USP – UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E COMPUTAÇÃO (ICMC)
DISCIPLINA: LABORATÓRIO DE INTRODUÇÃO A CIÊNCIA DA COMPUTAÇÃO II

RELATÓRIO 02 - EFICIÊNCIA DE ALGORITMOS DE ORDENAÇÃO

Carlos Filipe de Castro Lemos
nUSP: 12542630

Resumo

Este relatório faz uma sucinta análise assintótica dos mecanismos Quicksort e Heapsort quando ordenam vetores aleatórios ordenados e ordenados inversamente, variando-se a quantidade de casos de entrada (100, 1000, 10.000 e 100.000 elementos).
Palavras-chave: Quicksort, Heapsort, Notação Assintótica.

1 Introdução

Após a elaboração do *Relatório 01 - Eficiência de Algoritmos de Ordenação*, os alunos deveriam elaborar um segundo relatório, porém, dessa vez, com foco no *Quicksort* e no *Heapsort*. Os critérios de abordagem seriam basicamente os mesmos (constituição dos algoritmos, métodos de análise de eficiência, variáveis que interferiam em suas performances, notações assintóticas e outros). Assim, de forma resumida, podemos dizer que os alunos deveriam fazer a análise assintótica através da medição de marcos temporais (considerando a quantidade de entrada e a qualidade dos vetores). No entanto, além de fazer uma análise dos mecanismos em si mesmos, deveriam também fazer uma comparação com os outros três que foram objeto do relatório anterior (Bubble Sort, Insertion Sort e Merge Sort).

Assim, os alunos poderiam confrontar as informações ministradas na literatura especializada, bem como comprovar o conteúdo lecionado pelos professores em sala de aula.

2 Metodologia e Desenvolvimento

2.1 Metodologia

A metodologia utilizada no presente relatório está diretamente relacionada com a medição de eficiência dos algoritmos de ordenação (Quick Sort e Heap Sort). Nesse contexto, é

preciso esclarecer que o termo "eficiência", embora apresente variados significados, está empregado no sentido de *rapidez* ou *velocidade*. Além disso, é conveniente explicitar que foram tomados alguns cuidados no cálculo do tempo, na obtenção dos resultados e na sua posterior comparação.

No *cálculo do tempo*, observou-se as seguintes medidas mitigadoras de erros:

1. **Casos de Teste:** os casos testes consideraram a quantidade de elementos (100, 1000, 10000 e 100000) e a qualidade do vetor (pior e melhor caso). Para *casos aleatórios*, considerou-se vetores formados com números inteiros gerados aleatoriamente na faixa de 0 a 1000. Como *melhores casos*, considerou-se vetores formados com números devidamente ordenados. Enquanto que *piores casos*, considerou-se vetores que foram preenchidos de forma decrescente.

2. **Geração Vetores:** valendo-se da linguagem C, os casos de testes foram gerados de forma aleatória, por meio da biblioteca *<time.h>* (funções *rand()* e *srand()* - alimentadas por um *seed* baseado no horário em que foi feito o teste). Vale dizer que tal vetor foi copiado para outros de mesmo tamanho, de modo que cada medição de tempo tivesse os mesmos casos testes (não importando o método de ordenação). Essa medida teve a finalidade de assegurar solidez aos dados para posterior comparação de resultados.

3. **Medição do Tempo:** a medição do tempo foi realizada por meio da biblioteca *<time.h>*, que disponibiliza a função *clock()*. Essa função permitiu realizar a demarcação do tempo antes e depois da execução e, mediante uma simples subtração, obter o tempo de execução do código.

Na *consolidação dos resultados*, adotou-se um método de prevenção/diluição de erros. Isto é, com a finalidade de evitar variações de resultados, em razão de elementos impossíveis de serem controlados, realizou-se 10 medições para cada caso teste, sendo certo que, posteriormente, calculou-se a média aritmética simples do conjunto.

