Algoritmos de Ordenação

Análise Experimental

Leandro Feuser e Thiago Basso

***Resumo*—Artigo apresentado à disciplina de Análise de Algoritmo, que trata da elaboração de experimentos relacionados a algoritmos de ordenação codificados utilizando a ferramenta Octave. Foram implementados quatro algoritmos de ordenação com o intuito de comparar seus tempos de execução.**

***Index Terms*—Algoritmos de ordenação, InsertionSort, QuickSort, RandomizedQuickSort, MergeSort.**

I. Introdução

Algoritmos de ordenação apresentam uma área muito vasta de estudos dentro da Ciência da Computação, sendo um dos problemas mais clássicos da computação. Deve-se levar em conta que sua importância está em prover um arranjo ordenado não somente de elementos numéricos, mas também de dados satélites relacionados a esses elementos, ou seja, quando um elemento numérico do arranjo é ordenado, ele leva junto consigo uma quantidade de informações vinculadas ao seu registro. Devido a essa característica, a ordenação frequentemente é utilizada como sub-rotina para outros algoritmos [1].

Outro aspecto relacionado a ordenação é que se trata de um problema de interesse histórico. Pode-se observar inúmeras técnicas importantes usadas em projetos de algoritmos na ampla variedade de soluções que foram propostas durante os anos [1].

O presente artigo faz uma breve descrição de quatro estratégias para resolução do problema de ordenação e em seguida apresenta os procedimentos realizados para experimentação das mesmas e sua comparação. Os resultados coletados são comentados e por fim é feita uma breve conclusão acerca do tema.

II. Descrição dos Métodos

No decorrer desta seção os algoritmos serão descritos com base na movimentação de seu valor numérico, sem levar em conta adaptações necessárias quando se deseja fazer movimentação de dados satélites. Será exposto seu funcionamento, a técnica utilizada, um exemplo de sua execução e a complexidade apresentada por cada um dos algoritmos.

Assume-se que um algoritmo resolva o problema da ordenação se:

- Possui como entrada uma sequência n de números $(a1, a2, ...an)$, dispostos em qualquer ordem.
- Como saída apresenta a permutação desses números, onde $a1 <= a2 <= ... <= an$.

A. InsertionSort

A ideia do algoritmo de inserção é semelhante à forma como muitas pessoas ordenam cartas em um jogo de baralho. Inicialmente a mão que irá segurar as cartas está vazia e à medida que vão sendo recolhidas da mesa (uma a uma), são inseridas em sua posição correta. A representação algorítmica para essa ideia é representada no Algoritmo 1 a seguir.

Algoritmo 1: InsertionSort [1]

```
1  InsertionSort(A)
2  for j := 2 to A.comprimento do
3  |   chave := A[j]
   |   /* Inserir A[j] na sequência preordenada
   |   A[1...j-1]                                  */
4  |   i := j − 1
5  |   while i > 0 and A[i] > chave do
6  |   |   A[i + 1] = A[i]
7  |   |   i = i − 1
8  |   end
9  |   A[i+1] = chave
10 end
```

No pseudocódigo do Algoritmo 1, A representa o vetor de entrada que se pretende ordenar. No *loop for* das linhas de 2 a 10 o vetor é percorrido de seu segundo elemento até o fim, sendo que na linha 3 a variável $chave$ recebe o valor da posição do vetor que está sendo analisada no momento pela posição j. O *loop while* das linhas de 5 a 8 será executado enquanto a variável i tiver o valor maior que 0 (lembrando que ela recebe o valor de $j-1$ antes de iniciar a execução do laço) e o valor dado pela posição $A[i]$ for maior que $chave$, ou seja, o subvetor $A[1...j-1]$ será percorrido em direção decrescente de índice realizando troca em seus elementos até chegar a posição 0 ou encontrar um elemento que tenha valor menor do que a chave selecionada. Pela analogia da mão do baralho (figura 1), seria como se a carta em questão fosse pega ($chave$) e seus valores comparados dá carta da última posição da mão em direção ao início até encontrar a sua posição correta, sendo assim, quando sua posição correta é encontrada ela é inserida (linha 9) e as cartas maiores que elas foram movidas para frente (linhas 5 a 8). Ao final da execução do *loop* da linha 2 o vetor A estará ordenado.

