Instituto Federal de Educação, Ciência e Tecnologia da Paraíba

Curso: Análise e Desenvolvimento de Sistemas

Disciplina: Análise de Algoritmos

Docente: José Gomes Lopes Filho

Aluno: André Felipe Ferraz da Silva

Análise Experimental de Algoritmos

Monteiro – PB

2021

SUMÁRIO

1. Introdução

Este trabalho consiste em analisar o desempenho de um algoritmo de ordenação através de análises assintóticas, número de comparações de chaves, número de movimentações de registros e tempo total gasto de execução. Para a análise, foram usados vetores inteiros com tamanho: 10, 100, 1.000, 10.000, 100.000 e 1.000.000. Além disso, os vetores foram testados em arranjos ordenados, inversamente ordenados, quase ordenados e aleatórios, sendo testado 1 vez para cada algoritmo em vetores ordenados, 1 vez para cada algoritmo em vetores inversamente ordenados e 14 vezes para cada algoritmo em vetores quase ordenados e aleatórios.

2. Algoritmos

2.1. Bubble Sort

É um algoritmo que percorre várias vezes um mesmo vetor com a intenção de ordená-lo. Seu custo é de $(N + 2)2$ por isso, sua complexidade é $O(n^2)$ no pior e médio caso e $O(n)$ no melhor caso. O método Bubble Sort usado neste trabalho foi um Bubble Sort melhorado.

2.1.1. Implementação do Bubble Sort

```
public void execute(int[] vetor) {
    boolean troca = true;
    trocaChave = 0;
    testeChave = 0;
    iniciar();
    for (int i = 1; (i < vetor.length) && (troca); i++) {
        troca = false;
        for (int j = 0; j < vetor.length - 1; j++) {
            testeChave++;
            if (vetor[j] > vetor[j + 1]) {
                int aux;
                aux = vetor[j];
                vetor[j] = vetor[j + 1];
                vetor[j + 1] = aux;
                troca = true;
                trocaChave+=2;
            }
        }
    }
    finalizar();
}
```

2.2. Insertion Sort

É um algoritmo que, para ordenar um vetor, faz a inserção de um elemento por vez. Este algoritmo e bastante eficiente principalmente em casos pequenos, ele vai ordenando o vetor aos poucos e, ao ordenar aquela parte, introduz mais um elemento do vetor para ordena-lo. Seu custo é de $\frac{n^2}{2} - \frac{n}{2}$ por isso, sua complexidade é $O(n^2)$ no pior e médio caso e $O(n)$ no

melhor caso.

2.2.1. Implementação do Insertion Sort

```java
public int[] execute(int[] v) {
    iniciar();
    testeChave = 0;
    trocaChave = 0;
    for (int i = 1; i < v.length; i++) {
        int aux = v[i];
        int j = i;
        while ((j > 0) && (v[j - 1] > aux)) {
            v[j] = v[j - 1];
            j -= 1;
            trocaChave++;
            testeChave++;
        }
        testeChave++;
        trocaChave++;
        v[j] = aux;
    }
    finalizar();
    return v;
}
```

2.3. Quick Sort

É um algoritmo que, através da fragmentação do vetor e da escolha de um pivô como meta comparativa para fazer a troca de chaves, ordena todas as fragmentações criadas apartir do vetor. Seu custo é de $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ por isso, utilizando o método mestre, vemos que sua complexidade é $O(n^2)$ no pior caso e $O(n\ log\ n)$ no médio e no melhor caso.

2.3.1. Implementação do Quick Sort

```java
public void execute(int[] v, int i, int f) {

    int n = (int) (Math.random() * ((f) - i + 1) + i);
    int pivo = v[n];
    int e = i;
    int d = f;
    while (e <= d) {
        while (e <= f && v[e] < pivo) {
            e++;
            testeChave++;
        }
        testeChave++;

        while (d >= i && v[d] > pivo) {
            d--;
            testeChave++;

        }
        testeChave += 2;
        if (e <= d) {
            int aux = v[e];
            v[e] = v[d];
            v[d] = aux;
            e++;
            d--;
            trocaChave += 2;
        }
    }
    if (e < f)
        execute(v, e, f);
    if (d > i)
        execute(v, i, d);
}
```

2.4. Selection Sort

É um algoritmo baseado em percorrer o vetor a procura do menor elemento da vez e colocá-lo no começo do vetor, o que faz que ele seja muito pouco eficiente. Seu custo é de $\frac{n^2}{2} - \frac{n}{2}$ por isso sua complexidade é $O(n^2)$ no pior, médio e melhor caso.

2.4.1. Implementação do Selection Sort

```
public  void execute(int[] vetor, int tamanho) {
    iniciar();
    int i, j, menor, aux;

    for (i = 0; i < tamanho - 1; ++i) {
        menor = i;
        for (j = i + 1; j < tamanho; ++j) {
            testeChave++;
            if (vetor[j] < vetor[menor]) {
                menor = j;
            }
        }
        aux = vetor[i];
        vetor[i] = vetor[menor];
        vetor[menor] = aux;
        trocaChave+=2;
    }
    finalizar();
}
```

2.5. Merge Sort

É um algoritmo que divide o vetor em várias partes até chegar em um valor mínimo, ele começa a ordenar essas partes e vai juntando ao final todas as partes ordenadas, isso permite que este algoritmo seja bastante efetivo, mas tem um alto consumo de memória. Seu custo é de $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ por isso, utilizando o método mestre vemos que sua complexidade é sempre $O(n\ log\ n)$ no melhor, médio e no pior caso.

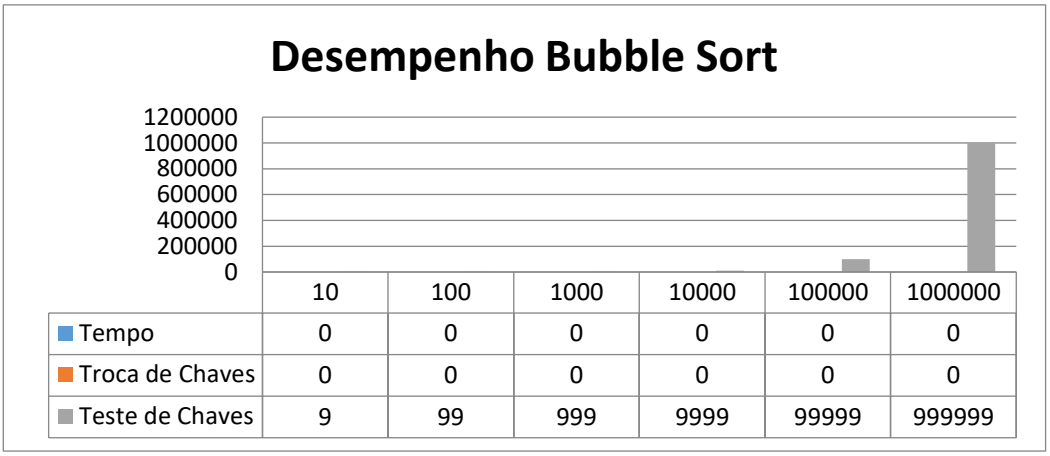
2.5.1. Implementação do Merge Sort

```java
/**
 * Método que ordena um vetor de elementos inteiros, utilizando o algoritmo
 * do Merge Sort.
 *
 * @param inicio - Posição inicial do vetor.
 * @param fim  - Posição final do vetor.
 * @param vetor - Vetor de números inteiros.
 */
private static void mergeSortRecursivo(int inicio, int fim, int[] vetor) {
  /* Se o inicio for menor que o fim menos 1, significa que tem elementos
       dentro do vetor. */
  if(inicio < fim - 1) {
    // Guarda a posição do meio do vetor.
    int meio = (inicio + fim) / 2;

    /* Chama este método recursivamente, indicando novas posições do
           inicio e fim do vetor. */
    mergeSortRecursivo(inicio, meio, vetor);

    /* Chama este método recursivamente, indicando novas posições do
           inicio e fim do vetor. */
    mergeSortRecursivo(meio, fim, vetor);

    // Chama o método que intercala os elementos do vetor.
    intercala(vetor, inicio, meio, fim);
  }
}

private static  void intercala(int[] vetor, int inicio, int meio, int fim) {
  /* Vetor utilizado para guardar os valors ordenados. */
  int novoVetor[] = new int[fim - inicio];
  /* Variavel utilizada para guardar a posicao do inicio do vetor. */
  int i = inicio;
  /* Variavel utilizada para guardar a posição do meio do vetor. */
  int m = meio;
  /* Variavel utilizada para guarda a posição onde os
    valores serão inseridos no novo vetor. */
  int pos = 0;

  /* Enquanto o inicio não chegar até o meio do vetor, ou o meio do vetor
    não chegar até seu fim, compara os valores entre o inicio e o meio,
    verificando em qual ordem vai guarda-los ordenado.*/
  while(i < meio && m < fim) {
    /* Se o vetor[i] for menor que o vetor[m], então guarda o valor do
      vetor[i] pois este é menor. */
    testeChave++;
    if(vetor[i] <= vetor[m]) {
      novoVetor[pos] = vetor[i];
      pos = pos + 1;
      i = i + 1;
      trocaChave++;
    // Senão guarda o valor do vetor[m] pois este é o menor.
    } else {
      novoVetor[pos] = vetor[m];
      pos = pos + 1;
      m = m + 1;
      trocaChave++;
    }
  }

    // Adicionar no vetor os elementos que estão entre o inicio e meio,
    // que ainda não foram adicionados no vetor.
    while(i < meio) {
      novoVetor[pos] = vetor[i];
      pos = pos + 1;
      i = i + 1;
      trocaChave++;
    }

    // Adicionar no vetor os elementos que estão entre o meio e o fim,
    // que ainda não foram adicionados no vetor.
    while(m < fim) {
      novoVetor[pos] = vetor[m];
      pos = pos + 1;
      m = m + 1;
      trocaChave++;
    }

    // Coloca no vetor os valores já ordenados.
    for(pos = 0, i = inicio; i < fim; i++, pos++) {
      vetor[i] = novoVetor[pos];
      trocaChave++;
    }
  }


public static void execute(int vetor[], int inicio, int fim) {
    trocaChave = 0;
    testeChave = 0;
    int meio;
    if (inicio < fim) {
        meio = (inicio + fim) / 2;
        mergeSortRecursivo(inicio, meio,vetor);
        mergeSortRecursivo(meio+1, fim,vetor);
        intercala(vetor, inicio, meio, fim);
    }
}
```

