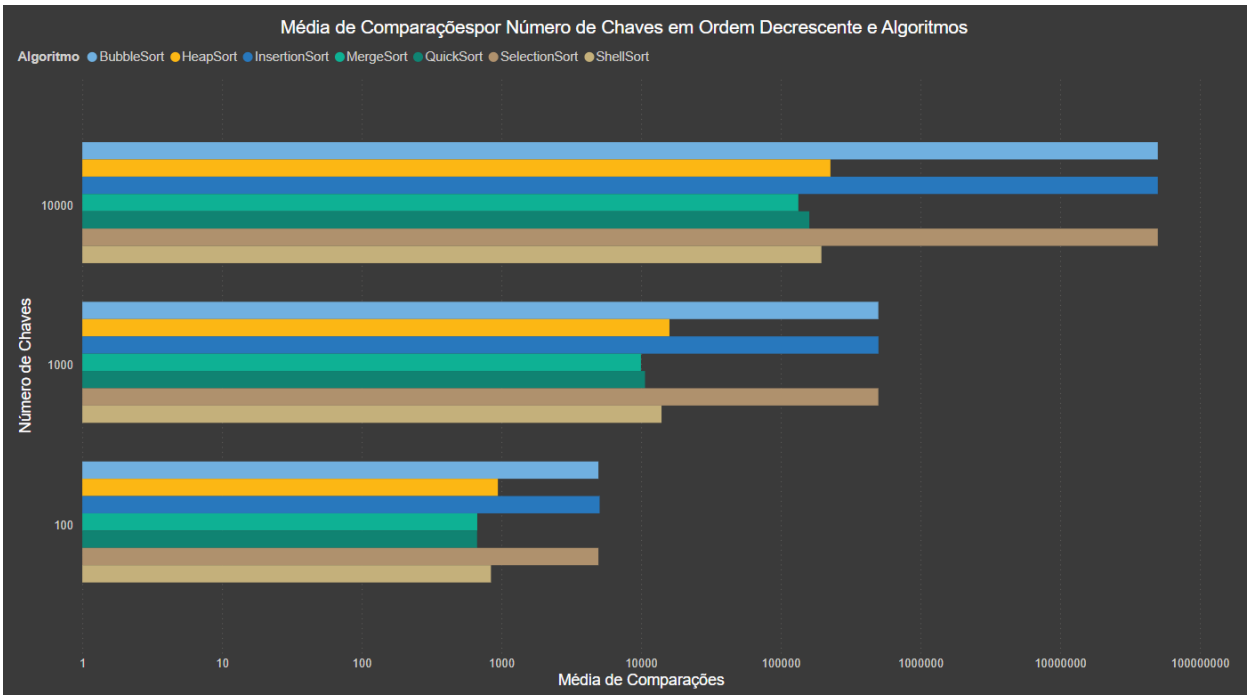
5.3.2 Chaves em Ordem Aleatória



Voltamos ao cenário mais comum no nosso dia a dia. Nele, percebemos novamente a ineficiência do Bubble e do InsertionSort em vetores que não estejam ordenados, liderando os algoritmos com maior número de comparações. O SelectionSort também continua o mesmo do gráfico anterior.

Os de complexidade $O(n \log n)$ começam a abrir vantegens cada vez maiores que os de complexidade $O(n^2)$. Ainda vale a pena dizer que o ShellSort apresenta um valor ligeiramente maior que o QuickSort e companhia.

5.3.3 Chaves em Ordem Decrescente



Com uma aparência quase idêntica ao gráfico anterior, nós chegamos ao último teste feito com esses algoritmos de ordenação.

Assim como o anterior, Bubble, Insertion e SelectionSort possuem os maiores números de comparações: 49995000, 50004999 e 49995000, respectivamente.

Merge, Quick, Shell e HeapSort apresentam os menores valores: 133616, 156571, 195433 e 226682, respectivamente.