Nas Figuras 2, 3 e 4 é possível acompanhar a movimentação no vetor durante a execução do *InsertionSort*. Em (a) temos o vetor de entrada original, a posição $A[j]$ é representada pela marcação em preto e a posição $A[i]$ pela marcação em cinza. Cada uma das figuras a, b, c, d, e e f, indicam a ação de tomar

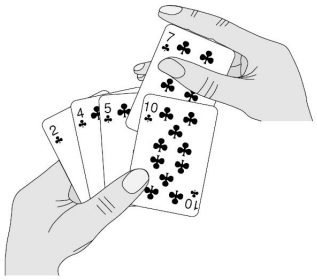

Figura 1. InsertionSort - Analogia com a mão do baralho [1]

.

uma carta nova na mão e procurar pela sua posição correta (linhas 1 a 9). As movimentações indicadas pelas setas cinzas representam os passos do laço *while* (linhas 6 a 8). E, por fim, a movimentação indicada pela seta preta indica o ato de inserir a chave (nova carta) na posição correta (linha 9).

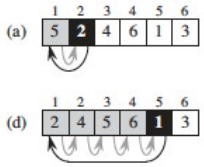

Figura 2. InsertionSort passo a passo, etapas a, d [1]

.

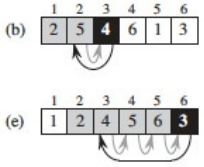

Figura 3. InsertionSort passo a passo, etapas b, e [1]

.

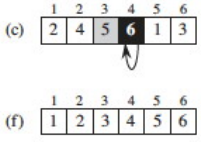

Figura 4. InsertionSort passo a passo, etapas c, f [1]

.

A complexidade do algoritmo *InsertionSort* é dada pelo tempo $O(n^2)$. Uma explicação detalhada de como chegar a essa definição pode ser vista em Cormen et. al[1].

B. QuickSort

Este método de ordenação é muitas vezes a melhor opção prática para ordenação. Apesar de seu tempo de execução no pior caso ser da ordem de $O(n^2)$, sua eficiência é esperada na média na ordem $O(n \log n)$. Além de ter a vantagem de não requisitar área extra de memória, pois a ordenação é feita em cima do próprio vetor (no local)[1].

O QuickSort aplica o paradigma de divisão e conquista em sua implementação [1].

- Divisão: O arranjo $A[p...r]$ é dividido em dois sub arranjos $A[p...q-1]$ e $A[q+1...r]$, sendo os elementos do primeiro sub arranjo menores ou iguais a $A[q]$ e o segundo sub arranjo maior ou igual a $A[q]$.
- Conquista: Ordenar os dois sub arranjos por chamadas recursivas ao *QuickSort*.
- Combinação: Após as chamadas recursivas, os sub arranjos estarão ordenados, como o algoritmo faz trocas locais o arranjo já estará ordenado.

A representação algorítmica da rotina do QuickSort é exibida no Algoritmo 2. O método partition é exibido no Algoritmo 3.