Por fim, foram realizadas *comparações* entre os resultados de tempos médios calculados nos passos anteriores. Isto é, comparou-se a eficiência de um mesmo método ordenação (em relação aos casos aleatórios, melhores e piores casos), bem como entre os diferentes mecanismos de ordenação (confrontando-se os tempos médios para casos aleatórios, melhores e piores). Além disso, para conferir a análise assintótica da literatura especializada, ou ministrada em sala de aula, tomou-se a liberdade de dividir o tempo médio pela quantidade de elementos do caso teste (100, 1000, 10000 e 100000 elementos), proporcionando uma verificação de variação assintótica em relação aos vários casos objeto de análise.

2.2 Quicksort e suas Características

2.2.1 Conceito

O método de ordenação do Quicksort traz como principal estratégia a ideia de dividir para conquistar. A grosso modo, isso ocorre porque, no momento de realizar a partição do vetor em dois segmentos, o algoritmo escolhe um pivô de referência e, logo em seguida, por meio de dois ponteiros, realiza a comparação dos elementos menores e maiores que o pivô. Ou seja, enquanto o primeiro ponteiro tem seu indice incrementado a partir do começo do vetor, o outro tem o índice decrementado a partir do final do vetor, sendo certo que ambos vão se deslocar até que suas posições se igualem ou se cruzem. Porém, antes desse limiar, caso encontrem elementos fora de posição irão realizar trocas entre si (posicionando os valores menores que o pivô antes dele e os maiores depois dele). No entanto, depois de se encontrarem ou se cruzarem, o pivô será inserido naquela posição do ponteiro decrementado, que é a posição ordenada para o pivô, e, desse modo, poderemos ter certeza de que os elementos que lhes são menores ou maiores estarão respectivamente alocados antes e depois dele. Em seguida, a função será chamada recursivamente para ambos os segmentos, com a finalidade de ordená-los também. O processo se repete até que os valores de inicio e fim do vetor sejam iguais ou o inicio seja maior que o fim.

2.2.2 Código-Fonte

O código fonte do Quicksort é um tanto quanto compacto e relativamente simples, porém apresenta um nível de dificuldade intermediário a partir do momento em que passa a utilizar recursão. Destaca-se três momentos: a) definição do pivô; b) particionamento; c) recursão. Vale acrescentar que o âmago da eficiência deste método de ordenação está no particionamento do vetor e na escolha do pivô. Para o presente relatório, considerou-se o pivô centralizado.

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

```
            j--;
            }
        }
    if(j > began) {quicksort(values, began, j+1);}
    if(i < end) {quicksort(values, i, end);}
}
```

2.2.3 Função de Eficiência e Casos Específicos

De acordo com a literatura de referência (CORMEN, 2002, p. 117-132), o limite de eficiência de tempo do Quicksort está diretamente ligado à variante de código que é escolhida para executar a tarefa. Isto é, devemos levar em consideração que existem adaptações para modificar a forma de particionamento do vetor (dois ou mais segmentos) ou para escolher o pivô (destacando-se a posição inicial, final, mediana, central, aleatório e outros), o que reflete em ganhos ou perdas substanciais de velocidade. Porém, convém ainda mencionar que a análise de eficiência costuma ser baseada no código tradicional com pivo aleatório.

Considerando essas ressalvas, verificamos que o limite superior de desempenho do Quick Sort acontecerá quando houver um desbalanceamento radical da árvore recursiva. Isto é, quando se produz um subproblema com (n-1) elementos de um lado e zero elementos de outro, pois, nesse cenário, a árvore recursiva será a maior possível, gerando um comportamento assintótico compatível com a ordem quadrática $\mathcal{O}(n^2)$. Por outro lado, o algoritmo apresentará limite inferior quanto mais equilibrada for a partição (ou seja, apresentando dois lados de tamanhos iguais ou aproximados), sendo certo que, nesse cenário, a rotina será a mais eficiente - governada pelo $\Omega(n \log n)$. Com relação a complexidade no espaço, verificamos que o algoritmo do Quicksort não ocupa memória extra e, por isso, trata-se de uma excelente opção para ser utilizado em sistemas que possuem memória limitada e precisam de grande eficiência.

