Universidade Federal de Lavras
Departamento de Ciência da Computação

Projeto e Análise de Algoritmos
Primeiro Trabalho Prático

Pedro Augusto Duarte de Almeida
Pedro Victor de Sousa Lima

Professor – Eric Fernandes de Mello Araújo
Tutor – Rodrigo Coelho

Lavras
Agosto de 2013

Sumário

1 – Introdução....3
1.1 Especificações técnicas....3
2 – Implementação do trabalho....4
public static void geraVetorOrdenado(int[] array)....4
public static void geraVetorInversamenteOrdenado(int[] array)....4
public static void geraVetorQuaseOrdenado(int[] array)....4
public static void geraVetorAleatorio(int[] array)....4
public static void geraLog(Object metodo, String log)....4
public static void removeLogsAntigos()....5
2.1 Códigos dos métodos utilizados....6
2.1.1 Mergesort....6
2.1.2 Heapsort 7
2.1.3 Quicksort....8
3 - Análise dos Testes....9
3.1 - Mergesort....9
3.1.1 Mergesort - Número de Comparações....10
3.1.2 Mergesort - Número de Atribuições....10
3.1.3 Mergesort - Tempo (ms)....11
3.2 Heapsort....11
3.2.1 Heapsort - Número de Comparações....13
3.2.2 Heapsort - Número de Atribuições....13
3.2.3 Heapsort - Tempo (ms)....14
3.3 Quicksort....14
3.3.1 Quicksort - Número de Comparações....16
3.3.2 Quicksort - Número de Atribuições....17
3.3.2 Quicksort - Tempo (ms)....17
3.4 Comparação dos algoritmos de ordenação....18
4 Conclusão....19
5 Bibliografia....19

1 – Introdução

Este trabalho consiste na implementação e comparação dos algoritmos de ordenação com complexidade O(n log n): Mergesort, Quicksort e Heapsort. Inicialmente, são feitas definições acerca de como o trabalho proposto foi desenvolvido, quais estratégias e estruturas de dados foram utilizadas para as implementações dos métodos além de considerações sobre cada um dos algoritmos definidos e posterior análise individual. A partir da avaliação dos algoritmos será feita uma análise em relação ao tempo de execução, tanto o tempo real quanto o custo computacional para tal. Por fim, são feitas comparações entre os algoritmos nas situações propostas e em seguida, a conclusão deste trabalho.

1.1 Especificações técnicas

- Computador utilizado para os testes:

  Modelo: Dell™ OptiPlex™ 790.
  Processador: Intel® Core™ i5-2400 CPU @ 3.10GHz × 4 6M Cache, up to 3.40 GHz, 4 cores, 64 bits.
  Memória: 8 GiB.

- Sistema Operacional:
  Ubuntu 12.04.2 LTS, codename Precise Pangolin.

- Ambiente de Desenvolvimento do código-fonte:
  Gedit, Sublime Text 2, NetBeans IDE 7.3.

- Linguagem utilizada:
  Java
  Compilador: javac 1.7.0_25 .
  Máquina Virtual: java version "1.7.0_25" , Java(TM) SE Runtime Environment (build 1.7.0_25-b15) , Java HotSpot(TM) Server VM (build 23.25-b01, mixed mode).

- Ambiente de desenvolvimento da documentação:
  LibreOffice Writer.

2 – Implementação do trabalho

A implementação do trabalho se deu de forma sistemática e em etapas. A primeira etapa definida pelo grupo foi a obtenção e o desenvolvimento dos algoritmos de ordenação na medida em que os métodos para a simulação das situações propostas (ordenação em vetor já ordenado, ordenação em vetor inversamente ordenado, ordenação em vetor aleatoriamente ordenado e ordenação em vetor parcialmente ordenado) foram desenvolvidos. A segunda etapa foi realizada no âmbito de definir a melhor forma de automatização dos testes para cada um dos algoritmos de ordenação, além das definições comportamentais: obtenção do tempo de ordenação de cada algoritmo, número de comparações e número de atribuições. A terceira e última etapa da implementação serviu para decidir a saída do processamento de forma inteligente e prática, para tal, ficou decidido que o formato de saída seria o **CSV** (*Comma Separated Values*) por sua simplicidade, e aceitação na maioria das ferramentas de planilhas eletrônicas, além de ser um formato de fácil entendimento e de ser fácil gerar tabelas a partir do mesmo. A implementação das situações propostas está definida como métodos que executam cada uma das situações e outros métodos para auxílio no trato dos arquivos de logs. Há um método que gera um log dos testes para posterior análise e outro para que cada vez que o programa é executado os logs antigos sejam removidos.

public static void geraVetorOrdenado(int[] array)

Método que recebe um vetor e o preenche de forma ordenada, conforme a repetição.

public static void geraVetorInversamenteOrdenado(int[] array)

Método que recebe um vetor e o preenche de forma inversamente ordenada.

public static void geraVetorQuaseOrdenado(int[] array)

Método que recebe um vetor e o preenche de forma aproximadamente ordenada. Para fins de consideração, decidimos que o vetor estaria aproximadamente ordenado caso 50% de sua ocupação estivesse ordenada e os outros 50% restantes estivesse aleatoriamente ordenado. Para tal, iniciamos a execução preenchendo o vetor ordenadamente até o meio, criamos uma lista auxiliar para comportar os valores que estariam do meio até o final e utilizamos o método **shuffle** da Classe **Collections** para embaralhar os valores na lista auxiliar, sendo que posterior randomicidade de valores pré-definidos retornamos a preencher o vetor, garantindo a não repetição de valores.

public static void geraVetorAleatorio(int[] array)

Método que recebe um vetor e o preenche de forma aleatoriamente ordenada. Para tal, criamos uma lista auxiliar para comportar os valores a preencherem o vetor, utilizamos o método **shuffle** da Classe **Collections** para embaralhar os valores na lista auxiliar, sendo que posterior randomicidade de valores pré-definidos preenchemos o vetor com os valores randômicos, garantindo a não repetição de valores.

public static void geraLog(Object metodo, String log)

Método que recebe um objeto de um dos métodos de ordenação utilizados no momento apenas para o fim de identificar qual é o método de ordenação está sendo analisado em um determinado momento, e uma String contendo os logs a serem impressos em arquivo.

public static void removeLogsAntigos()

Método que remove todos os logs pré-existentes na pasta de logs.

```java
    /*Método que gera preenche um vetor passado como parâmetro de forma ordenada*/
    public static void geraVetorOrdenado(int[] array) {
        for (int i = 0; i < array.length; i++) {
            array[i] = i;
        }
    }

    /*Método que gera preenche um vetor passado como parâmetro de forma inversamente ordenada*/
    public static void geraVetorInversamenteOrdenado(int[] array) {
        int k = array.length;
        for (int i = 0; i < array.length; i++) {
            array[i] = k;
            k--;
        }
    }

    /*Método que preenche um vetor passado como parâmetro de forma aproximadamente ordenada*/
    public static void geraVetorQuaseOrdenado(int[] array) {

        /*Preenchimento do vetor ordenadamente até a metade*/
        int i = 0;
        for (; i < array.length / 2; i++) {
            array[i] = i;
        }

        /*Criação de uma lista que comportará os números a serem embaralhados*/
        ArrayList<Integer> list = new ArrayList<Integer>();

        /*Adição de números na lista*/
        int j = i;
        for (; j < array.length; j++) {
            list.add(j);
        }

        /*Embaralhamento de números*/
        Collections.shuffle(list);
    /*Adição de números embaralhados no vetor inicial*/
    for (j = 0; i < array.length; i++) {
        array[i] = list.get(j);
        j++;
    }
}

/*Método que gera preenche um vetor passado como parâmetro de forma aleatória*/
public static void geraVetorAleatorio(int[] array) {
    ArrayList<Integer> list = new ArrayList<Integer>();
    for (int i = 0; i < array.length; i++) {
        list.add(i);
    }
    Collections.shuffle(list);
    for (int i = 0; i < list.size(); i++) {
        array[i] = list.get(i);
    }
}

/*Método que gera um log em csv*/
public static void geraLog(Object metodo, String log) {
    String nomemetodo = (metodo.getClass().getName().replaceAll("([a-zA-Z]+\\.)([a-zA-Z]+)", "$2"));
    try {
        FileWriter x = new FileWriter("./logs/" + nomemetodo + ".csv", true);
        log += "\n\r";
        x.write(log);
        x.close();
    } catch (Exception e) {
    }
}
  /*Método que remove os logs pre-existentes*/
  public static void removeLogsAntigos() {

      File dir = new File("./logs");
      String[] allFiles = dir.list();

      for (String file : allFiles) {
          new File("./logs/" + file).delete();
      }
  }
```

Para mais organização, os algoritmos foram estruturados em classes três classes distintas, Quicksort, Mergesort e Heapsort, cada classe com atributos definidos para contagem de número de comparações e atribuições, além dos métodos de ordenação pertinentes e seus sub-métodos participantes. Além destas três, há a classe **ClassTeste** que é a classe princiál do programa, na qual são feitas todas as instâncias e chamadas dos métodos de ordenação, testes e geração de logs de resultado.