Algoritmo 2: QuickSort [1]

```
1  QuickSort(A, p, r)
2  if p <r then
3  |   q := Partition(A, p, r)
4  |   QuickSort(A, p, q − 1)
5  |   QuickSort(A, q + 1, r)
6  end
```

Algoritmo 3: Partition [1]

```
1  Partition(A, p, r)
2  x := A[r]
3  i := p − 1
4  for j := p to r − 1 do
5  |   if A[j] <=x then
6  |   |   i := i + 1
7  |   |   trocaA[i]porA[j]
8  |   end
9  end
10 trocaA[i + 1]porA[r]
11 return i + 1
```

Na Figura 5 podemos ver o comportamento da execução da rotina *Partition*. Acompanhando a execução pelo 3, o Piv (atribuição de x na linha 1) é selecionado como sendo o último elemento do arranjo. O valor de i (linha 2) é inicializado com $p\check{}1$, sendo p a posição inicial do arranjo. No laço *for* da linha 3 a 6, o arranjo é percorrido até o penúltimo elemento (lembrando que o último é o pivô) e trocas vão sendo realizadas caso o valor em questão seja menor que o valor do pivô, de modo que os elementos do índice p até i tenham seus valores menores que o pivô. Ao final do *loop*, todo o arranjo foi percorrido e a variável i guarda a posição do índice onde foi colocado o último elemento menor que o pivô, e então na linha 7 o pivô é inserido em $i+1$, garantindo que todos os elementos antes dele sejam menores ou iguais ao seu valor, e todos elementos depois dele tenham seus valores maior ou

igual a ele. Por fim, na linha 8 é retornado o valor de $i + 1$, que corresponde a variável q do algoritmo *Quaicksort*.

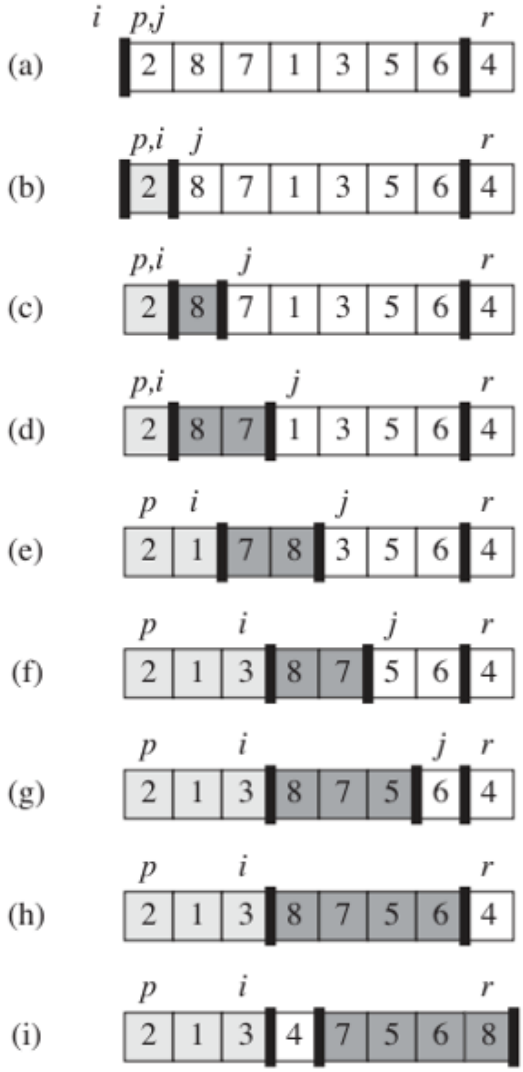

Figura 5. Partition passo a passo [1]

.

C. RandomizedQuickSort

No algoritmo *QuickSort*, a escolha do pivô irá influenciar no balanceamento das partições. Se o particionamento não for balanceado, ou seja, apresentar quantidades muito diferentes de elementos em cada uma das partições (ex.: uma partição com um elemento e a outra com o restante), seu tempo de execução irá tender ao seu pior caso que é $O(n^2)$ [1].

A versão aleatória do *QuickSort*: *RandomizedQuickSort* e RandomizedPartition, tenta diminuir a influência do pivô na hora de montar as partições, escolhendo um elemento aleatório no arranjo. As modificações necessárias para a versão aleatória são exibidas nos algoritmos 5 e 4, respectivamente. Com essa modificação, o tempo esperado para execução de RandomizedQuickSort passa a ser de $O(nlogn)$, conforme demonstrado por Cormen et. al em [1].

