UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

AVALIAÇÃO UNIDADE II
RELATÓRIO SOBRE ALGORITMOS DE ORDENAÇÃO

Docente: Eduardo de Lucena Falcão
Discente: Tiago Felipe de Souza
Disciplina: Algoritmos e Estruturas de Dados I

Natal/RN

Iniciaremos nosso relatório com algumas explicações sobre ordenação, para que fique mais clara a utilização e a importância de algoritmos mais eficientes em questões de desempenho. Ordenação vem do verbo ordenar, que significa: arrumar, dispor, colocar em ordem, ou seja, dispor harmoniosamente determinados elementos com alguma finalidade como a de facilitar alguma provável busca como por exemplo, saber qual o maior ou menor elemento em uma sequência ou coleção.

Agora, pense como seria buscar um número em um catálogo telefônico se os nomes não estivessem listados em ordem alfabética? Pois é, seria muito complicado. Então, podemos utilizar uma estratégia para generalizar e facilitar nossos trabalhos, em que, primeiro coloquemos os elementos ou objetos em ordem, e depois decidimos a operação ou o que fazer com eles. Na computação existe uma série de algoritmos que utilizam vários métodos e diferentes técnicas para ordenar um conjunto de dados e são chamados algoritmos de ordenação.

Os métodos de ordenação de algoritmos se classificam em:

- Ordenação Interna (In-Place): Onde todos os elementos a serem ordenados estão dentro do mesmo array, limitando e organizando no mesmo espaço da memória os elementos.
- Ordenação Externa (Out-of-Place): Onde os elementos que serão ordenados precisam de um array externo ao principal para auxiliar no método de ordenação do array principal, alocando mais espaço na memória.

Feitas as considerações iniciais, vamos iniciar a resolução da avaliação II da disciplina de Algoritmos e Estruturas de Dados I.

Considere os seguintes vetores:

- a = [3, 6, 2, 5, 4, 3, 7, 1]
- b = [7, 6, 5, 4, 3, 3, 2, 1]

- Ilustre, em detalhes, o funcionamento dos seguintes algoritmos com os seguintes vetores: **(4.0)**
    a. BubbleSort (melhor versão) **com o vetor a**

```
void bubbleSort(int* v, int n){
    for(int varredura = 0; varredura < n-1; varredura++){
        bool trocou = false;
        for(int i = 0; i < n-varredura-1; i++){
            if(v[i]>v[i+1]){
                int temp = v[i];
                v[i] = v[i+1];
                v[i+1] = temp;
                trocou = true;
            }
        }
        if(trocou==false)
            return;
    }
}
```

O algoritmo de ordenação BubbleSort propõe que testando 2 a 2 elementos a partir do início do array (posições = 0 e 1), descobrir se o elemento da posição anterior é maior e assim, comparando as posições, trocar estas posições desses elementos e dessa forma fazer com que o maior elemento do array esteja na última posição do array na primeira varredura, “Flutuando” o maior elemento até o última posição do array, o segundo maior elemento do array esteja na penúltima posição na segunda verredura, e assim sucessivamente até que o array esteja totalmente ordenado.

Considerações: Aplicando o algoritmo, vemos alguns detalhes que podem ser melhorados quanto a otimização.

- Não é necessário que realizemos todas as varreduras possíveis para o tamanho do array, pois na última varredura do algoritmo o array já estará ordenado, visto que o elemento de posição = 1 será maior que o de posição = 0 e não será necessário executar esta operação, economizando processamento e memória.
- Não é necessário que sempre temos que ir até o final no array para testar quem é maior ou não, pois como já dito, toda vez que executamos o algoritmo nós descobrimos um elemento que será o maior e dessa forma consumirá mais memória e processamento,

então, em cada varredura nós descontamos uma posição na verificação de quem é o elemento maior ou não.

- Na melhor possibilidade de todas, o algoritmo deverá ordenar um array já ordenado, dessa forma nós não precisamos ordenar um array já ordenado, logo, se após a primeira varredura do array, for constatado que não houve nenhuma alteração no teste da ordenação, significa que o array já está ordenado e não precisa executar mais o algoritmo.

Por conseguinte, temos:

Vetor a = [3, 6, 2, 5, 4, 3, 7, 1]

Varredura	Teste	Array (comparação)	if (v[i] > v[i+1])
0	1	**3, 6,** 2, 5, 4, 3, 7, 1	3 < 6, não troca
	2	3, **6, 2,** 5, 4, 3, 7, 1	6 > 2, troca (int temp = 6 ; v[i] = 2 ; v[i+1] = 6)
	3	3, 2, **6, 5,** 4, 3, 7, 1	6 > 5, troca
	4	3, 2, 5, **6, 4,** 3, 7, 1	6 > 4, troca
	5	3, 2, 5, 4, **6, 3,** 7, 1	6 > 3, troca
	6	3, 2, 5, 4, 3, **6, 7,** 1	6 < 7, não troca
	7	3, 2, 5, 4, 3, 6, **7, 1**	7 > 1, troca
	****	3, 2, 5, 4, 3, 6, 1, **7**	7 é o maior

Varredura	Teste	Array (comparação)	if (v[i] > v[i+1])
1	1	**3, 2,** 5, 4, 3, 6, 1, **7**	3 > 2, troca
	2	2, **3, 5,** 4, 3, 6, 1, **7**	3 < 5, não troca
	3	2, 3, **5, 4,** 3, 6, 1, **7**	5 > 4, troca
	4	2, 3, 4, **5, 3,** 6, 1, **7**	5 > 3, troca
	5	2, 3, 4, 3, **5, 6,** 1, **7**	5 < 6, não troca
	6	2, 3, 4, 3, 5, **6, 1,** 7	6 > 1, troca
	****	2, 3, 4, 3, 5, 1, **6, 7**	6 e 7 ordenados

Varredura	Teste	Array (comparação)	if (v[i] > v[i+1])
2	1	**2, 3,** 4, 3, 5, 1, **6, 7**	2 < 3, não troca
	2	2, **3, 4,** 3, 5, 1, **6, 7**	3 < 4, não troca
	3	2, 3, **4, 3,** 5, 1, **6, 7**	4 > 3, troca
	4	2, 3, 3, **4, 5,** 1, **6, 7**	4 < 5, não troca
	5	2, 3, 3, 4, **5, 1,** 6, **7**	5 > 1, troca
	****	2, 3, 3, 4, 1, **5, 6, 7**	5, 6 e 7 ordenados

Varredura	Teste	Array (comparação)	if (v[i] > v[i+1])
3	1	**2, 3,** 3, 4, 1**, 5, 6, 7**	2 < 3, não troca
	2	2, **3, 3,** 4, 1**, 5, 6, 7**	3 = 3, não troca
	3	2, 3, **3, 4,** 1**, 5, 6, 7**	3 < 4, não troca
	4	2, 3, 3, **4, 1, 5, 6, 7**	4 > 1, troca
	****	2, 3, 3, 1**, 4, 5, 6, 7**	4, 5, 6, 7 ordenados

Varredura	Teste	Array (comparação)	if (v[i] > v[i+1])
4	1	**2, 3,** 3, 1**, 4, 5, 6, 7**	2 < 3, não troca
	2	2, **3, 3,** 1**, 4, 5, 6, 7**	3 = 3, não troca
	3	2, 3, **3, 1**, 4, 5, 6, 7**	3 > 1, troca
	****	2, 3, 1**, 3, 4, 5, 6, 7**	3, 4, 5, 6, 7 ordenados

Varredura	Teste	Array (comparação)	if (v[i] > v[i+1])
5	1	**2, 3,** 1**, 3, 4, 5, 6, 7**	2 < 3, não troca
	2	2, **3, 1**, 3, 4, 5, 6, 7**	3 > 1, troca
	****	2, 1**, 3, 3, 4, 5, 6, 7**	3, 3, 4, 5, 6, 7 ordenados

Varredura	Teste	Array (comparação)	if (v[i] > v[i+1])
6	1	**2, 1**, 3, 3, 4, 5, 6, 7**	2 > 1, troca
	****	**1, 2, 3, 3, 4, 5, 6, 7**	Vetor a = [1, 2, 3, 3, 4, 5, 6, 7] ordenado!

b. InsertionSort (in-place, melhor versão) **com o vetor b**

```
void insertionSortIP(int* v, int tamanho) {
    for (int i = 1; i < tamanho; i++) {
        for (int j = i; j > 0 && v[j - 1] > v[j]; j--) {
            int temp = v[j - 1];
            v[j - 1] = v[j];
            v[j] = temp;
        }
    }
}
```

O algoritmo de ordenação InsertioSort In-Place, diferentemente da versão Out-of-Place que temos que utilizar dois arrays, um com os elementos do array original e outro que colocaremos em ordem, elemento a elemento, propõe que devemos abstrair a visão do array original em dois subarrays menores, um estando ordenado e outro não estando ordenado, começando pelo array de um elemento, pois o array com um elemento é considerado um array ordenado. Bem, dessa forma, nós poderemos ir

fazendo comparações entre os subarrays internos do array original para poder inserir ordenadamente alterando as posições do vetor original, dependendo se o valor que está mais à esquerda é menor ou maior, e se for maior, troca.

Considerações: Aplicando o algoritmo, vemos alguns detalhes que podem ser melhorados quanto a otimização.

- Esta versão do InsertionSort é uma versão que "parece" um BubbleSort invertido, pois nós vamos fazendo testes com o elemento que está mais a esquerda e caso o elemento anterior seja maior do que ele, será feita a troca de posições, assim, abre-se o espaço necessário para a inserção iterativa até que não tenha mais comparações para se fazer, e então, o array estará ordenado.
- Na melhor possibilidade de todas, o algoritmo deverá ordenar um array já ordenado, dessa forma nós não precisamos ordenar um array já ordenado, logo, o for interno nunca será executado.

Por conseguinte, temos:

Vetor b = [7, 6, 5, 4, 3, 3, 2, 1]

Teste	Array (detalhamento)	i = 1; i < tamanho ; i++ j = i; j > 0 && v[j-1] > v[j] ; j--
1	**7, 6,** 5, 4, 3, 3, 2, 1	[7] ordenado, [6, 5, 4, 3, 3, 2, 1] desordenado, 7 > 6, troca (int temp = 7 ; v[j-1] = 6 ; v[j] = 7)
2	**6, 7, 5,** 4, 3, 3, 2, 1	[6, 7] ordenado, [5, 4, 3, 3, 2, 1] desordenado, 7 > 5, troca
3	**6, 5, 7,** 4, 3, 3, 2, 1	6 > 5, troca
4	**5, 6, 7, 4,** 3, 3, 2, 1	[5, 6, 7] ordenado, [4, 3, 3, 2, 1] desordenado, 7 > 4, troca
5	**5, 6, 4, 7,** 3, 3, 2, 1	6 > 4, troca
6	**5, 4, 6, 7,** 3, 3, 2, 1	5 > 4, troca
7	**4, 5, 6, 7, 3,** 3, 2, 1	[4, 5, 6, 7] ordenado, [3, 3, 2, 1] desordenado, 7 > 3, troca
8	**4, 5, 6, 3, 7,** 3, 2, 1	6 > 3, troca
9	**4, 5, 3, 6, 7,** 3, 2, 1	5 > 3, troca
10	**4, 3, 5, 6, 7,** 3, 2, 1	4 > 3, troca
11	**3, 4, 5, 6, 7, 3,** 2, 1	[3, 4, 5, 6, 7] ordenado, [3, 2, 1] desordenado, 7 > 3, troca
12	**3, 4, 5, 6, 3, 7,** 2, 1	6 > 3, troca
13	**3, 4, 5, 3, 6, 7,** 2, 1	5 > 3, troca
14	**3, 4, 3, 5, 6, 7,** 2, 1	4 > 3, troca
15	**3, 3, 4, 5, 6, 7,** 2, 1	3 > 3, não troca

16	**3, 3, 4, 5, 6, 7, 2,** 1	[3, 3, 4, 5, 6, 7] ordenado, [2, 1] desordenado, 7 > 2, troca
17	**3, 3, 4, 5, 6, 2, 7,** 1	6 > 2, troca
18	**3, 3, 4, 5, 2, 6, 7,** 1	5 > 2, troca
19	**3, 3, 4, 2, 5, 6, 7,** 1	4 > 2, troca
20	**3, 3, 2, 4, 5, 6, 7,** 1	3 > 2, troca
21	**3, 2, 3, 4, 5, 6, 7,** 1	3 > 2, troca
22	**2, 3, 3, 4, 5, 6, 7,** 1	[2, 3, 3, 4, 5, 6, 7] ordenado, [1] desordenado, 7 > 1, troca
23	**2, 3, 3, 4, 5, 6, 1, 7**	6 > 1, troca
24	**2, 3, 3, 4, 5, 1, 6, 7**	5 > 1, troca
25	**2, 3, 3, 4, 1, 5, 6, 7**	4 > 1, troca
26	**2, 3, 3, 1, 4, 5, 6, 7**	3 > 1, troca
27	**2, 3, 1, 3, 4, 5, 6, 7**	3 > 1, troca
28	**2, 1, 3, 3, 4, 5, 6, 7**	2 > 1, troca
29	**1, 2, 3, 3, 4, 5, 6, 7**	Vetor b = [1, 2, 3, 3, 4, 5, 6, 7] ordenado!