5.4 Gráfico de Linhas

Estes próximos três gráficos possuem os mesmos dados dos três primeiros que analisamos anteriormente. Entretanto, fica mais fácil de visualizar o comportamento e tendência deles deste modo:

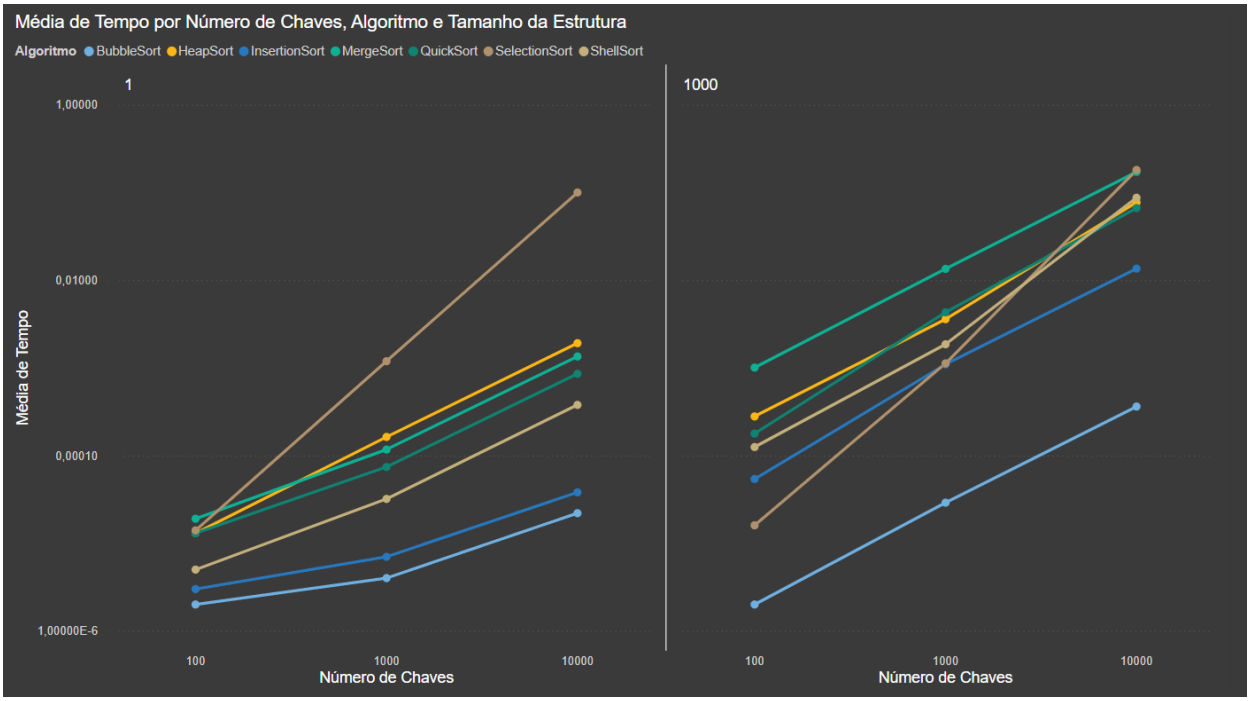


Figure 1: Chaves em Ordem Crescente

Neste, conseguimos ver como o BubbleSort é rápido em vetores já ordenados, independente do tamanho da estrutura. Vemos também que o SelectionSort apresenta uma linha com um coeficiente de inclinação muito maior que os demais nesse cenário, revelando seu péssimo desempenho em arrays ordenados.