3. Análise dos Algoritmos

3.1. Vetores Ordenados

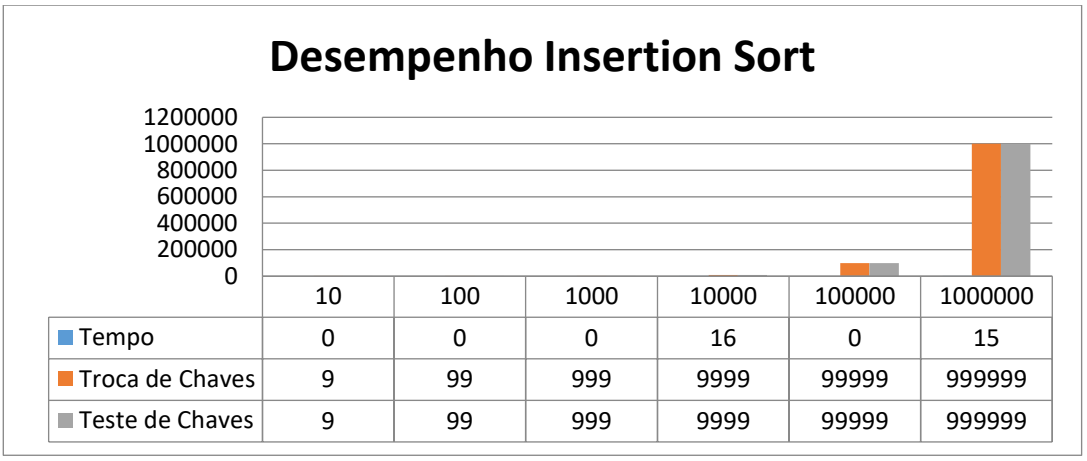
3.1.1. Desempenho Bubble Sort



	10	100	1000	10000	100000	1000000
Tempo	0	0	0	0	0	0
Troca de Chaves	0	0	0	0	0	0
Teste de Chaves	9	99	999	9999	99999	999999

Esse gráfico descreve o desempenho do método Bubble Sort melhorado para ordenar vetores com 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de elementos já ordenados. Usando vetores já ordenados o Bubble Sort melhorado sempre fica com sua taxa de tempo de processamento em 0 milissegundos e seus testes de chaves sempre vão ser o tamanho do vetor – 1.

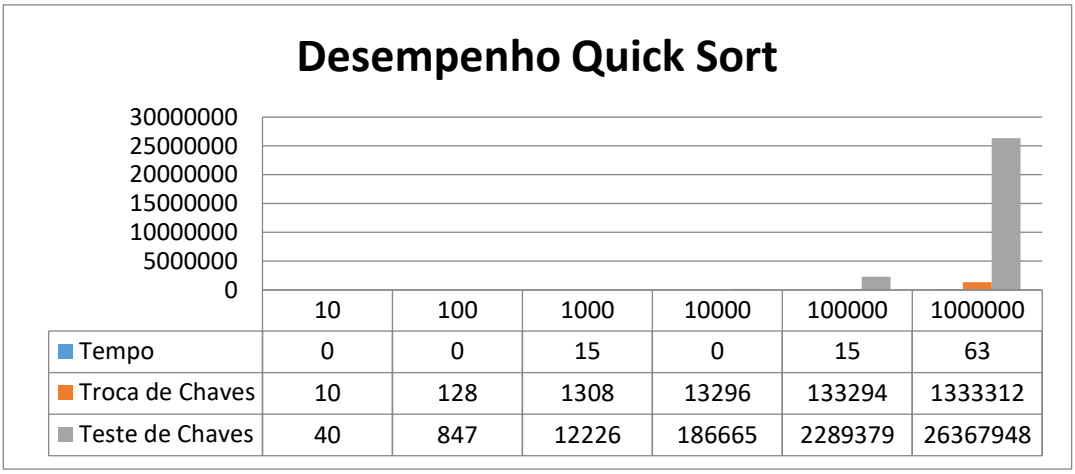
3.1.2. Desempenho Insertion Sort



	10	100	1000	10000	100000	1000000
Tempo	0	0	0	16	0	15
Troca de Chaves	9	99	999	9999	99999	999999
Teste de Chaves	9	99	999	9999	99999	999999

Esse gráfico descreve o desempenho do método Insertion Sort para ordenar vetores com 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de elementos já ordenados. Usando vetores já ordenados o Insertion Sort mantem sua taxa de tempo de processamento abaixo de 20 milissegundos, seus testes e suas trocas de chaves sempre vão ser o tamanho do vetor – 1.

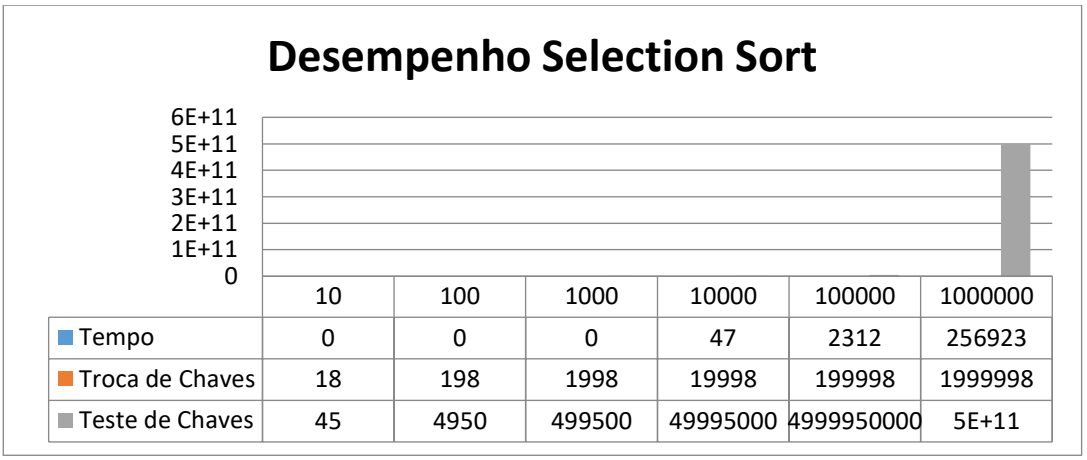
3.1.3. Desempenho Quick Sort



	10	100	1000	10000	100000	1000000
Tempo	0	0	15	0	15	63
Troca de Chaves	10	128	1308	13296	133294	1333312
Teste de Chaves	40	847	12226	186665	2289379	26367948

Esse gráfico descreve o desempenho do método Quick Sort para ordenar vetores com 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de elementos já ordenados. Este metódo de ordenação, mesmo usando vetores já ordenados, realiza muitas trocas e testes de chaves, mas isso não afeta muito em seu tempo de processamento, isso se da pelo fato que o Quick Sort trabalha fragmentando o vetor.