A realização dos testes é feita instanciando cada uma das classes que representam os métodos de ordenação, e a partir destas instâncias, são realizados os testes, que consistem em montar vetores com os tamanhos pré-definidos e realizar a ordenação, calculando o tempo em milissegundo e nanosegundo, este último definido para uma maior precisão do tempo real de ordenação. Enquanto os testes são executados, jogamos os resultados em um objeto **StringBuilder** e ao final da execução de todos os testes para todas as situações é gerado um log em csv com as informações obtidas pelo algoritmo.

2.1 Códigos dos métodos utilizados

2.1.1 Mergesort

```
public void mergesort(int[] data, int first, int n) {

    /*Duas atribuições*/
    this.numeroatribuicoes += 2;

    int n1; // Size of the first half of the array
    int n2; // Size of the second half of the array

    /*Uma comparação*/
    this.numerocomparacoes++;

    if (n > 1) {
        // Compute sizes of the two halves

        /*Duas atribuições*/
        this.numeroatribuicoes += 2;

        n1 = n / 2;
        n2 = n - n1;


        mergesort(data, first, n1);      // Sort data[first] through data[first+n1-1]
        mergesort(data, first + n1, n2); // Sort data[first+n1] to the end

        // Merge the two sorted halves.
        merge(data, first, n1, n2);
    }
}
```

2.1.2 Heapsort

```
public void heapsort(int[] num) {

    constructHeap(num);

    /*Uma atribuição*/
    this.numeroatribuicoes++;
    int end = num.length - 1;

    while (end > 0) {
        /*Uma comparação a cada repetição*/
        this.numerocomparacoes++;

        /*Três atribuições a cada repetição*/
        this.numeroatribuicoes += 3;
        int temp = num[0];
        num[0] = num[end];
        num[end] = temp;
        bubbleDown(num, 0, end - 1);
        end--;
    }
}

public void constructHeap(int[] num) {

    /*Duas atribuições*/
    int start = (num.length / 2) - 1; // Starting from last parent
    int end = num.length - 1;
    this.numeroatribuicoes += 2;

    /*Uma comparação a cada iteração*/
    while (start >= 0) {

        this.numerocomparacoes++;

        bubbleDown(num, start, end);

        /*Uma comparação*/
        this.numerocomparacoes++;
        if (start == 0) {
            break;
        }

        /*Uma atribuição*/
        start = start - 1;
        this.numeroatribuicoes++;
    }
}
```

2.1.3 Quicksort

```
public void quick_srt(int array[], int low, int n) {

    /*Duas atribuições*/
    int lo = low;
    int hi = n;
    this.numeroatribuicoes += 2;

    /*Uma comparação*/
    this.numerocomparacoes++;
    if (lo >= n) {
        return;
    }

    /*Uma atribuição*/
    int mid = array[(lo + hi) / 2];
    this.numeroatribuicoes++;

    while (lo < hi) {

        /*Uma comparação a cada iteração*/
        this.numerocomparacoes++;


        while (lo < hi && array[lo] < mid) {
            lo++;

            /*Duas comparações e uma atribuição a cada iteração*/
            this.numerocomparacoes += 2;
            this.numeroatribuicoes++;
        }
         while (lo < hi && array[hi] > mid) {
             hi--;

             /*Duas comparações e uma atribuição a cada iteração*/
             this.numerocomparacoes += 2;
             this.numeroatribuicoes++;
         }

         /*Uma comparação*/
         this.numerocomparacoes++;
         if (lo < hi) {

             /*três atribuições*/
             this.numeroatribuicoes += 3;

             int T = array[lo];
             array[lo] = array[hi];
             array[hi] = T;
         }
     }

     /*Uma comparação*/
     this.numerocomparacoes++;
     if (hi < lo) {
         /*três atribuições*/
         this.numeroatribuicoes += 3;
         int T = hi;
         hi = lo;
         lo = T;
     }
        /*Chamadas recursivas*/
        quick_srt(array, low, lo);
        quick_srt(array, lo == low ? lo + 1 : lo, n);
    }
}
```

3 - Análise dos Testes

Tabulação, construção dos gráficos a partir dos dados gerados pelo programa e análise dos resultados.

3.1 - Mergesort

Mergesort é um exemplo de algoritmo de ordenação do tipo dividir-para-conquistar. Primeiro cria-se uma sequência ordenada a partir de duas outras também ordenadas. Para isso, ele divide a sequência original em pares de dados, ordena-as; depois as agrupa em sequências de quatro elementos, e assim por diante, até ter toda a sequência dividida em apenas duas partes. Apresenta complexidade Θ(n log2 n).

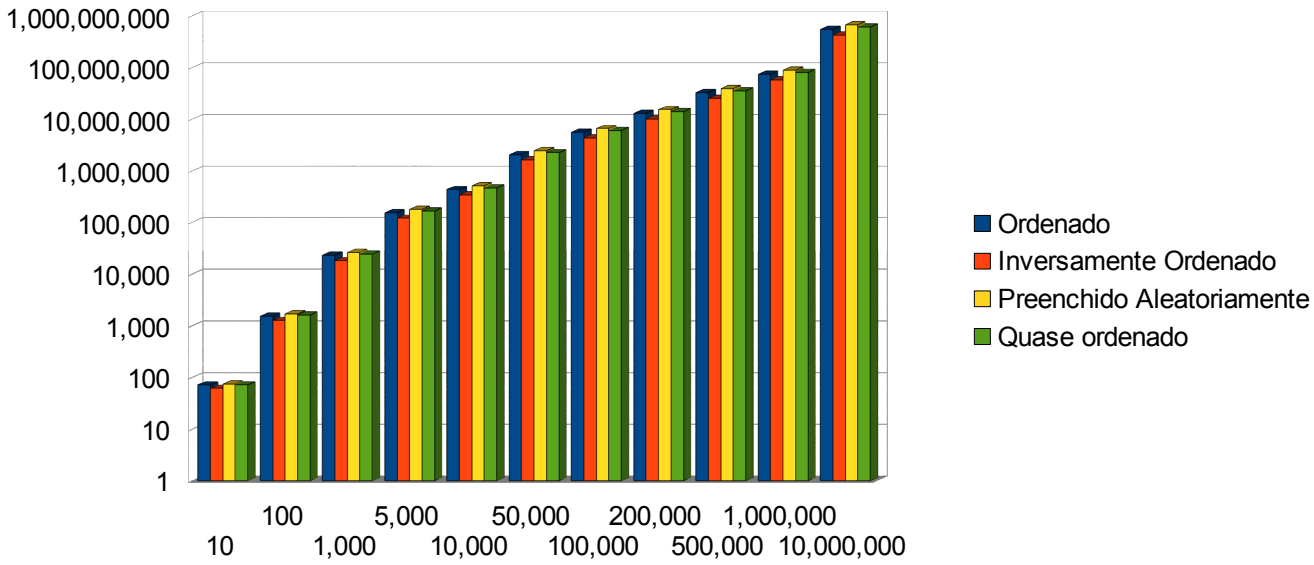
MergeSort Ordenado				
Número de entradas	Número de comparações	Número de atribuições	Tempo(ms)	Tempo(ns)
10	72	148	0	0.019
100	1557	2556	1	0.205
1000	23367	33466	2	2.759
5000	154752	212034	22	22.747
10000	437550	589216	8	8.155
50000	2087000	2708090	42	41.53
100000	5585935	7145890	51	50.353
200000	12983843	16421544	44	44.311
500000	32844502	40872908	102	102.478
1000000	74565862	91775694	204	204.042
10000000	556657838	668221192	2248	2247.41

