Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Curso de Bacharelado em Sistemas de Informação

Algoritmos de ordenação

Bruno Máximo
Débora Ngan Guei Lai
Pedro Fernando Christofoletti dos Santos
Heitor Camilo de Freitas e Oliveira
Henrique Delgado Franzin

Monografia apresentada na disciplina SSC0503 -
Introdução à Ciência de Computação II, como
requisito parcial para obtenção da aprovação na mesma.

São Carlos/SP, 16 de Setembro de 2019

SUMÁRIO

Resumo: O tema desta monografia é Algoritmos de ordenação. Será explicado o que são e suas características tais como: complexidade, estabilidade, funcionamento, implementações e gráficos e/ou tabelas contendo informações sobre seu desempenho. Serão apresentadas comparações entre os algoritmos a fim de determinar quais são os mais eficientes para determinados casos.

1. Introdução

Algoritmo de ordenação em ciência da computação é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem -- em outras palavras, efetua sua ordenação completa ou parcial. As ordens mais usadas são a numérica e a lexicográfica. Existem várias razões para se ordenar uma sequência. Uma delas é a possibilidade de acessar seus dados de modo mais eficiente.

A partir dos nossos estudos sobre esses algoritmos,, foram feitas oito implementações diferentes, sendo elas: *BubbleSort*, *BubbleSort* com sentinela, *SelectionSort*, *Insertion Sort*, *HeapSort*, *ShellSort*, *QuickSort* e *MergeSort*.

A análise dos algoritmos em questão foi feita usando vetores de tamanho 10^K onde K ε N variou de 2 à 6 para todos os seguintes cenários:

- Vetores Aleatórios
- Vetores Ordenados
- Vetores Inversamente Ordenados
- Vetores Quase ordenados (10% ordenado)

Na análise de cada algoritmo foi coletada as informações de:

- Número de trocas feitas
- Número de comparações
- Tempo de execução

A partir dos dados coletados foram gerados gráficos plotados em R usando a biblioteca *ggplot2* a fim de melhor visualização do desempenho de cada algoritmo. Além disso, será explicado a maneira com a qual cada um trabalha para solucionar o problema. Dessa forma será possível compreender mais sobre suas complexidade e o modo como se comportam em cada caso. Haja vista que foram expostos em seu Pior caso, Caso Médio e Melhor Caso.

Tendo em vista o profundo estudo dessas tecnologias é possível justificar a constante criação e busca de um algoritmo de ordenação que possua a melhor eficácia possível na maioria dos casos, algo que, dentre os algoritmos em questão, será apresentado.

Esses algoritmos são utilizados no dia a dia, falando em linguagens estruturadas, assim a necessidade de reduzir os tempos de execução é fundamental. Afinal de contas, o volume de dados só aumenta, gerando tempos maiores de execução que podem, à medida do tempo, formar gargalos em sistemas que utilizam esses processos. Um bom exemplo é a folha de pagamento, onde cada funcionário de uma empresa possui dados tais como: salário, cargo, nome etc, e que devem ser ordenados possibilitando que cada indivíduo receba o respectivo salário.

2. Como foi feito a aplicação

A aplicação consiste de um programa escrito em C com interface de linha de comando onde o algoritmos de ordenação a ser usado, o tipo de vetor e a ordem de grandeza dele são passados por argumento da linha de comando para o programa.

Os argumentos que definem o algoritmo a ser usado são:

- -b: bubblesort sem sentinela
- -g: bubblesort com sentinela
- -s: selectionsort
- -i: insertionsort
- -m: mergesort
- -h: heapsort
- -q: quicksort
- -z: shellsort

Os argumentos que definem o tipo de vetor e ordem de grandeza são:

- -r: vetor aleatório
- -a: vetor ordenado de forma crescente
- -d: vetor inversamente ordenado
- -p: vetor parcialmente (10%) ordenado
- -k <valor>: ordem de grandeza.

O programa é seguido de um shell script que automatiza toda as execuções de cada combinação de argumentos chamado runall.sh. O programa escreve na saída padrão o número de trocas, comparações e tempo de execução, além do nome do algoritmo. As linhas referentes a saída do vetor na tela estão comentadas para não poluir o terminal.

A utilização do programa se dá através do seguinte comando:

./programa -<algoritmo> -<vetor> -k <ordem>

Só é garantido o funcionamento do programa se o usuário cumprir o uso desses argumentos. Não nos responsabilizamos pelo mau funcionamento do programa se caso esses argumentos não sejam passados. O runall.sh automaticamente compila o código e gera um executável básico a.out.

3. Fundamentação Teórica

3.1 Estabilidade de um algoritmo

Um algoritmo de ordenação é considerado estável se possui uma ordem de registros de chaves iguais. O que significa que esses registros aparecem em uma mesma sequência ordenada tanto no fim da ordenação quanto na sequência inicial. Uma boa forma de se pensar em registros e chaves é representando respectivamente os mesmos por números e naipes de cartas de um baralho, embora se embaralhe e escolha um número **n** de cartas, os naipes se mostram como uma classe desses números, assim essas cartas serão ordenadas primeiro levando em conta seus naipes e depois os números.
Um algoritmo não estável não utiliza uma o conceito das chaves, o que pode fazer com que uma ordenação nem sempre seja a exata, tendo em mente mais de uma característica de cada elemento.

Por exemplo, um algoritmo estável ordenando a sequência de números (chaves) com letras associadas (registros):

```
3[a], 2[b], 2[c], 1[d]
```

obrigatoriamente retornará:

```
1[d], 2[b], 2[c], 3[a]
```

enquanto algoritmos não estáveis sujeitam os elementos associados aos objetos a serem ordenados a mudanças:

```
1[d], 2[c], 2[b], 3[a]
```

3.2 Notação assintótica

Para se falar sobre algoritmos de ordenação, é necessário explicar o conceito de complexidade e como isso afeta o Tempo de execução, Espaço utilizado na memória, Número de trocas e comparações realizadas.

Se o quantidade de elementos a serem ordenados é pequena, o tempo de execução é pode ser desprezado, entretanto, ao aumentar essa quantidade, o tempo irá subir de acordo com a notação assintótica do algoritmo, a qual é representada por **O(f(n))**, onde n = número de elementos a serem ordenados e f(n) o comportamento (a função) que determina o funcionamento do algoritmo.

Exemplo:

- $O(n^2)$
- O(n log n)
- O(n)

Esse O significa que independente da quantidade de elementos e o modo como estão organizados, o vetor em questão jamais terá um caso pior do que essa complexidade..

Além da notação tipo O, também existem outras anotações como: **Ω(ômega)**, que representa um mínimo, uma visão otimista, do desempenho de um algoritmo. Pode ser algo impreciso, pensando em uma função, ela pode estabelecer que o valor exato está acima do valor dado, sendo imprecisa, mas ainda sim entrega algo verdadeiro. Outro tipo de notação é o **Θ(theta)**. Essa notação define duas constantes k1 e k2, as quais multiplicarão f(n), entretanto somente para termos de complexidade alta, como n^3, n^2 e n!. Na prática é como se os termos de ordem baixa, como n^1 fossem descartados. Dessa forma, k1 f(n) formará uma função que terá valores menores que a função dada por k2 f(n), assim haverá uma área entre as duas funções, e é essa área onde estará o desempenho exato do algoritmo.

4. Algoritmos

4.1 BubbleSort

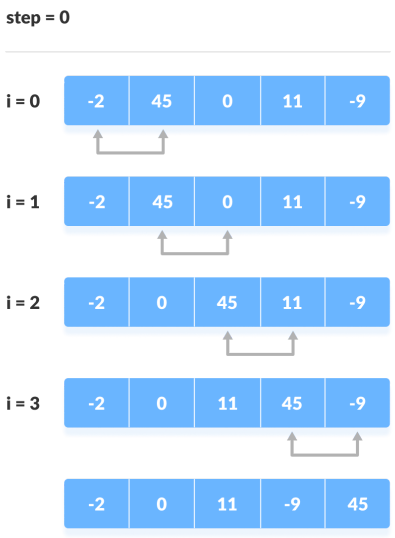
4.1.1 Descrição

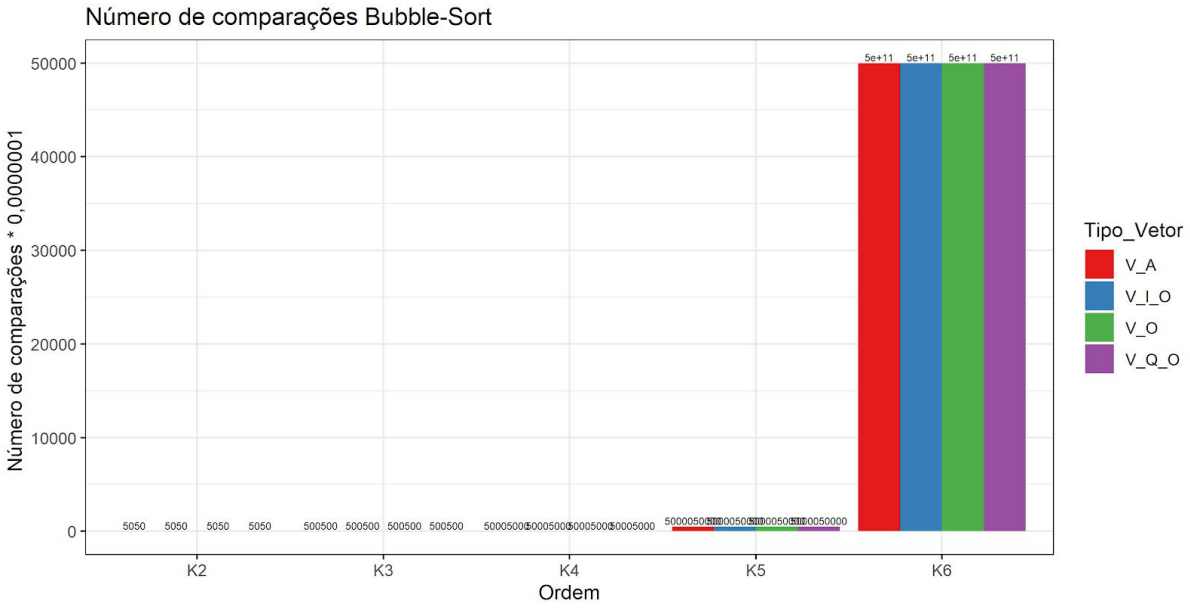
Bubble sort é um algoritmo de ordenação que faz comparações entre elementos adjacentes da esquerda para direita de modo que a cada passagem, o maior elemento fique em última posição. Em contexto de ordenações crescentes, a cada comparação, caso o segundo elemento seja maior ou ambos terem o mesmo valor, não é realizado nenhuma troca e segue a comparação com o próximo elemento, caso contrário, será feito troca de suas respectivas posições e a comparação continua com os próximos elementos.