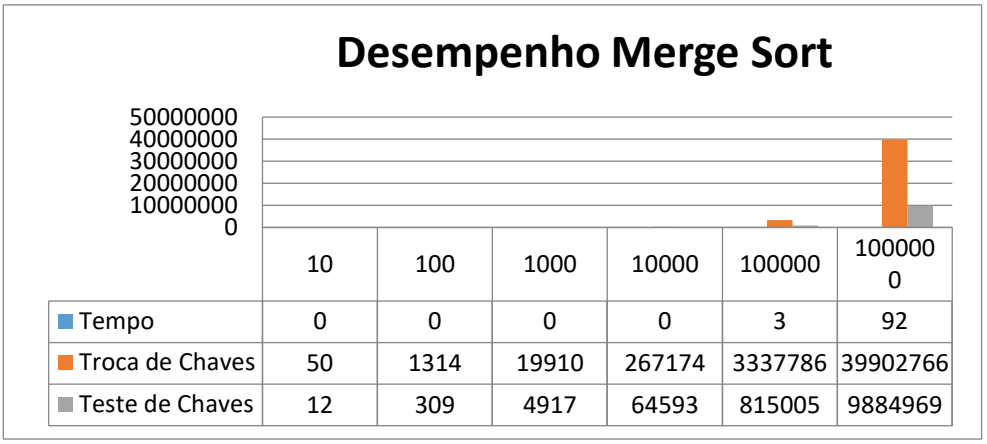
3.1.4. Desempenho Selection Sort



	10	100	1000	10000	100000	1000000
Tempo	0	0	0	47	2312	256923
Troca de Chaves	18	198	1998	19998	199998	1999998
Teste de Chaves	45	4950	499500	49995000	4999950000	5E+11

Esse gráfico descreve o desempenho do método Selection Sort para ordenar vetores com 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de elementos já ordenados. Dos metódos de ordenação usados em vetores ordenados, o Selection Sort foi o que teve o pior desempenho disparado, pois ele, em qualquer situação, sempre vai ser $O(N^2)$. Só aprensentou baixo tempo de processamento para entradas pequenas mas, mesmo assim, ele realiza trocas desnecessárias.

3.1.5. Desempenho Merge Sort

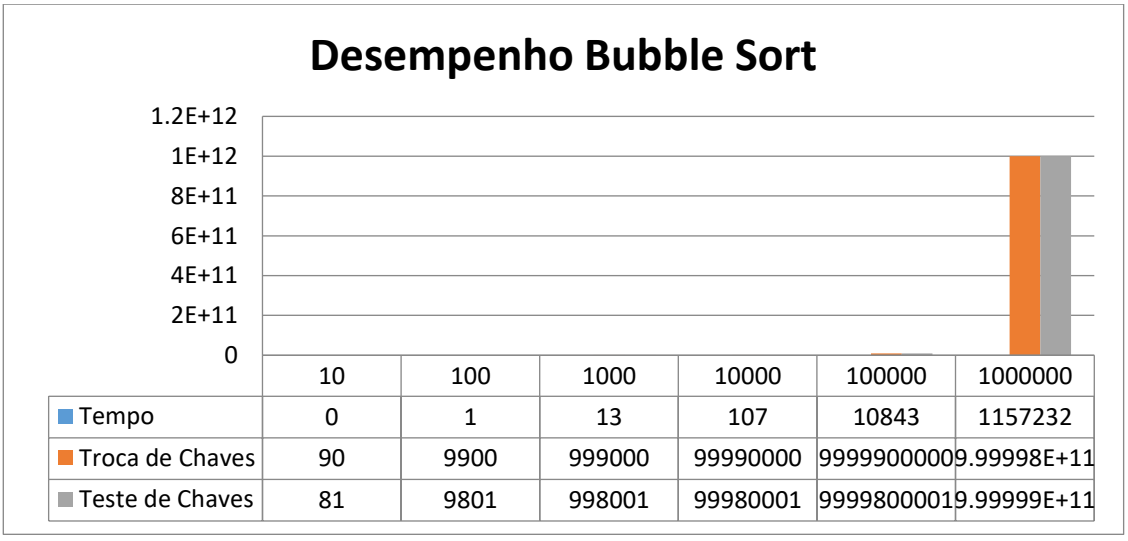


	10	100	1000	10000	100000	1000000
Tempo	0	0	0	0	3	92
Troca de Chaves	50	1314	19910	267174	3337786	39902766
Teste de Chaves	12	309	4917	64593	815005	9884969

Esse gráfico descreve o desempenho do método Merge Sort para ordenar vetores com 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de elementos já ordenados. O Merge Sort se mostrou um algortimo que realiza muitas trocas de chaves (Mesmo com o vetor já ordenado) seu tempo de processamento é relativamente baixo, quase se aproximando do tempo do Quick Sort.

3.2. Vetores Ordenados Inversamente

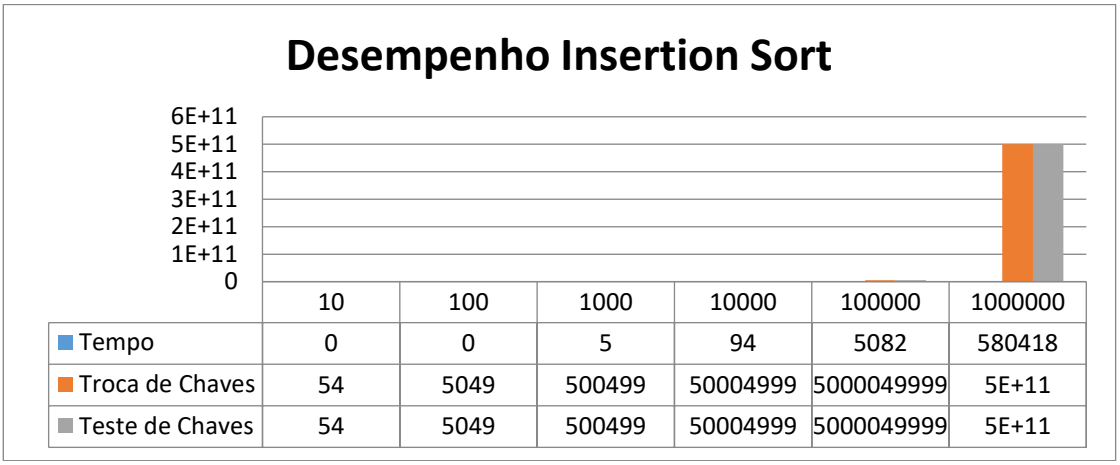
3.2.1. Desempenho Bubble Sort



	10	100	1000	10000	100000	1000000
Tempo	0	1	13	107	10843	1157232
Troca de Chaves	90	9900	999000	99990000	9999900000	9.99998E+11
Teste de Chaves	81	9801	998001	99980001	9999800001	9.99999E+11

Esse gráfico descreve o desempenho do método Bubble Sort para ordenar vetores com 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de elementos ordenados inversamente. Por os vetores estarem inversamente ordenados, o Bubble Sort sempre entrava em seu pior caso, que é $O(N^2)$. Tanto seu tempo de processamento, seus testes de chaves e trocas de chaves alcançaram números altissímos, sendo muito inviável usar este metódo de orndenação para este tipo de problema.

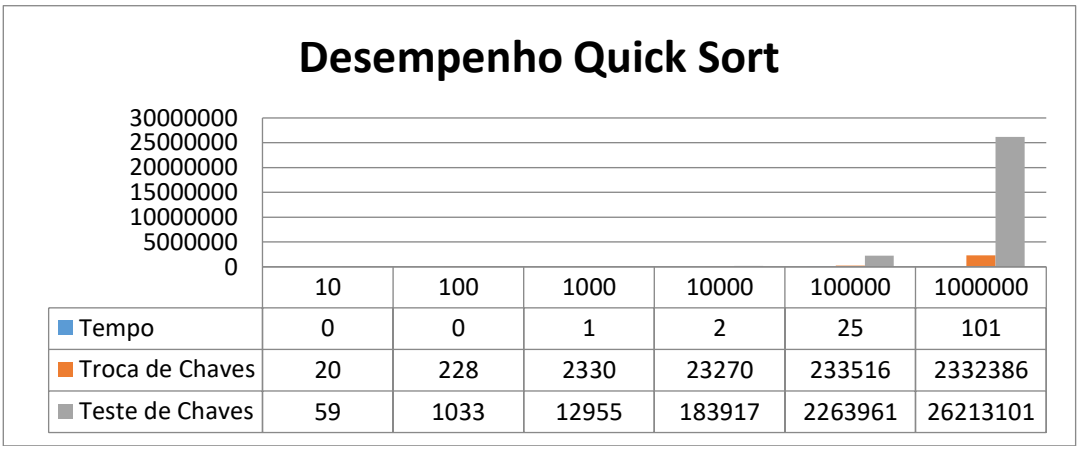
3.2.2. Desempenho Insertion Sort



	10	100	1000	10000	100000	1000000
Tempo	0	0	5	94	5082	580418
Troca de Chaves	54	5049	500499	50004999	5000049999	5E+11
Teste de Chaves	54	5049	500499	50004999	5000049999	5E+11

Esse gráfico descreve o desempenho do método Insertion Sort para ordenar vetores com 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de elementos ordenados inversamente. A mesma coisa que acontece com o Bubble Sort acontece aqui, sempre entra em seu pior caso ($O(N^2)$), mas o Insertion Sort ainda se saiu melhor que o algoritmo anterior, cortando seu tempo de processamento quase pela metade.