Mergesort Inversamente Ordenado				
Número de entradas	Número de comparações	Número de atribuições	Tempo(ms)	Tempo(ns)
10	62	148	0	0.004
100	1275	2556	0	0.011
1000	18278	33466	1	0.11
5000	122066	212034	0	0.595
10000	344665	589216	1	1.243
50000	1631053	2708090	7	6.879
100000	4353856	7145890	14	14.252
200000	10099490	16421544	30	29.875
500000	25358386	40872908	78	77.788
1000000	57376209	91775694	163	163.952
10000000	429387114	668221192	1875	1875.246

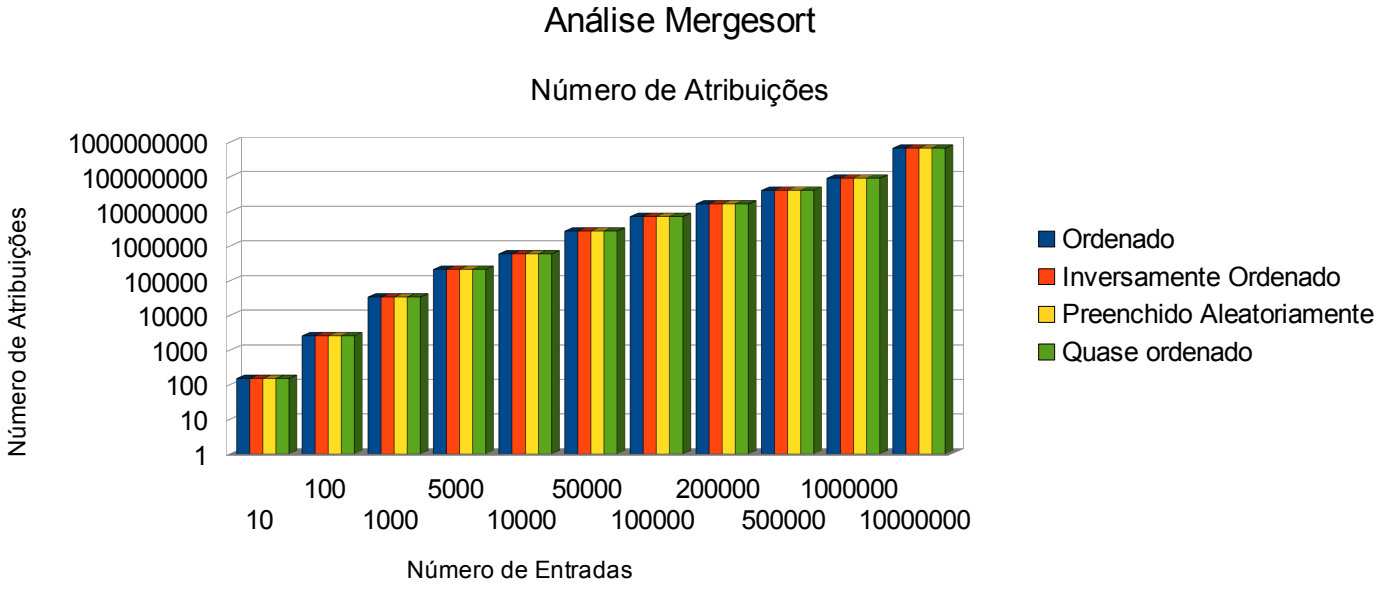
Mergesort Preenchido Aleatoriamente				
Número de entradas	Número de comparações	Número de atribuições	Tempo(ms)	Tempo(ns)
10	75	148	0	0.004
100	1720	2556	0	0.018
1000	26687	33466	1	0.206
5000	180767	212034	1	1.17
10000	513904	589216	2	2.497
50000	2471462	2708090	14	14.349
100000	6636473	7145890	31	30.096
200000	15466391	16421544	64	63.68
500000	39167956	40872908	169	169.325
1000000	89070699	91775694	377	376.702
10000000	671333046	668221192	4515	4515.484

Mergesort Quase Ordenado				
Número de entradas	Número de comparações	Número de atribuições	Tempo(ms)	Tempo(ns)
10	72	148	0	0.01
100	1621	2556	0	0.016
1000	24755	33466	0	0.161
5000	166284	212034	1	0.89
10000	471866	589216	2	1.878
50000	2262580	2708090	11	10.86
100000	6069689	7145890	23	22.823
200000	14133458	16421544	48	48.264
500000	35789959	40872908	151	151.383
1000000	81353092	91775694	273	273.425
10000000	611029202	668221192	3287	3286.424

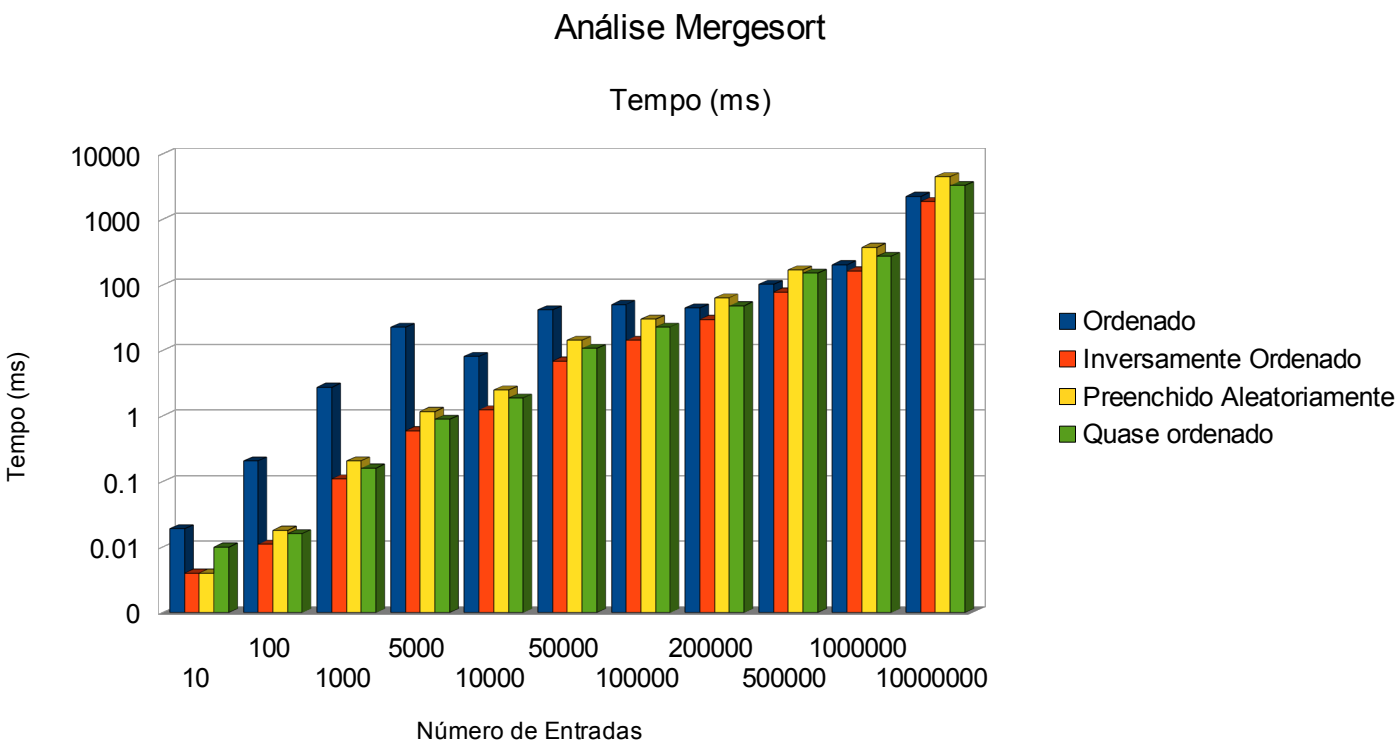
3.1.1 Mergesort - Número de Comparações



3.1.2 Mergesort - Número de Atribuições



3.1.3 Mergesort - Tempo (ms)



3.2 Heapsort

O heapsort utiliza uma estrutura de dados chamada heap, para ordenar os elementos a medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada, lembrando-se sempre de manter a propriedade de max-heap. Apresenta complexidade Θ(n log2 n).