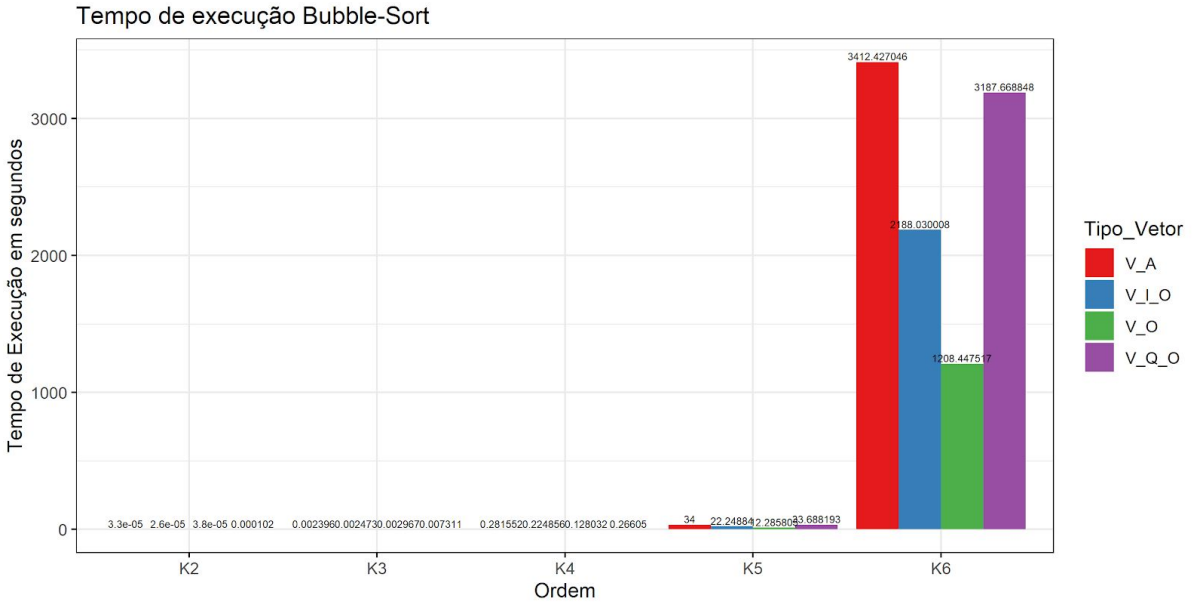
4.1.2 Características

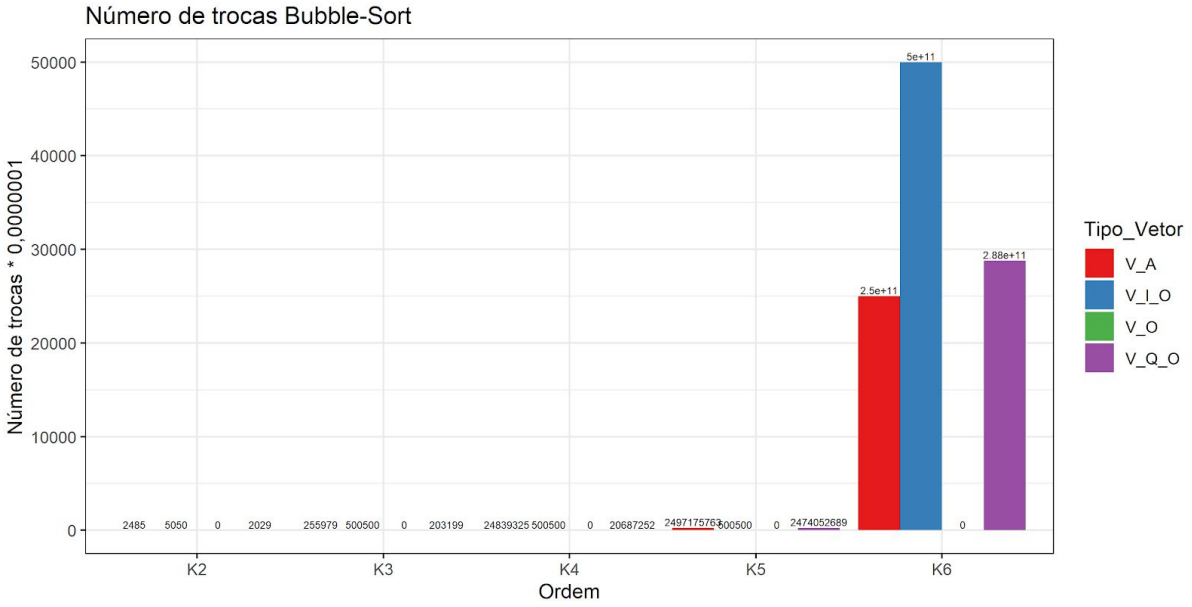
- Algoritmo intuitivo, de simples implementação e manutenção;
- Adequado para pequenas estruturas;
- Complexidade de espaço: O(1);
- Complexidade de troca: $O(n^2)$;
- Complexidade de tempo em todos os casos: $O(n^2)$;
- Algoritmo estável: não altera a ordem relativa dos elementos com valores iguais.

4.1.3 Gráficos e Explicações ilustrativas









4.1.4 Implementação em C

```
void bubble(int *vetor, int n){
     int i, j, aux;
     for(i = n-1; i >= 0; i--){
          for(j = 0; j < i; j++){
               if(v[j] > v[j+1]){
                         aux = v[j];
                         v[j] = v[j+1];
                         v[j+1] = aux;
                    }
          }
     }
}
```

4.2 BubbleSort com sentinela

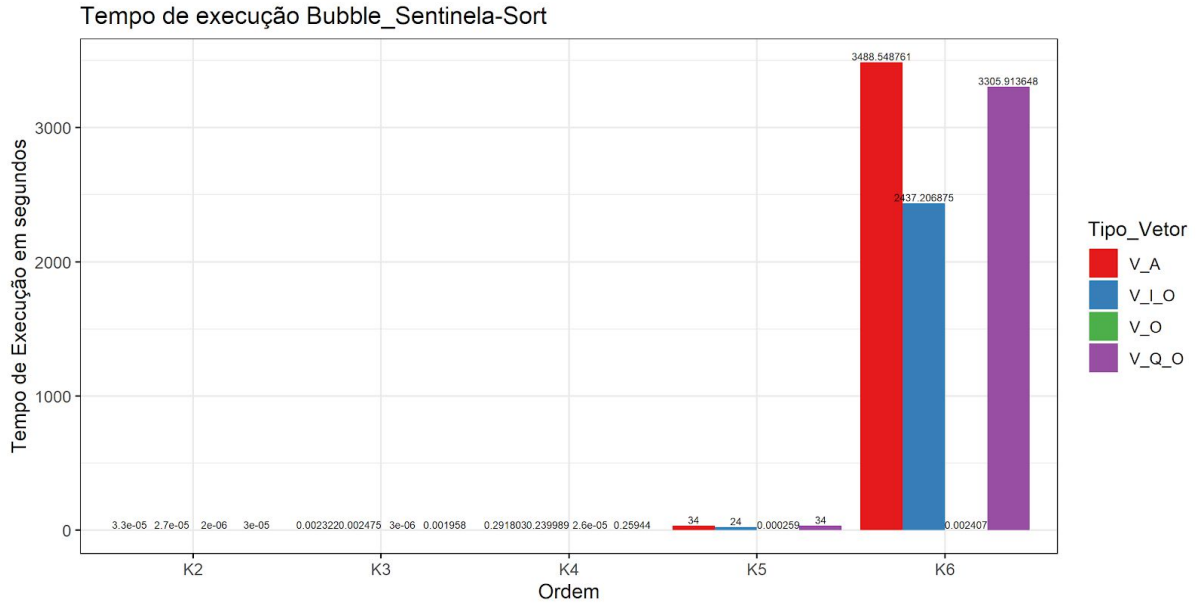
4.2.1 Descrição

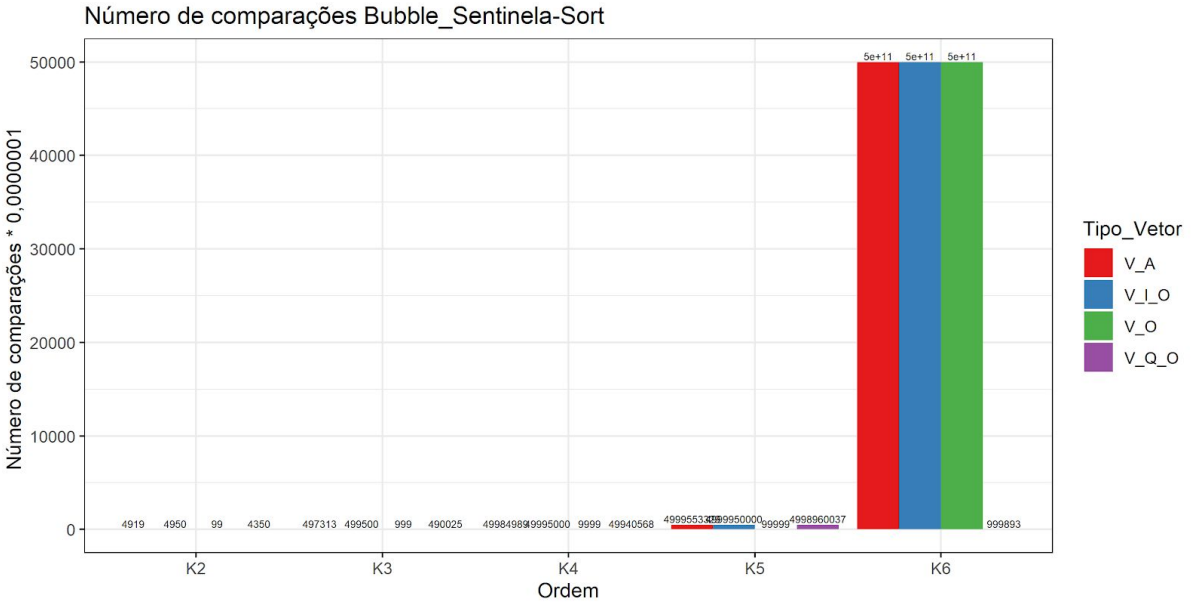
O BubbleSort com sentinela segue o mesmo raciocínio do Bubblesort comum porém com uma otimização, já que faz uso de uma sentinela para reconhecer quando os elementos já estão ordenados, sem necessitar de demais comparações até que todos os elementos sejam verificados.

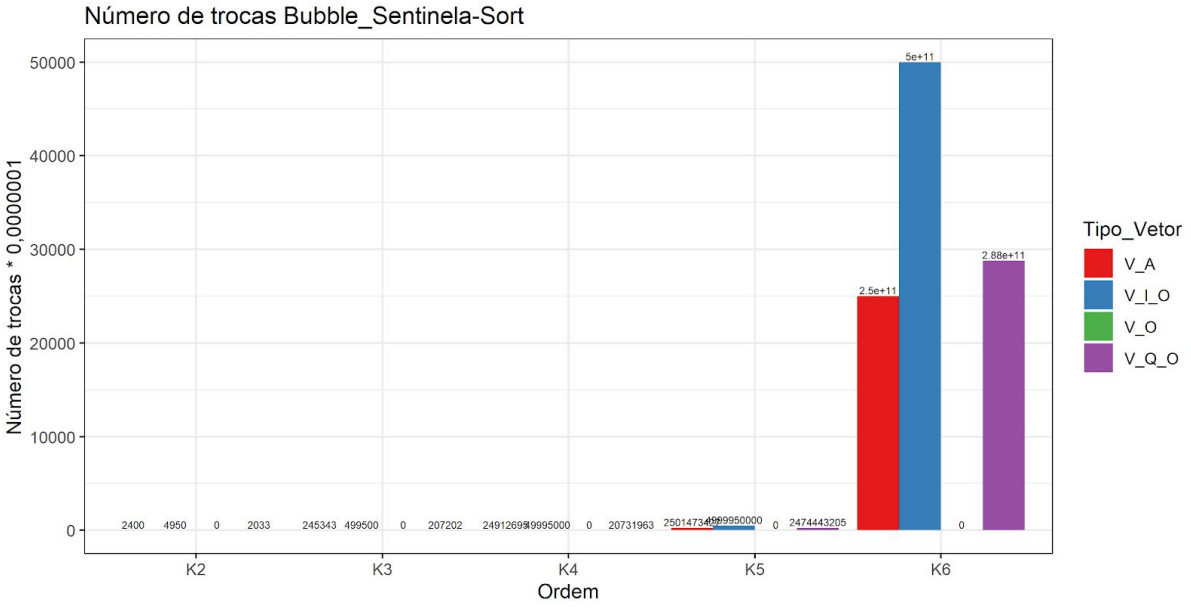
4.2.2 Características

- Algoritmo intuitivo, de simples implementação e manutenção;
- Adequado para pequenas estruturas;
- Complexidade de espaço: O(1);
- Complexidade de tempo: constante em todos os casos: O(n^2);
- Complexidade de trocas: O(n^2);
- Algoritmo estável, ou seja, não altera a ordem relativa dos elementos com valores iguais.

4.2.3 Gráficos







4.2.4 Implementação em C

```
void bubbleSentinela(int *vetor, int n){
    int j, u, i, temp;
    j = n-1;
    while( j > 0 ){
        u = -1;
        for( i = 0; i < j; i++ ){
            if( vetor[i] > vetor[i+1] ){
                temp = vetor[i];
                vetor[i] = vetor[i+1];
                vetor[i+1] = temp;
                u = i;
            }
```

```
                }
                j = u;
            }
        }
```