Algoritmo 4: RandomizedPartition [1]

```
1 RandomizedPartition(A, p, r)
2 i := Random(p, r)
3 trocaA[p]porA[i]
4 return Partition(A, p, r)
```

Algoritmo 5: RandomizedQuickSort [1]

```
1 RandomizedQuickSort(A, p, r)
2 if p <r then
3 |   q := RandomizedPartition(A, p, r)
4 |   RandomizedQuickSort(A, p, q − 1)
5 |   RandomizedQuickSort(A, q + 1, r)
6 end
```

D. MergeSort

O *MergeSort* é um método de ordenação por intercalação que segue a abordagem de dividir e conquistar. Tal abordagem divide o problema em vários subproblemas, os resolve recursivamente e combina as várias soluções encontradas em uma única solução para o problema original[1]. A abordagem de dividir e conquistar é dividida nos três seguintes passos:

- Dividir o problema em um determinado número de subproblemas.
- Conquistar o subproblemas, resolvendo-os recursivamente ou de forma direta se forem pequenos o bastante.
- Combinar as soluções encontradas para os subproblemas, a fim de formar a solução para o problema original.

Para o algoritmo do Mergesort os três passos são descritos como:

- Dividir: Divide de forma recursiva a sequência de n elementos a serem ordenados em duas subsequências de $n/2$ elementos cada uma, até não ser mais possível a divisão.
- Conquistar: Ordena recursivamente as duas subsequências utilizando intercalação. Não há nenhum trabalho com sequências de comprimento um, pois já estão ordenadas.
- Combinar: Faz a intercalação das duas sequências ordenadas, de modo a produzir uma terceira sequência de comprimento maior que as anteriores e já ordenada.

Como pode ser visto na Figura 6, o *MergeSort* divide a sequência desordenada em várias subsequências, ordena essas subsequências e as intercala até produzir uma única sequência que contenha todos os símbolos iniciais ordenados.

A operação principal do *Mergesort* é a intercalação de duas sequências ordenadas. Para executar a intercalação, usamos a função $Merge(vet\_esq, n, vet\_dir, m)$, onde vet_esq e vet_dir são os dois vetores a serem intercalados sendo n e m os seus respectivos tamanhos. Para realizar a intercalação é utilizado um vetor auxiliar, portando este método não é *in-place*. Sempre o símbolo de menor valor será inserido primeiro no vetor auxiliar, caso uma sequência termine antes da outra, não é mais necessário comparar os símbolos, o vetor auxiliar será preenchido com os símbolos da sequência que restou. A função $MergeSort(vetor\_desordenado, n)$ é responsável