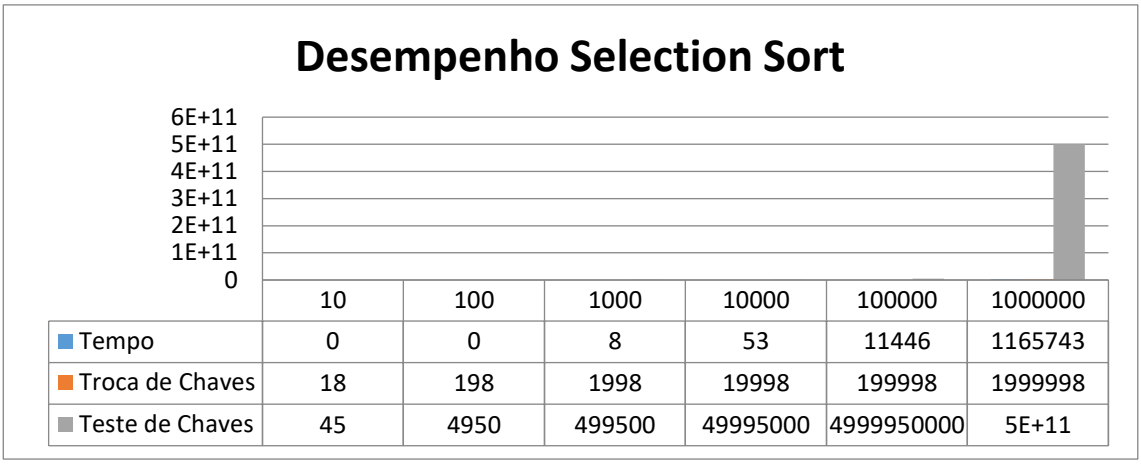
3.2.3. Desempenho Quick Sort



	10	100	1000	10000	100000	1000000
Tempo	0	0	1	2	25	101
Troca de Chaves	20	228	2330	23270	233516	2332386
Teste de Chaves	59	1033	12955	183917	2263961	26213101

Esse gráfico descreve o desempenho do método Quick Sort para ordenar vetores com 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de elementos ordenados inversamente. Comparando com o vetor já ordenado, este metódo de ordenação teve um leve crescimento em seu tempo de processamento.

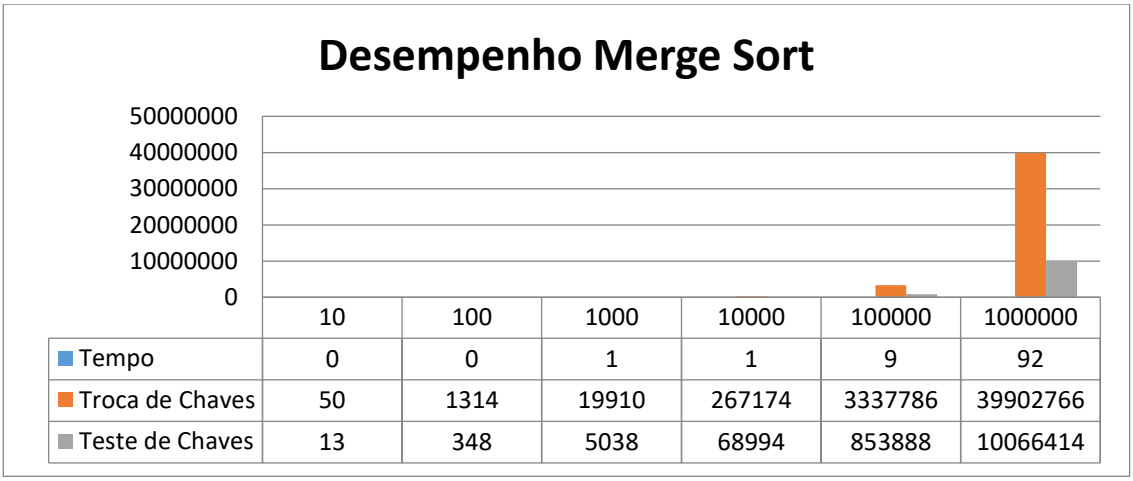
3.2.4. Desempenho Selection Sort



	10	100	1000	10000	100000	1000000
Tempo	0	0	8	53	11446	1165743
Troca de Chaves	18	198	1998	19998	199998	1999998
Teste de Chaves	45	4950	499500	49995000	4999950000	5E+11

Esse gráfico descreve o desempenho do método Selection Sort para ordenar vetores com 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de elementos ordenados inversamente. Não há muito o que falar do Selection Sort, ele sempre terá a mesma quantidade de troca e teste de chaves, apenas seu tempo de processamento que é mudado e, como vemos aqui, seu tempo teve um grande aumento, seu tempo é quase equivalente ao do Bubble Sort.

3.2.5. Desempenho Merge Sort



	10	100	1000	10000	100000	1000000
Tempo	0	0	1	1	9	92
Troca de Chaves	50	1314	19910	267174	3337786	39902766
Teste de Chaves	13	348	5038	68994	853888	10066414

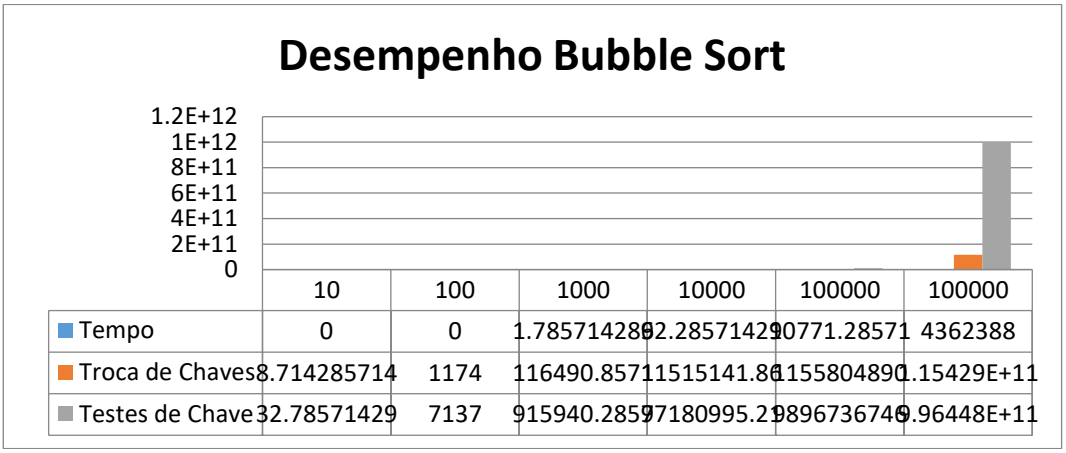
Esse gráfico descreve o desempenho do método Merge Sort para ordenar vetores com 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de elementos ordenados inversamente. O Merge Sort mantem a mesma quantidade de trocas de chaves, mesmo com um número alto de troca de chaves ele teve um tempo de processamento melhor que o do Quick Sort.

3.3. Vetores Quase Ordenados

Para está análise, foi usado o mesmo vetor para todos os metódos de ordenação (5), os metódos foram rodados 14 vezes, o que significa que foram testados 14 tipos de vetores quase ordenados. O que será aprensentado é a média do tempo, troca de chaves e teste de chaves

desses 14 vetores.

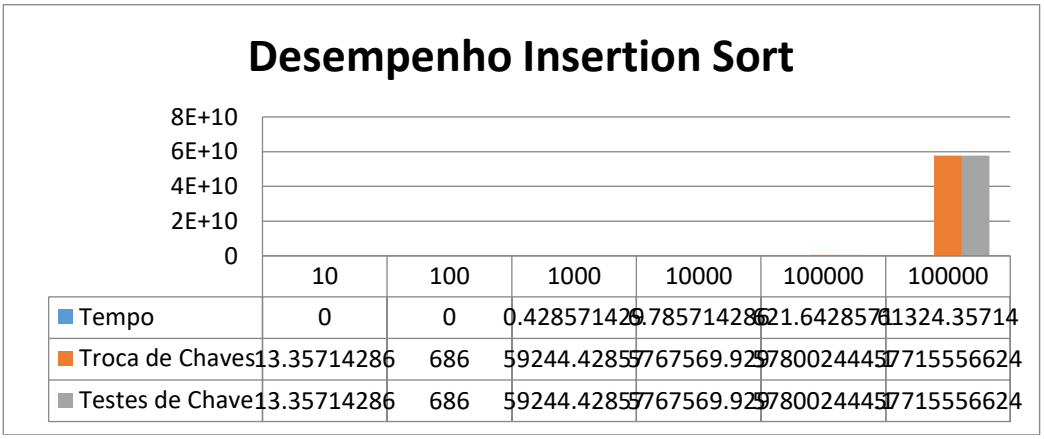
3.3.1. Desempenho Bubble Sort



	10	100	1000	10000	100000	100000
Tempo	0	0	1.785714286	2.285714286	10771.28571	4362388
Troca de Chaves	8.714285714	1174	116490.8571	11515141.86	1155804890	1.15429E+11
Testes de Chave	32.78571429	7137	915940.2857	97180995.29	9896736748	9.96448E+11

O Bubble Sort, como de se esperando, apresentou números bons para entradas pequenas e pessímos números para entradas grandes, mesmo com 10% porcento do vetor desorganizado, o Bubble Sort leva muito tempo e faz muitos teste de chaves para poder ordenar os números.