4.3 SelectionSort

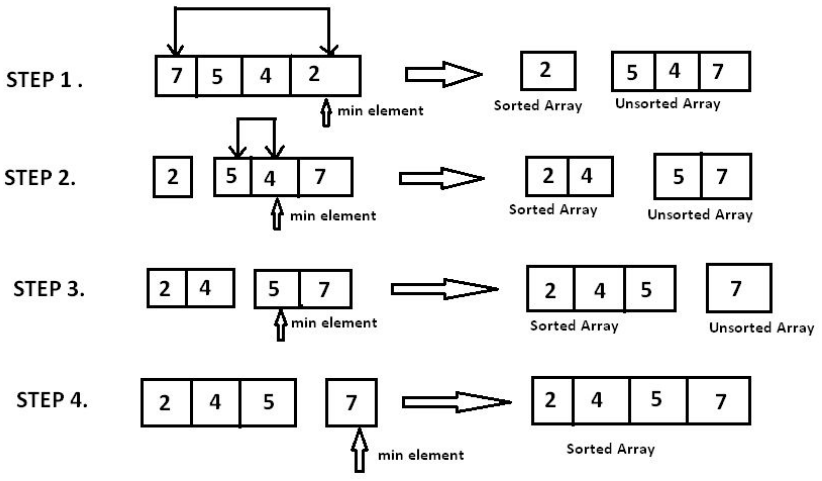
4.3.1 Descrição

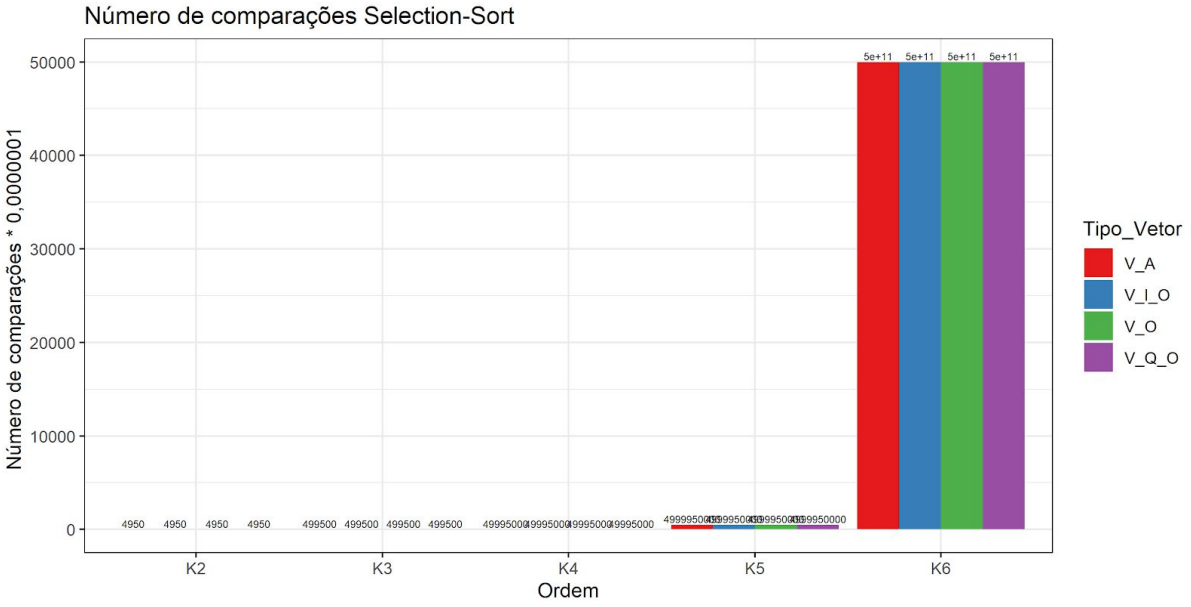
O algoritmo de ordenação *selection* tem por objetivo percorrer a estrutura a ser ordenada até encontrar o menor elemento, em seguida, o coloca em primeira posição, posteriormente, percorre novamente a estrutura até encontrar o segundo menor elemento e o coloca em segunda posição, a execução termina somente quando o maior elemento ingressar em última posição.

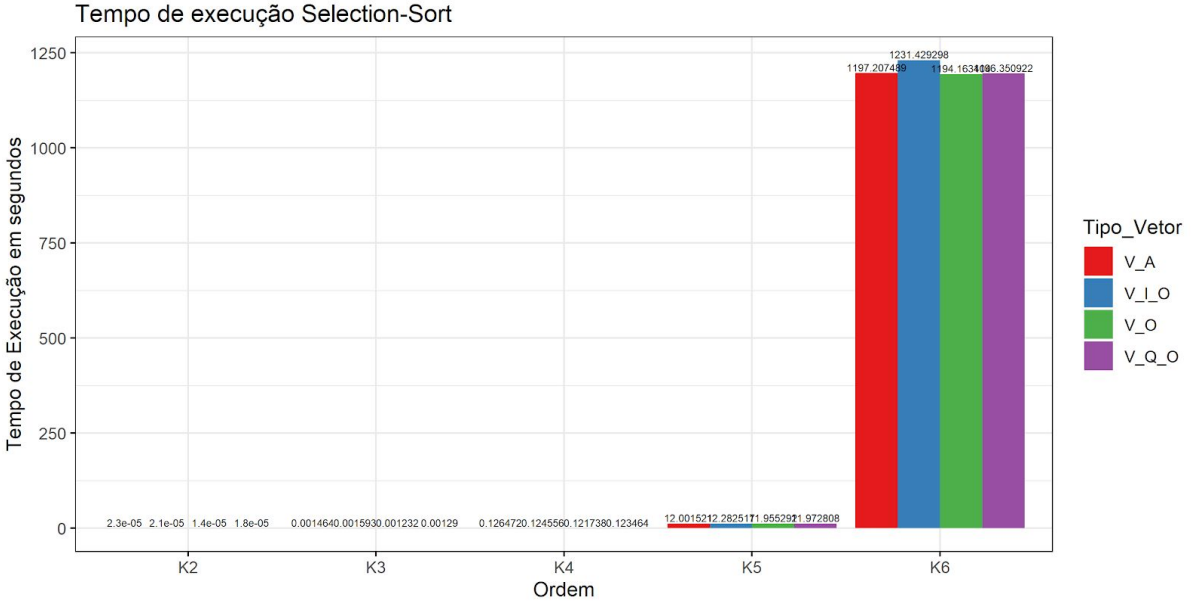
4.3.2 Características

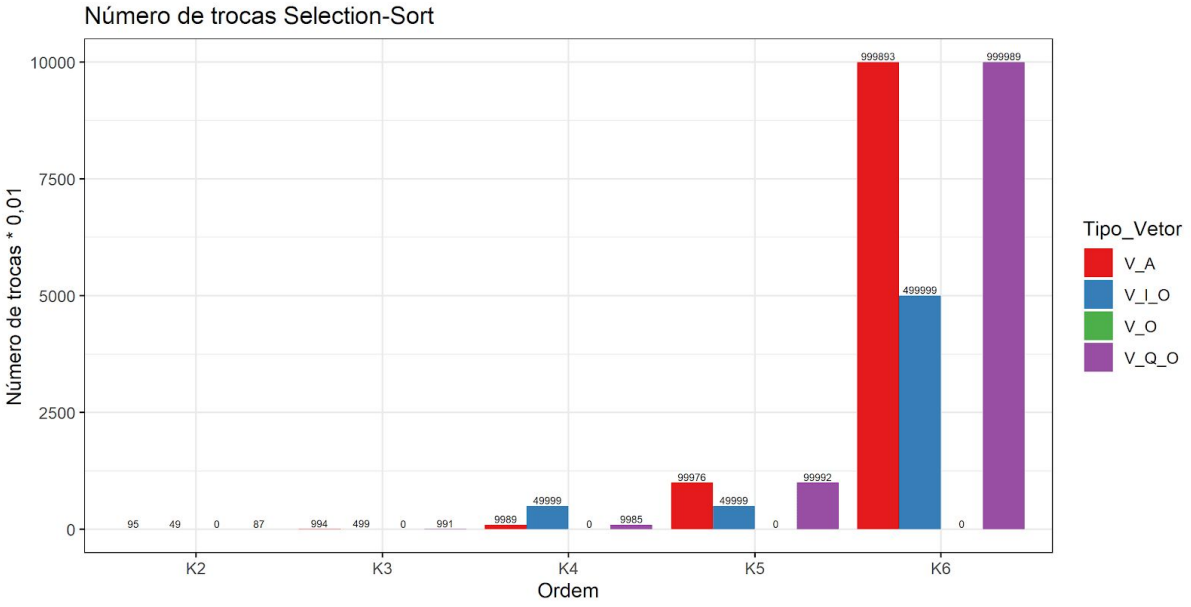
- Algoritmo intuitivo, de simples implementação e manutenção;
- Adequado para pequenas estruturas;
- Complexidade de espaço: O(1);
- Complexidade de tempo: constante em todos os casos, $O(n^2)$;
- Complexidade de trocas: O(n);
- Não é um algoritmo estável, ou seja, pode alterar a ordem relativa dos elementos com valores iguais.

4.3.3 Gráficos e Explicações ilustrativas




Número de comparações Selection-Sort
Número de comparações * 0,0000001
50000
40000
30000
20000
10000
0
5e+11
4950
499500
K2
K3
K4
K5
K6
Ordem
Tipo_Vetor
V_A
V_I_O
V_O
V_Q_O


Tempo de execução Selection-Sort
Tempo de Execução em segundos
1250
1000
750
500
250
0
1231.429298
1197.207489
2.3e-05
2.1e-05
1.4e-05
1.8e-05
0.001464
0.001593
0.001232
0.00129
0.126472
0.124556
0.121738
0.123464
12.001521
12.282517
11.955292
11.972808
K2
K3
K4
K5
K6
Ordem
Tipo_Vetor
V_A
V_I_O
V_O
V_Q_O



4.3.4 Implementação em C

```
void selection(int *v, int n){
     int i, j, min, aux;
     for (i = 0; i < n-1; i++){
          min = i;
        for (j = i+1; j < n; j++){
             if(v[j] < v[min]){
               min = j;
             }
        }
        if(i != min){
            aux = v[i];
            v[i] = v[min];
            v[min] = aux;
        }

 }
```

4.4 InsertionSort

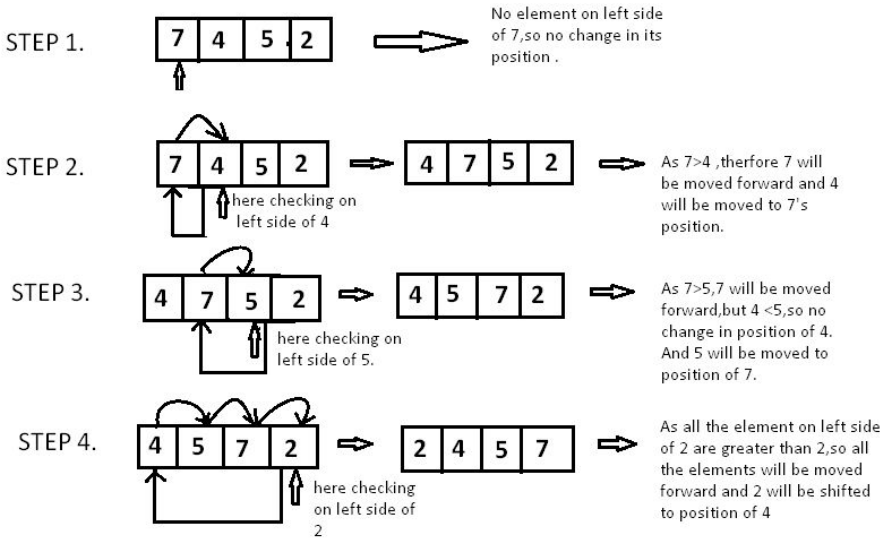
4.4.1 Descrição

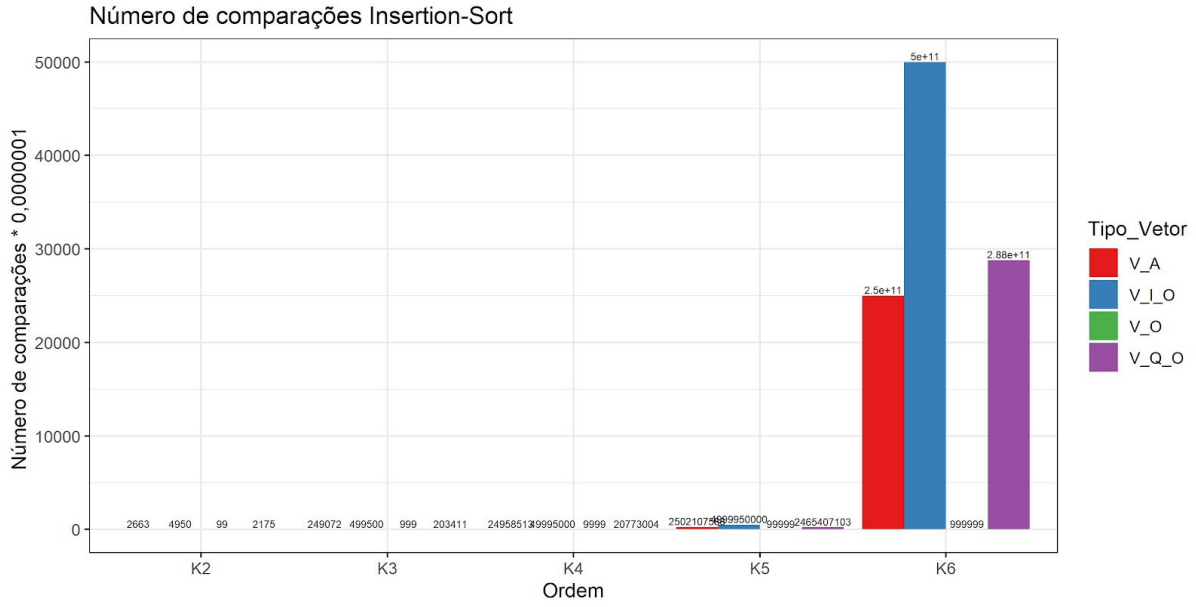
O objetivo desse algoritmo é fazer inserção dos elementos em sua posição correspondente da estrutura. Em caso de ordenação crescente e levando em consideração o sentido da esquerda para a direita, as realizando das inserções são feitas a cada vez que passa pela estrutura, fazendo comparações entre os elementos da estrutura, com o objetivo de encontrar o menor elemento. A execução é finalizada quando não houver mais trocas entre elementos a serem realizadas.