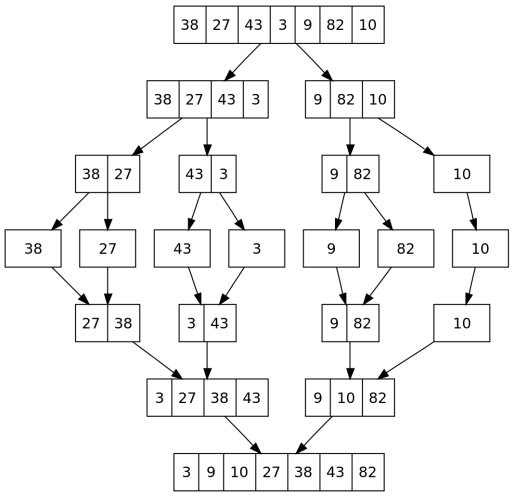

Figura 6. Mergesort recursivo ordenando um vetor de 7 elementos

por dividir, de forma recursiva, o vetor desordenado o máximo possível de vez. Após não ser mais possível realizar a divisão, a função $Merge$ é chamada para ordenar e intercalar as subsequências.A Figura 7 apresenta o programa em *Matlab* para o método de *Mergesort*.

```
function[vet_ord] = Mergesort(vet_desord,n)
  if n==1
    vet_ord = vet_desor;
  else
    m = floor(n/2);
    vet_esq = Mergesort(vet_desord(1:m),m);
    vet_dir = Mergesort(vet_desord(m+1:n),n-m);
    vet_ord = Merge(vet_esq,m,vet_dir,n-m);
  end
end

function[aux] = Merge(vet_esq,n,vet_dir,m)
  aux = zeros(1,n+m);
  x = 1;
  y = 1;
  for i=1:(n+m)
    if x > n
      aux(i) = vet_dir(y);
      y++;
    elseif y > m
      aux(i) = vet_esq(x);
      x++;
    elseif vet_esq(x) <= vet_dir(y)
      aux(i) = vet_esq(x);
      x++;
    else
      aux(i) = vet_dir(y);
      y++;
    end
  end
end
```

Figura 7. Mergesort em *Matlab*

Como o número de símbolos do vetor é reduzido pela metade a cada chamada da função e o tempo gasto em cada uma é n, a complexidade do algoritmo *MergeSort* é dada pelo tempo $O(n \log n)$). Uma explicação detalhada de como chegar a essa definição pode ser vista em Cormen et. al[1].

III. Metodologia Experimental

O presente trabalho compõe atividade solicitada no estudo da disciplina de Experimentação Algorítmica. Fora solicitada a implementação de algoritmos de ordenação da ordem de n^2 e $n \log n$, obrigatoriamente. Solicitou-se também, a plotagem de gráficos de tempos a partir da execução com algumas amostras.

Para implementação das rotinas necessárias fora utilizada a ferramenta *GNU Octave*, versão 4.0.3. Esta ferramenta foi a indicada para a execução durante a disciplina, devido a sua facilidade de plotar gráficos e executar atividades propostas à experimentação de algoritmos. O ambiente para execução dos experimentos é uma máquina rodando Windows 10 Home – 64bits, cuja configuração é: processador Intel Core i5 @ 2.40GHz, com 4GB de memória RAM.

O projeto é composto por diversos módulos organizados da seguinte forma:

- InsertionSort.m: contempla a implementação do método de ordenação InsertionSort.
- QuickSort.m: contempla as rotinas necessárias à implementação do método de ordenação QuickSort.
- RandomizedQuickSort.m: contempla as rotinas necessárias à implementação do método de ordenação RandomizedQuickSort.
- MergeSort.m: contempla as rotinas necessárias à implementação do método de ordenação MergeSort.
- Main.m: script utilizado para experimentação, sua rotina é uma série de testes que visam extrair tempos de execução gasto na ordenação para cada um dos algoritmos escritos. Também plota gráficos com os comparativos dos tempos de execução.

Os testes foram conduzidos executando o script Main.m. A rotina de testes foi construída visando a obtenção do tempo de execução de cada um dos algoritmos descritos acima. Sendo assim, a cada interação, um valor para n definia o tamanho do vetor a ser ordenado. Para cada interação foram criados três vetores diferentes:

- Melhor Caso: apresenta elementos de 1 a n já ordenados.
- Caso Médio: apresenta elementos com valores de 1 a n, dispostos de forma aleatória dentro do vetor.
- Pior Caso: apresenta elementos com valores de n a 1 ordenados de forma inversa.

Os resultados obtidos são discutidos na próxima seção.

IV. Resultados e Discussões

Os experimentos realizados mediram o tempo de execução para ordenação de elementos em cada um dos algoritmos construídos. Foram realizadas rodadas de testes com vetores de cinco, cem, duzentos e cinquenta, quinhentos, oitocentos e mil elementos. Para cada rodada do experimento era executada a ordenação para o vetor de melhor caso, caso médio e pior caso.