2.3 Heapsort e suas Características

2.3.1 Conceito

O Heapsort é baseado na estrutura de árvores binárias (ou seja, cada nó da estrutura possui no máximo dois graus, isto é, duas ligações, aos próximos elementos). Nesse caso, um nó denominado *raiz* irá apontar para outro nó que irá possuir duas configurações básicas: ou este novo nó apresentará uma ou duas ramificações (denominadas subárvores) ou será um nó *folha* (isto é, não teremos subdivisões). Essa estrutura também pode ser representada linearmente por meio de um vetor, organizando-se os elementos pais na posição “i” que irão apresentar conexão com no máximo dois elementos filhos “2i+1” e “2i+2”. Assim, a árvore estará ordenada quando apresentar um mesmo padrão vertical e horizontal, ou seja, quando

os nós filhos (ou derivados) sejam menores (Max-Heap) ou maiores (Min-Heap) que os nós pais, mas, ao mesmo tempo, seus elementos estejam organizados da esquerda para a direita.

2.3.2 Código-Fonte

O código fonte do Heapsort é compacto e utiliza poucas linhas de instruções (baseando-se na estrutura de dados heap), porém apresenta um nível intermediário de dificuldade uma vez que utiliza recursão. Destaca-se por utilizar-se de duas funções:

a) *HeapSort:* responsável por construir a estrutura heap e ordená-la.

b) *MaxHeapify:* ordena a árvore em maxheap, ou seja, o elemento pai possui valor maior que os elementos filhos. Assim, os maiores elementos são posicionados nos menores níveis e vice-versa. No código abaixo, esta função é denominada *peneira()*.

```
void peneira(int *vet, int raiz, int fundo);
void heapSort(int *vet, int n) {
   int i, tmp;
   for (i = (n / 2); i >= 0; i--) {peneira(vet, i, n - 1);}
   for (i = n-1; i >= 1; i--) {
      tmp = vet[0];
      vet[0] = vet[i];
      vet[i] = tmp;
      peneira(vet, 0, i-1);
   }
}

void peneira(int *vet, int raiz, int fundo) {
   int pronto, filhoMax, tmp;
   pronto = 0;
   while ((raiz*2 <= fundo) && (!pronto)) {
      if (raiz*2 == fundo) {
         filhoMax = raiz * 2;
      }
      else if (vet[raiz * 2] > vet[raiz * 2 + 1]) {
         filhoMax = raiz * 2;
      }
      else {filhoMax = raiz * 2 + 1;}
      if (vet[raiz] < vet[filhoMax]) {
         tmp = vet[raiz];
         vet[raiz] = vet[filhoMax];
```

```
            vet[filhoMax] = tmp;
            raiz = filhoMax;
        }
        else {pronto = 1;}
    }
}
```

2.3.3 Função de Eficiência e Casos Específicos

De acordo com a literatura de referência (CORMEN, 2002, p. 103-111), a função de eficiência de tempo do Heapsort é determinada pela estrutura em forma de árvore binária, variando de acordo com a altura da árvore $\Theta(\log n)$ e pelas funções MAX-HEAPIFY (que mantem a ordenação maxheap - $\mathcal{O}(\log n)$) e a HEAPSORT (que ordena um arranjo local - $\mathcal{O}(n \log n)$). Isso acontece porque, se considerarmos o consumo de tempo proporcional ao número de comparações, teremos que a função de eficiência irá variar de acordo com a função $\mathcal{O}(n \log n)$, afinal o algoritmo realizará constantemente duas comparações entre elementos de cada vez. Assim, essa rotina se repete tanto nos casos aleatórios, quanto nos melhores e piores casos. Sendo assim, matematicamente, verificamos que o limite superior máximo do Heapsort é determinado pela função $\mathcal{O}(n \log n)$. Por outro lado, é conveniente esclarecer que, embora esse contexto seja bastante animador ao usuário, a constante de proporcionalidade da função Heapsort costuma ser maior do que aquelas outras existentes no Merge Sort e no Quicksort, razão pela qual o Heapsort é ligeiramente mais lento que esses dois métodos de ordenação.