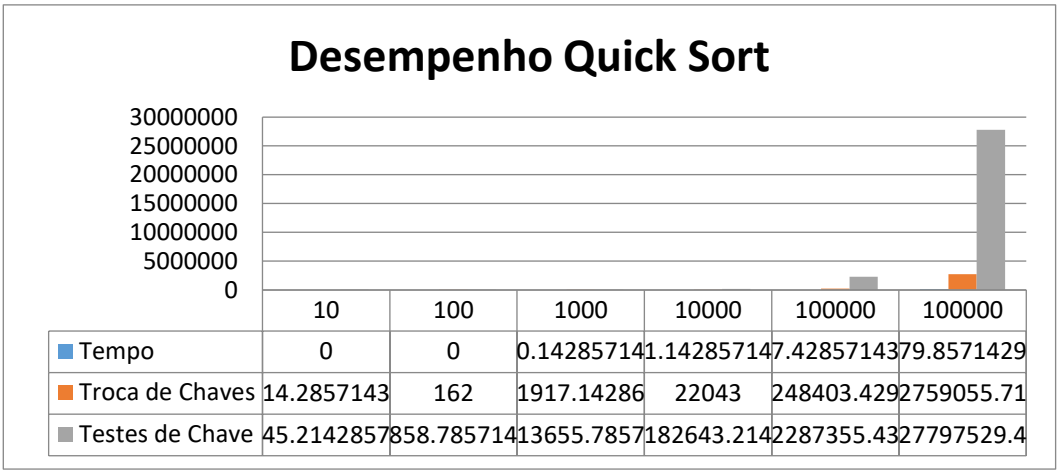
3.3.2. Desempenho Insertion Sort



	10	100	1000	10000	100000	100000
Tempo	0	0	0.428571429	0.785714286	621.6428571	1324.35714
Troca de Chaves	13.35714286	686	59244.42857	5767569.929	578002444	57715556624
Testes de Chave	13.35714286	686	59244.42857	5767569.929	578002444	57715556624

O Insertion Sort continua se saindo melhor que o Bubble Sort, ficando atrás apenas em vetores de tamanho 10. Com vetores grandes seu tempo, trocas e testes de chaves aumentam muito, ficando inviável de se usar para esses problemas.

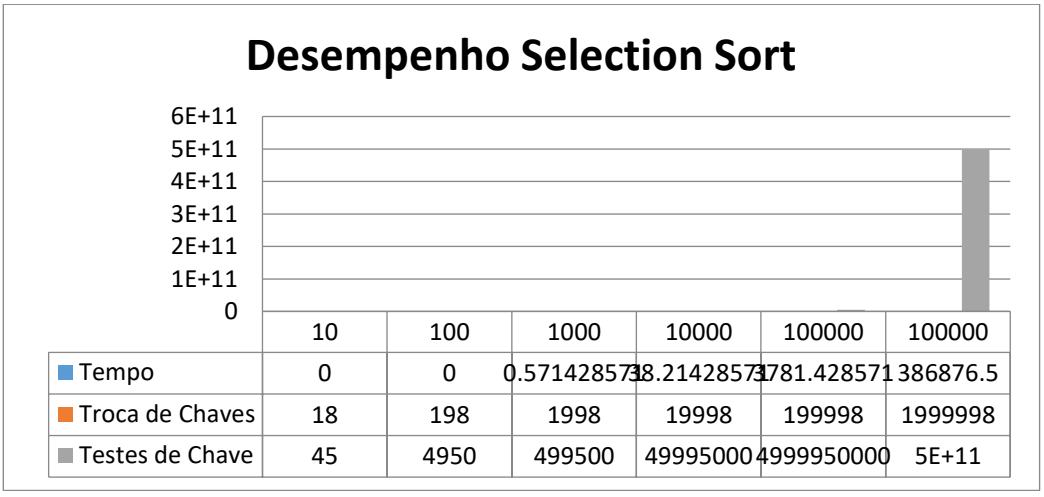
3.3.3. Desempenho Quick Sort



	10	100	1000	10000	100000	100000
Tempo	0	0	0.14285714	1.14285714	7.42857143	79.8571429
Troca de Chaves	14.2857143	162	1917.14286	22043	248403.429	2759055.71
Testes de Chave	45.2142857	858.785714	13655.7857	182643.214	2287355.43	27797529.4

O Quick Sort é o que apresenta os melhores resultados no quesito tempo de processamento, ele ainda continua apresentando um número de teste de chaves bem alto, mas nada que o comprometa.

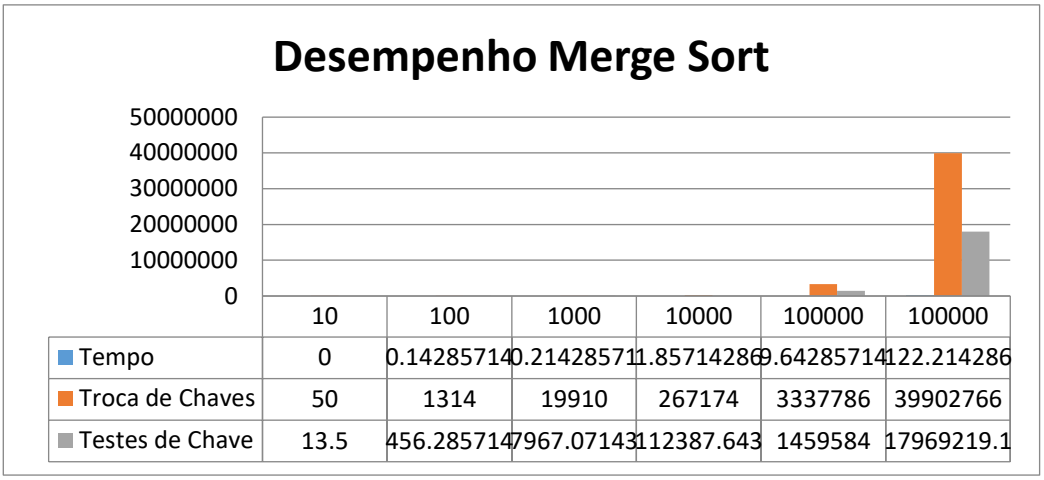
3.3.4. Desempenho Selection Sort



	10	100	1000	10000	100000	100000
Tempo	0	0	0.57142857	18.2142857	1781.42857	386876.5
Troca de Chaves	18	198	1998	19998	199998	1999998
Testes de Chave	45	4950	499500	49995000	4999950000	5E+11

Todos os seus casos são O(N^2). O algoritmo aprensentou ser mais eficais que o Bubble Sort para vetores de tamanho grande.

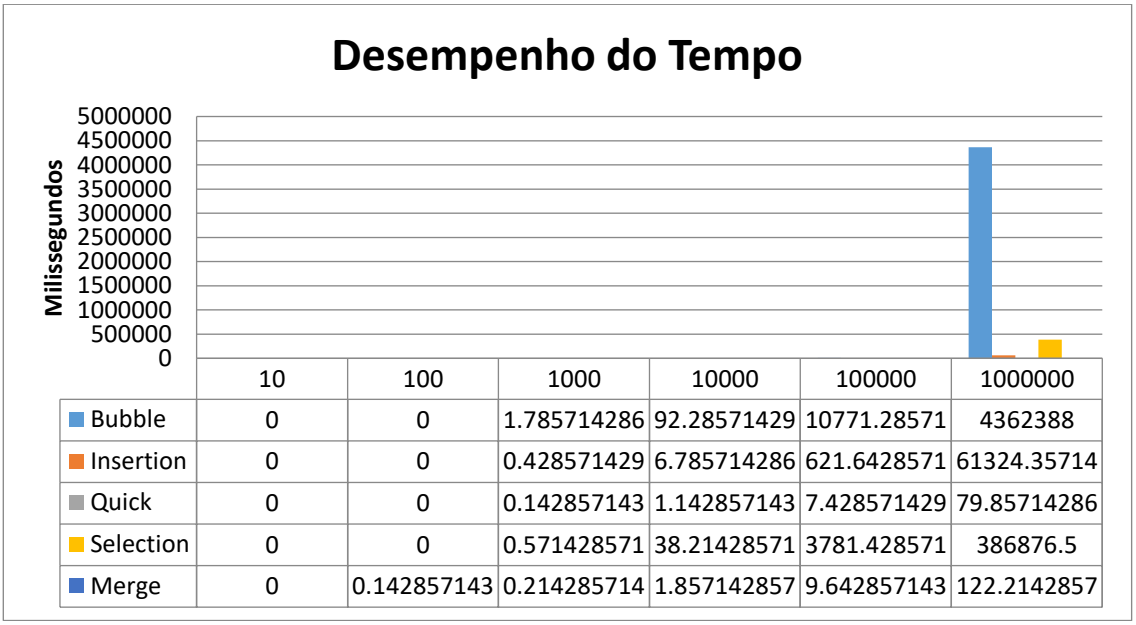
3.3.5. Desempenho Merge Sort



	10	100	1000	10000	100000	100000
Tempo	0	0.14285714	0.21428571	1.85714286	9.64285714	122.214286
Troca de Chaves	50	1314	19910	267174	3337786	39902766
Testes de Chave	13.5	456.285714	7967.07143	112387.643	1459584	17969219.1