O *Octave* teve problemas em executar testes com mais de mil e duzentos elementos em um vetor, em todas tentativas o Ocatave travou. Apesar disso, com os resultados para vetores

de até mil elementos obtivemos uma boa análise do comportamento de cada um dos métodos de ordenação.

Tabela I
TEMPOS DE EXECUÇÃO DE CADA ALGORITMO (EM SEGUNDOS)

	5	100	250	500	800	1000
Melhor caso						
InsertionSort	0.0031	0.0021	0.0048	0.0102	0.0157	0.0188
QuickSort	0.0031	0.1404	0.9393	3.5233	9.2298	13.772
Mergesort	0.0031	0.0245	0.0689	0.1446	0.2661	0.3089
RQuickSort	0.0072	0.1496	0.9158	3.6081	8.5303	13.870
Caso médio						
	5	**100**	**250**	**500**	**800**	**1000**
InsertionSort	0.0011	0.0492	0.2570	1.0841	2.8002	4.2075
QuickSort	0.0011	0.0194	0.0514	0.1215	0.2090	0.2501
Mergesort	0.0011	0.0275	0.0760	0.1628	0.2703	0.3742
RQuickSort	0.0002	0.0217	0.0549	0.1263	0.1965	0.2398
Pior caso						
	5	**100**	**250**	**500**	**800**	**1000**
InsertionSort	0.0001	0.0843	0.5454	2.2322	5.8875	8.8285
QuickSort	0.0001	0.0915	0.5439	2.2725	5.3569	8.2566
Mergesort	0.0011	0.0275	0.0691	0.1479	0.2646	0.3395
RQuickSort	0.0011	0.0508	0.2442	1.5717	2.4927	6.7829

Outro pequeno problema encontrado no Octave é que por padrão ele aceita um número pequeno de chamadas recursivas, foi necessário acrescentar a linha $max\_recursion\_depth(1000)$, ao script de testes (Main.m) para aumentar o número de recursões possíveis.

Na Tabela I são apresentados os tempos de execução obtidos nos testes realizados. Os tempos foram coletados pela função *tic-toc* do *Octave*, e estão exibidos em segundos. Destacamos que algumas alterações nos tempos coletados podem ocorrer devido a multitarefa do processador, como por exemplo, o *InsertionSort* no melhor caso executou em $0,0031s$ para cinco elementos, ao passo que para cem elementos seu tempo foi de $0,0021s$.

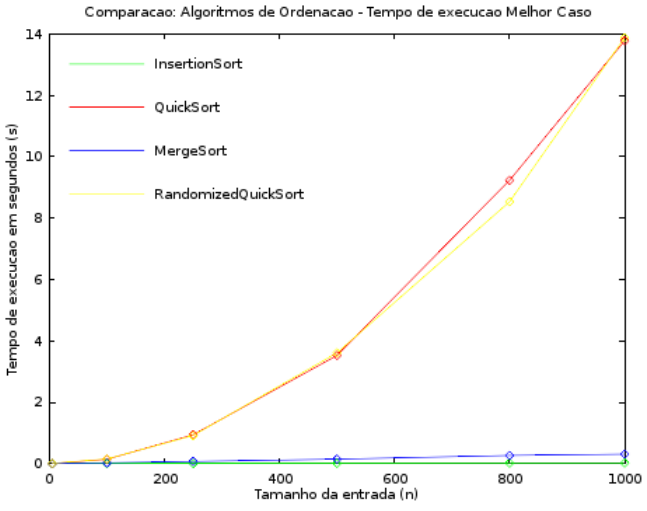

Figura 8. Tempos de execução de cada algoritmo no melhor caso

Na Figura 8 podemos ver o tempo de execução para o melhor caso. Apesar de ter tempo de execução de $O(n^2)$, o algoritmo *InsertionSort* foi o que apresentou o melhor resultado. Como nesta situação o vetor já se encontra ordenado, o algoritmo de *InsertionSort* irá percorrer o vetor todo apenas uma vez, pois todos os elementos já estão em sua posição correta, e como seu corpo de execução tem menos instruções, sua performance será melhor para esse caso.