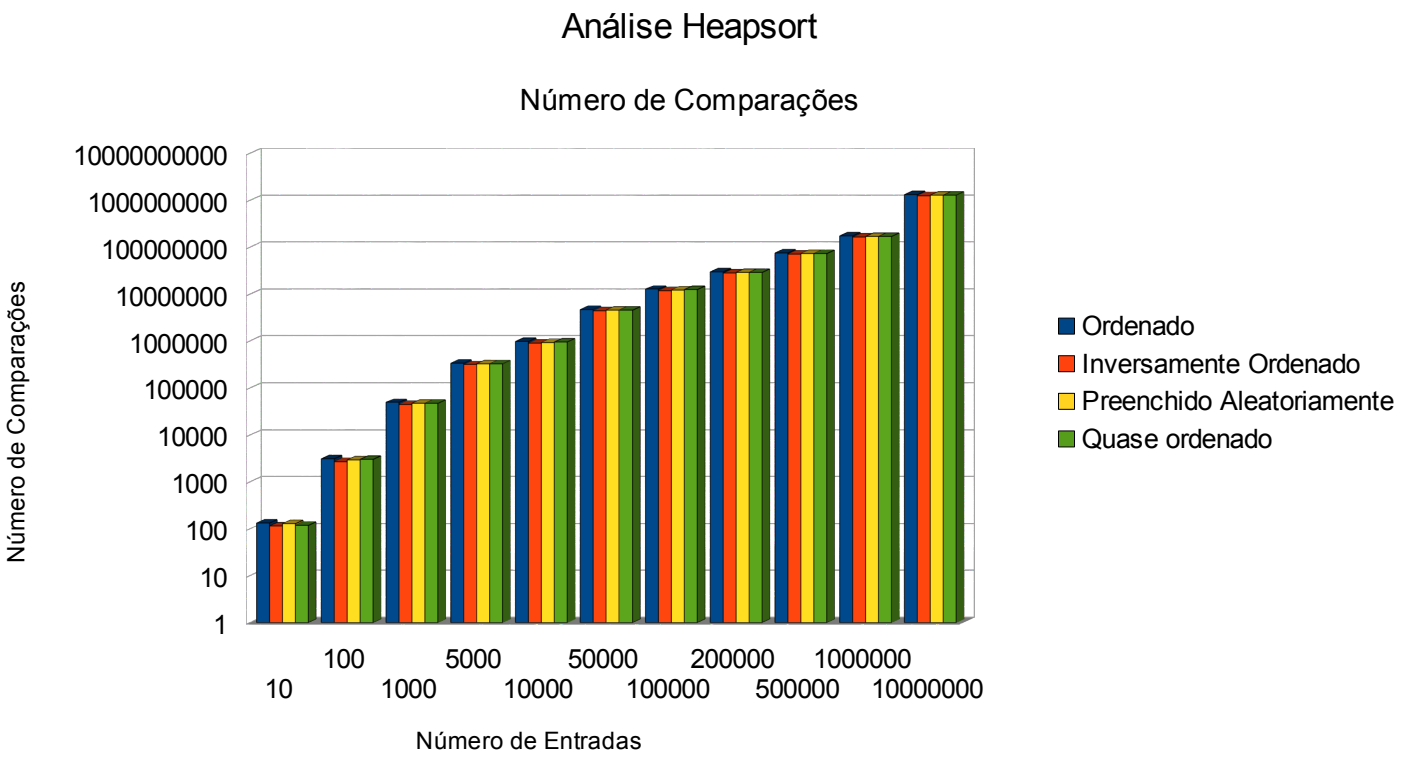
Heapsort Ordenado				
Número de entradas	Número de comparações	Número de atribuições	Tempo(ms)	Tempo(ns)
10	134	229	0	0.024
100	3068	5327	1	0.347
1000	49132	84563	5	4.778
5000	339596	584262	7	7.407
10000	971035	1669536	5	5.334
50000	4709724	8070804	13	12.68
100000	12691928	21746722	19	18.588
200000	29660667	50788991	34	34.143
500000	75274611	128725540	90	90.672
1000000	171597910	293561174	191	190.757
10000000	1304506369	2229421478	2245	2244.715

Heapsort Inversamente Ordenado				
Número de entradas	Número de comparações	Número de atribuições	Tempo(ms)	Tempo(ns)
10	114	168	0	0.002
100	2703	4329	0	0.009
1000	44657	72908	0	0.112
5000	312816	515910	1	0.652
10000	899615	1486395	1	1.392
50000	4414994	7320823	7	7.671
100000	11931878	19825502	18	17.929
200000	27973262	46545153	34	33.769
500000	71419671	119055358	90	89.843
1000000	163427125	272627089	190	189.44
10000000	1250764699	2092496317	2219	2218.357

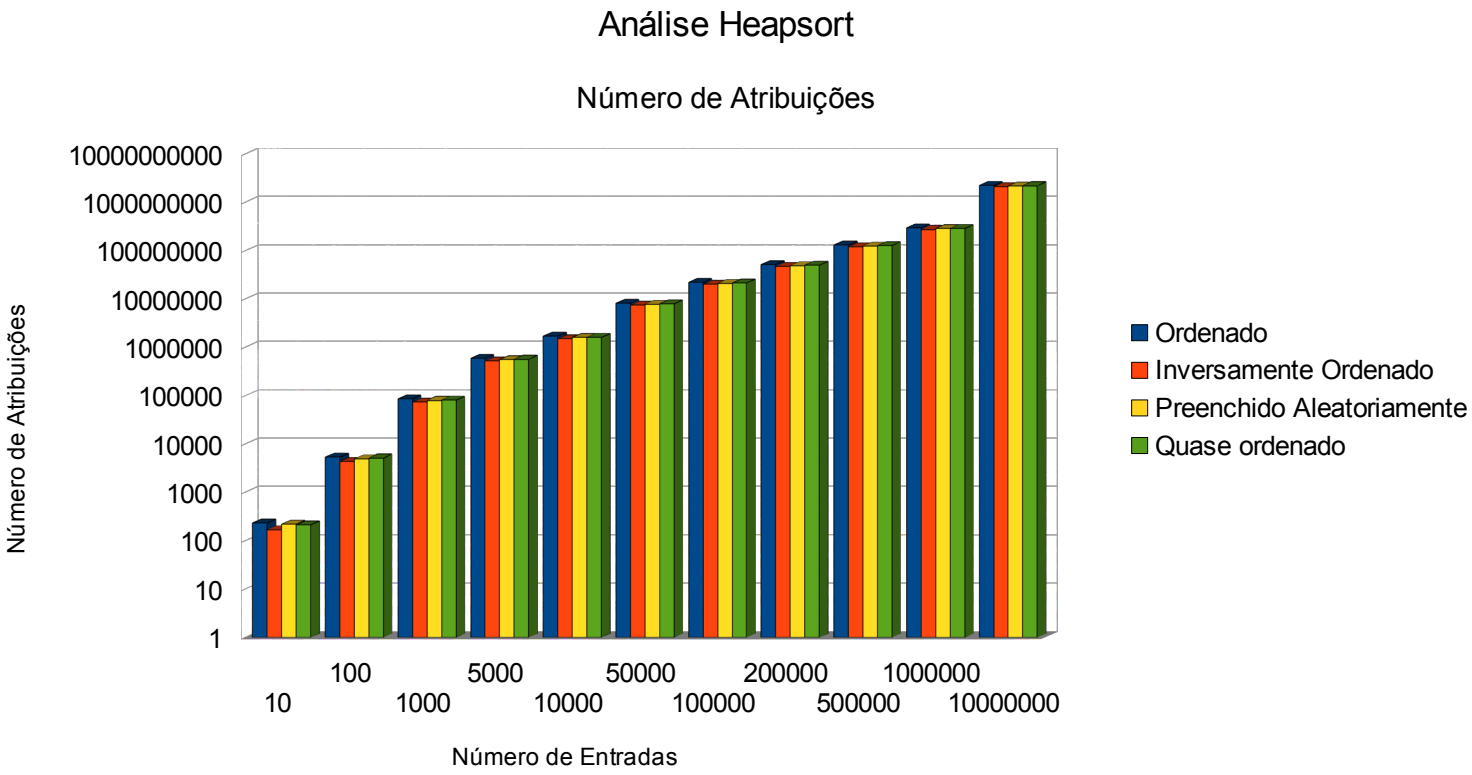
HeapSort Preenchido Aleatoriamente				
Número de entradas	Número de comparações	Número de atribuições	Tempo(ms)	Tempo(ns)
10	129	219	0	0.002
100	2923	4900	0	0.009
1000	47152	79308	0	0.127
5000	326241	549631	0	0.774
10000	934955	1576471	2	2.599
50000	4559689	7698341	10	10.102
100000	12308653	20790495	22	21.739
200000	28806722	48673998	48	47.155
500000	73298321	123925958	133	132.362
1000000	167281565	282928685	299	299.285
10000000	1273898709	2156108203	4644	4644.35