c. MergeSort **com o vetor a**

```
void mergeSort(int* v, int tamV){
    if(tamV>1){
        //primeiro quebramos o vetor em 2 vetores menores
        int meio = tamV/2;

        int tamV1 = meio;
        int* v1 = (int*)malloc(tamV1*sizeof(int));
        for(int i = 0; i < meio; i++){
            v1[i] = v[i];
        }

        int tamV2 = tamV-meio;
        int* v2 = (int*)malloc(tamV2*sizeof(int));
        for(int i = meio; i < tamV; i++){
            v2[i-meio] = v[i];
        }

        mergeSort(v1,tamV1);
        mergeSort(v2,tamV2);
        merge(v,tamV,v1,tamV1,v2,tamV2);

        free(v1);
        free(v2);
    }
}
```

```
void merge(int* v, int tamV, int* e, int tamE, int* d, int tamD){
    int indexV = 0;
    int indexE = 0;
    int indexD = 0;
    while(indexE < tamE && indexD < tamD){
        if(e[indexE] <= d[indexD]){
            v[indexV] = e[indexE];
            indexE++;
        } else{
            v[indexV] = d[indexD];
            indexD++;
        }
        indexV++;
    }
    //ainda poderíamos ter elementos no vetor E que não foram copiados para V
    while(indexE < tamE){
        v[indexV] = e[indexE];
        indexE++;
        indexV++;
    }
    //ainda poderíamos ter elementos no vetor D que não foram copiados para V
    while(indexD < tamD){
        v[indexV] = d[indexD];
        indexD++;
        indexV++;
    }
}
```

O algoritmo de ordenação MergeSort propõe que o array seja dividido em tamanho menores, recursivamente, até que se chegue no menor array que se possa ter ordenado. Já comentamos anteriormente, que o vetor que possui apenas uma posição é um vetor que já é ordenado e dessa forma, independentemente do tamanho do array, o algoritmo divide todo array em vetores de uma posição e depois combina todos os arrays de uma posição só que nas posições já ordenando-os, ou seja, quando todo o vetor for construído novamente, ele já estará ordenado com os vetores “unitários” já nas suas posições corretas.

Considerações: Aplicando o algoritmo, vemos alguns detalhes que:

- O MergeSort é um algoritmo de divisão e conquista, ou seja, vamos dividir todo o array até que ele não possa mais ser dividido e em seguida, vamos combiná-lo novamente, de forma ordenada, para conquistar o vetor totalmente ordenado.
- Como já falado, esse algoritmo é divido em duas partes: 1. Divisão e 2. Conquista. A etapa da conquista (merge) consiste em combinar todos os arrays unitários e comparar os primeiros elementos de cada

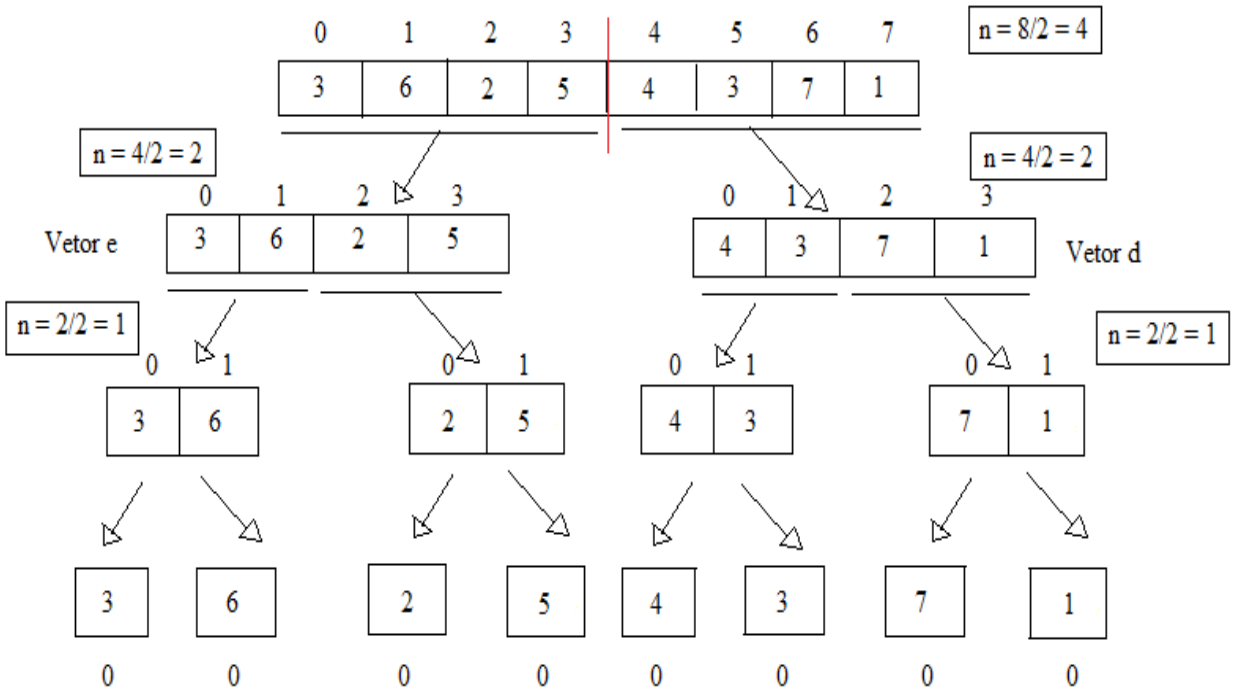
"subarray" e ir perguntando quem é menor ou igual entre os "subarrays" esquerdo e direito de forma a colocar no vetor original "a", a combinação ordenada dos arrays conquistados e transformá-lo no vetor totalmente ordenado.

- Já para a etapa da divisão nós vamos fazer divisões recursivas no array original "a" até que não se tenha mais como dividí-lo, transformando-o em vários n vetores de tamanho 1, e dessa forma nós podemos utilizar o algoritmo do merge para poder unir todos os n vetores unitários.

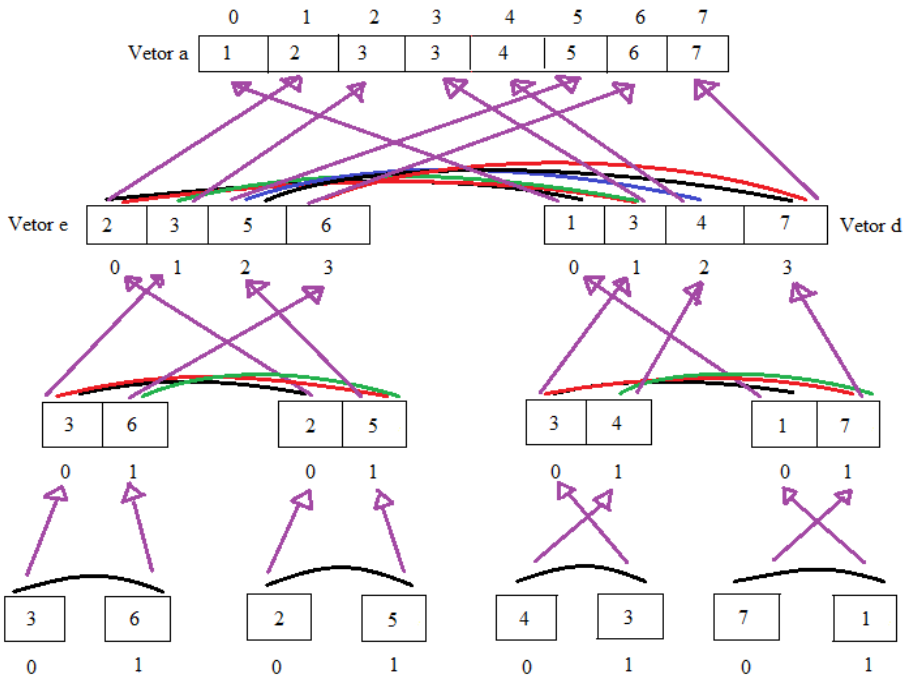
Por conseguinte, temos:

Vetor a = [3, 6, 2, 5, 4, 3, 7, 1]

*Divisão



*Conquista



Algoritmo do merge

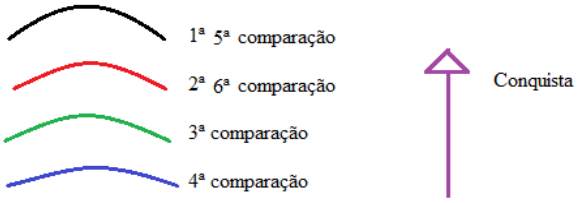
Lado Vetor e
* 2 <= 1, não, 1 assume a posição = 0 do vetor a
* 2 <= 3, sim, 2 assume a posição = 1 do vetor a
* 3 <= 3, sim, 3 assume a posição = 2 do vetor a
* 5 <= 3, não, 3 assume a posição = 3 do vetor a
* 5 <= 4, não, 4 assume a posição = 4 do vetor a
* 5 <= 7, sim, 5 assume a posição = 5 do vetor a
* 6 <= 7, sim, 6 assume a posição = 6 do vetor a e copia o resto

Lado Vetor e
* 3 <= 2, não, 2 assume a posição = 0 acima
* 3 <= 5, sim, 3 assume a posição = 1 acima
* 6 <= 5, não, 5 assuma a posição = 2 acima e copia o resto

Lado Vetor d
* 3 <= 1, não, 1 assume a posição = 0 acima
* 3 <= 7, sim, 3 assume a posição = 1 acima
* 4 <= 7, sim, 4 assume a posição = 2 acima e copia o resto

Lado Vetor e
* 3 <= 6, sim, 3 assume a posição = 0 acima e copia o resto
* 2 <= 5, sim, 2 assume a posição = 0 acima e copia o resto

Lado Vetor d
* 4 <= 3, não, 3 assume a posição = 0 acima e copia o resto
* 7 <= 1, não, 1 assume a posição = 0 acima e copia o resto



- Os algoritmos Merge e MergeSort executarão toda a divisão e conquista do lado esquerdo no vetor **e,** e então só após, que executarão a divisão e conquista para o lado direito no vetor **d**.

d. QuickSort (s/ randomização de pivô) **com o vetor b**

```
void quickSort(int* v, int ini, int fim){
    if(fim>ini){
        int indexPivo = particiona(v,ini,fim);
        quickSort(v,ini,indexPivo-1);
        quickSort(v,indexPivo+1,fim);   //indexPivo já está no seu local
    }
}
```

```c
void swap(int* v, int i, int j){
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

int particiona(int* v, int ini, int fim){
    int pIndex = ini;
    int pivo = v[fim];
    for(int i = ini; i < fim; i++){
        if(v[i] <= pivo){
            swap(v,i,pIndex);
            pIndex++;
        }
    }
    swap(v,pIndex,fim);
    return pIndex;
}
```

O algoritmo de ordenação do QuickSort (sem randomização do pivô) propõe que num determinado array escolhe-se um elemento pivô, e após escolhido esse pivô, coloca-se todos os elementos menores que esse pivô a sua esquerda e os elementos maiores a sua direita. Executando recursivamente esse algoritmo (procedimento) para os subarrays esquerdo e direito após a "divisão" pelo elemento pivô, não precisando o auxílio de outros vetores, como no formato Out-of-Place, ou seja, toda a implementação e execução do algoritmo é no formato In-Place.

Considerações: Essa operação de divisão do array pelo elemento pivô, que pode ser qualquer elemento do array, é denominada particionamento. Essa operação consiste em dividir virtualmente esse array original abstraindo em dois subarrays, o subarray esquerdo e o direito. Como já falado, essa operação não necessita de arrays auxiliares, e o que delimita essas operações com o elemento pivô em cada chamada a função são os próprios índices do array original e dos subarrays divididos.

- Pela natureza do algoritmo do QuickSort, ao fazer a chamada e execução da função particiona, ela já retorna um valor de pivô inteiro já na disposição correta e ordenada no array original.
- Importante salientar que o algoritmo do QuickSort executa o algoritmo começando pelo subarray do lado esquerdo do pivô e só

depois o subarray dividido em subarrays ordenados de tamanho 1, que ele passará a executar o algoritmo no subarray direito ao pivô.