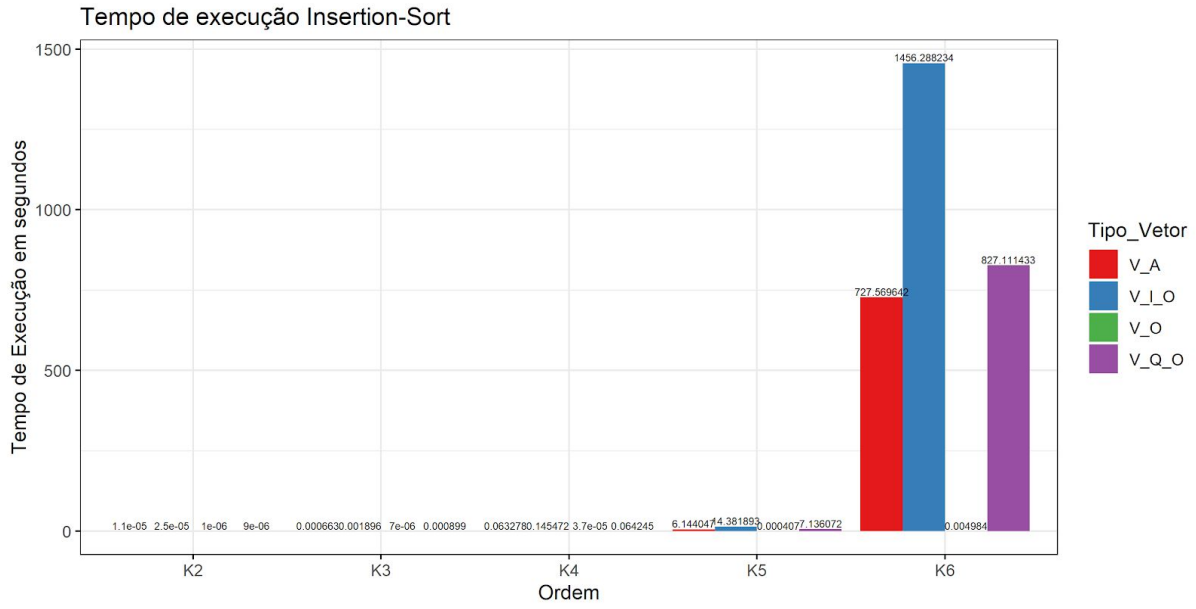
4.4.2 Características

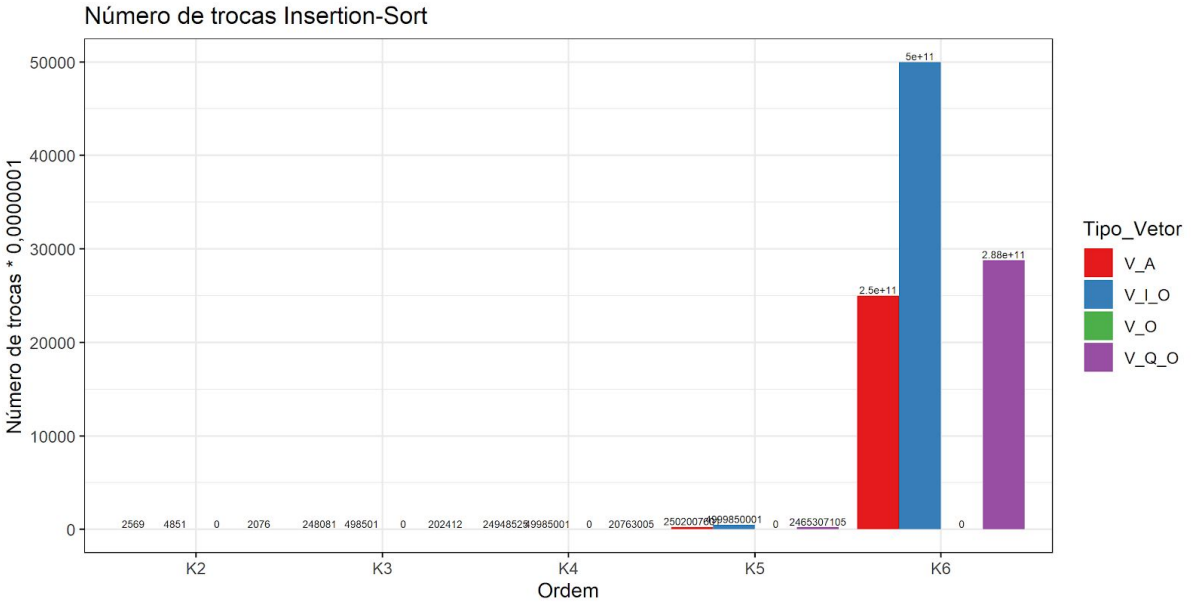
- Simples implementação e manutenção;
- Adequado para pequenas estruturas;
- Complexidade de espaço O(1);
- Complexidade de tempo no pior caso: O(n^2), quando a estrutura está inversamente ordenada;
- Complexidade de tempo no caso médio: O(n^2);
- Complexidade de tempo no melhor caso: O(n), quando a estrutura está ordenada;
- Algoritmo estável, ou seja, não altera a ordem relativa dos elementos com valores iguais.

4.4.3 Gráficos e Explicações ilustrativas




Número de comparações Insertion-Sort
Número de comparações * 0,0000001
50000
40000
30000
20000
10000
0
5e+11
2.88e+11
2.5e+11
2663
4950
99
2175
249072
499500
999
203411
999999
K2
K3
K4
K5
K6
Ordem
Tipo_Vetor
V_A
V_I_O
V_O
V_Q_O


Tempo de execução Insertion-Sort
Tempo de Execução em segundos
1500
1000
500
0
1456.288234
827.111433
727.569642
1.1e-05
2.5e-05
1e-06
9e-06
0.000899
0.064245
0.004984
K2
K3
K4
K5
K6
Ordem
Tipo_Vetor
V_A
V_I_O
V_O
V_Q_O



4.4.4 Implementação em C

```
void insertion(int *v, int n){
    int i, j, atual;
    for (i = 0; i < n; i++)
     {
          atual = v[i];
          j = i - 1;

          while ((j >= 0) && (atual < v[j])){
               v[j + 1] = v[j];
               j--;
          }
          v[j + 1] = atual;
     }
}
```

4.5 HeapSort

4.5.1 Descrição

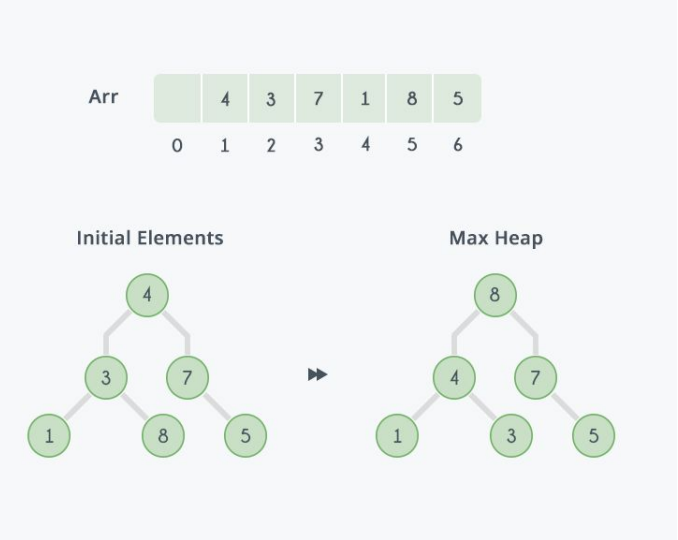
O algoritmo *heapsort* foi desenvolvido em 1964 por Robert W. Floyd e J.W.J Williams, baseada em árvores que faz ordenação por seleção.Tal algoritmo utiliza uma estrutura de dados chamada heap para ordenar os elementos à medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada.

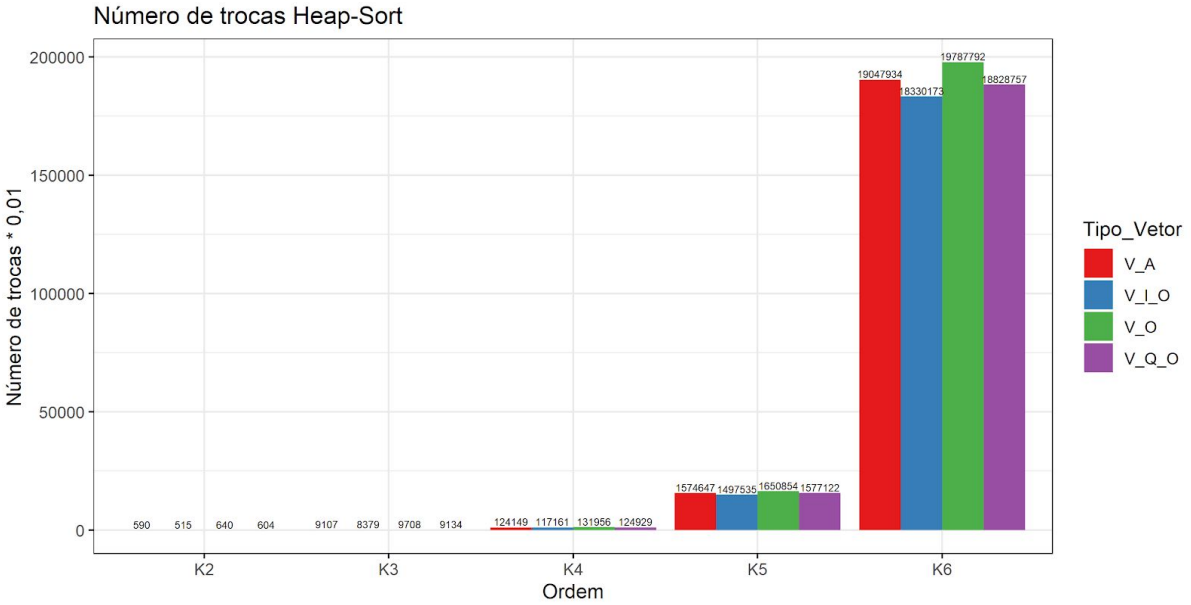
4.5.2 Características

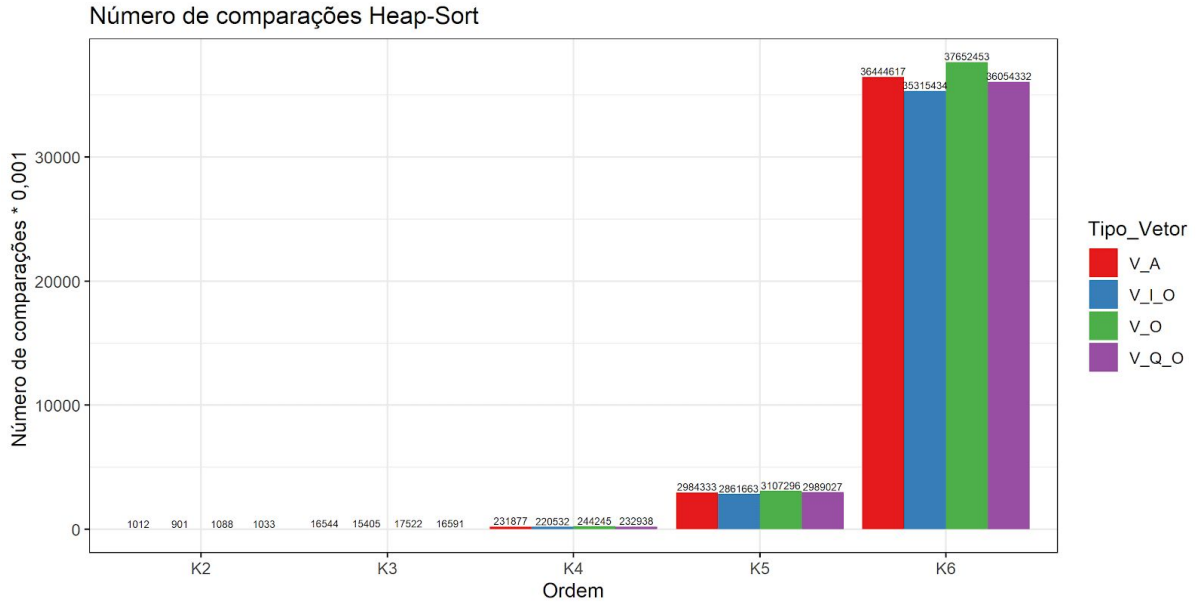
- Adequado para grandes estruturas;
- Complexidade de espaço O(1);