Na Figura 9 podemos ver o tempo de execução do caso médio. Era esperado que o *QuickSort* tivesse o melhor desempenho, conforme discutido na seção II, e foi o que observamos. O *MergeSort* apresenta desempenho bem próximo e o *InsertionSort* apresentou o pior desempenho.

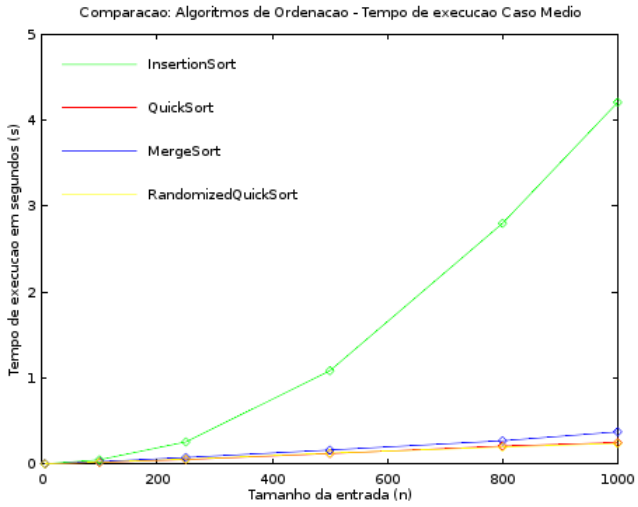

Figura 9. Tempos de execução de cada algoritmo no caso médio

Na Figura 10 podemos ver o tempo de execução do pior caso. Conforme discutido na seção II, o *QuickSort* apresenta tempo de execução na ordem de $O(n^2)$ para esse caso, e podemos observar isso, pois seu tempo ficou próximo ao do *InsertionSort*. O *MergeSort* apresentou o melhor desempenho, conforme esperado, porém a versão randômica do *QuickSort* obteve um resultado abaixo do previsto.

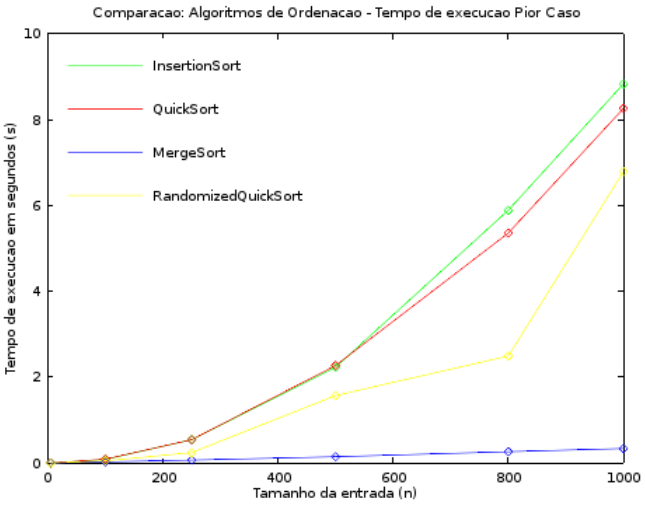

Figura 10. Tempos de execução de cada algoritmo no pior caso

V. CONCLUSÕES

Com relação aos tempos gastos na ordenação, o *InsertionSort* foi o mais eficiente no melhor caso, isso devido ao fato do

vetor já estar ordenado. O *QuickSort* e o *RandomizedQuickSort* obtiveram os melhores tempo no caso médio mas foram muito ineficientes no melhor caso. No geral o algoritmo que obteve melhores resultados foi o *MergeSort*, foi o mais eficiente no pior caso e ficou muito pouco abaixo dos outros no melhor caso e no caso médio.

REFERÊNCIAS

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Algoritmos: Teoria e Prática. Rio de Janeiro: Elsevier, 2012. 3ª edição.