Por conseguinte, temos:

Vetor b = [7, 6, 5, 4, 3, 3, 2, 1]

Ini = 0	1	2	3	4	5	6	fim = 7
7	6	5	4	3	3	2	1
pindex	i						pivô

- V[i] <= pivô | 6 <= 1? Não. Incrementa apenas o i, i++;

Ini = 0	1	2	3	4	5	6	fim = 7
7	6	5	4	3	3	2	1
pindex		i					pivô

- V[i] <= pivô | 5 <= 1? Não. Incrementa apenas o i, i++;

Ini = 0	1	2	3	4	5	6	fim = 7
7	6	5	4	3	3	2	1
pindex			i				pivô

- V[i] <= pivô | 4 <= 1? Não. Incrementa apenas o i, i++;

Ini = 0	1	2	3	4	5	6	fim = 7
7	6	5	4	3	3	2	1
pindex				i			pivô

- V[i] <= pivô | 3 <= 1? Não. Incrementa apenas o i, i++;

Ini = 0	1	2	3	4	5	6	fim = 7
7	6	5	4	3	3	2	1
pindex				i			pivô

- V[i] <= pivô | 3 <= 1? Não. Incrementa apenas o i, i++;

Ini = 0	1	2	3	4	5	6	fim = 7
7	6	5	4	3	3	2	1
pindex						i	pivô

- V[i] <= pivô | 2 <= 1? Não. Faz o swap entre pivô e pindex;

Ini = 0	1	2	3	4	5	6	fim = 7
7	6	5	4	3	3	2	1
pindex							pivô

- Retorna a função particiona e o elemento 1 já está no seu lugar ordenado no array.
- Nova chamada a função QuickSort.

Ini = 0	1	2	3	4	5	6	fim = 7
1	6	5	4	3	3	2	7
	pindex	i					pivô

- V[i] <= pivô | 5 <= 7? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	5	6	4	3	3	2	7
		pindex	i				pivô

- V[i] <= pivô | 4 <= 7? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	5	4	6	3	3	2	7
			pindex	i			pivô

- V[i] <= pivô | 3 <= 7? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	5	4	3	6	3	2	7
				pindex	i		pivô

- V[i] <= pivô | 3 <= 7? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	5	4	3	3	6	2	7
					pindex	i	pivô

- V[i] <= pivô | 2 <= 7? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	5	4	3	3	2	6	7
						pindex	pivô

- Retorna a função particiona e o elemento 6 já está no seu lugar ordenado no array.
- Nova chamada a função QuickSort.

Ini = 0	1	2	3	4	5	6	fim = 7
1	5	4	3	3	2	6	7
	pindex	i				pivô	

- V[i] <= pivô | 4 <= 6? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	4	5	3	3	2	6	7
		pindex	i			pivô	

- V[i] <= pivô | 3 <= 6? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	4	3	5	3	2	6	7
			pindex	i		pivô	

- V[i] <= pivô | 3 <= 6? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	4	3	3	5	2	6	7
				pindex	i	pivô	

- V[i] <= pivô | 2 <= 6? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	4	3	3	2	5	6	7
					pindex	pivô	

- Retorna a função particiona e o elemento 5 já está no seu lugar ordenado no array.
- Nova chamada a função QuickSort.

Ini = 0	1	2	3	4	5	6	fim = 7
1	4	3	3	2	5	6	7
	pindex	i			pivô		

- V[i] <= pivô | 4 <= 5? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	3	4	3	2	5	6	7
		pindex	i		pivô		

- V[i] <= pivô | 3 <= 5? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	3	3	4	2	5	6	7
			pindex	i	pivô		

- V[i] <= pivô | 2 <= 5? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	3	3	2	4	5	6	7
				pindex	pivô		

- Retorna a função particiona e o elemento 4 já está no seu lugar ordenado no array.
- Nova chamada a função QuickSort.

Ini = 0	1	2	3	4	5	6	fim = 7
1	3	3	2	4	5	6	7
	pindex	i		pivô			

- V[i] <= pivô | 3 <= 4? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	3	3	2	4	5	6	7
		pindex	i	pivô			

- V[i] <= pivô | 2 <= 4? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	3	2	3	4	5	6	7
			pindex	pivô			

- Retorna a função particiona e o elemento 3 já está no seu lugar ordenado no array.
- Nova chamada a função QuickSort.

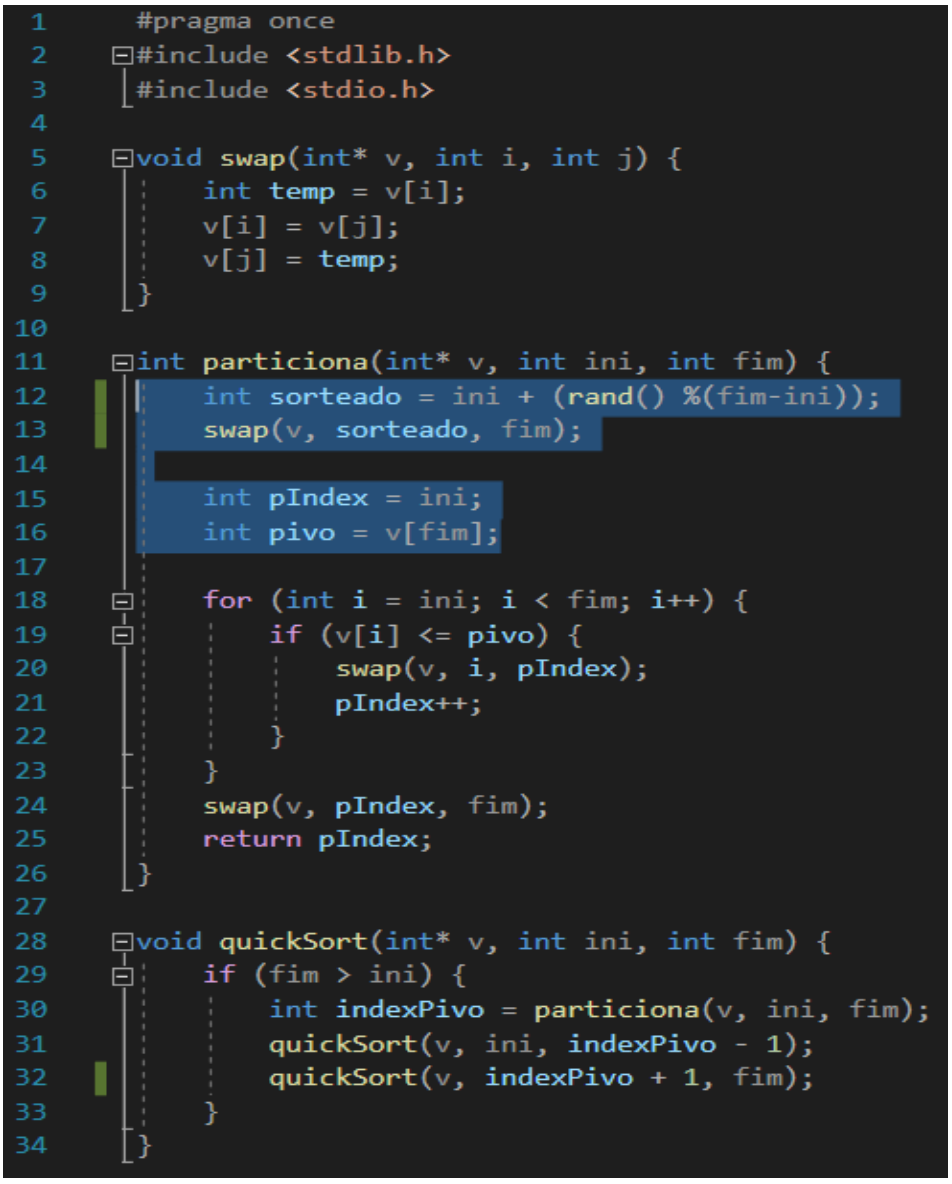
Ini = 0	1	2	3	4	5	6	fim = 7
1	3	2	3	4	5	6	7
	pindex	i	pivô				

- V[i] <= pivô | 2 <= 3? Sim. Swap com i e pindex, incrementa pindex++ e o i++ do próximo laço do for;

Ini = 0	1	2	3	4	5	6	fim = 7
1	2	3	3	4	5	6	7
		pindex	pivô				

- Retorna a função particiona, o elemento 3 já está no seu lugar ordenado no array, retorna para a função QuickSort e o vetor está ordenado!!!

- Implemente o QuickSort com seleção randomizada do pivô. **(1.0)**



```c
#pragma once
#include <stdlib.h>
#include <stdio.h>

void swap(int* v, int i, int j) {
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

int particiona(int* v, int ini, int fim) {
    int sorteado = ini + (rand() %(fim-ini));
    swap(v, sorteado, fim);

    int pIndex = ini;
    int pivo = v[fim];

    for (int i = ini; i < fim; i++) {
        if (v[i] <= pivo) {
            swap(v, i, pIndex);
            pIndex++;
        }
    }
    swap(v, pIndex, fim);
    return pIndex;
}

void quickSort(int* v, int ini, int fim) {
    if (fim > ini) {
        int indexPivo = particiona(v, ini, fim);
        quickSort(v, ini, indexPivo - 1);
        quickSort(v, indexPivo + 1, fim);
    }
}
```

```cpp
#include "pch.h"
#include <C:\Users\tiago souza\source\repos\QuickSort_Random_pivo\Cabeçalho.h>

class SortingAlgorithmsTest : public ::testing::Test {
protected:
    virtual void TearDown() {
        free(v);
    }

    virtual void SetUp() {
        v = (int*)malloc(tamanho * sizeof(int));
        for (int i = 0; i < tamanho; i++) {
            v[i] = rand() % 1000;
        }
    }

    int* v;
    int tamanho = 1000;
    int nTestes = 1;
};


TEST_F(SortingAlgorithmsTest, QuickSort) {
    for (int i = 0; i < nTestes; i++) {
        SetUp();
        quickSort(v, 0, tamanho - 1);
        for (int i = 0; i < tamanho - 1; i++) {
            EXPECT_TRUE(v[i] <= v[i + 1]);
        }
        if (i != nTestes - 1)   //evita executar free 2x no ultimo teste
            TearDown();
    }
}

TEST(TestCaseName, TestName) {
  EXPECT_EQ(1, 1);
  EXPECT_TRUE(true);
}
```

```
Console de Depuração do Microsoft Visual Studio
Running main() from c:\a\1\s\thirdparty\googletest\googletest\src\gtest_main.cc
[==========] Running 2 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 1 test from SortingAlgorithmsTest
[ RUN      ] SortingAlgorithmsTest.QuickSort
[       OK ] SortingAlgorithmsTest.QuickSort (2 ms)
[----------] 1 test from SortingAlgorithmsTest (4 ms total)

[----------] 1 test from TestCaseName
[ RUN      ] TestCaseName.TestName
[       OK ] TestCaseName.TestName (0 ms)
[----------] 1 test from TestCaseName (7 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 2 test cases ran. (18 ms total)
[  PASSED  ] 2 tests.

O C:\Users\tiago souza\source\repos\QuickSort_Random_pivo\Debug\QuickSort_Random_pivo.exe (processo 1224) foi encerrado com o código 0.
Para fechar o console automaticamente quando a depuração parar, habilite Ferramentas -> Opções -> Depuração -> Fechar o console automaticamente quando a depuração parar.
Pressione qualquer tecla para fechar esta janela...
```

- Vamos fazer alguns **experimentos** com os seguintes algoritmos: **SelectionSort (in-place), BubbleSort (melhor versão), InsertionSort (in-place, melhor versão), MergeSort, QuickSort, QuickSort (com seleção randomizada de pivô) e CountingSort.** Crie vetores com os seguintes **tamanhos** 10^1, 10^3, 10^5 (se julgar interessante, pode escolher outros tamanhos). Para cada tamanho, você criará um **vetor ordenado**, um **vetor com valores aleatórios**, e um **vetor ordenado de forma decrescente** (use sementes para obter valores iguais). Para cada combinação de fatores, execute 30 repetições. Compute a média e mediana dessas 30 execuções para cada combinação de fatores. Faça uma análise dissertativa sobre a performance dos algoritmos para diferentes vetores e tamanhos, explicando quais algoritmos têm boa performance em quais situações. **(5.0)**

Inicialmente, vamos mostrar os algoritmos implementados nos arquivos de cabeçalho.h (é o mesmo para as três main.cpp) e main.cpp para vetores ordenados de forma crescente, aleatório e decrescente, respectivamente, em que cada função que é chamada na main.cpp e são coletadas as informações de média e mediana para cada combinação de fatores, executando 30 repetições.