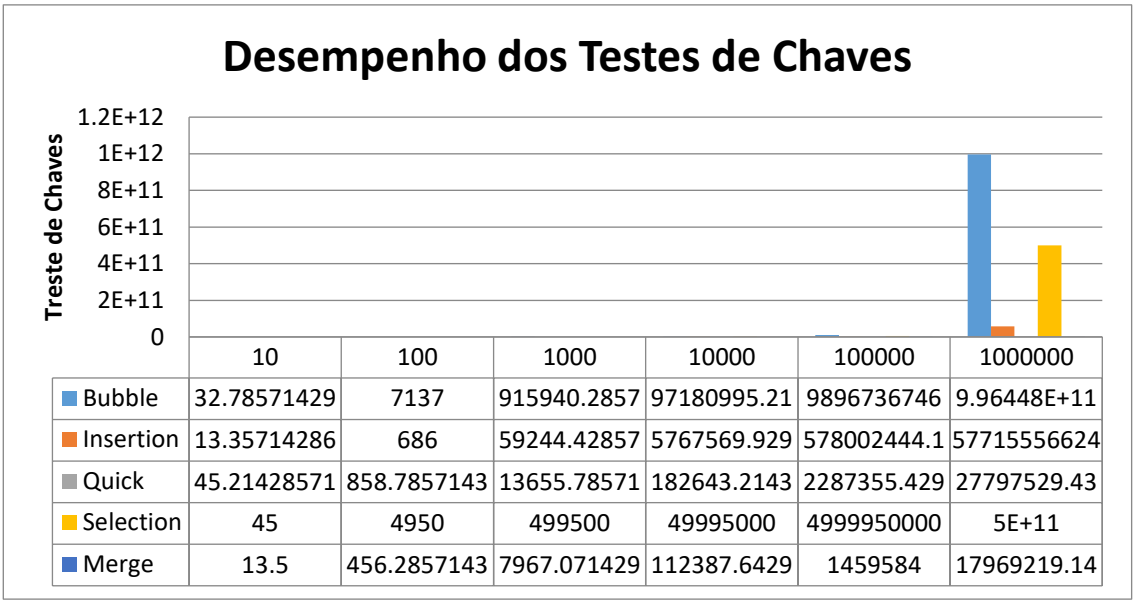
Realiza menos teste de chaves que o Quick Sort, mas realiza mais trocas de chaves, seu tempo é um pouco maior que o do Quick Sort. Sua troca de chaves é fixa para cada tamanho de vetor.

3.3.6. Comparação dos Algoritmos



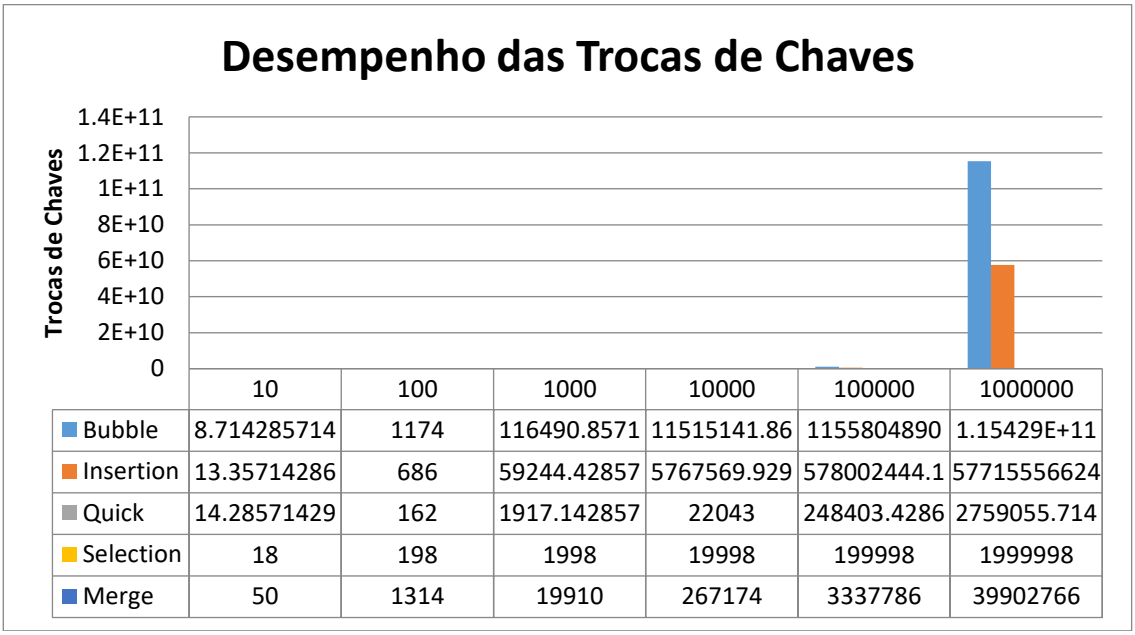
	10	100	1000	10000	100000	1000000
Bubble	0	0	1.785714286	92.28571429	10771.28571	4362388
Insertion	0	0	0.428571429	6.785714286	621.6428571	61324.35714
Quick	0	0	0.142857143	1.142857143	7.428571429	79.85714286
Selection	0	0	0.571428571	38.21428571	3781.428571	386876.5
Merge	0	0.142857143	0.214285714	1.857142857	9.642857143	122.2142857

Quadro 1 – Desempenho do Tempo dos Algoritmos



	10	100	1000	10000	100000	1000000
Bubble	32.78571429	7137	915940.2857	97180995.21	9896736746	9.96448E+11
Insertion	13.35714286	686	59244.42857	5767569.929	578002444.1	57715556624
Quick	45.21428571	858.7857143	13655.78571	182643.2143	2287355.429	27797529.43
Selection	45	4950	499500	49995000	4999950000	5E+11
Merge	13.5	456.2857143	7967.071429	112387.6429	1459584	17969219.14

Quadro 2 – Desempenho dos Testes de Chaves dos Algoritmos



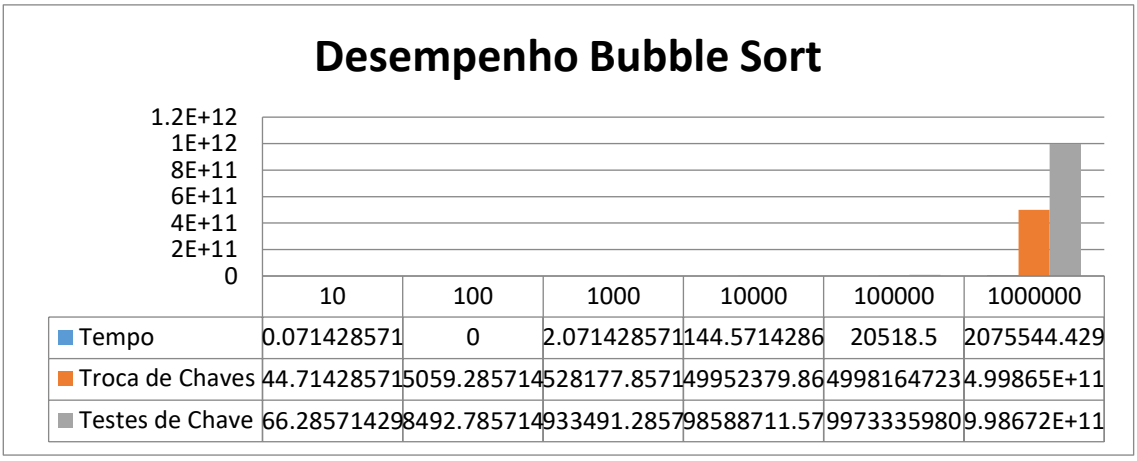
	10	100	1000	10000	100000	1000000
Bubble	8.714285714	1174	116490.8571	11515141.86	1155804890	1.15429E+11
Insertion	13.35714286	686	59244.42857	5767569.929	578002444.1	57715556624
Quick	14.28571429	162	1917.142857	22043	248403.4286	2759055.714
Selection	18	198	1998	19998	199998	1999998
Merge	50	1314	19910	267174	3337786	39902766

Quadro 3 – Desempenho das Trocas de Chaves dos Algoritmos

3.4. Vetores Aleatórios

Para está análise, foi usado o mesmo vetor para todos os metódos de ordenação (5), os metódos foram rodados 14 vezes, o que significa que foram testados 14 tipos de vetores quase ordenados. O que será aprensentado é a média do tempo, troca de chaves e teste de chaves desses 14 vetores.