Heapsort quase ordenado				
Número de entradas	Número de comparações	Número de atribuições	Tempo(ms)	Tempo(ns)
10	119	210	0	0.001414
100	2978	5098	0	0.009095
1000	47647	81379	0	0.124914
5000	329786	563065	1	0.746306
10000	943915	1611565	2	1.60838
50000	4598624	7845487	13	13.389116
100000	12408993	21166377	21	20.71828
200000	29028407	49504357	45	44.997316
500000	73807401	125861632	124	124.539213
1000000	168373575	287094785	280	280.639316
10000000	1281116714	2183039483	4332	4332.370572

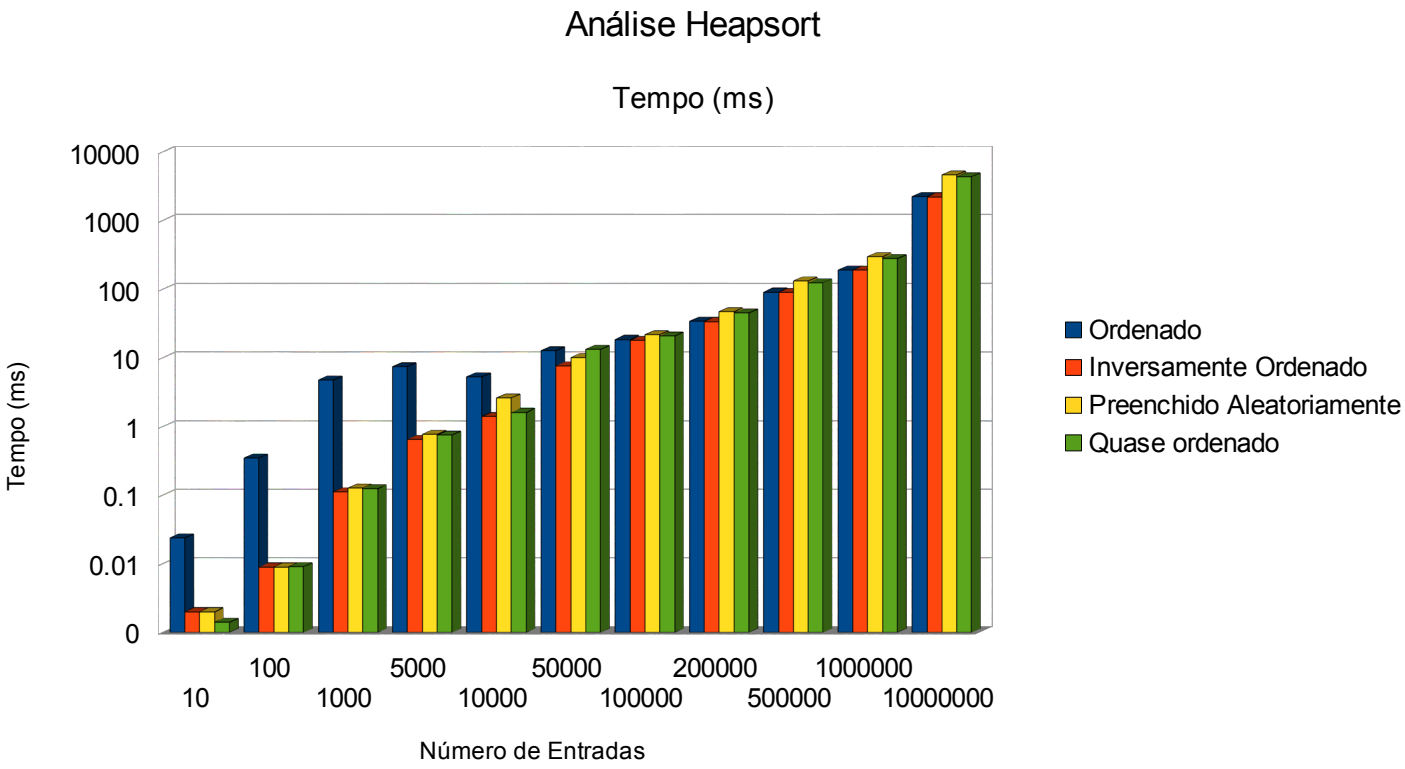
3.2.1 Heapsort - Número de Comparações



3.2.2 Heapsort - Número de Atribuições



3.2.3 Heapsort - Tempo (ms)



3.3 Quicksort

A estratégia básica do quicksort é a de "dividir para conquistar". Inicia-se com a escolha de um elemento da lista, designado pivô; a lista é então rearranjada de forma que todos os elementos maiores do que o pivô fiquem de um dos lados do pivô e todos os elementos menores fiquem do outro lado (ficando assim o pivô na sua posição definitiva); recursivamente, repete-se este processo para cada sub-lista e, no final, o resultado é uma lista ordenada.

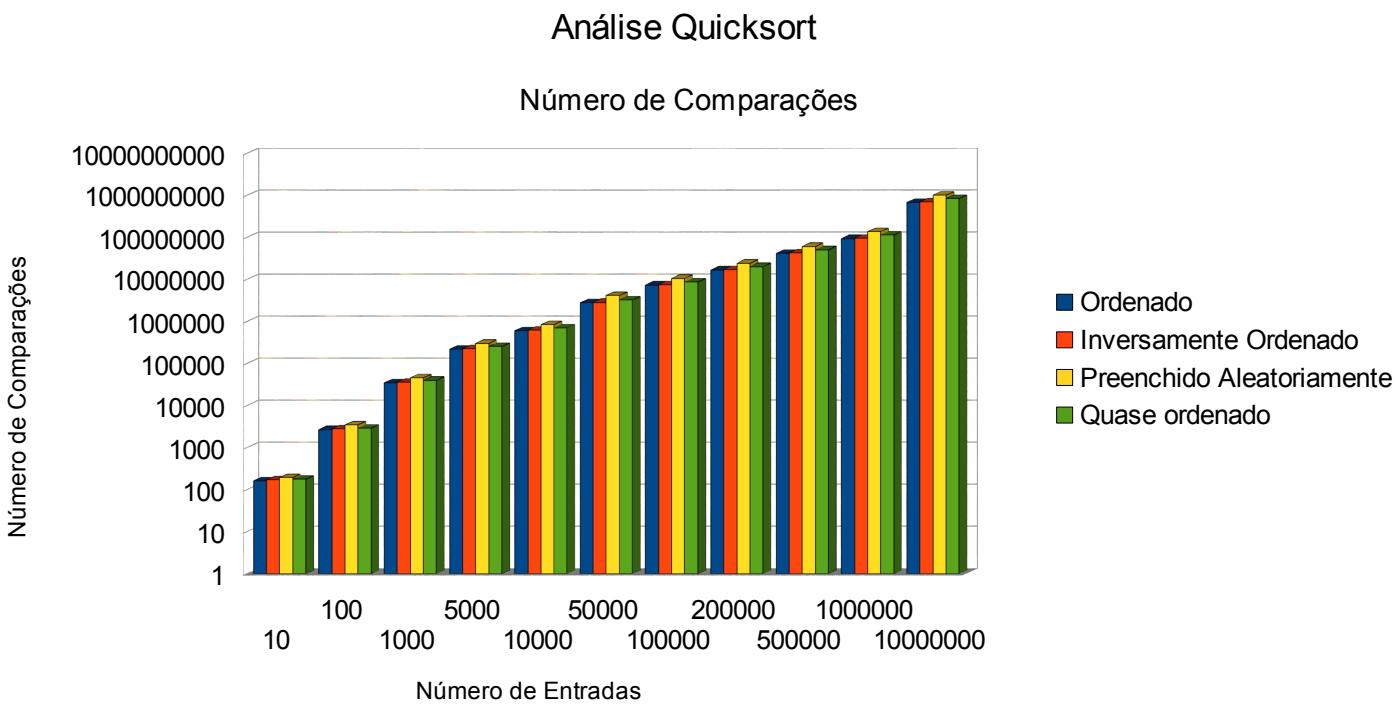
Quicksort ordenado				
Número de entradas	Número de comparações	Número de atribuições	Tempo(ms)	Tempo(ns)
10	162	125	0	0.020
100	2674	1875	0	0.194
1000	34588	22826	3	2.505
5000	218160	139606	44	43.915
10000	605346	383193	1	0.590
50000	2774224	1717626	3	3.064
100000	7312028	4486522	9	8.329
200000	16787686	10224345	13	12.816
500000	41739054	25200023	34	33.947
1000000	93641844	56151412	65	65.850
10000000	680087346	399374157	748	748.260