- Cabeçalho.h:

```
#pragma once
#include <stdlib.h>

//selectionSort in-Place
//troca deve ser declarado antes de selectionSortIP
void troca(int* v, int i, int j) {
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

void selectionSortIP(int* v, int tamanho) {
    for (int i = 0; i < (tamanho - 1); i++) {
        int iMenor = i;
        for (int j = i + 1; j < tamanho; j++) {
            if (v[j] < v[iMenor]) {
                iMenor = j;
            }
        }

        troca(v, i, iMenor);
    }
}

//BubbleSort
```

```
void bubbleSort(int* v, int n) {
    for (int varredura = 0; varredura < n - 1; varredura++) {
        bool trocou = false;
        for (int i = 0; i < n - varredura - 1; i++) {
            if (v[i] > v[i + 1]) {
                int temp = v[i];
                v[i] = v[i + 1];
                v[i + 1] = temp;
                trocou = true;
            }
        }
        if (trocou == false)
            return;
    }
}

//insertionSort In-Place
void insertionSortIPV4(int* v, int tamanho) {
    for (int i = 1; i < tamanho; i++) {
        for (int j = i; j > 0 && v[j - 1] > v[j]; j--) {
            int temp = v[j - 1];
            v[j - 1] = v[j];
            v[j] = temp;
        }
    }
}

void merge(int* v, int tamV, int* e, int tamE, int* d, int tamD) {
    int indexV = 0;
    int indexE = 0;
    int indexD = 0;
    while (indexE < tamE && indexD < tamD) {
        if (e[indexE] <= d[indexD]) {
            v[indexV] = e[indexE];
            indexE++;
        }
        else {
            v[indexV] = d[indexD];
            indexD++;
        }
        indexV++;
    }
    while (indexE < tamE) {
        v[indexV] = e[indexE];
        indexE++;
        indexV++;
    }
    while (indexD < tamD) {
        v[indexV] = d[indexD];
        indexD++;
        indexV++;
    }
}
//MergeSort
void mergeSort(int* v, int tamV) {
    if (tamV > 1) {
        int meio = tamV / 2;
        int tamV1 = meio;
        int* v1 = (int*)malloc(tamV1 * sizeof(int));
        for (int i = 0; i < meio; i++) {
            v1[i] = v[i];
        }
```

```
        int tamV2 = tamV - meio;
        int* v2 = (int*)malloc(tamV2 * sizeof(int));
        for (int i = meio; i < tamV; i++) {
            v2[i - meio] = v[i];
        }

        mergeSort(v1, tamV1);
        mergeSort(v2, tamV2);
        merge(v, tamV, v1, tamV1, v2, tamV2);

        free(v1);
        free(v2);
    }
}

void swap(int* v, int i, int j) {
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

int particiona(int* v, int ini, int fim) {
    int pIndex = ini;
    int pivo = v[fim];
    for (int i = ini; i < fim; i++) {
        if (v[i] <= pivo) {
            swap(v, i, pIndex);
            pIndex++;
        }
    }
    swap(v, pIndex, fim);
    return pIndex;
}

//QuickSort sem Randomização de Pivô
void quickSort(int* v, int ini, int fim) {
    if (fim > ini) {
        int indexPivo = particiona(v, ini, fim);
        quickSort(v, ini, indexPivo - 1);
        quickSort(v, indexPivo + 1, fim);
    }
}

int particionaRP(int* v, int ini, int fim) {
    int sorteado = ini + (rand() % (fim - ini));
    swap(v, sorteado, fim);

    int pIndex = ini;
    int pivo = v[fim];

    for (int i = ini; i < fim; i++) {
        if (v[i] <= pivo) {
            swap(v, i, pIndex);
            pIndex++;
        }
    }
    swap(v, pIndex, fim);
    return pIndex;
}

//QuickSort com Randomização de Pivô
```

```
void quickSortRP(int* v, int ini, int fim) {
    if (fim > ini) {
        int indexPivo = particiona(v, ini, fim);
        quickSort(v, ini, indexPivo - 1);
        quickSort(v, indexPivo + 1, fim);
    }
}

//CountingSort
void countingSort(int** v, int tam) {
    int iMenorValor = 0;
    int iMaiorValor = 0;
    for (int i = 0; i < tam; i++) {
        if ((*v)[i] < (*v)[iMenorValor])
            iMenorValor = i;
        if ((*v)[i] > (*v)[iMaiorValor])
            iMaiorValor = i;
    }

    int tamContagem = (*v)[iMaiorValor] - (*v)[iMenorValor] + 1;
    int* contagem = (int*)calloc(tamContagem, sizeof(int));

    for (int i = 0; i < tam; i++) {
        int indiceEmContagem = (*v)[i] - (*v)[iMenorValor];
        contagem[indiceEmContagem]++;
    }

    for (int i = 1; i < tamContagem; i++)
        contagem[i] += contagem[i - 1];

    int* ordenado = (int*)malloc(tam * sizeof(int));
    bool* adicionado = (bool*)calloc(tam, sizeof(bool));

     for (int i = 0; i < tam; i++) {
        int indiceOrdenado = contagem[(*v)[i] - (*v)[iMenorValor]] - 1;
        while (adicionado[indiceOrdenado])
            indiceOrdenado--;
        ordenado[indiceOrdenado] = (*v)[i];
        adicionado[indiceOrdenado] = true;

    }

    (*v) = ordenado;
       free(contagem);
       free(ordenado);
       free(adicionado);
}
```

- Main.cpp (Vetores Ordenados de Forma Crescente)

```
#include "Cabeçalho.h"

#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <chrono>
#include <thread>

#define tam 100
```

```
using namespace std;

int main() {
	double tempo[30];
	int* Vetor = (int*)malloc(tam * sizeof(int));

	cout << "Vetor com valores aleatorios: ";

	int aux[tam];

	// Gerando vetor crecente fixo
	for (int k = 0; k < tam; k++) {
		aux[k] = k;
	}

	// Mostrando vetor crescente fixo
	for (int k = 0; k < tam; k++) {
		cout << aux[k] << " ";
	}

	cout << endl;
	cout << endl;

	//Contagem de tempo
	for (int j = 0; j < 7; j++) {
		double media = 0;
		double mediana = 0;
		for (int i = 0; i < 30; i++) {

			// Gerando vetor aleatório
			for (int k = 0; k < tam; k++) {
				Vetor[k] = aux[k];
			}

			if (j == 0) {
				auto start = chrono::high_resolution_clock::now();
				selectionSortIP(Vetor, tam);
				auto end = chrono::high_resolution_clock::now();
				std::chrono::duration<double, std::milli> float_ms =
end - start;
				tempo[i] = float_ms.count();
				if (i == 0)
					cout << "*SelectionSortIP*" << endl;

			}
			else if (j == 1) {
				auto start = chrono::high_resolution_clock::now();
				bubbleSort(Vetor, tam);
				auto end = chrono::high_resolution_clock::now();
				std::chrono::duration<double, std::milli> float_ms =
end - start;
				tempo[i] = float_ms.count();
				if (i == 0)
					cout << "*BubbleSort*" << endl;
			}
			else if (j == 2) {
				auto start = chrono::high_resolution_clock::now();
				insertionSortIPV4(Vetor, tam);
				auto end = chrono::high_resolution_clock::now();
				std::chrono::duration<double, std::milli> float_ms =
end - start;
				tempo[i] = float_ms.count();
```

```
                        if (i == 0)
                                cout << "*InsertionSortIP*" << endl;
                }
                else if (j == 3) {
                        auto start = chrono::high_resolution_clock::now();
                        mergeSort(Vetor, tam);
                        auto end = chrono::high_resolution_clock::now();
                        std::chrono::duration<double, std::milli> float_ms =
end - start;
                        tempo[i] = float_ms.count();
                        if (i == 0)
                                cout << "*MergeSort*" << endl;
                }
                else if (j == 4) {
                        auto start = chrono::high_resolution_clock::now();
                        quickSort(Vetor, 0, tam-1);
                        auto end = chrono::high_resolution_clock::now();
                        std::chrono::duration<double, std::milli> float_ms =
end - start;
                        tempo[i] = float_ms.count();
                        if (i == 0)
                                cout << "*QuickSortIP*" << endl;
                }
                else if (j == 5) {
                        auto start = chrono::high_resolution_clock::now();
                        quickSortRP(Vetor, 0, tam-1);
                        auto end = chrono::high_resolution_clock::now();
                        std::chrono::duration<double, std::milli> float_ms =
end - start;
                        tempo[i] = float_ms.count();
                        if (i == 0)
                                cout << "*QuickSortRP*" << endl;
                }
                else {
                        auto start = chrono::high_resolution_clock::now();
                        countingSort(&Vetor, tam);
                        auto end = chrono::high_resolution_clock::now();
                        std::chrono::duration<double, std::milli> float_ms =
end - start;
                        tempo[i] = float_ms.count();
                        if (i == 0)
                                cout << "*CountingSortIP*" << endl;
                }

        }

        for (int i = 0; i < 30; i++) {
                //cout << tempo[i] << " ";
                media += tempo[i];
                if (i == 14 || i == 15) {
                        mediana += tempo[i];
                }

        }

        cout << endl << "Vetor Ordenado: ";

        for (int i = 0; i < tam; i++) {
                cout << Vetor[i] << " ";
        }

        cout << endl;
```

```
			cout << endl << "Media: " << media / 30 << endl;
			cout << endl << "Mediana: " << mediana / 2 << endl << endl << endl;
		}
		free(Vetor);
		return 0;
}
```

- Main.cpp (Vetores Ordenados de Forma Aleatória)

```
#include "Cabeçalho.h"

#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <chrono>
#include <thread>

#define tam 100

using namespace std;

int main() {
	double tempo[30];
	int* Vetor = (int*)malloc(tam * sizeof(int));

	cout << "Vetor com valores aleatorios: ";

	int aux[tam];

	// Gerando vetor aleatório fixo
	for (int k = 0; k < tam; k++) {
		aux[k] = rand() % tam;
	}

	// Mostrando vetor aleatório fixo
	for (int k = 0; k < tam; k++) {
		cout << aux[k] << " ";
	}

	cout << endl;
	cout << endl;

	//Contagem de tempo
	for (int j = 0; j < 7; j++) {
		double media = 0;
		double mediana = 0;
		for (int i = 0; i < 30; i++) {

			// Gerando vetor aleatório
			for (int k = 0; k < tam; k++) {
				Vetor[k] = aux[k];
			}

			if (j == 0) {
				auto start = chrono::high_resolution_clock::now();
				selectionSortIP(Vetor, tam);
				auto end = chrono::high_resolution_clock::now();
				std::chrono::duration<double, std::milli> float_ms =
end - start;
```

```
                        tempo[i] = float_ms.count();
                        if (i == 0)
                                cout << "*SelectionSortIP*" << endl;

                }
                else if (j == 1) {
                        auto start = chrono::high_resolution_clock::now();
                        bubbleSort(Vetor, tam);
                        auto end = chrono::high_resolution_clock::now();
                        std::chrono::duration<double, std::milli> float_ms =
end - start;
                        tempo[i] = float_ms.count();
                        if (i == 0)
                                cout << "*BubbleSort*" << endl;
                }
                else if (j == 2) {
                        auto start = chrono::high_resolution_clock::now();
                        insertionSortIPV4(Vetor, tam);
                        auto end = chrono::high_resolution_clock::now();
                        std::chrono::duration<double, std::milli> float_ms =
end - start;
                        tempo[i] = float_ms.count();
                        if (i == 0)
                                cout << "*InsertionSortIP*" << endl;
                }
                else if (j == 3) {
                        auto start = chrono::high_resolution_clock::now();
                        mergeSort(Vetor, tam);
                        auto end = chrono::high_resolution_clock::now();
                        std::chrono::duration<double, std::milli> float_ms =
end - start;
                        tempo[i] = float_ms.count();
                        if (i == 0)
                                cout << "*MergeSort*" << endl;
                }
                else if (j == 4) {
                        auto start = chrono::high_resolution_clock::now();
                        quickSort(Vetor, 0, tam-1);
                        auto end = chrono::high_resolution_clock::now();
                        std::chrono::duration<double, std::milli> float_ms =
end - start;
                        tempo[i] = float_ms.count();
                        if (i == 0)
                                cout << "*QuickSortIP*" << endl;
                }
                else if (j == 5) {
                        auto start = chrono::high_resolution_clock::now();
                        quickSortRP(Vetor, 0, tam-1);
                        auto end = chrono::high_resolution_clock::now();
                        std::chrono::duration<double, std::milli> float_ms =
end - start;
                        tempo[i] = float_ms.count();
                        if (i == 0)
                                cout << "*QuickSortRP*" << endl;
                }
                else {
                        auto start = chrono::high_resolution_clock::now();
                        countingSort(&Vetor, tam);
                        auto end = chrono::high_resolution_clock::now();
                        std::chrono::duration<double, std::milli> float_ms =
end - start;
                        tempo[i] = float_ms.count();
```

```
                        if (i == 0)
                                cout << "*CountingSortIP*" << endl;
                }

        }

        for (int i = 0; i < 30; i++) {
                //cout << tempo[i] << " ";
                media += tempo[i];
                if (i == 14 || i == 15) {
                        mediana += tempo[i];
                }

        }

        cout << endl << "Vetor Ordenado: ";

        for (int i = 0; i < tam; i++) {
                cout << Vetor[i] << " ";
        }

        cout << endl;
        cout << endl << "Media: " << media / 30 << endl;
        cout << endl << "Mediana: " << mediana / 2 << endl << endl << endl;
    }
    free(Vetor);
    return 0;
}
```