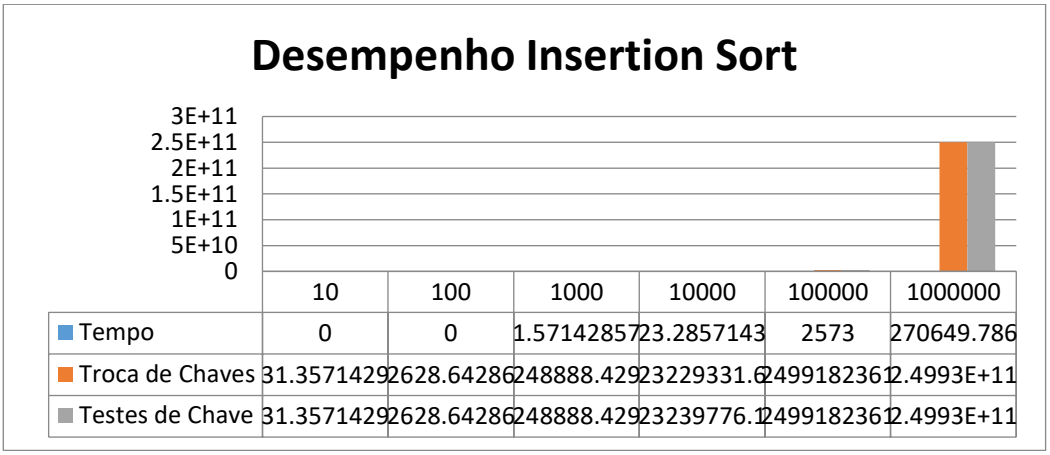
3.4.1. Desempenho Bubble Sort



	10	100	1000	10000	100000	1000000
Tempo	0.071428571	0	2.071428571	144.5714286	20518.5	2075544.429
Troca de Chaves	44.71428571	5059.285714	528177.8571	49952379.86	4998164723	4.99865E+11
Testes de Chave	66.28571429	8492.785714	933491.2857	98588711.57	9973335980	9.98672E+11

Com valores aleatórios no vetor o Bubble Sort continou como o pior algoritmo para tratar vetores deste tipo, com seu tempo de processamento chegando à quase 35 minutos e inumeros testes e trocas de chaves.

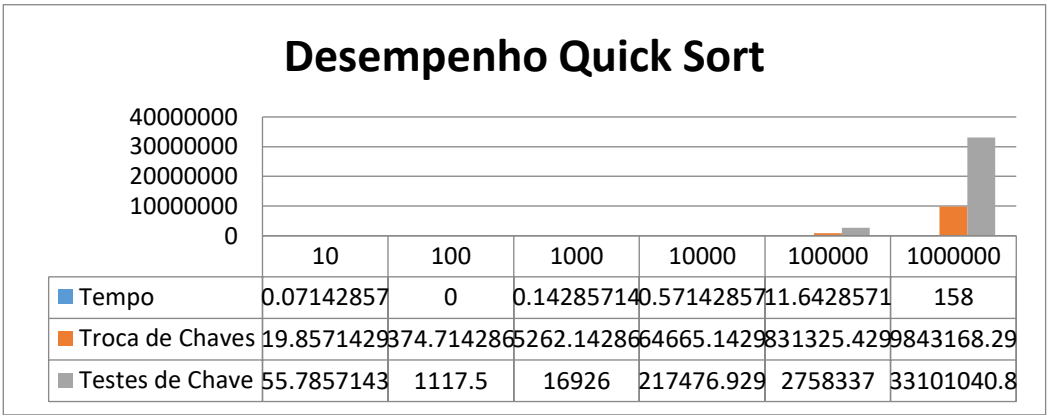
3.4.2. Desempenho Insertion Sort



	10	100	1000	10000	100000	1000000
Tempo	0	0	1.571428571	23.2857143	2573	270649.786
Troca de Chaves	31.35714286	2628.642857	248888.4286	23229331.6	2499182361	2.4993E+11
Testes de Chave	31.35714286	2628.642857	248888.4286	23239776.1	2499182361	2.4993E+11

O Insertion Sort se sai bem melhor que o algoritmo de ordenação anterior, seu tempo pra ordenar um vetor com um milhão de elementos fica na casa dos 4 minutos.

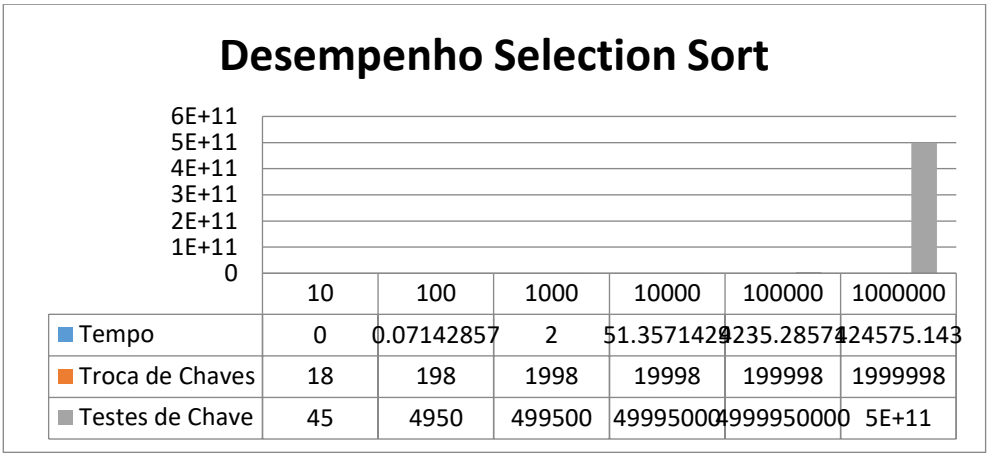
3.4.3. Desempenho Quick Sort



	10	100	1000	10000	100000	1000000
Tempo	0.07142857	0	0.142857143	0.571428571	11.6428571	158
Troca de Chaves	19.85714286	374.7142857	5262.142857	64665.14286	831325.4286	9843168.286
Testes de Chave	55.78571429	1117.5	16926	217476.9286	2758337	33101040.86

Continua apresentando números satisfatórios, tendo numeros de tempos superiores a 10 milisegundo apartir de vetores de 100 mil elementos.

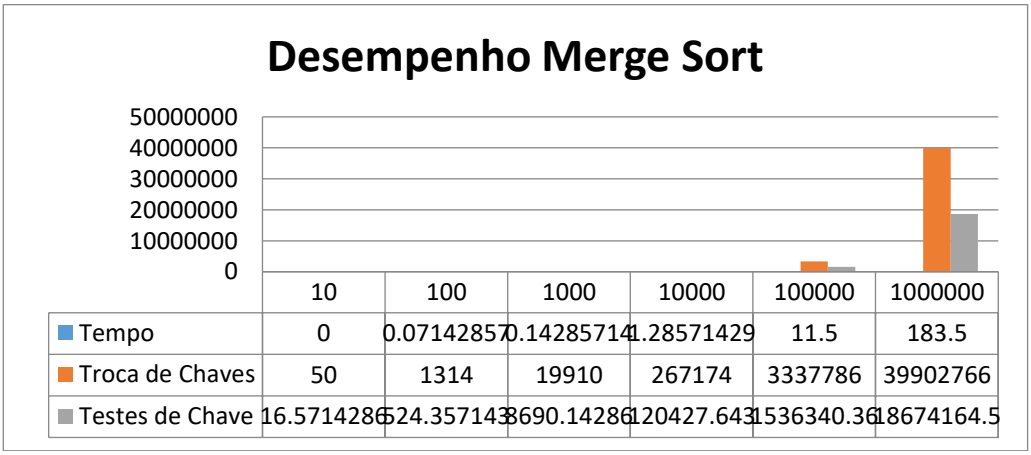
3.4.4. Desempenho Selection Sort



	10	100	1000	10000	100000	1000000
Tempo	0	0.07142857	2	51.35714286	4235.285714	424575.143
Troca de Chaves	18	198	1998	19998	199998	1999998
Testes de Chave	45	4950	499500	49995000	4999950000	5E+11

É caracteristica do Selection Sort ter resutados até que bons para vetores pequenos e resultados ruins para grandes vetores, e aqui, não teve um resultado diferente.