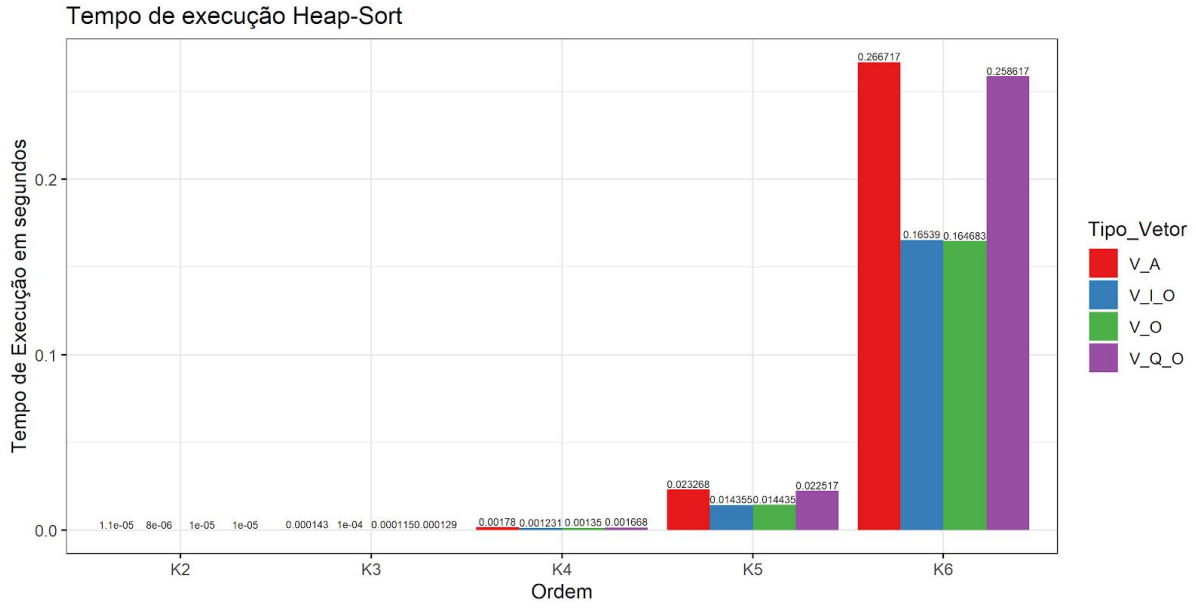
- Complexidade de tempo em todos os casos: O(n log n);
- Complexidade de troca no pior caso: O(n log n);
- Algoritmo não estável, ou seja, altera a ordem relativa dos elementos com valores iguais.

4.5.3 Gráficos e Explicações ilustrativas









4.5.4 Implementação em C

```
void heap (int a[], int n){
  int i = n / 2, pai, filho, t;
      while (1){
          if (i > 0){
         i--;
         t = a[i];
          }else{
               n--;
               if (n <= 0){
                    return;
               }
               t = a[n];
```

```
                    a[n] = a[0];
                }
                pai = i;
                filho = i * 2 + 1;
                while (filho < n)   {
                    if ((filho + 1 < n) && (a[filho + 1]
> a[filho]))
                        filho++;
                   if (a[filho] > t) {
                        a[pai] = a[filho];
                        pai = filho;
                        filho = pai * 2 + 1;
                    }else  break;
                }
                    a[pai] = t;
            }
        }
```

4.6 ShellSort

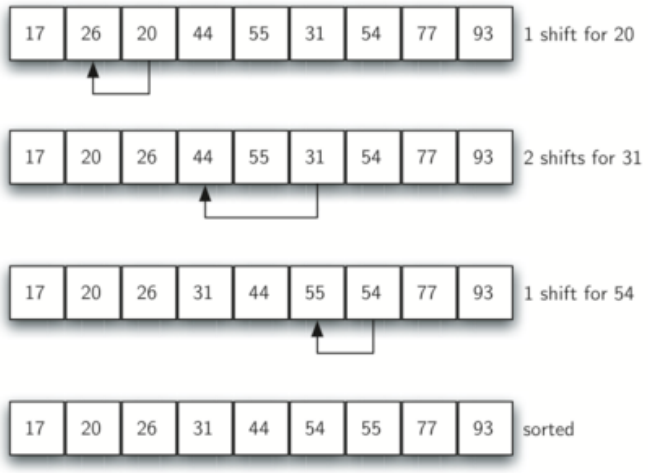
4.6.1 Descrição

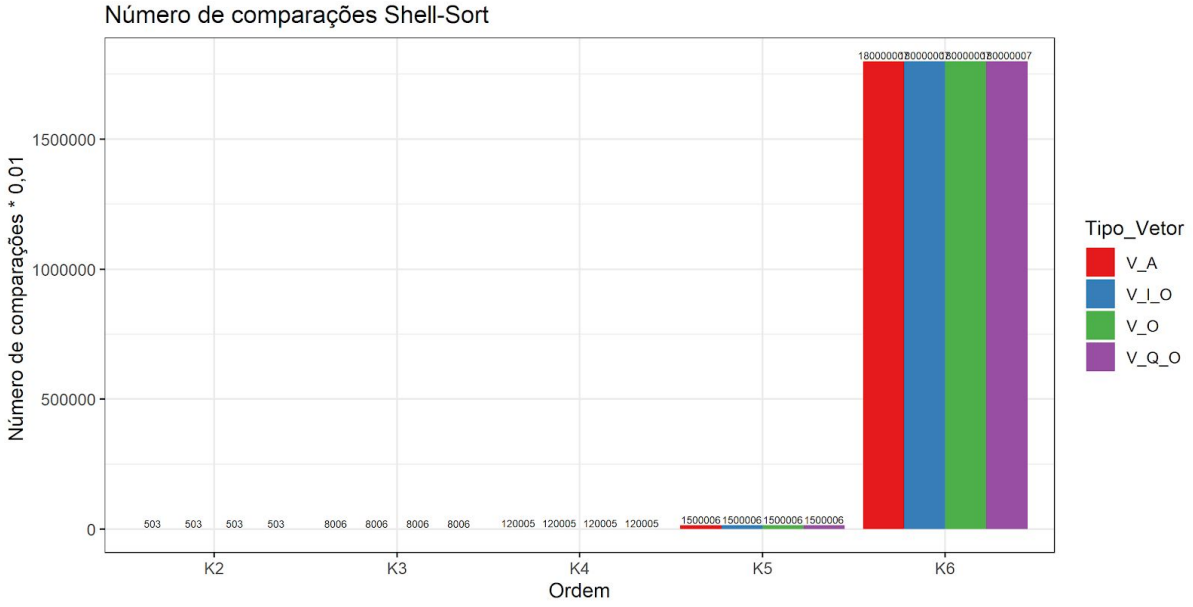
O shellsort foi criado por *Donald Shell* em 1959, o método é uma extensão do algoritmo de ordenação por inserção. Sua proposta consiste em re-arrumar objetos com intervalos maiores que irão diminuir até chegar ao intervalo 1, onde sua execução se espelha na do insertion. Ele troca elementos, que estão a uma distância de por exemplo n/2 no vetor, sendo n o número de elementos, sempre fazendo com que o maior vá para a direita e o menor para a esquerda. O processo em questão começa no primeiro elemento e quando as trocas forem até o elemento n/2, as trocas irão começar novamente do primeiro elemento, mas serão feitas trocas entre elementos que estejam à n/4 de distância no vetor, assim sucessivamente até os valores estejam somente a uma casa de distância, onde ocorrerá o método de inserção comum.

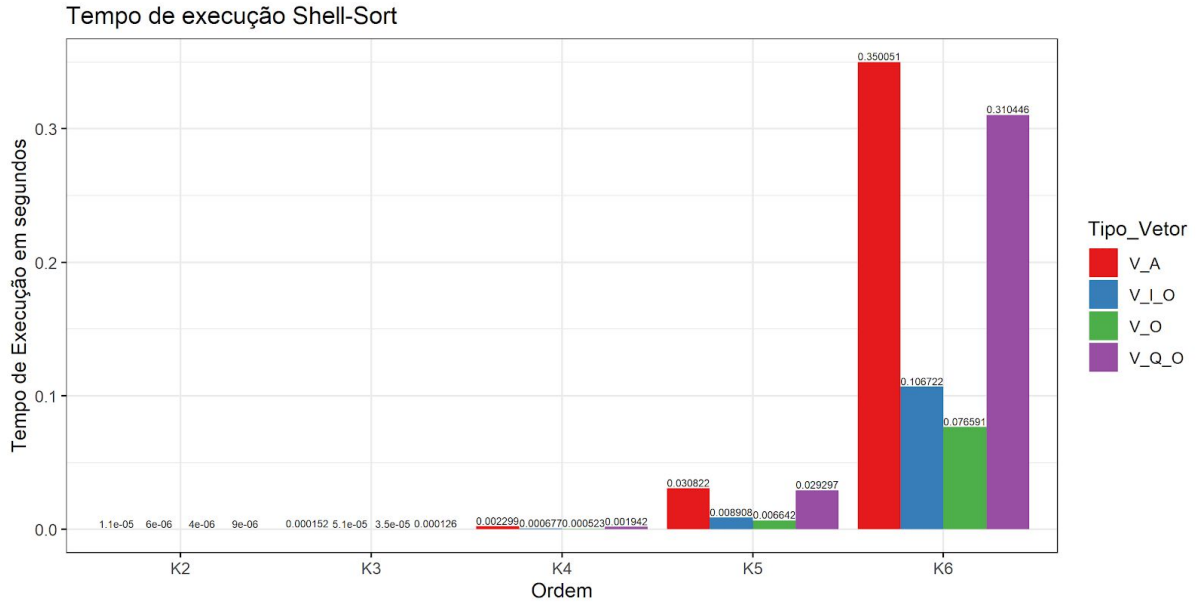
4.6.2 Características

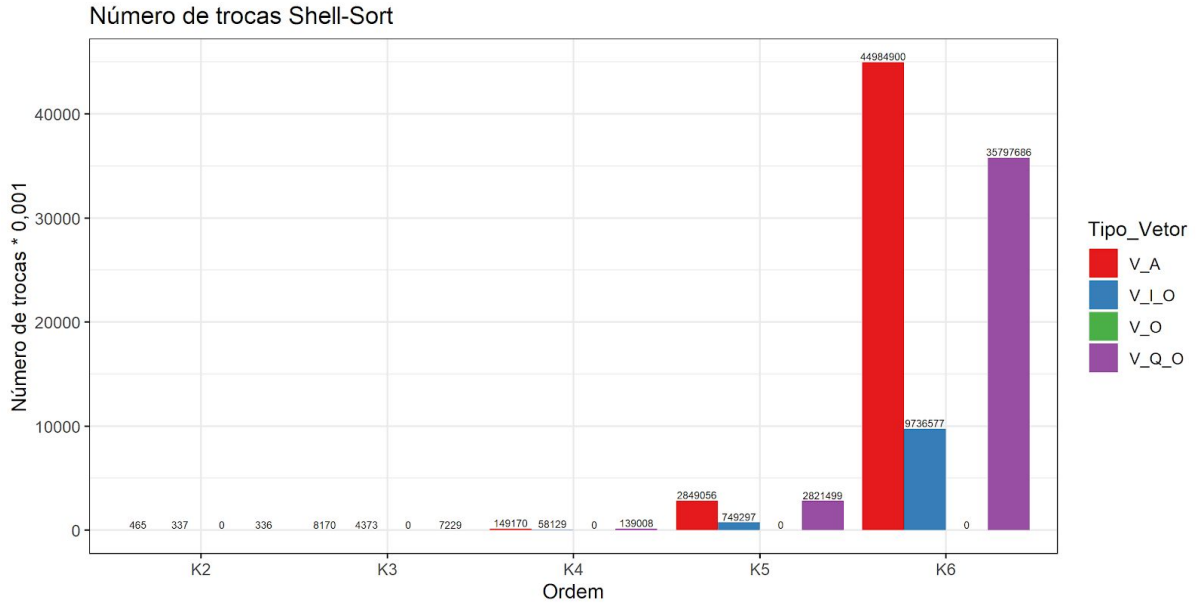
- Complexidade de espaço O(1);
- Permite a troca de registros distantes um do outro;
- Algoritmo não estável, ou seja, altera a ordem relativa dos elementos com valores iguais.