- Main.cpp (Vetores Ordenados de Forma Decrescente)

```
#include "Cabeçalho.h"

#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <chrono>
#include <thread>

#define tam 100

using namespace std;

int main() {
    double tempo[30];
    int* Vetor = (int*)malloc(tam * sizeof(int));

    cout << "Vetor com valores decrescentes: ";

    int aux[tam];

    // Gerando vetor decrescente
    for (int k = tam-1; k >=0; k--) {
        aux[k] = k;
    }

    // Imprimindo vetor decrescente
    for (int k = tam-1; k >= 0; k--) {
        cout << aux[k] << " ";
    }
```

```
	cout << endl;
	cout << endl;

	//Contagem de tempo
	for (int j = 0; j < 7; j++) {
		double media = 0;
		double mediana = 0;
		for (int i = 0; i < 30; i++) {

			// Copiando vetor decrescente
			for (int k = tam-1; k >= 0; k--) {
				Vetor[k] = aux[k];
			}

			if (j == 0) {
				auto start = chrono::high_resolution_clock::now();
				selectionSortIP(Vetor, tam);
				auto end = chrono::high_resolution_clock::now();
				std::chrono::duration<double, std::milli> float_ms =
end - start;
				tempo[i] = float_ms.count();
				if (i == 0)
					cout << "*SelectionSortIP*" << endl;

			}
			else if (j == 1) {
				auto start = chrono::high_resolution_clock::now();
				bubbleSort(Vetor, tam);
				auto end = chrono::high_resolution_clock::now();
				std::chrono::duration<double, std::milli> float_ms =
end - start;
				tempo[i] = float_ms.count();
				if (i == 0)
					cout << "*BubbleSort*" << endl;
			}
			else if (j == 2) {
				auto start = chrono::high_resolution_clock::now();
				insertionSortIPV4(Vetor, tam);
				auto end = chrono::high_resolution_clock::now();
				std::chrono::duration<double, std::milli> float_ms =
end - start;
				tempo[i] = float_ms.count();
				if (i == 0)
					cout << "*InsertionSortIP*" << endl;
			}
			else if (j == 3) {
				auto start = chrono::high_resolution_clock::now();
				mergeSort(Vetor, tam);
				auto end = chrono::high_resolution_clock::now();
				std::chrono::duration<double, std::milli> float_ms =
end - start;
				tempo[i] = float_ms.count();
				if (i == 0)
					cout << "*MergeSort*" << endl;
			}
			else if (j == 4) {
				auto start = chrono::high_resolution_clock::now();
				quickSort(Vetor, 0, tam-1);
				auto end = chrono::high_resolution_clock::now();
				std::chrono::duration<double, std::milli> float_ms =
end - start;
```

```
                        tempo[i] = float_ms.count();
                        if (i == 0)
                                cout << "*QuickSortIP*" << endl;
                }
                else if (j == 5) {
                        auto start = chrono::high_resolution_clock::now();
                        quickSortRP(Vetor, 0, tam-1);
                        auto end = chrono::high_resolution_clock::now();
                        std::chrono::duration<double, std::milli> float_ms =
end - start;
                        tempo[i] = float_ms.count();
                        if (i == 0)
                                cout << "*QuickSortRP*" << endl;
                }
                else {
                        auto start = chrono::high_resolution_clock::now();
                        countingSort(&Vetor, tam);
                        auto end = chrono::high_resolution_clock::now();
                        std::chrono::duration<double, std::milli> float_ms =
end - start;
                        tempo[i] = float_ms.count();
                        if (i == 0)
                                cout << "*CountingSortIP*" << endl;
                }

        }

        for (int i = 0; i < 30; i++) {
                //cout << tempo[i] << " ";
                media += tempo[i];
                if (i == 14 || i == 15) {
                        mediana += tempo[i];
                }

        }

        cout << endl << "Vetor Ordenado: ";

        for (int i = 0; i < tam; i++) {
                cout << Vetor[i] << " ";
        }

        cout << endl;
        cout << endl << "Media: " << media / 30 << endl;
        cout << endl << "Mediana: " << mediana / 2 << endl << endl << endl;
    }
    free(Vetor);
    return 0;
}
```

Considerações:

- Nestes algoritmos da main.cpp, definimos uma variável chamada “tam” com o comando “#define”, e nessa variável foi definido os tamanhos dos vetores.
- Foi implementado um grande for e com o auxílio das bibliotecas <thread> e <chrono>, foi recolhido o tempo médio e mediano das 30

repetições de um grande comando “for”, que em cada tempo/varredura, chamava uma função de ordenação e implementava o código chamado.

- Ao final de cada tempo/varredura é mostrado o tempo médio e mediano em **milissegundos.**
- Logo abaixo, é mostrado algumas tabelas com alterações da variável definida **tam** da função main.cpp para cada caso (Vetor Crescente, Aleatório e Decrescente) em que o tamanho será = 10^1, 10^2, 10^3, 10^4, 10^5, com 30 repetições, recolhendo as médias e medianas. Vamos recolher as informações dos 7 algoritmos de ordenação, são eles: **SelectionSort (in-place), BubbleSort (melhor versão), InsertionSort (in-place, melhor versão), MergeSort, QuickSort, QuickSort (com seleção randomizada de pivô) e CountingSort.** Depois faremos análises sobre cada algoritmo.

Média (milissegundos) - Vetor Crescente					
Algoritmo/Tamanho	10^1	10^2	10^3	10^4	10^5
SelectionSortIP	0.000863333	0.0193867	1.76625	171.924	16446.4
BubbleSort	0.000376667	0.0005	0.00446667	171.956	16446.7
InsertionsortIP	0.0004	0.000816667	0.00527333	171.992	16447.1
MergeSort	0.0161433	0.19485	1.42317	181.633	16546.4
QuickSort	0.00300333	0.187313	15.6213	Stack Overflow	
QuickSortRP	0.00281333	0.18097	11.4078		
CountingSort	0.00358667	0.00906333	0.0322233	0.245311	2.61883

Mediana (milissegundos) - Vetor Crescente					
Algoritmo/Tamanho	10^1	10^2	10^3	10^4	10^5
SelectionSortIP	0.00085	0.0193	1.6748	156.331	1597.7
BubbleSort	0.00035	0.0004	0.0045	156.363	15958
InsertionsortIP	0.00035	0.0008	0.00515	156.397	15958.3
MergeSort	0.0136	0.1608	1.1619	166.912	16060.6
QuickSort	0.0031	0.18395	14.8653	Stack Overflow	
QuickSortRP	0.00275	0.17395	10.6195		
CountingSort	0.01525	0.0048	0.05345	0.25332	2.60829

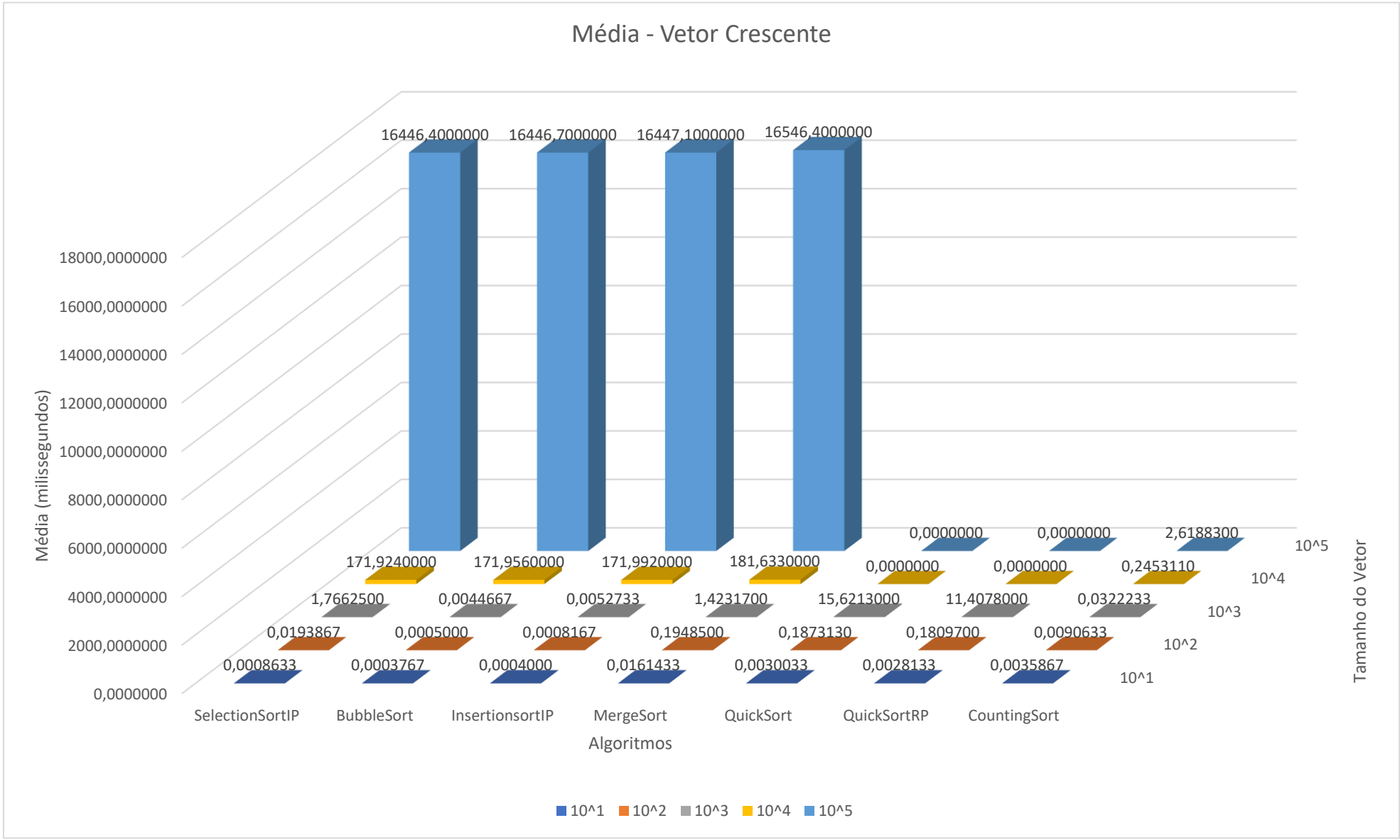
Média (milissegundos) - Vetor Aleatório					
Algoritmo/Tamanho	10^1	10^2	10^3	10^4	10^5
SelectionSortIP	0.000823333	0.0209433	1.79452	178.579	15850
BubbleSort	0.00066	0.0316667	3.61548	369.562	35883.5
InsertionsortIP	0.00057	0.0212167	1.94555	136.028	12738.9
MergeSort	0.0190367	0.21336	1.33121	11.7748	108.807
QuickSort	0.0014	0.0302067	0.314083	3.59902	41.4992
QuickSortRP	0.0014	0.0202633	0.488943	3.43401	41.539
CountingSort	0.00316333	0.00848	0.04484	0.39704	4.53199