3 Resultados

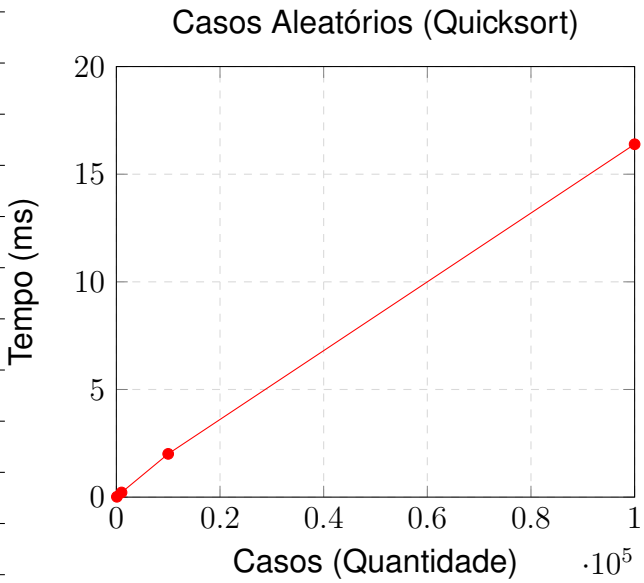
3.1 Analise Assintótica do Quicksort

3.1.1 Casos Aleatórios (100, 1000, 10000 e 100000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

CASOS ALEATÓRIOS QUICKSORT (em ms)

Teste	A100	A1000	A10000	A100000
1	0,01000	0,23600	1,32100	22,39500
2	0,00900	0,26200	2,17600	16,53500
3	0,01000	0,50200	1,57100	15,97700
4	0,00900	0,10800	2,31500	14,97300
5	0,07600	0,17600	1,52700	15,58400
6	0,01000	0,10600	1,30000	15,20200
7	0,01000	0,30400	3,52100	15,42500
8	0,00900	0,17100	1,27400	16,26800
9	0,00900	0,10600	1,45600	15,81500
10	0,01000	0,19300	3,62900	15,74100
Média	**0,01620**	**0,21640**	**2,00900**	**16,39150**



Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,00016 ms para calcular uma unidade no caso de 100 elementos. Da mesma forma, demorou 0,000216 ms no caso de 1000 elementos, 0,0002009 ms para 10000 elementos e 0,00016392 ms para 100000 elementos. Assim, houve um aumento de 1,35 vezes quando se passou do caso de 100 para o de 1000 e, da mesma forma, 1,25 vezes do de 100 para o de 10000 e 1,02 vezes de 100 para o de 100000.