4.6.3 Gráficos e Explicações ilustrativas









4.6.4 Implementação em C

```
void shell(int *v, int n){
    int i, g, m,j;
    for(g = n/2; g >= 1; g /= 2){
        for(i = 0; i < n-g; i++){
            m = v[i+g];
            j=i;

            while(j>=0 && v[j]>m){
                v[j+g] = v[j];
                j-=g;
            }
            v[j+g]=m;}}}
```

4.7 QuickSort

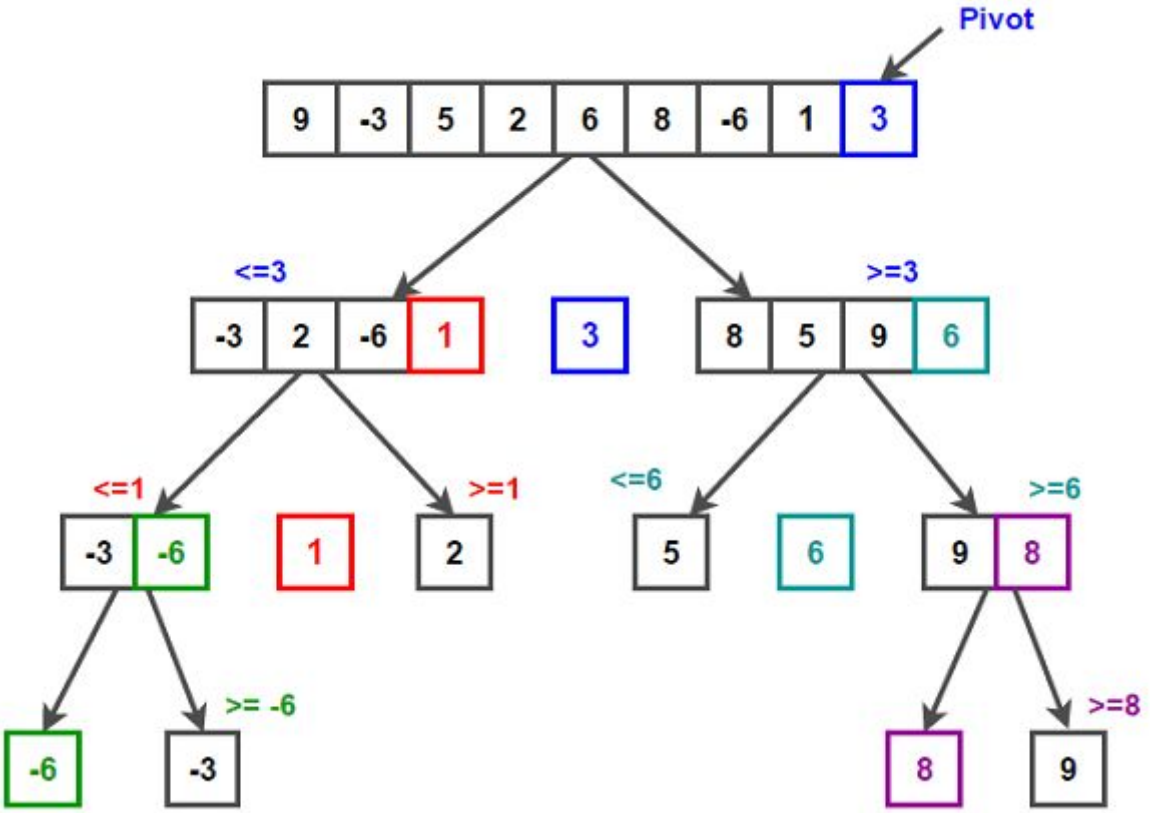
4.7.1 Descrição

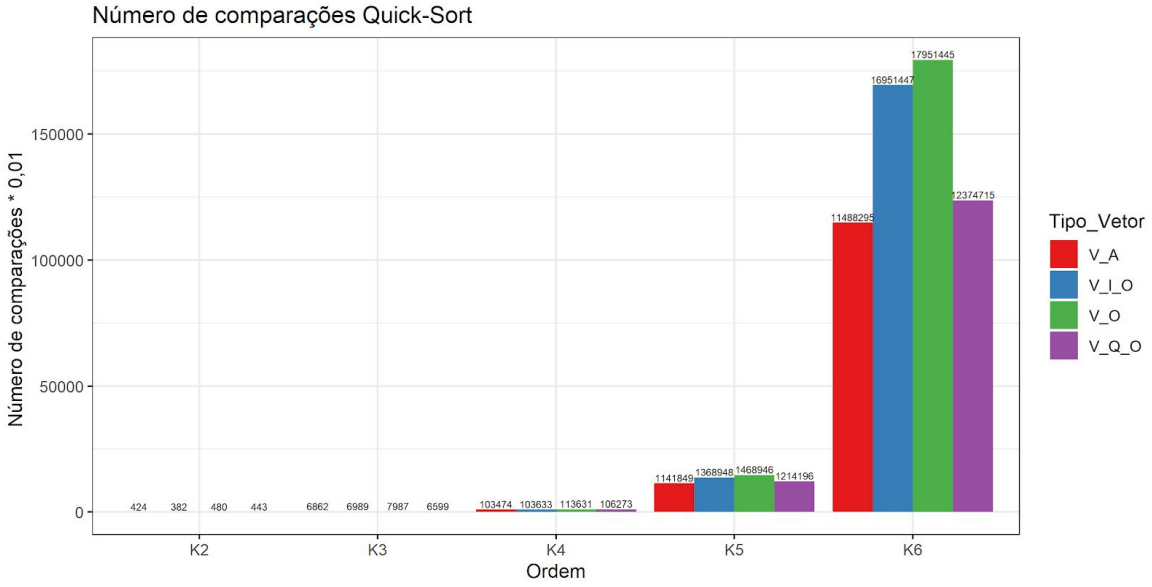
O algoritmo de ordenação quicksort foi criado por *C. A. R. Hoare* em 1960. Considerando um pivô aleatório, o algoritmo divide o vetor que recebe dois vetores menores, através do pivô, todos os valores à sua direita devem ser maiores que o mesmo e à esquerda devem ser menores. De forma recursiva, dentro desses vetores serão escolhidos outros pivôs e ocorrerá o mesmo processo, até que o vetor esteja totalmente ordenado.

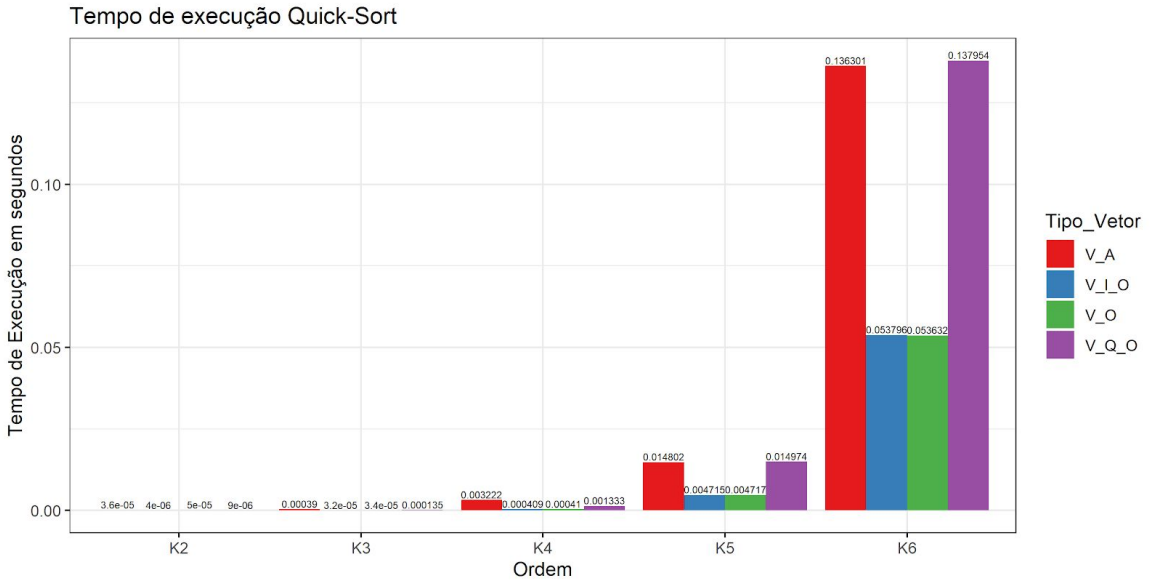
4.7.2 Características

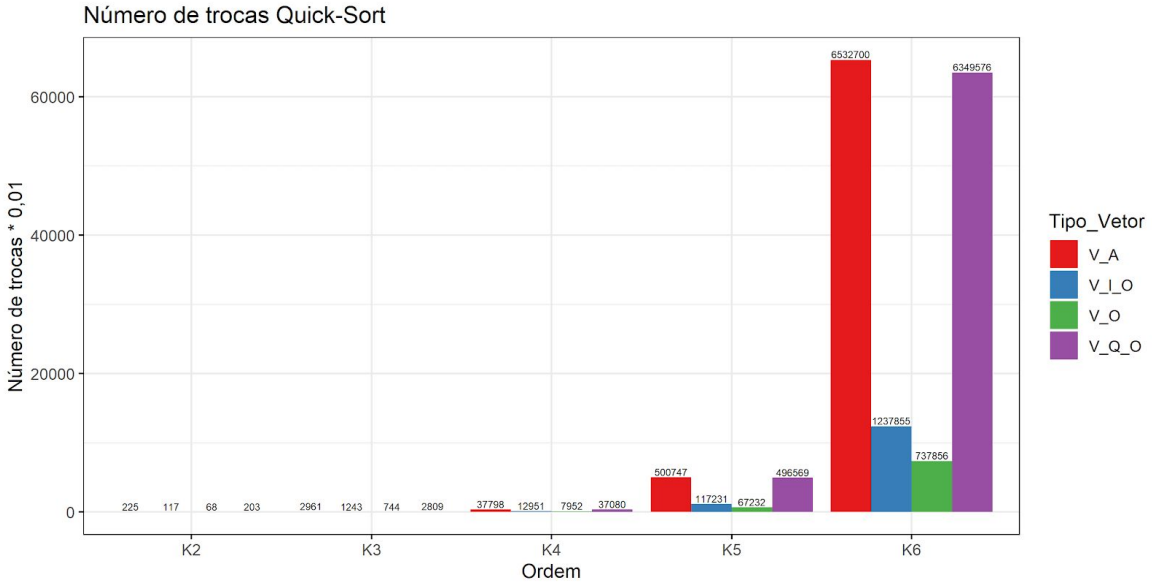
- Implementação não intuitiva e necessita de detalhes, no entanto, dificulta sua manutenção;
- Complexidade de espaço O(n);
- Complexidade de comparações no pior caso: $O(n^2)$;
- Algoritmo não estável, ou seja, altera a ordem relativa dos elementos com valores iguais.

4.7.3 Gráficos e Explicações ilustrativas




Número de comparações Quick-Sort
Número de comparações * 0,01
150000
100000
50000
0
424 382 480 443 6862 6989 7987 6599 103474 103633 113631 106273 1141849 1368948 1468946 1214196 11488295 16951447 17951445 12374715
K2 K3 K4 K5 K6
Ordem
Tipo_Vetor
V_A
V_I_O
V_O
V_Q_O


Tempo de execução Quick-Sort
Tempo de Execução em segundos
0.10
0.05
0.00
3.6e-05 4e-06 5e-05 9e-06 0.00039 3.2e-05 3.4e-05 0.000135 0.003222 0.000409 0.00041 0.001333 0.014802 0.004715 0.004717 0.014974 0.136301 0.053796 0.053632 0.137954
K2 K3 K4 K5 K6
Ordem
Tipo_Vetor
V_A
V_I_O
V_O
V_Q_O


Número de trocas Quick-Sort
Número de trocas * 0,01
60000
40000
20000
0
225 117 68 203 2961 1243 744 2809 37798 12951 7952 37080 500747 117231 67232 496569 6532700 1237855 737856 6349576
K2 K3 K4 K5 K6
Ordem
Tipo_Vetor
V_A
V_I_O
V_O
V_Q_O

4.7.4 Implementação em C

```
void quicksort(int values[], int began, int end)
{
     int i, j, pivo, aux;
     i = began;
     j = end-1;
     pivo = values[(began + end) / 2];
     while(i <= j)
     {
          while(values[i] < pivo && i < end)
          {
               i++;
          }
          while(values[j] > pivo && j > began)
          {
               j--;
          }
          if(i <= j)
          {
               aux = values[i];
               values[i] = values[j];
               values[j] = aux;
               i++;
               j--;
          }
     }
     if(j > began)
          quicksort(values, began, j+1);
     if(i < end)
          quicksort(values, i, end);
}
```

4.8 MergeSort

4.8.1 Descrição

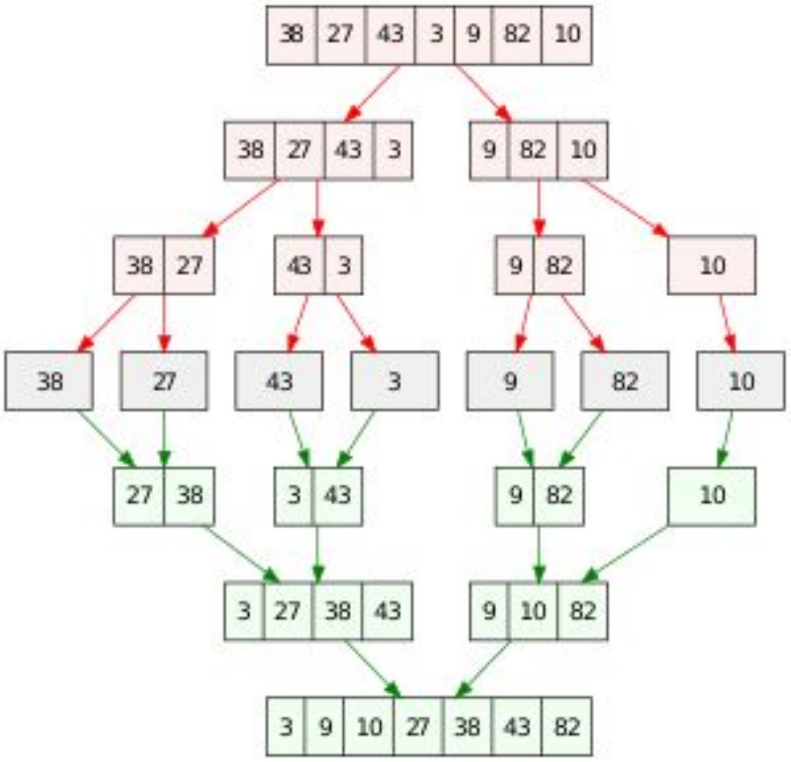
Outro algoritmo recursivo é o MergeSort, criado em 1945 pelo matemático americano *John Von Neumann*que. O algoritmo divide o vetor em dois diversas vezes, até que fiquem dois (ou um caso o número de elementos seja ímpar) elementos. Eles serão ordenados e então se juntarão a outra dupla, de forma que esse quarteto também será ordenado. Isso acontecerá recursivamente até que se juntem as duas metades do vetor e essas sejam ordenadas como o vetor final.