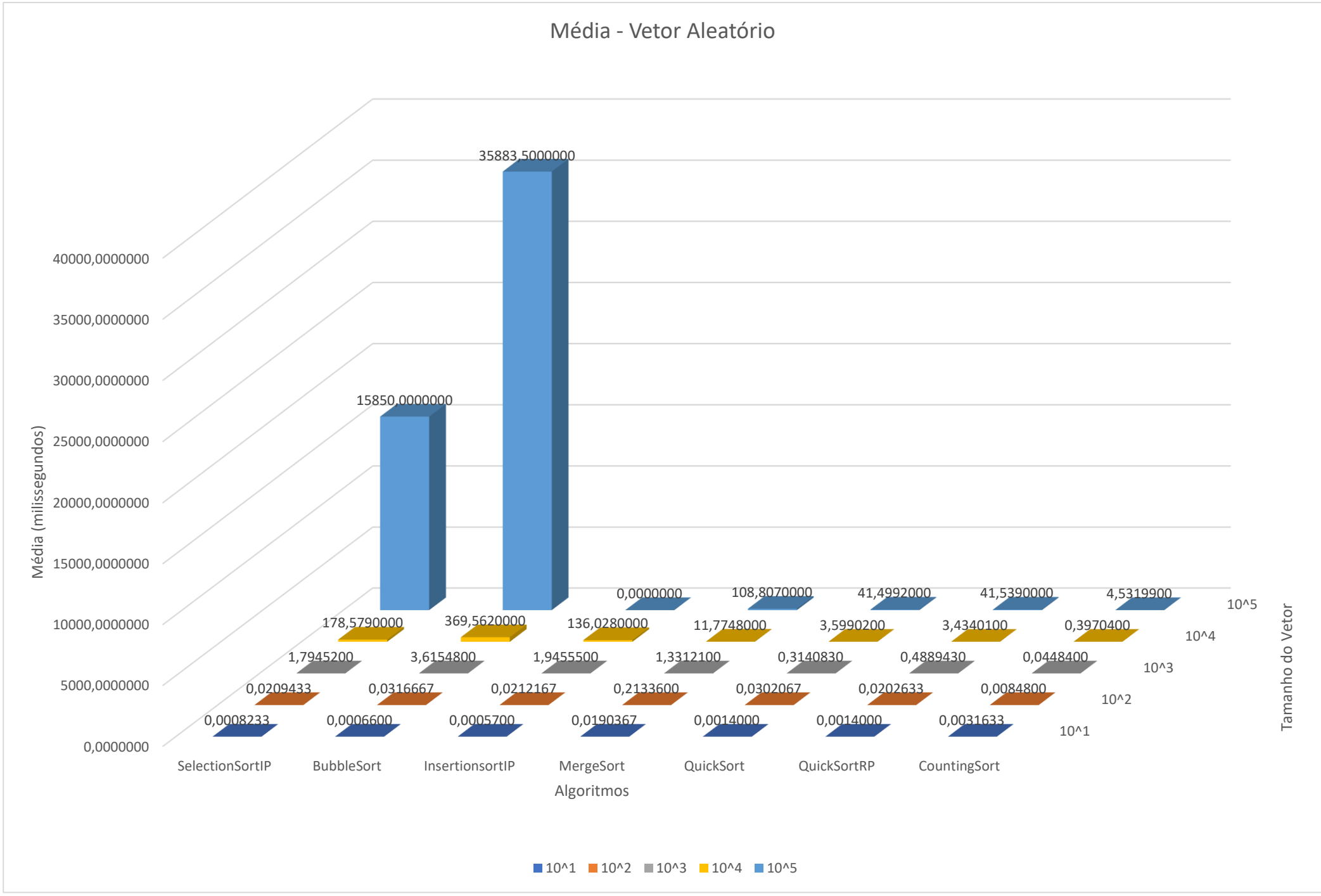
Mediana (milissegundos) - Vetor Aleatório					
Algoritmo/Tamanho	10^1	10^2	10^3	10^4	10^5
SelectionSortIP	0.0008	0.0206	1.71485	168.163	15715.5
BubbleSort	0.0005	0.031	3.006	381.607	35780.7
InsertionsortIP	0.0005	0.0212	1.8883	129.409	12588.2
MergeSort	0.01775	0.1822	1.1896	11.0747	106.628
QuickSort	0.0013	0.02375	0.38545	3.85415	40.0069
QuickSortRP	0.0013	0.0218	0.3524	3.1973	42.0626
CountingSort	0.01395	0.005	0.044	0.3717	4.28865

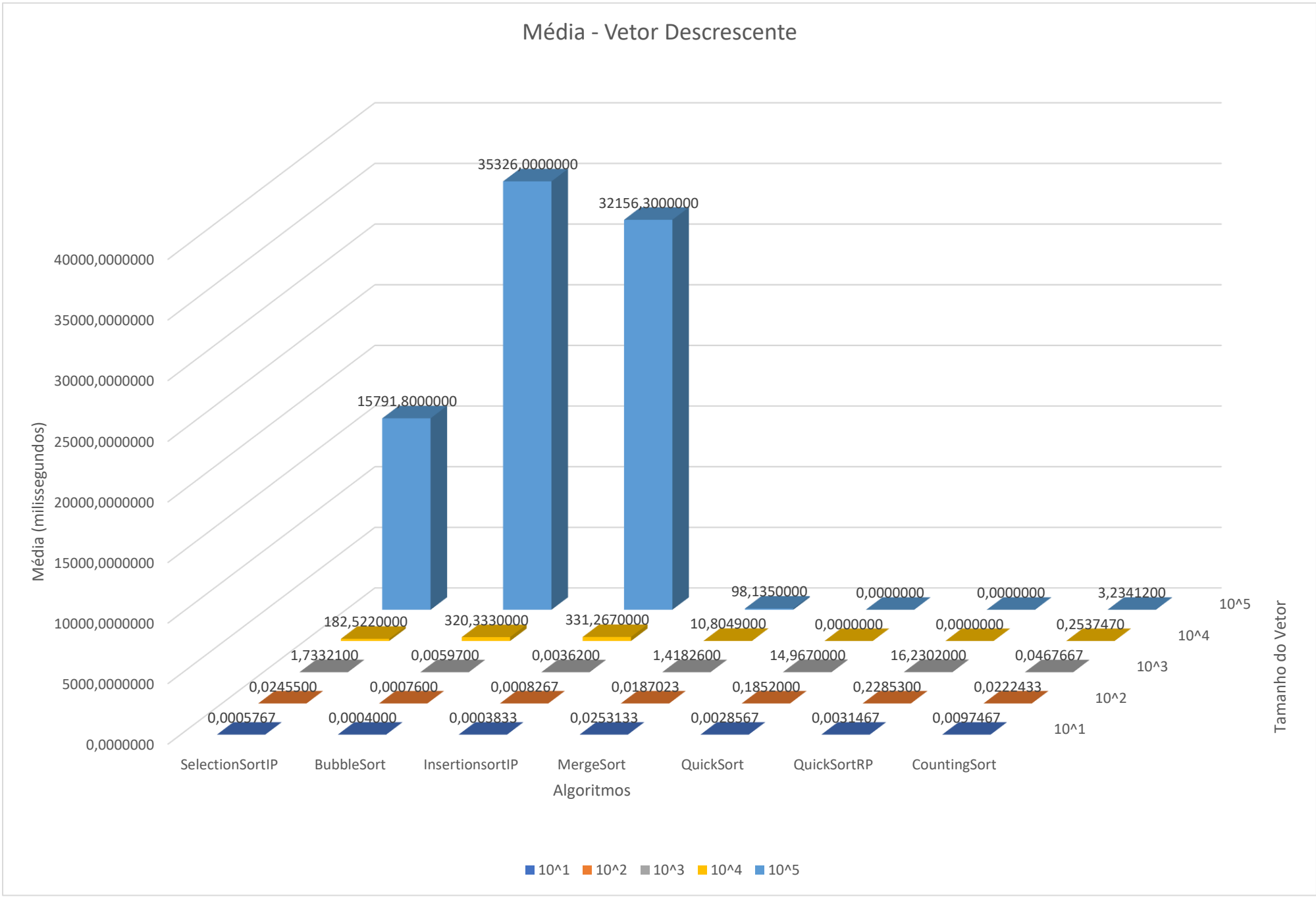
Média (milissegundos) - Vetor Decrescente					
Algoritmo/Tamanho	10^1	10^2	10^3	10^4	10^5
SelectionSortIP	0.000576667	0.02455	1.73321	182.522	15791.8
BubbleSort	0.0004	0.00076	0.00597	0.0320333	0.35326
InsertionsortIP	0.000383333	0.000826667	0.00362	0.0331267	0.321563
MergeSort	0.0253133	0.0187023	1.41826	10.8049	98.135
QuickSort	0.002856667	0.1852	14.967	Stack Overflow	
QuickSortRP	0.00314667	0.22853	16.2302		
CountingSort	0.00974667	0.0222433	0.0467667	0.253747	3.23412

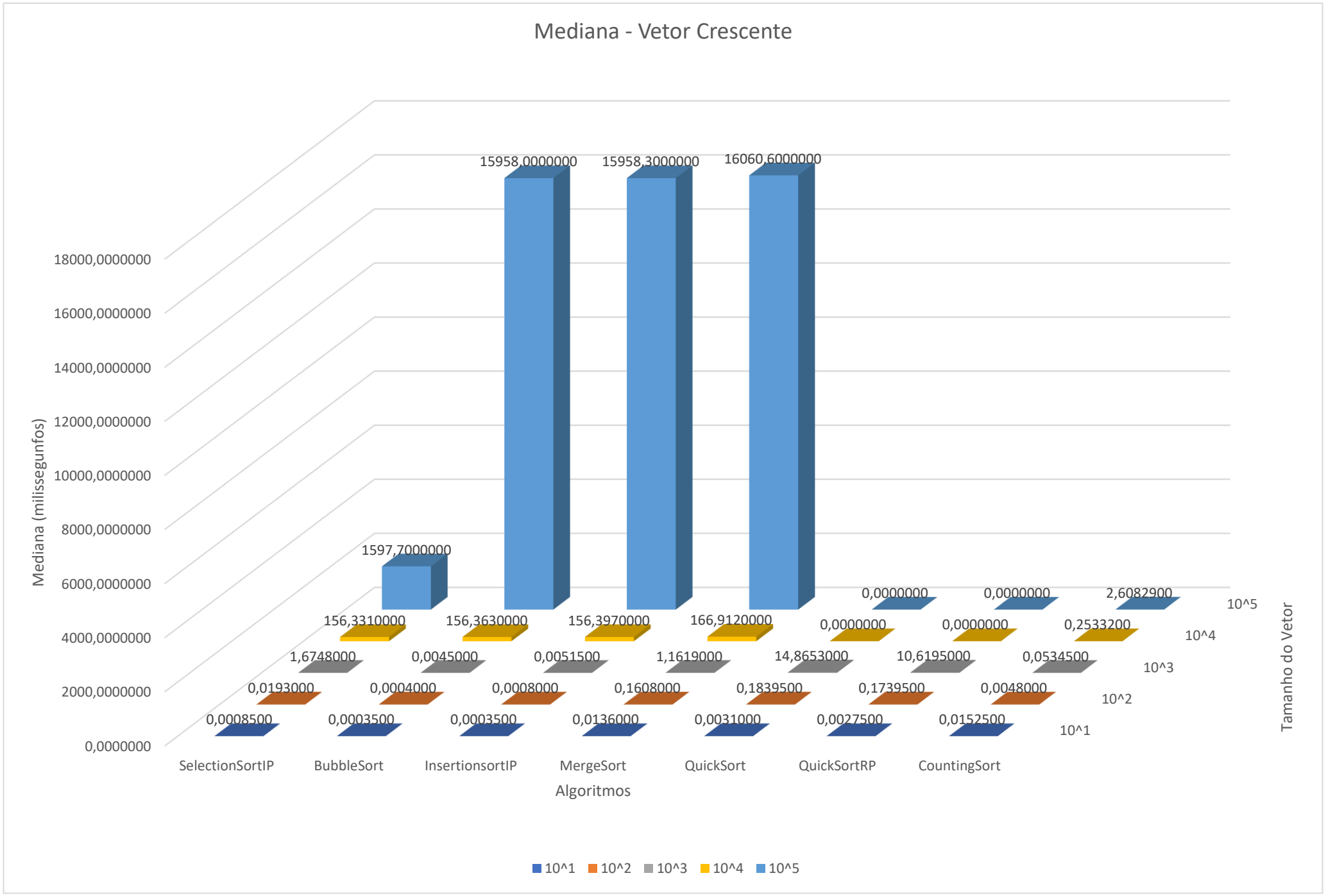
Mediana (milissegundos) - Vetor Decrescente					
Algoritmo/Tamanho	10^1	10^2	10^3	10^4	10^5
SelectionSortIP	0.0005	0.02445	1.672	165.515	15680
BubbleSort	0.0003	0.00075	0.00405	0.0322	0.358
InsertionsortIP	0.0004	0.00085	0.0035	0.03305	0.31585
MergeSort	0.0163	0.153	1.0276	13.2995	95.9591
QuickSort	0.0027	0.18365	16.7699	Stack Overflow	
QuickSortRP	0.0032	0.2441	14.7317		
CountingSort	0.10565	0.00515	0.0708	0.29615	2.8258