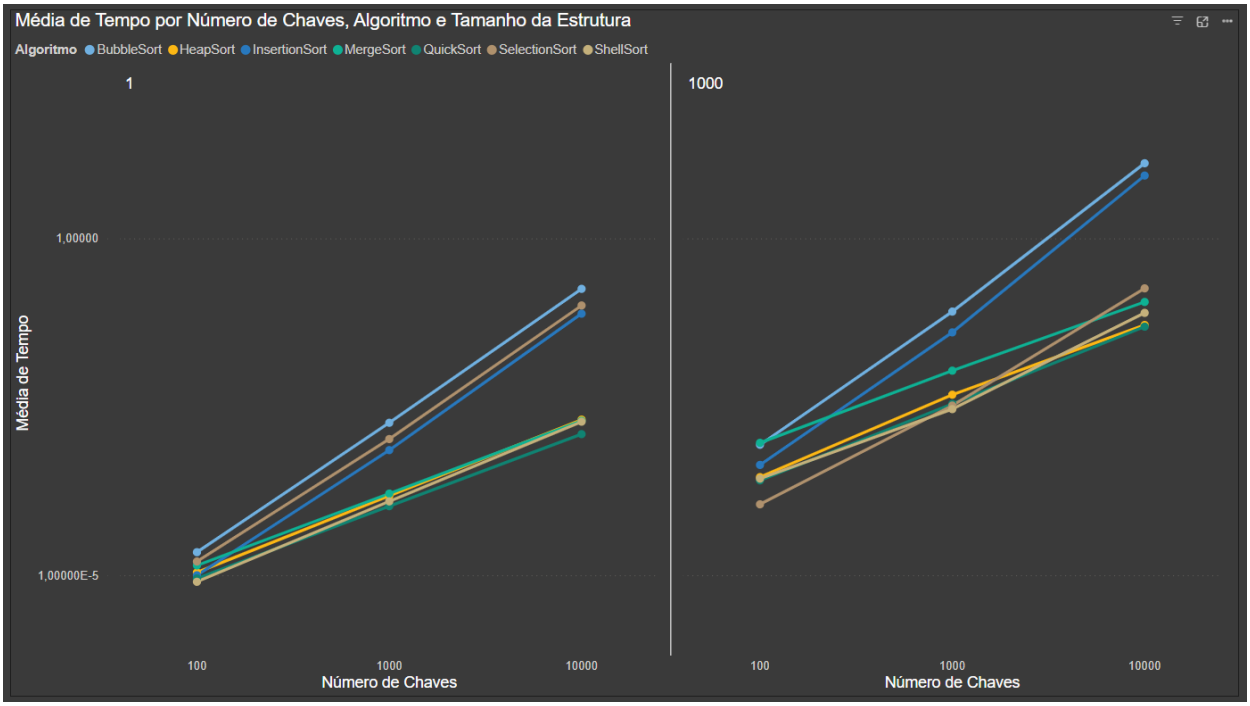


Figure 2: Chaves em Ordem Aleatória

Agora, percebemos claramente a divisão de desempenho entre os algoritmos $O(n \log n)$ e $O(n^2)$ no gráfico à esquerda (tamanho da estrutura 1). Por outro lado, também conseguimos ver o caso em que o SelectionSort se aproxima do QuickSort e amigos (gráfico à direita), como abordado no assunto de número de movimentações.