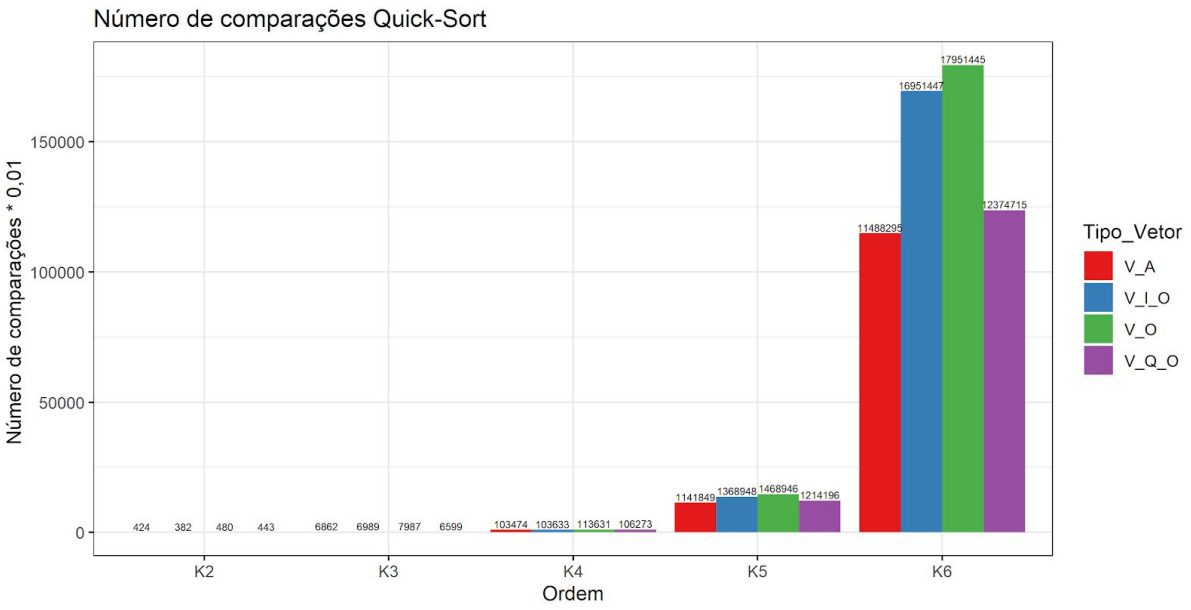
4.8.2 Características

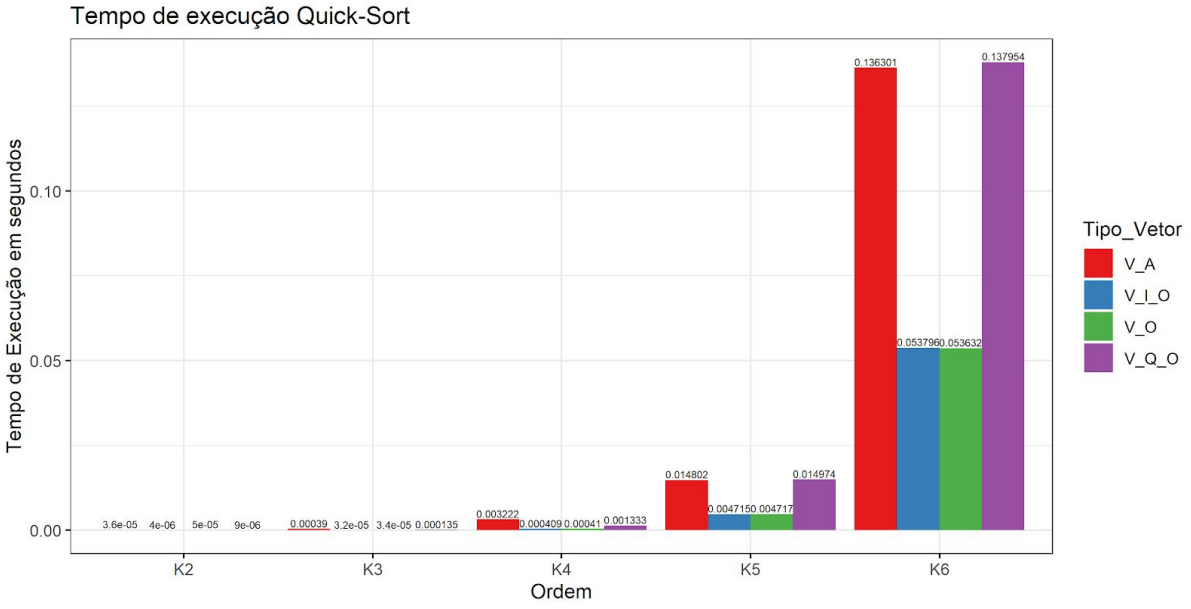
- Complexidade de espaço: O(n);

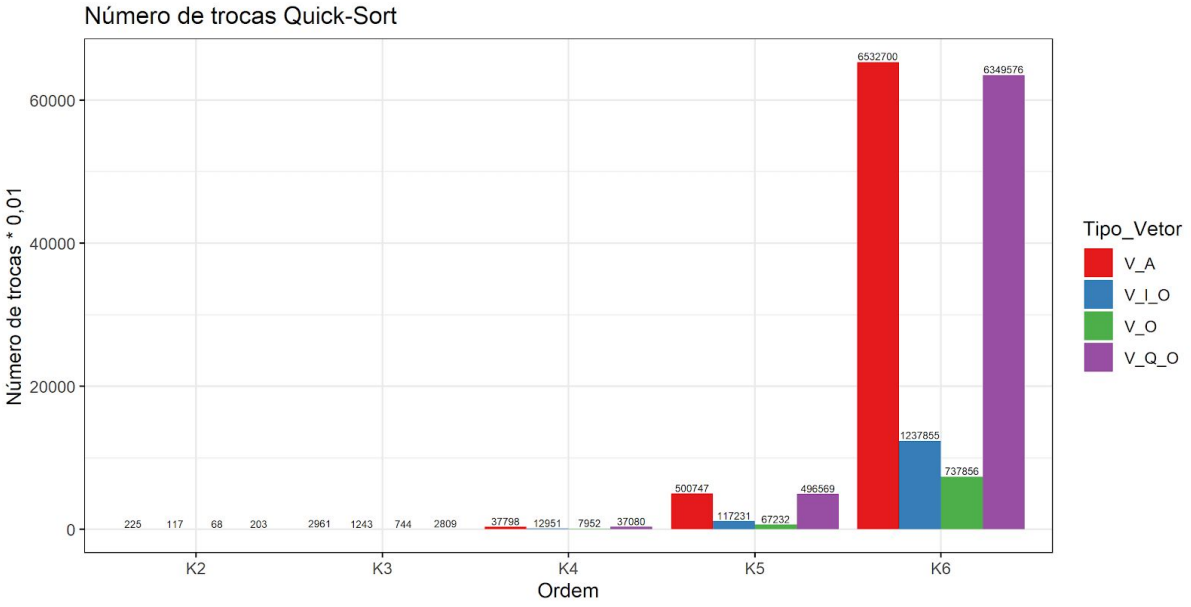
- Complexidade de tempo em todos os casos: O(n log n);
- Complexidade de troca: O(n log n);
- Algoritmo estável, ou seja, não altera a ordem relativa dos elementos com valores iguais.

4.8.3 Gráficos e Explicações ilustrativas




Número de comparações Quick-Sort
Número de comparações * 0,01
150000
100000
50000
0
424
382
480
443
6862
6989
7987
6599
103474
103633
113631
106273
1141849
1368948
1468946
1214196
11488295
16951447
17951445
12374715
K2
K3
K4
K5
K6
Ordem
Tipo_Vetor
V_A
V_I_O
V_O
V_Q_O


Tempo de execução Quick-Sort
Tempo de Execução em segundos
0.10
0.05
0.00
3.6e-05
4e-06
5e-05
9e-06
0.00039
3.2e-05
3.4e-05
0.000135
0.003222
0.000409
0.00041
0.001333
0.014802
0.0047150.004717
0.014974
0.136301
0.0537960.053632
0.137954
K2
K3
K4
K5
K6
Ordem
Tipo_Vetor
V_A
V_I_O
V_O
V_Q_O



4.8.4 Implementação em C

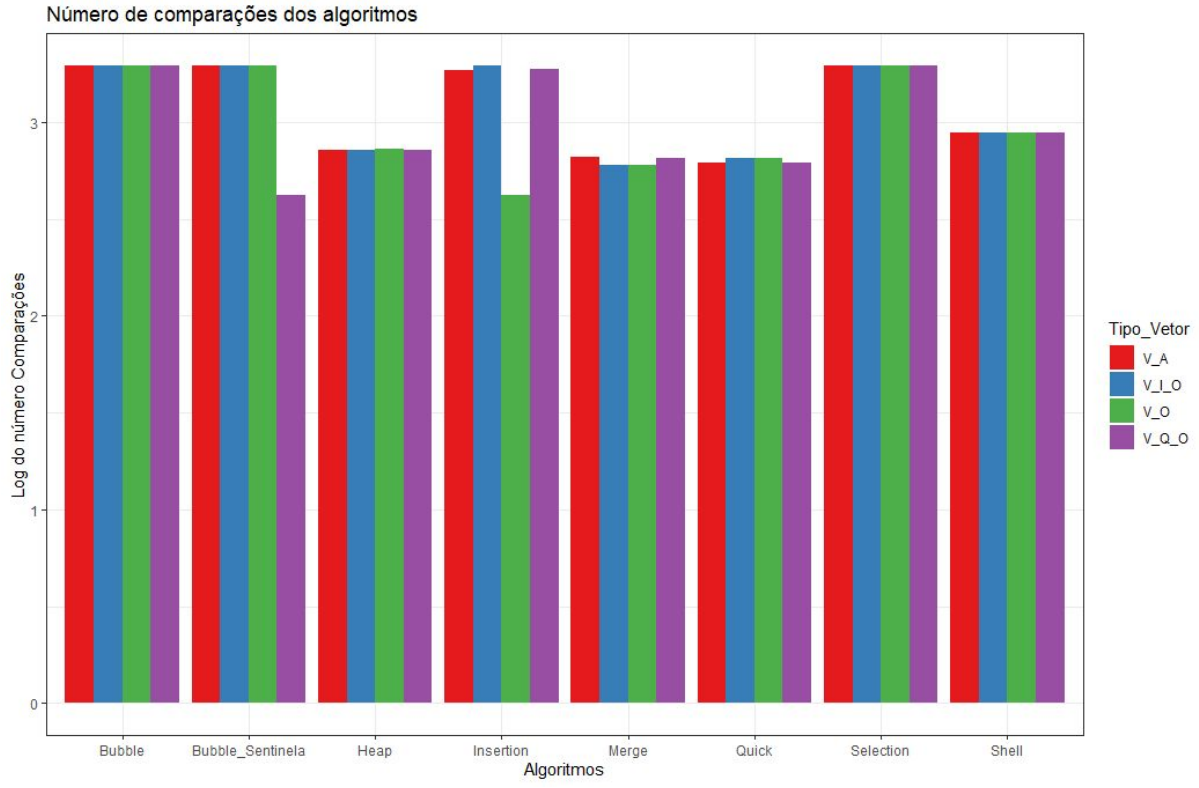
```
    void merge(int vetor[], int comeco, int meio, int
fim) {
        int com1 = comeco, com2 = meio+1, comAux = 0,
tam = fim-comeco+1;
        int *vetAux;
        vetAux = (int*)malloc(tam * sizeof(int));

        while(com1 <= meio && com2 <= fim){
            if(vetor[com1] < vetor[com2]) {
                vetAux[comAux] = vetor[com1];
                com1++;
            } else {
                vetAux[comAux] = vetor[com2];
                com2++;
            }
            comAux++;
        }

        while(com1 <= meio){  //Caso ainda haja
elementos na primeira metade
            vetAux[comAux] = vetor[com1];
            comAux++;
            com1++;
        }
```