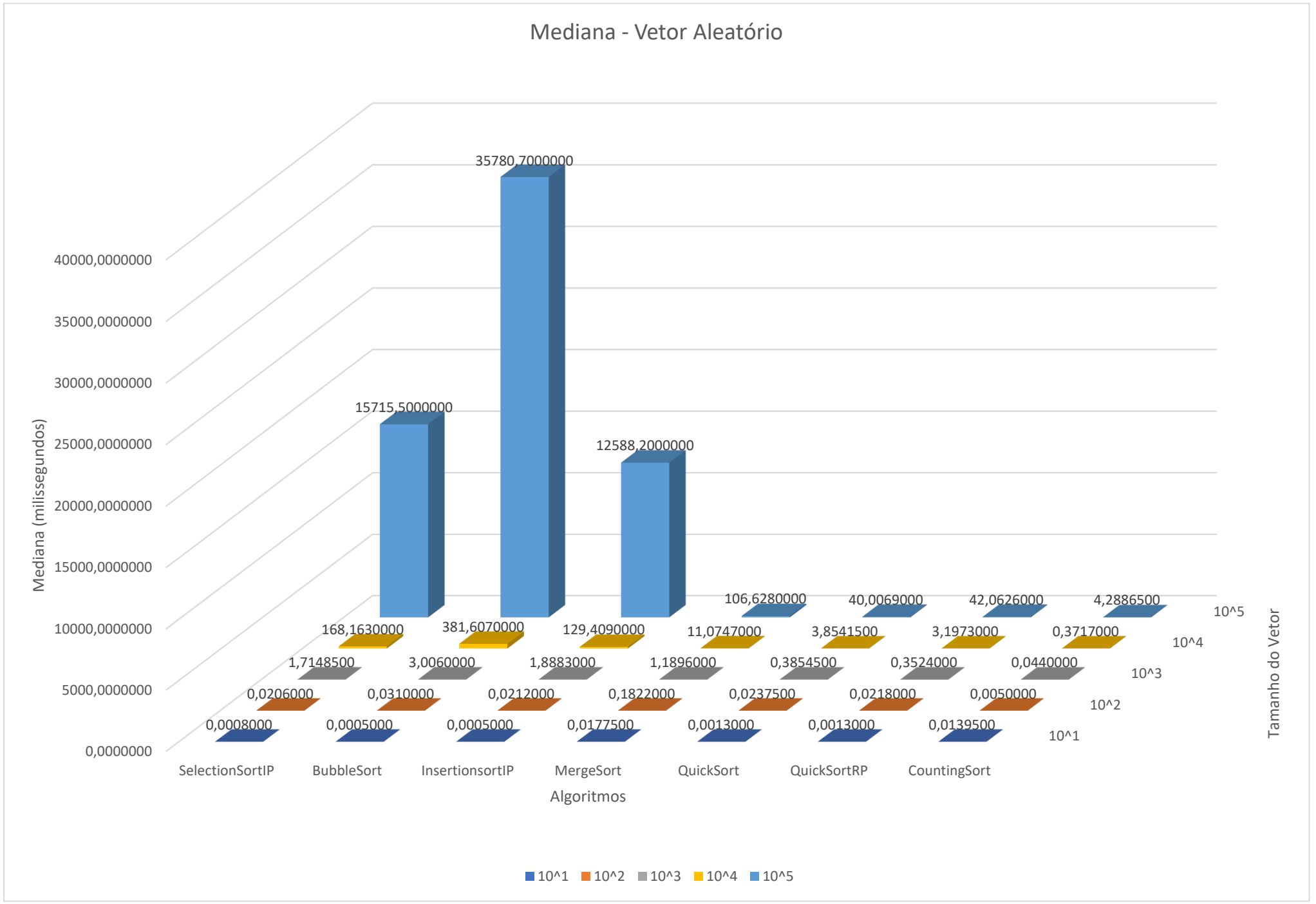
Gráficos

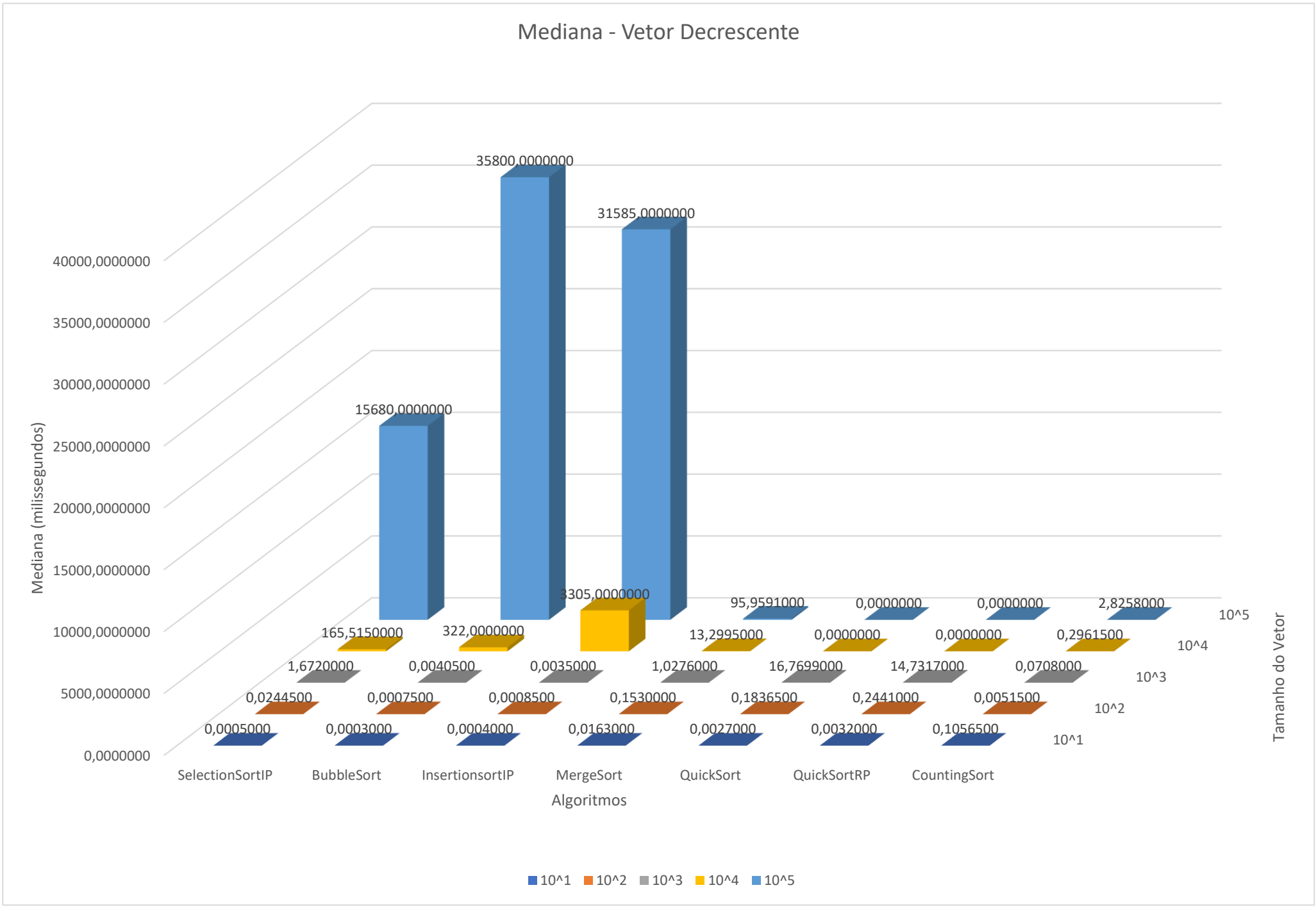

Média - Vetor Crescente
Média (milissegundos)
18000,0000000
16000,0000000
14000,0000000
12000,0000000
10000,0000000
8000,0000000
6000,0000000
4000,0000000
2000,0000000
0,0000000
16446,4000000
16446,7000000
16447,1000000
16546,4000000
171,9240000
171,9560000
171,9920000
181,6330000
1,7662500
0,0044667
0,0052733
1,4231700
0,0193867
0,0005000
0,0008167
0,1948500
0,1873130
0,0008633
0,0003767
0,0004000
0,0161433
0,0030033
0,0000000
0,0000000
0,0000000
0,0000000
15,6213000
11,4078000
0,2453110
0,0322233
0,1809700
0,0090633
0,0028133
0,0035867
2,6188300
SelectionSortIP
BubbleSort
InsertionsortIP
MergeSort
QuickSort
QuickSortRP
CountingSort
Algoritmos
10^5
10^4
10^3
10^2
10^1
Tamanho do Vetor
10^1
10^2
10^3
10^4
10^5


Média - Vetor Aleatório
Média (milissegundos)
40000,0000000
35000,0000000
30000,0000000
25000,0000000
20000,0000000
15000,0000000
10000,0000000
5000,0000000
0,0000000
35883,5000000
15850,0000000
178,5790000
369,5620000
136,0280000
11,7748000
3,5990200
3,4340100
0,3970400
1,7945200
3,6154800
1,9455500
1,3312100
0,3140830
0,4889430
0,0448400
0,0209433
0,0316667
0,0212167
0,2133600
0,0302067
0,0202633
0,0084800
0,0008233
0,0006600
0,0005700
0,0190367
0,0014000
0,0014000
0,0031633
0,0000000
108,8070000
41,4992000
41,5390000
4,5319900
10^5
10^4
10^3
10^2
10^1
Tamanho do Vetor
SelectionSortIP
BubbleSort
InsertionsortIP
MergeSort
QuickSort
QuickSortRP
CountingSort
Algoritmos
10^1
10^2
10^3
10^4
10^5


Média - Vetor Descrescente
Média (milissegundos)
40000,0000000
35000,0000000
30000,0000000
25000,0000000
20000,0000000
15000,0000000
10000,0000000
5000,0000000
0,0000000
35326,0000000
32156,3000000
15791,8000000
182,5220000
320,3330000
331,2670000
98,1350000
0,0000000
0,0000000
3,2341200
1,7332100
0,0059700
0,0036200
10,8049000
0,0000000
0,0000000
0,2537470
0,0245500
0,0007600
0,0008267
1,4182600
14,9670000
16,2302000
0,0467667
0,0005767
0,0004000
0,0003833
0,0187023
0,1852000
0,2285300
0,0222433
0,0253133
0,0028567
0,0031467
0,0097467
SelectionSortIP
BubbleSort
InsertionsortIP
MergeSort
QuickSort
QuickSortRP
CountingSort
Algoritmos
10^5
10^4
10^3
10^2
10^1
Tamanho do Vetor
10^1
10^2
10^3
10^4
10^5


Mediana - Vetor Crescente
Mediana (milissegunfos)
18000,0000000
16000,0000000
14000,0000000
12000,0000000
10000,0000000
8000,0000000
6000,0000000
4000,0000000
2000,0000000
0,0000000
15958,0000000
15958,3000000
16060,6000000
1597,7000000
156,3310000
156,3630000
156,3970000
166,9120000
0,0000000
0,0000000
2,6082900
0,0000000
0,0000000
0,2533200
1,6748000
0,0045000
0,0051500
1,1619000
14,8653000
10,6195000
0,0534500
0,0193000
0,0004000
0,0008000
0,1608000
0,1839500
0,1739500
0,0048000
0,0008500
0,0003500
0,0003500
0,0136000
0,0031000
0,0027500
0,0152500
SelectionSortIP
BubbleSort
InsertionsortIP
MergeSort
QuickSort
QuickSortRP
CountingSort
Algoritmos
10^1
10^2
10^3
10^4
10^5
Tamanho do Vetor
10^1
10^2
10^3
10^4
10^5
10^1 10^2 10^3 10^4 10^5


Mediana - Vetor Aleatório
Mediana (milissegundos)
40000,0000000
35000,0000000
30000,0000000
25000,0000000
20000,0000000
15000,0000000
10000,0000000
5000,0000000
0,0000000
35780,7000000
15715,5000000
12588,2000000
106,6280000
40,0069000
42,0626000
4,2886500
168,1630000
381,6070000
129,4090000
11,0747000
3,8541500
3,1973000
0,3717000
1,7148500
3,0060000
1,8883000
1,1896000
0,3854500
0,3524000
0,0440000
0,0206000
0,0310000
0,0212000
0,1822000
0,0237500
0,0218000
0,0050000
0,0008000
0,0005000
0,0005000
0,0177500
0,0013000
0,0013000
0,0139500
SelectionSortIP
BubbleSort
InsertionsortIP
MergeSort
QuickSort
QuickSortRP
CountingSort
Algoritmos
10^5
10^4
10^3
10^2
10^1
Tamanho do Vetor
10^1
10^2
10^3
10^4
10^5


Mediana - Vetor Decrescente
Mediana (milissegundos)
40000,0000000
35000,0000000
30000,0000000
25000,0000000
20000,0000000
15000,0000000
10000,0000000
5000,0000000
0,0000000
35800,0000000
31585,0000000
15680,0000000
3305,0000000
322,0000000
165,5150000
95,9591000
13,2995000
1,6720000
0,0040500
0,0035000
1,0276000
16,7699000
14,7317000
0,0708000
0,0244500
0,0007500
0,0008500
0,1530000
0,1836500
0,2441000
0,0051500
0,0005000
0,0003000
0,0004000
0,0163000
0,0027000
0,0032000
0,1056500
0,0000000
0,0000000
0,0000000
2,8258000
0,2961500
SelectionSortIP
BubbleSort
InsertionsortIP
MergeSort
QuickSort
QuickSortRP
CountingSort
Algoritmos
Tamanho do Vetor
10^5
10^4
10^3
10^2
10^1
10^1
10^2
10^3
10^4
10^5

Análises

Podemos observar que durante os testes, algumas combinações de vetores e algoritmos não executaram ou demoraram muito a executar e mostrar o resultado. Vemos na tabela supracitada que os casos em que os algoritmos do QuickSort e QuickSort com Randomização de Pivô, não executaram e mostraram um erro “Stack Overflow” para vetores com tamanhos de 10^4 e 10^5, mesmo executando um só por vez, para os casos em que o vetor já estava ordenado de forma crescente e de forma decrescente. E isso, mostra de forma experimental os conceitos que para estes casos o algoritmo do QuickSort e QuickSortRP, irão levar ao pior caso, que é O(n^2), e que provavelmente nem executaram.