Quicksort inversamente ordenado				
Número de entradas	Número de comparações	Número de atribuições	Tempo(ms)	Tempo(ns)
10	172	140	0	0.003
100	2784	2040	0	0.007
1000	35698	24491	1	0.060
5000	224270	148771	0	0.313
10000	621456	407358	2	1.241
50000	2840334	1816791	4	4.096
100000	7478138	4735687	7	6.962
200000	17153796	10773510	13	13.783
500000	42605164	26499188	37	36.899
1000000	95507954	58950577	72	72.301
10000000	691953456	417173322	797	796.991

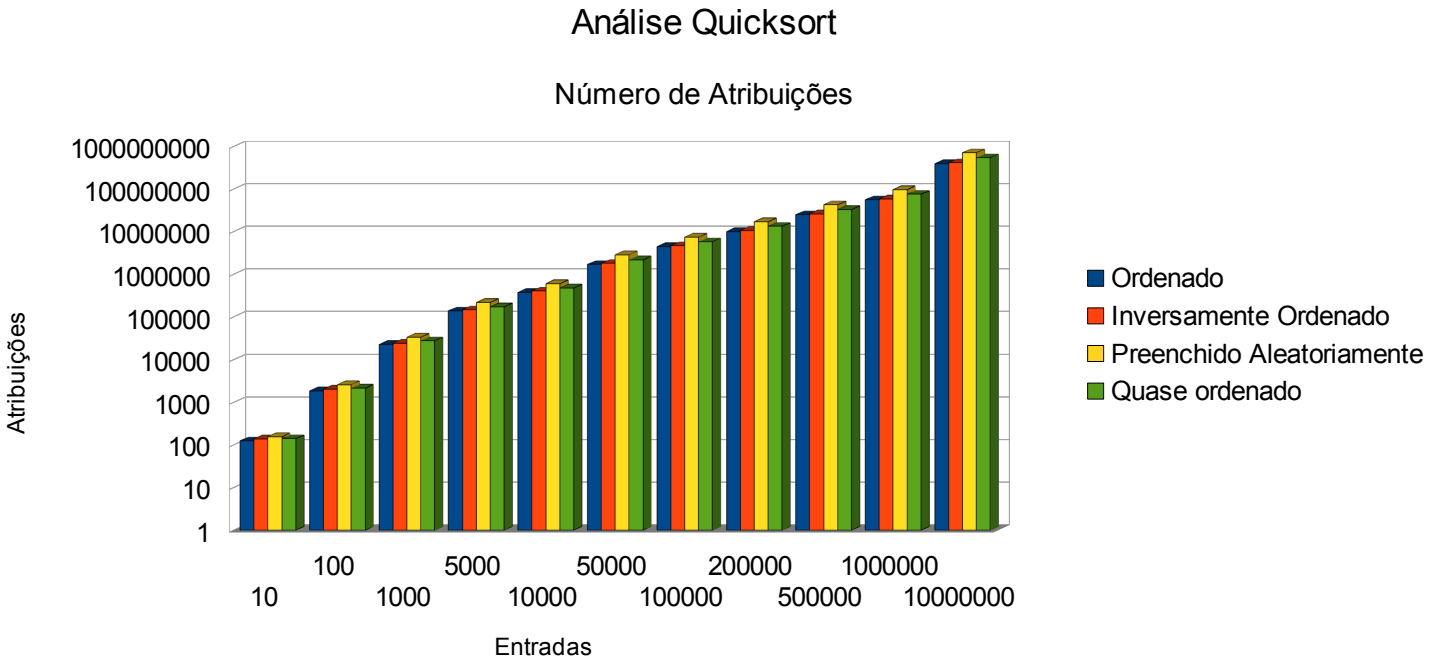
Quicksort aleatoriamente ordenado				
Número de entradas	Número de comparações	Número de atribuições	Tempo(ms)	Tempo(ns)
10	192	158	0	0.003
100	3442	2607	0	0.016
1000	46011	34063	0	0.171
5000	302445	219510	1	0.940
10000	837263	607127	3	3.125
50000	4116325	2916873	10	10.619
100000	10606349	7567355	22	22.129
200000	24303663	17323042	46	46.108
500000	60705388	43180300	121	121.851
1000000	137410185	97397858	254	253.995
10000000	1011096550	712795221	2879	2878.672

Quicksort quase ordenado				
Número de entradas	Número de comparações	Número de atribuições	Tempo(ms)	Tempo(ns)
10	177	141	0	0.00857
100	2914	2148	0	0.011315
1000	39838	27858	0	0.108845
5000	253515	174579	0	0.58056
10000	708253	484128	2	1.221876
50000	3280955	2215939	7	6.701087
100000	8667945	5829492	13	13.812665
200000	20167640	13482718	28	28.595058
500000	50535561	33602299	76	75.957397
1000000	114513190	75762420	157	156.962881
10000000	836513227	547443879	1773	1772.605794