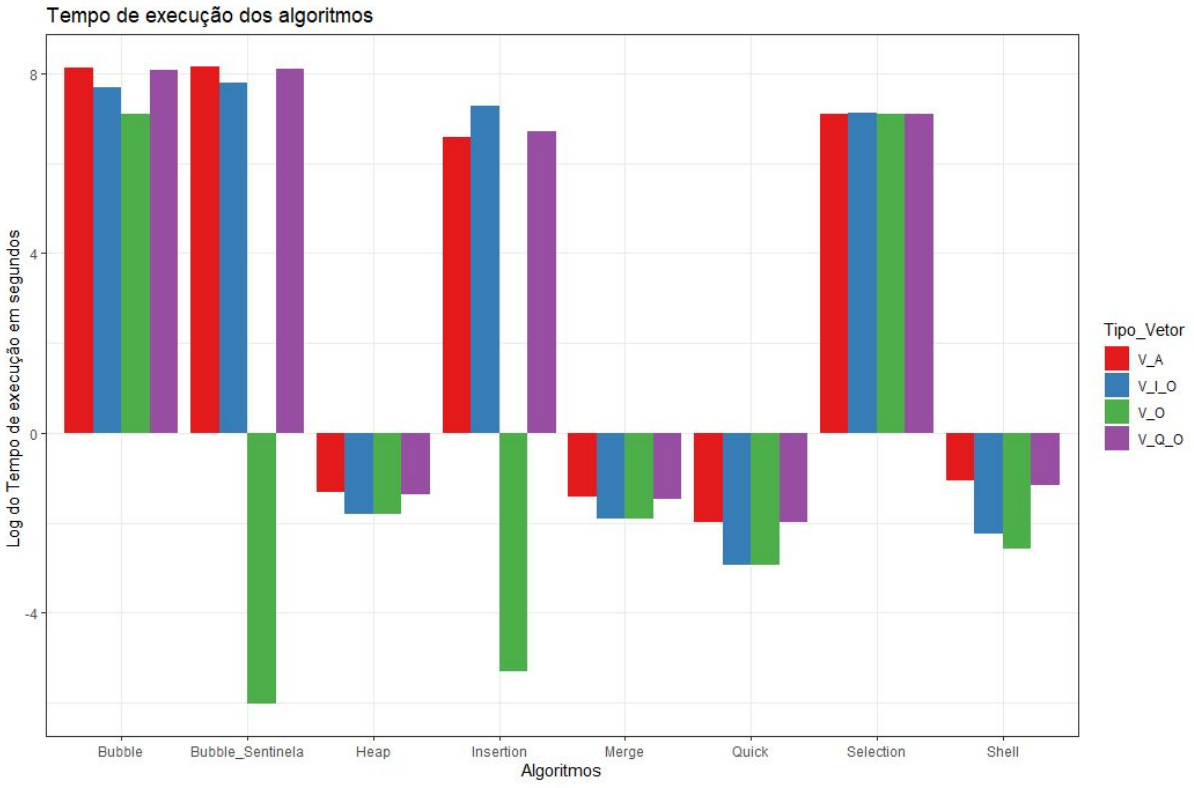
```
            while(com2 <= fim) {    //Caso ainda haja
elementos na segunda metade
                vetAux[comAux] = vetor[com2];
                comAux++;
                com2++;
            }

            for(comAux = comeco; comAux <= fim; comAux++){
//Move os elementos de volta para o vetor original
                vetor[comAux] = vetAux[comAux-comeco];
            }
            free(vetAux);
        }

        void mergeSort(int vetor[], int comeco, int fim){
            if (comeco < fim) {
                int meio = (fim+comeco)/2;
                mergeSort(vetor, comeco, meio);
                mergeSort(vetor, meio+1, fim);
                merge(vetor, comeco, meio, fim);
            }
        }
```

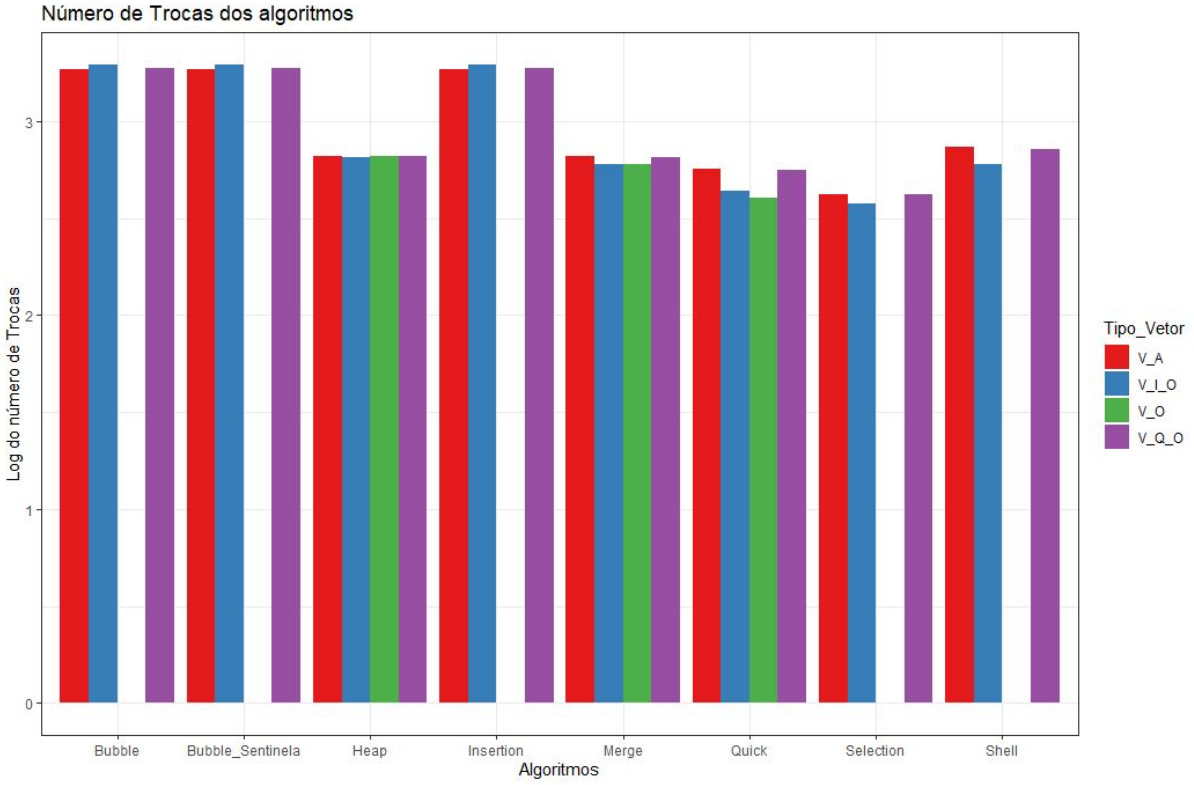
5. Resultados e Discussões

5.1 Gráficos

Para K = 6, usando Log a fim de melhor visualização:







5.2 Melhores casos para cada algoritmo

- Bubble tem melhor desempenho com vetor ordenado.
- Bubble com sentinela apresenta maior eficiência com vetor ordenado.

- Selection tem melhor rendimento com vetor ordenado, entretanto, o vetor quase ordenado não deixa uma diferença notável.
- Insertion é mais prudente em casos com o vetor ordenado.
- Merge: tanto o vetor ordenado quanto inversamente ordenado extraíram o melhor desempenho do algoritmo.
- Heap: tanto o vetor ordenado quanto inversamente ordenado extraíram o melhor desempenho do algoritmo.
- Quick: tanto o vetor ordenado quanto inversamente ordenado obtiveram o melhor desempenho do algoritmo.
- Shell: o melhor caso é o com vetor ordenado.

5.3 Melhores escolhas de algoritmo para K=6

- O algoritmo com melhor desempenho foi o Bubble sort com sentinela.
- O algoritmo com melhor desempenho foi o QuickSort.
- O algoritmo com melhor desempenho foi o QuickSort.
- O algoritmo com melhor desempenho foi o QuickSort.

5.4 Algoritmo Vencedor

O algoritmo vencedor, tendo em vista principalmente os vetores com 10^6 elementos, foi o QuickSort, o qual apresentou o menor tempo de execução em grande parte dos casos, deixando-o em destaque visto que esse é um dos mais fortes objetivos.

5.4.1 Tempo

Tendo em vista o tempo como parâmetro, o algoritmo em questão sai na frente de todos os outros nos casos de vetor inversamente ordenado, quase ordenado e aleatório. Somente no caso do vetor ordenado não consegue a primeira posição, mas ainda sim fica em terceiro, demonstrando sua eficiência sempre presente.

5.4.2 Trocas

Com um constante baixo número de trocas, o quick só não possui uma das menores médias pois algoritmos como o Bubble e Selection não realizam troca alguma quando o vetor está ordenado. Entretanto nos outros casos fica sempre com o segundo menor número de trocas, perdendo novamente para o Selection.

5.4.3 Comparações

Em relação ao log do número comparações, o Quick e o Merge apresentam as melhores médias, as quais possuem uma diferença desprezível, sendo assim vale dividir o posto de melhor média para ambos, mostrando novamente um motivo pelo qual o Quick é o melhor algoritmo.

6. Considerações Finais

Métodos de simples implementação são adequados para pequenas listas e requerem $O(n^2)$ em tempo de execução.

Métodos mais eficientes são adequados para arquivos maiores e requerem O(n log n) comparações e tempo, para isso, são mais complexas nos detalhes.

Portanto, não existe um método de ordenação considerado universalmente melhor que todos os outros. É necessário analisar o problema e, com base nas características dos dados, decidir qual o melhor método deverá ser aplicado. Averiguando sempre as complexidades de tempo e espaço dos algoritmos.

Referências

https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao/amp/#referrer=https://www.google.com Acesso em: <14/set/2019>

https://pt.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort#targetText=Assim%20como%20o%20merge%20sort,como%20o%20merge%20sort%20faz.&targetText=Quicksort%20funciona%20localmente. Acesso em: <14/set/2019>

https://pt.wikipedia.org/wiki/Bubble_sort Acesso em: <14/set/2019>

https://pt.wikipedia.org/wiki/Selection_sort Acesso em: <14/set/2019>

https://pt.wikipedia.org/wiki/Insertion_sort Acesso em: <14/set/2019>

https://pt.wikipedia.org/wiki/Heapsort Acesso em: <14/set/2019>

https://pt.wikipedia.org/wiki/Ordena%C3%A7%C3%A3o_est%C3%A1vel Acesso em: <14/set/2019>

https://pt.wikipedia.org/wiki/Shell_sort#targetText=Criado%20por%20Donald%20Shell%20em,do%20m%C3%A9todo%20de%20inser%C3%A7%C3%A3o%20direta.&targetText=Basicamente%20o%20algoritmo%20passa%20v%C3%A1rias,o%20grupo%20maior%20em%20menores. Acesso em: <14/set/2019>

https://www.youtube.com/watch?v=M3bS6w1R434 Acesso em: <14/set/2019>

https://pt.wikipedia.org/wiki/Ordenação_estável Acesso em: <14/set/2019>

http://www.facom.ufu.br/~backes/gsi011/Aula06-Ordenacao.pdf Acesso em: <14/set/2019>

https://pt.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation Acesso em: <14/set/2019>

https://pt.stackoverflow.com/questions/325088/ordenação-estável-vs-instável/325124 Acesso em: <14/set/2019>

https://pt.wikipedia.org/wiki/Ordena%C3%A7%C3%A3o_est%C3%A1vel Acesso em: <14/set/2019>

https://www.embarcados.com.br/algoritmos-de-ordenacao-bubble-sort/. Acesso em: <14/set/2019>

http://www.decom.ufop.br/reinaldo/site_media/uploads/2013-02-bcc202/aula_12_-_bubblesort-selectionsort-insertionsort_(v1).pdf. Acesso em: <14/set/2019>

https://www.hackerearth.com/pt-br/practice/algorithms/sorting/insertion-sort/tutorial/ Acesso em: <15/set/2019>

https://www.hackerearth.com/pt-br/practice/algorithms/sorting/selection-sort/tutorial/ Acesso em: <15/set/2019>

https://www.programiz.com/dsa/bubble-sort Acesso em: <15/set/2019>

https://www.hackerearth.com/pt-br/practice/algorithms/sorting/heap-sort/tutorial/ Acesso em: <15/set/2019>

https://upload.wikimedia.org/wikipedia/commons/thumb/e/e6/Merge_sort_algorithm_diagram.svg/300px-Merge_sort_algorithm_diagram.svg.png Acesso em: <15/set/2019>

https://www.techiedelight.com/quicksort/ Acesso em: <15/set/2019>

https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheShellSort.html Acesso em: <15/set/2019>

https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao/.Acesso em: <15/set/2019>