Em geral a performance do algoritmo SelectionSort é o pior de todos os algoritmos de ordenação, pois o fato de ter que “varrer” todo o vetor múltiplas vezes, causa mais trabalho e processamento, tornando o processo mais custoso, de forma que na primeira vez que o algoritmo varre o vetor, ele visita n-1 índices, dessa forma, na segunda vez, ele vista n-2 índices, por isso, ele se torna um algoritmo O(n^2).

Nos casos em que o vetor é muito grande, o algoritmo do BubbleSort leva bastante desvantagem, em que para um vetor muito grande e decrescente, a ordenação se torna mais custosa até que o próprio SelectionSort, se tornando O(n^2), no pior caso.

Para o InsertioSort, no melhor caso, em que o vetor já estava ordenado, ele se tornou O(n), pois o for interno não foi executado e ele só visitou os índices do vetor apenas uma vez e verificou que já estava ordenado, e para o caso de vetor ordenado de forma decrescente, ele se tornou O(n^2) ao visitar várias posições várias vezes.

O algoritmo MergeSort, é um algoritmo O(nlog(n)) no pior caso e isso foi mostrado no caso do vetor ordenado de forma decrescente, mas infelizmente, por sua implementação, Out_of_Place, não o ajudou muito para vetores muito grandes, porém com média e mediana muito próximas ao QuickSort e QuickSortRP.

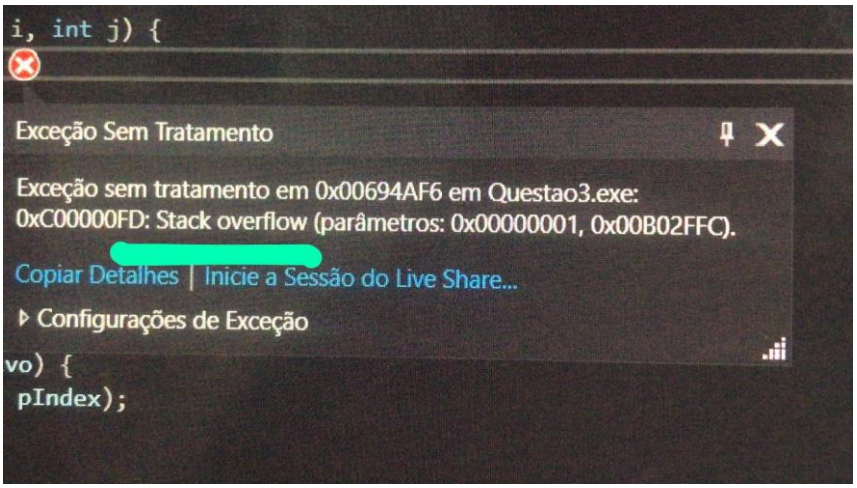
Os algoritmos QuickSort e QuickSort com Randomização de pivô foram muito bons, para os casos de vetores muito pequenos, como todos os outros, se tornando O(1), para os casos médios teve complexidade muito próxima ao do MergeSort, O(nlog(n)), nos casos em que o vetor é

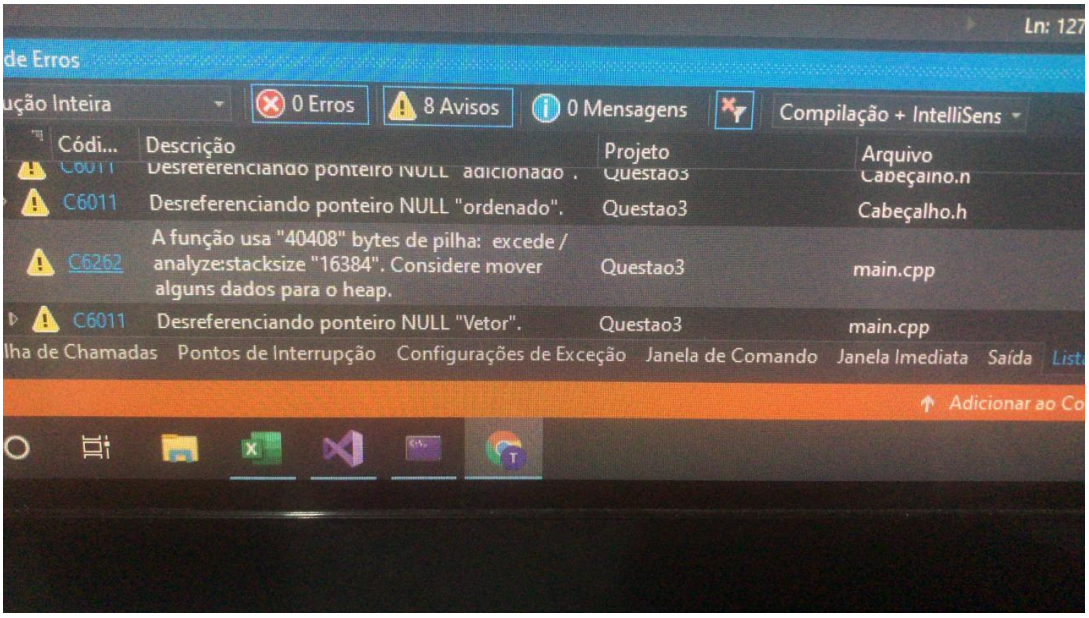
muito grande e ordenado de forma decrescente, teoricamente, o QuickSort é pior chegando ao caso de O(n^2), que pôde ser dirimido com o QuickSort com Randomização de Pivô, porém, na prática teve desempenho próximo em questões de média e mediana das 30 repetições dos vetores.

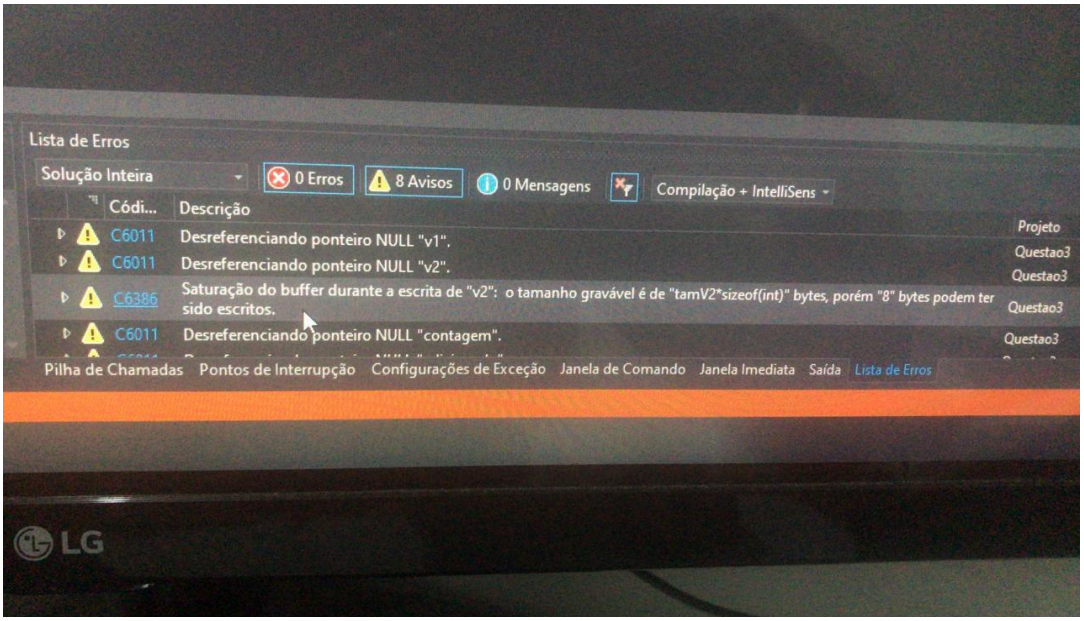
Para o caso do algoritmo de ordenação, apesar de na sua estrutura necessitar de outros vetores alocados na memória, o que ocupa mais espaço, foi o melhor de todos para os casos médios e maiores com relação ao tamanho do vetor, tornando quase que O(1) para todos os casos, de tamanhos 10^1 até 10^5.

De forma geral, todos os algoritmos para vetores de tamanhos muito pequenos, se tornam O(1), tanto é que tive que usar o tempo em milissegundos mais algumas casas decimais a mais para poder fazer as verificações de processamento, tempo e desempenho. Já, com vetores muito grande, os algoritmos SelectionSort, BubbleSort e InsertionSort se tornaram muito custosos, levando em torno de quase 10 minutos ou mais para fazer as ordenações de um vetor decrescente, já os algoritmos MergeSort e QuickSort, foram algoritmos que responderam melhor aos Vetores um pouco maiores, em 10^3 e 10^4, talvez em virtude de sua estratégia de ordenação, já o QuickSort com Randomização de Pivô teve algumas diferenças com relação aos QuickSort e MergeSort, porém, para os casos aqui estudados não teve tanta diferença, já o algoritmo CountingSort, se destacou amplamente para os casos de vetores medianos e muito grandes em torno de 10^3, 10^4, 10^5 e um pequeno teste que fiz com o 10^6, em que nenhum outro algoritmo conseguiu alcançar seus tempos de execução, média e mediana.

Making of


i, int j) {
Exceção Sem Tratamento
Exceção sem tratamento em 0x00694AF6 em Questao3.exe:
0xC00000FD: Stack overflow (parâmetros: 0x00000001, 0x00B02FFC).
Copiar Detalhes | Inicie a Sessão do Live Share...
Configurações de Exceção
vo) {
pIndex);


Ln: 127
de Erros
ução Inteira
0 Erros
8 Avisos
0 Mensagens
Compilação + IntelliSens
Códi...
Descrição
Projeto
Arquivo
C6011
Desreferenciando ponteiro NULL "ordenado".
Questao3
Cabeçalho.h
C6262
A função usa "40408" bytes de pilha: excede / analyze:stacksize "16384". Considere mover alguns dados para o heap.
Questao3
main.cpp
C6011
Desreferenciando ponteiro NULL "Vetor".
Questao3
main.cpp
lha de Chamadas
Pontos de Interrupção
Configurações de Exceção
Janela de Comando
Janela Imediata
Saída
Adicionar ao Co


Lista de Erros
Solução Inteira
0 Erros
8 Avisos
0 Mensagens
Compilação + IntelliSens
Códi...
Descrição
Projeto
C6011
Desreferenciando ponteiro NULL "v1".
Questao3
C6011
Desreferenciando ponteiro NULL "v2".
Questao3
C6386
Saturação do buffer durante a escrita de "v2": o tamanho gravável é de "tamV2*sizeof(int)" bytes, porém "8" bytes podem ter sido escritos.
Questao3
C6011
Desreferenciando ponteiro NULL "contagem".
Questao3
Pilha de Chamadas
Pontos de Interrupção
Configurações de Exceção
Janela de Comando
Janela Imediata
Saída
Lista de Erros
LG