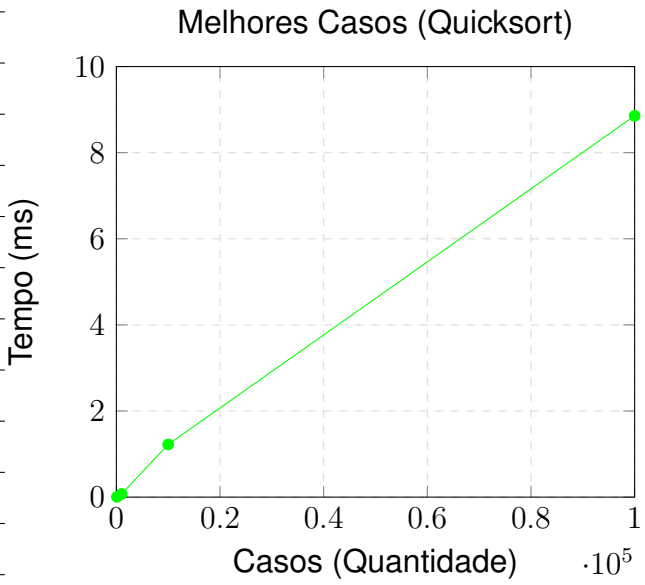
Em outras palavras, o algoritmo não se mostrou estável nos casos testes (apresentando variações significativas), mas, na média, apresentou resultado compatível com o esperado. Porém, não apresentou significativas alterações no tempo médio de ordenação, razão pela qual não apresentou complexidade quadrática.

3.1.2 Melhores Casos (100, 1000, 10000 e 100000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

MELHORES CASOS QUICKSORT (em ms)

Teste	M100	M1000	M10000	M100000
1	0,00700	0,05700	0,79400	8,27600
2	0,00500	0,14000	0,87600	9,14300
3	0,00500	0,05800	3,96400	8,66000
4	0,00500	0,13800	1,08400	9,38500
5	0,00500	0,05600	0,86300	8,92400
6	0,00600	0,05700	0,91800	9,05600
7	0,00500	0,05500	1,24700	9,04200
8	0,00500	0,05700	0,80900	8,55900
9	0,00500	0,05500	0,73700	8,71300
10	0,00400	0,05600	0,93900	8,80300
Média	**0,00520**	**0,07290**	**1,22310**	**8,85610**



Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,000052 ms para calcular uma unidade no caso de 100 elementos. Da mesma forma, demorou 0,0000729 ms no caso de 1000 elementos, 0,00012231 ms para 10000 elementos e 0,000088561 ms para 100000 elementos. Assim, houve um aumento de 1,4 vezes quando se passou do caso de 100 para o de 1000 e, da mesma forma, 23,52 vezes do de 100 para o de 10000 e 1,70 vezes de 100 para o de 100000.

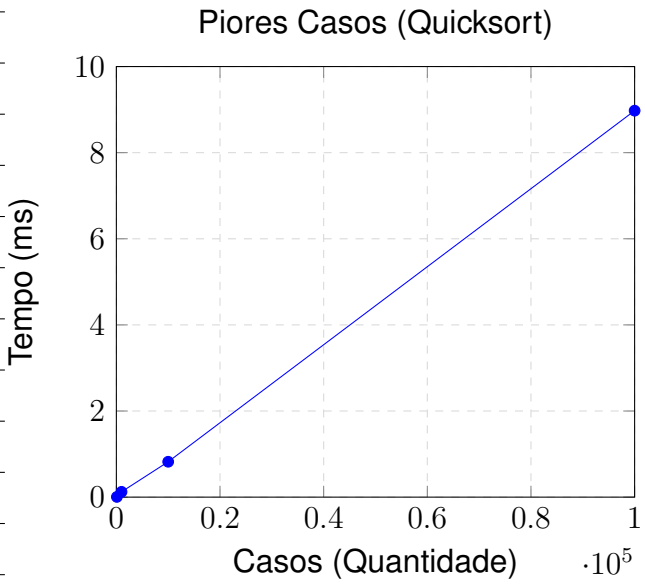
Em outras palavras, o algoritmo não se mostrou estável nos casos testes (apresentando variações significativas), mas, na média, apresentou resultado compatível com o esperado. Porém, não apresentou significativas alterações no tempo médio de ordenação de uma unidade (exceto pelo vetor intermediário de 10.000 elementos). De qualquer forma, não apresentou complexidade quadrática.