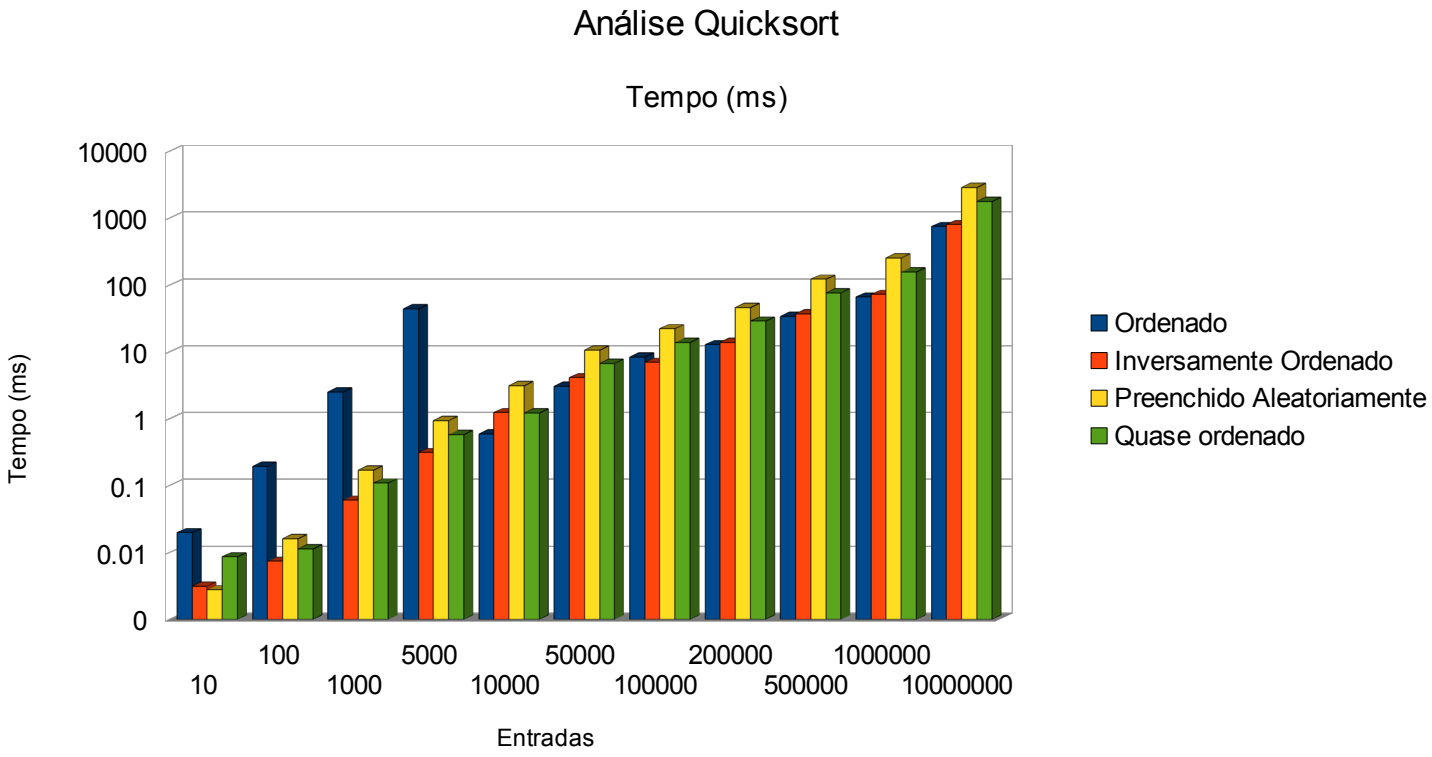
3.3.1 Quicksort - Número de Comparações



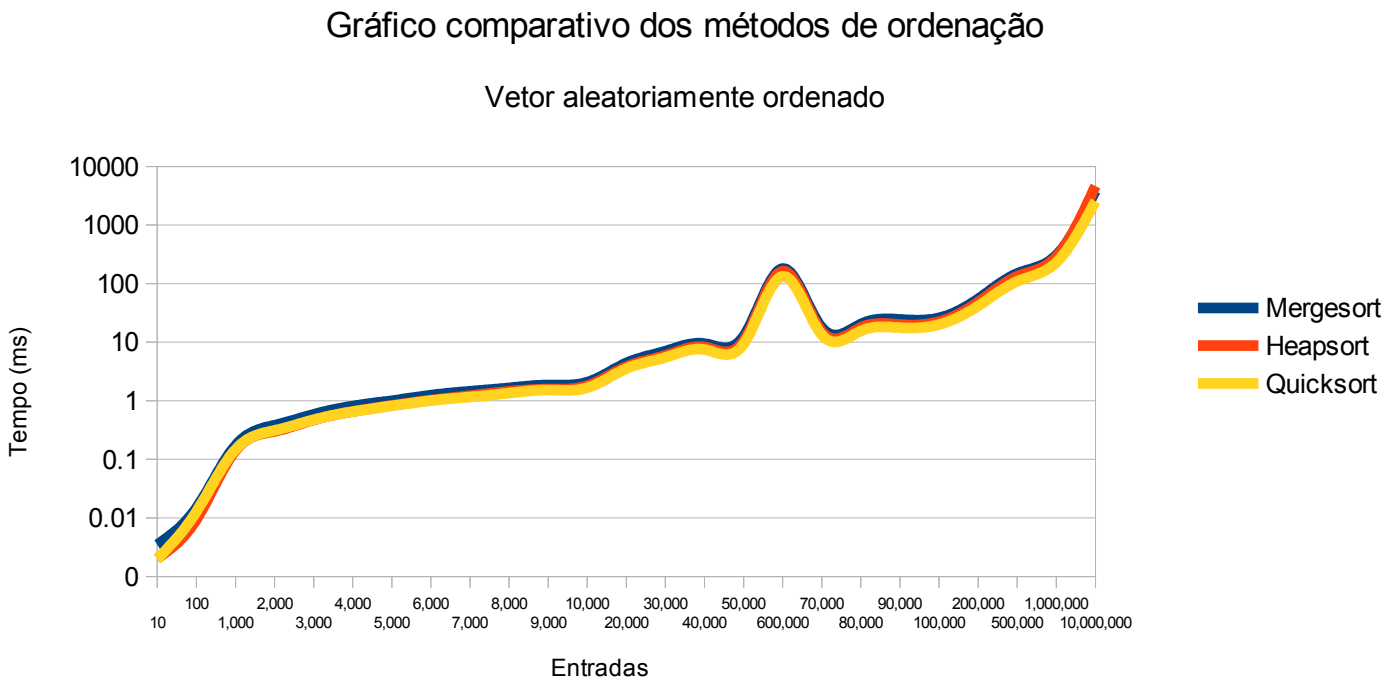
3.3.2 Quicksort - Número de Atribuições

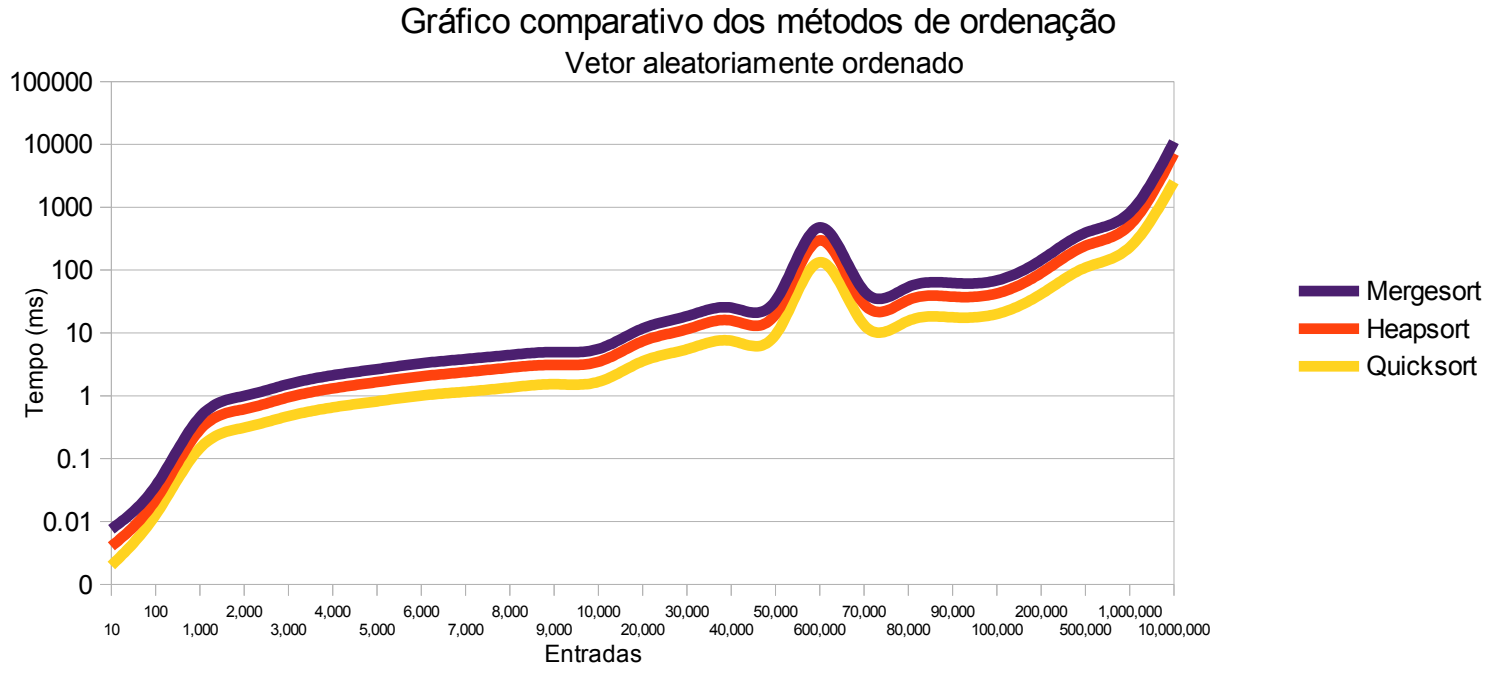


3.3.3 Quicksort - Tempo (ms)



3.4 Comparação dos algoritmos de ordenação





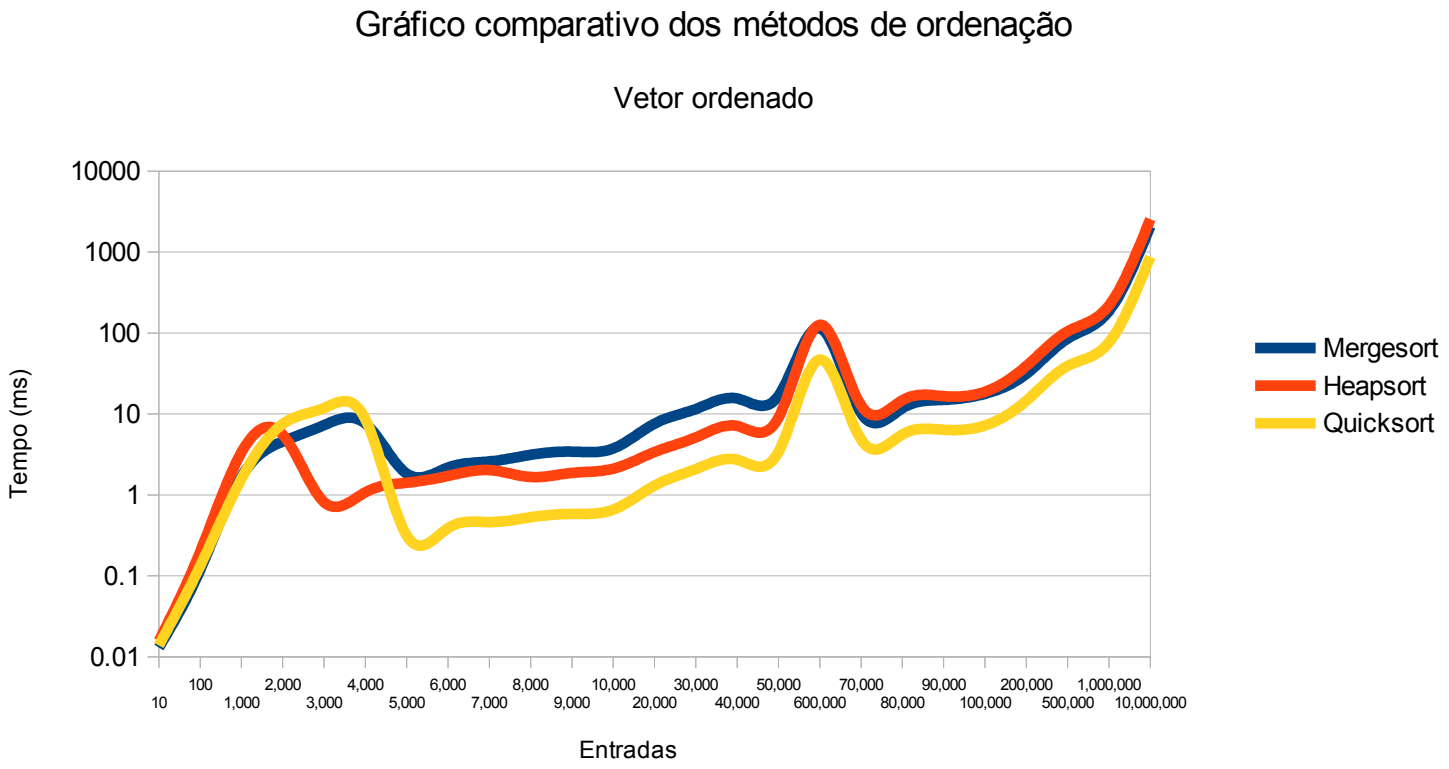


Gráfico comparativo dos métodos de ordenação
Vetor ordenado
Tempo (ms)
10000
1000
100
10
1
0.1
0.01
10
100
1,000
2,000
3,000
4,000
5,000
6,000
7,000
8,000
9,000
10,000
20,000
30,000
40,000
50,000
600,000
70,000
80,000
90,000
100,000
200,000
500,000
1,000,000
10,000,000
Entradas
Mergesort
Heapsort
Quicksort

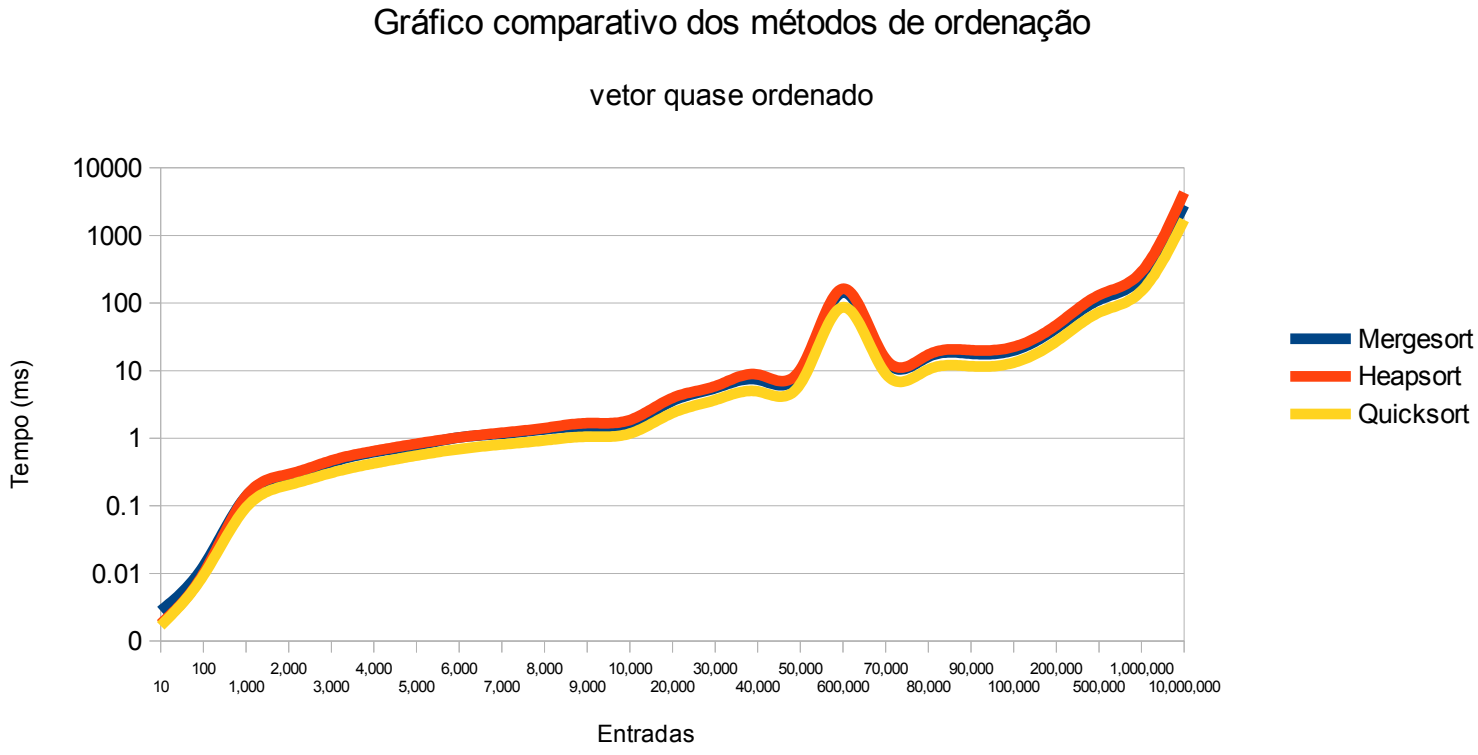


Gráfico comparativo dos métodos de ordenação
vetor quase ordenado
Tempo (ms)
10000
1000
100
10
1
0.1
0.01
0
10
100
1,000
2,000
3,000
4,000
5,000
6,000
7,000
8,000
9,000
10,000
20,000
30,000
40,000
50,000
600,000
70,000
80,000
90,000
100,000
200,000
500,000
1,000,000
10,000,000
Entradas
Mergesort
Heapsort
Quicksort

4 - Conclusão

Neste trabalho, após elaboração de tabelas e gráficos e uma análise dos mesmos, podemos concluir que:

- O Mergesort, no geral, é o pior de todos. Apresentou bons resultados somente para ordenação de poucos elementos, sendo os outros algoritmos mais recomendados para a ordenação de valores maiores.
- Para um número maior de elementos o Quicksort deve ser usado porque, apesar de sua maior utilização de espaço, ele é consideravelmente o mais rápido do que qualquer outro algoritmo quando o vetor é suficientemente grande.
- O Quicksort é o algoritmo mais eficiente de todos levando em conta números de comparações, atribuições e tempo com uma visão mais geral.
- Devido a sua natureza, o algoritmo Heapsort apresenta ser o mais eficiente que o Quicksort na ordenação de vetores com mais de cem milhões de elementos.
- O Quicksort realiza consideravelmente menos atribuições do que comparações, ao contrário de todos os outros algoritmos.

5 Bibliografia/Referências

[1] http://www.sorting-algorithms.com/
[2] http://stackoverflow.com/
[3] http://www.csd.uwo.ca/courses/CS1037a/notes/topic13_AnalysisOfAlgs.pdf
[4] http://home.westman.wave.ca/~rhenry/sort/
[5] http://www.vogella.com/algorithms.html