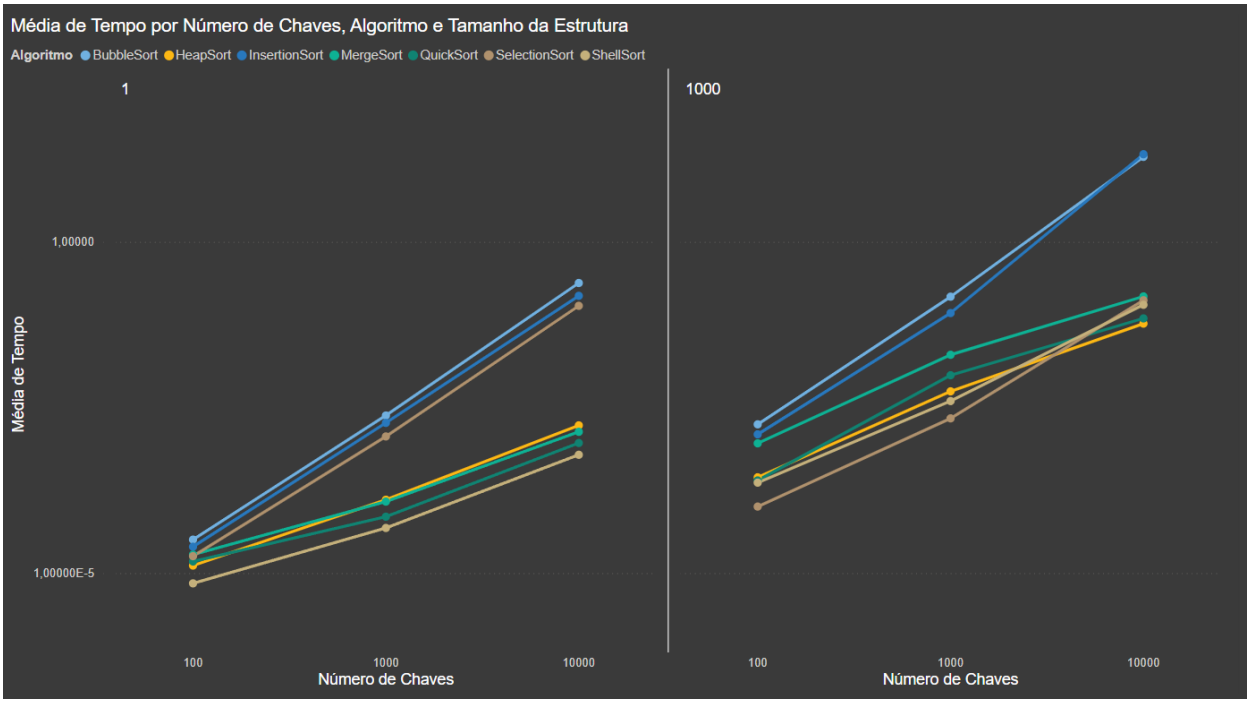


Figure 3: Chaves em Ordem Decrescente

Por fim, se olharmos para o ponto em que o número de chaves é igual a 1000, percebemos que o Quick, Heap e MergeSort começam a diminuir o coeficiente de inclinação de suas retas, revelando uma melhora de desempenho em vetores maiores. Enquanto isso, nesse mesmo ponto o Selection e o ShellSort acabam aumentando seu ângulo de inclinação e, portanto, revelando sua fraqueza em arrays de tamanhos maiores.

6 Conclusão

6.1 Casos de Vetores Ordenados Crescentemente

Vimos então que quando o vetor já está ordenado, o BubbleSort e o InsertionSort disparam em questão de tempo de ordenação. Isso é devido à condição de verificação de troca do BubbleSort e da inserção no local correto do InsertionSort, diminuindo o tempo que é gasto com movimentações como os outros algoritmos fazem.

6.2 Casos de Vetores Ordenados Aleatoriamente

Agora quando o vetor se encontra da maneira como geralmente presenciamos no dia a dia, o Bubble e o Insertion deixam de ser opções viáveis. Chegam em cena então os algoritmos eficientes: QuickSort, HeapSort e MergeSort. Além disso, temos um intermediário entre esses algoritmos $O(n^2)$ e $O(n \log n)$: o ShellSort, $O(n \log^2 n)$ ou $O(n^{1,25})$.

Nesse caso, esses algoritmos são a escolha da vez, apresentando um melhor tempo de ordenação e relativamente próximo um dos outros.

Temos que lembrar também do SelectionSort, que, devido ao seu baixo número de movimentação em relação aos outros algoritmos, consegue chegar perto da média de tempo de ordenação dos algoritmos eficientes. Porém, para vetores com mais de 10 mil elementos, ele começa a revelar sua fraqueza.

6.3 Casos de Vetores Ordenados Decrescentemente

As informações a serem concluídas neste caso são basicamentes as mesmas do caso anterior. Talvez valha a pena citar a competitividade do ShellSort com os algortimos eficientes, que conseguiu se manter com tempo de execução, movimentações e comparações próximos ou menores do que as deles. Ainda assim, para vetores maiores que 10 mil chaves, os gráficos de linhas mostram que eles irão se separar mais.