3.1.3 Pior Caso (100, 1000, 10000 e 100000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

PIORES CASOS QUICKSORT (em ms)

Teste	P100	P1000	P10000	P100000
1	0,00700	0,05900	0,74800	10,62200
2	0,00500	0,12200	0,75600	8,64800
3	0,00500	0,05700	0,66400	8,50600
4	0,00500	0,12100	0,81900	8,48600
5	0,00500	0,12200	1,01400	9,56500
6	0,00600	0,05700	0,75900	8,80800
7	0,00600	0,20800	0,87900	8,55000
8	0,00500	0,27000	0,66300	9,21400
9	0,00500	0,05700	1,16600	8,76600
10	0,00500	0,15800	0,74700	8,58800
Média	**0,00540**	**0,12310**	**0,82150**	**8,97530**



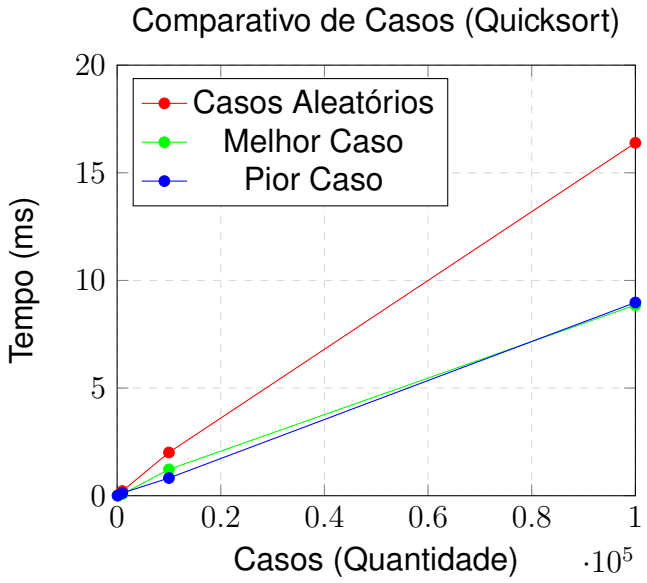
Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,000054 ms para calcular uma unidade no caso de 100 elementos. Da mesma forma, demorou 0,0001231 ms no caso de 1000 elementos, 0,00008215 ms para 10000 elementos e 0,000089753 ms para 100000 elementos. Assim, houve um aumento de 2,24 vezes quando se passou do caso de 100 para o de 1000 e, da mesma forma, 1,52 vezes do de 100 para o de 10000 e 1,66 vezes de 100 para o de 100000.

Em outras palavras, o algoritmo não se mostrou estável nos casos testes (apresentando variações significativas), mas, na média, apresentou resultado compatível com o esperado. Porém, não apresentou significativas alterações no tempo médio de ordenação de uma unidade. De qualquer forma, não apresentou complexidade quadrática.

3.1.4 Comparativo de Casos

Em um primeiro momento, convém mencionar que a escolha do pivô (como elemento central do vetor) interferiu na performance do algoritmo, pois, se tivesse sido utilizado o código tradicional (com a escolha do pivô no primeiro ou no último elemento do vetor), teríamos um eficiência do tipo quadrática quando o vetor estivesse ordenado (seja em ordem crescente ou decrescente, respectivamente, melhores ou piores casos), o que não foi confirmado nos resultados. Além disso, os dados obtidos nos itens anteriores revelam uma compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório, uma vez que denotam eficiência logarítmica do tempo ($\mathcal{O}(n \log n)$).

No entanto, comparando-se os casos do acima, por meio do gráfico abaixo, contatamos que o Quicksort é um mecanismo de ordenação muito eficiente em todos os casos, mas demonstra melhor desempenho para pequenas e médias ordenações.