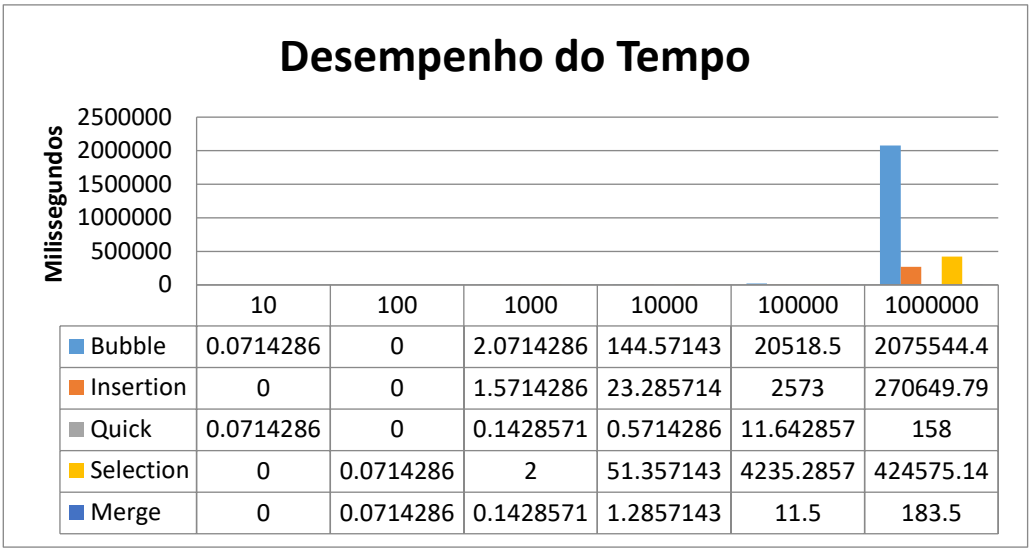
3.4.5. Desempenho Merge Sort



	10	100	1000	10000	100000	1000000
Tempo	0	0.07142857	0.14285714	1.28571429	11.5	183.5
Troca de Chaves	50	1314	19910	267174	3337786	39902766
Testes de Chave	16.57142857	524.3571429	8690.142857	120427.6429	1536340.357	18674164.5

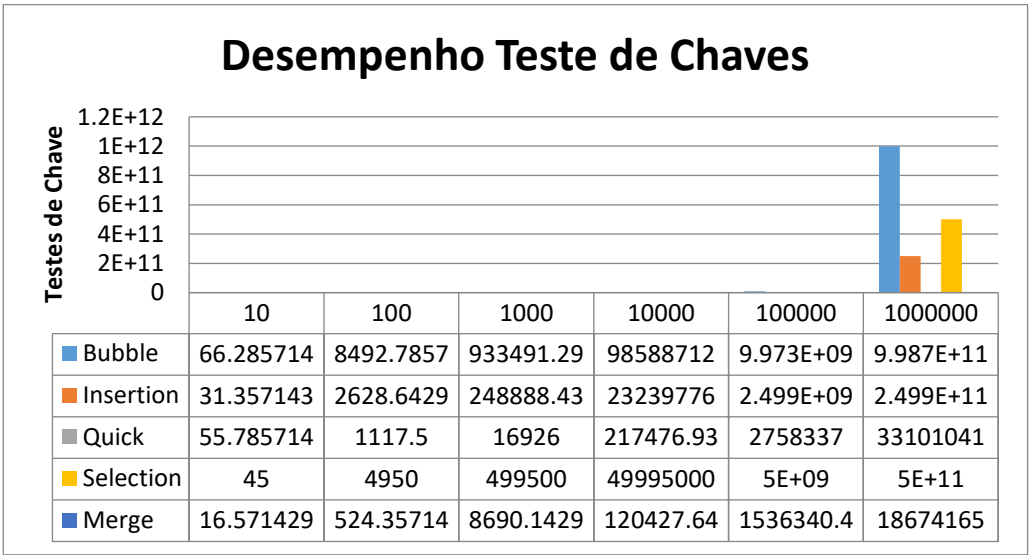
O Merge Sort continua se saindo muito bem, mesmo sendo um algoritmo recursivo, seu tempo com vetores grandes não aumenta tanto assim.

3.4.6. Comparação dos Algoritmos



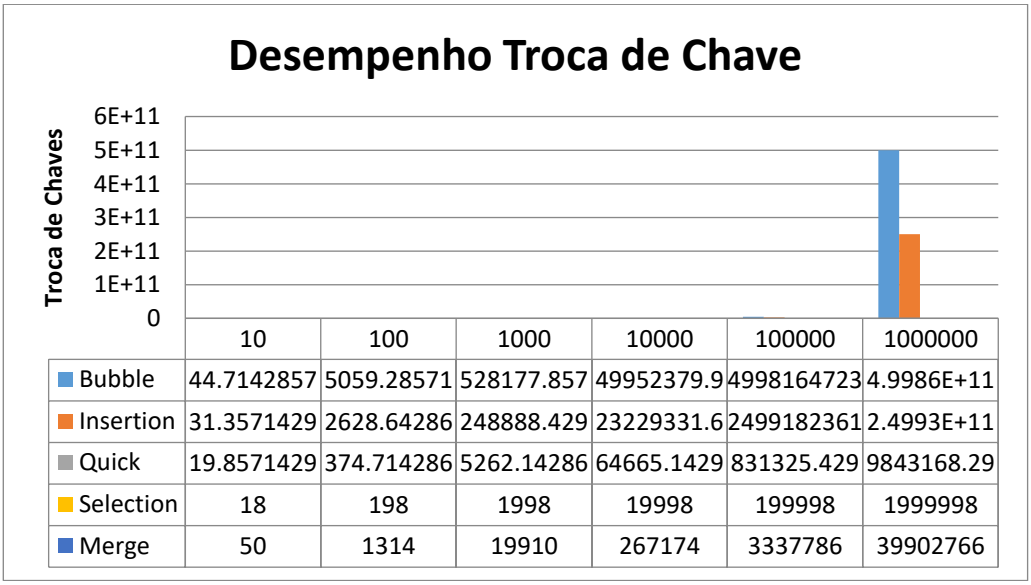
	10	100	1000	10000	100000	1000000
Bubble	0.0714286	0	2.0714286	144.57143	20518.5	2075544.4
Insertion	0	0	1.5714286	23.285714	2573	270649.79
Quick	0.0714286	0	0.1428571	0.5714286	11.642857	158
Selection	0	0.0714286	2	51.357143	4235.2857	424575.14
Merge	0	0.0714286	0.1428571	1.2857143	11.5	183.5

Quadro 1 – Desempenho do Tempo dos Algoritmos



	10	100	1000	10000	100000	1000000
Bubble	66.285714	8492.7857	933491.29	98588712	9.973E+09	9.987E+11
Insertion	31.357143	2628.6429	248888.43	23239776	2.499E+09	2.499E+11
Quick	55.785714	1117.5	16926	217476.93	2758337	33101041
Selection	45	4950	499500	49995000	5E+09	5E+11
Merge	16.571429	524.35714	8690.1429	120427.64	1536340.4	18674165

Quadro 2 – Desempenho dos Testes de Chaves dos Algoritmos



	10	100	1000	10000	100000	1000000
Bubble	44.7142857	5059.28571	528177.857	49952379.9	4998164723	4.9986E+11
Insertion	31.3571429	2628.64286	248888.429	23229331.6	2499182361	2.4993E+11
Quick	19.8571429	374.714286	5262.14286	64665.1429	831325.429	9843168.29
Selection	18	198	1998	19998	199998	1999998
Merge	50	1314	19910	267174	3337786	39902766

Quadro 3 – Desempenho das Trocas de Chaves dos Algoritmos

4. Conclusão

Tanto o Quick Sort quanto o Merge Sort são ótimos algoritmos de ordenação, alcançando tempos baixos para ordenar vetores de grande porte, sendo o Quick o maior realizador de testes chaves entre os dois e o Merge o que realiza mais trocas (Pelo fato de não reconhecer se um vetor já está ordenado). Já os outros três algortimos apresentaram números bons para pequenos vetores, mas com grandes vetores eles não se sairam tão bem, sendo o Bubble Sort o pior entre eles, seguido do Selection Sort e o Insertion Sort.

Comparando todos os algoritmos, o tempo de cada um para vetores pequenos é igual ou quase equivalente, já a partir de 100 mil elementos em um vetor os números começam a ter uma diferença griantante entre o Quick e Merge com os outros três algoritmos, não que o tempo dos dois não aumente, teve sim um aumento de tempo, mas não tão alto quanto os outros três, o aumento é minimo.

Portanto, pelo dados aprensentado neste estudo, o Quick e o Merge Sort são algoritmos bons para se trabalhar tanto com vetores grandes quanto com vetores pequenos, já os outros três só são viáveis de serem usados para vetores pequenos.