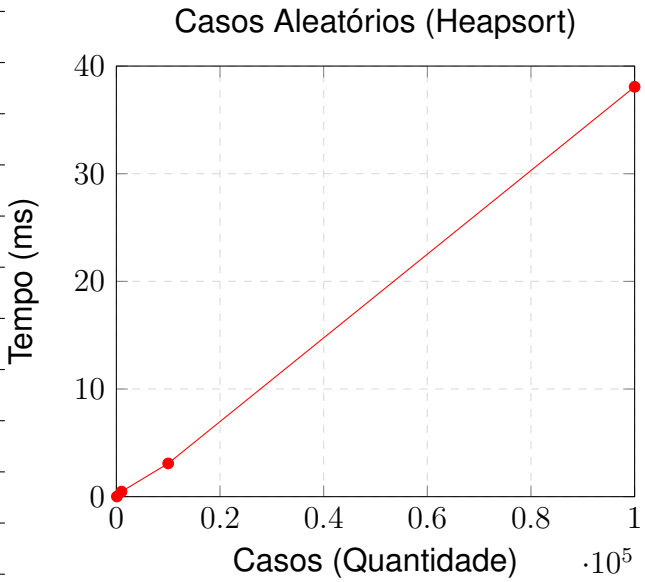
3.2 Analise Assintótica do Heapsort

3.2.1 Casos Aleatórios (100, 1000, 10000 e 100000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

CASOS ALEATÓRIOS HEAPSORT (em ms)

Teste	A100	A1000	A10000	A100000
1	0,01300	0,61500	3,14600	43,56600
2	0,01200	0,17900	2,84500	38,12500
3	0,01300	0,35400	3,34800	37,45400
4	0,01400	1,62000	3,12600	34,65800
5	0,01400	0,17900	2,89400	36,57400
6	0,01400	0,78000	3,94500	34,18900
7	0,01400	0,26000	3,93800	40,90100
8	0,01300	0,39300	2,51200	38,03100
9	0,01400	0,17400	2,66800	40,05000
10	0,01500	0,18000	2,44700	37,20400
Média	**0,01360**	**0,47340**	**3,08690**	**38,07520**



Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,000136 ms para calcular uma unidade no caso de 100 elementos. Da mesma forma, demorou 0,00004734 ms no caso de 1000 elementos, 0,00030869 ms para 10000 elementos e 0,000380752 ms para 100000 elementos. Assim, houve uma redução de 0,348 vezes quando se passou do caso de 100 para o de 1000, mas um aumento de 2,84

vezes do de 100 para o de 10000 e 2,79 vezes de 100 para o de 100000.

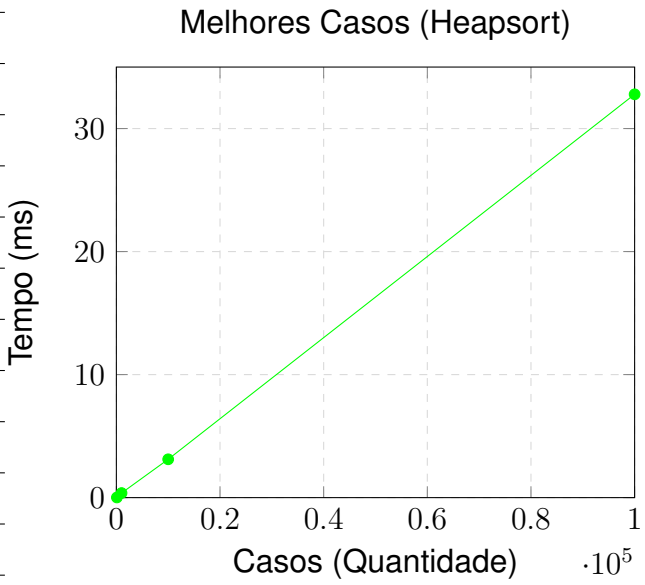
Em outras palavras, o algoritmo não se mostrou estável nos casos testes (apresentando variações significativas), sendo certo que, no caso teste de 1000 elementos, apresentou eficiência máxima. No entanto, de modo geral, apresentou resultado compatível com o esperado. Vale acrescentar que, embora tenha se aumentado significativamente a quantidade de elementos dos casos testes, o algoritmo não apresentou padrão de complexidade do tipo quadrática.

3.2.2 Melhores Casos (100, 1000, 10000 e 100000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

MELHORES CASOS HEAPSORT (em ms)

Teste	M100	M1000	M10000	M100000
1	0,01400	0,16300	3,22100	32,91200
2	0,01300	0,16300	3,53200	32,38000
3	0,01300	0,16200	2,83300	33,00400
4	0,01400	0,16300	2,83300	32,73400
5	0,01300	0,16200	2,65200	33,44900
6	0,01400	0,16100	4,25200	32,87600
7	0,01300	0,16200	2,50700	32,95000
8	0,01300	0,37600	2,70000	31,73200
9	0,01300	0,16200	2,54000	32,15500
10	0,01300	2,04000	4,08600	33,65500
Média	**0,01330**	**0,37140**	**3,11560**	**32,78470**



Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,0001330 ms para calcular uma unidade no caso de 100 elementos. Da mesma forma, demorou 0,00003714 ms no caso de 1000 elementos, 0,00031156 ms para 10000 elementos e 0,000327847 ms para 100000 elementos. Assim, houve uma redução para 0,27 vezes quando se passou do caso de 100 para o de 1000, mas um aumento de 2,34 vezes do de 100 para o de 10000 e 2,46 vezes de 100 para o de 100000.

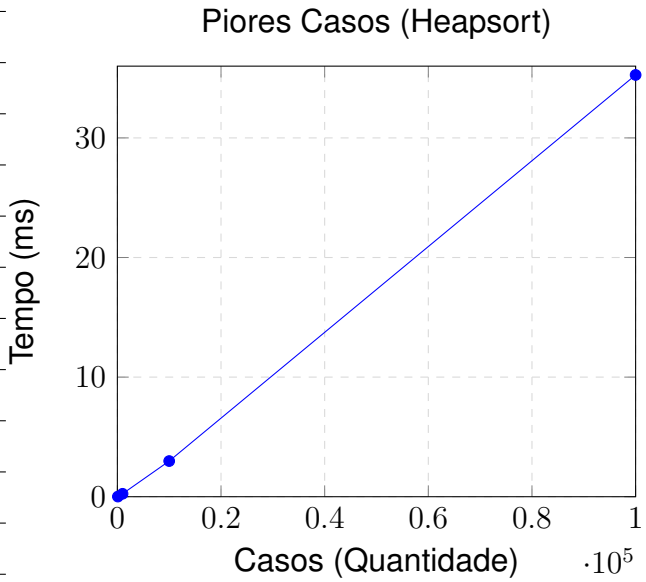
Em outras palavras, o algoritmo não se mostrou estável nos casos testes (apresentando variações significativas), sendo certo que, no caso teste de 1000 elementos, apresentou eficiência máxima. No entanto, de modo geral, apresentou resultado compatível com o esperado. Vale acrescentar que, embora tenha se aumentado significativamente a quantidade de elementos dos casos testes, o algoritmo não apresentou padrão de complexidade do tipo quadrática.

3.2.3 Pior Caso (100, 1000, 10000 e 100000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

PIORES CASOS HEAPSORT (em ms)

Teste	P100	P1000	P10000	P100000
1	0,01200	0,21900	2,70300	36,61600
2	0,01200	0,49500	2,60400	35,14000
3	0,01100	0,15100	3,53800	38,24600
4	0,01200	0,28200	3,58800	34,29300
5	0,01200	0,15100	2,45500	35,11000
6	0,01200	0,34800	2,39800	33,85200
7	0,01200	0,21600	3,18000	33,62300
8	0,01200	0,28100	3,32400	34,48000
9	0,01200	0,15300	3,39500	35,24200
10	0,01100	0,15100	2,62500	36,03200
Média	**0,01180**	**0,24470**	**2,98100**	**35,26340**

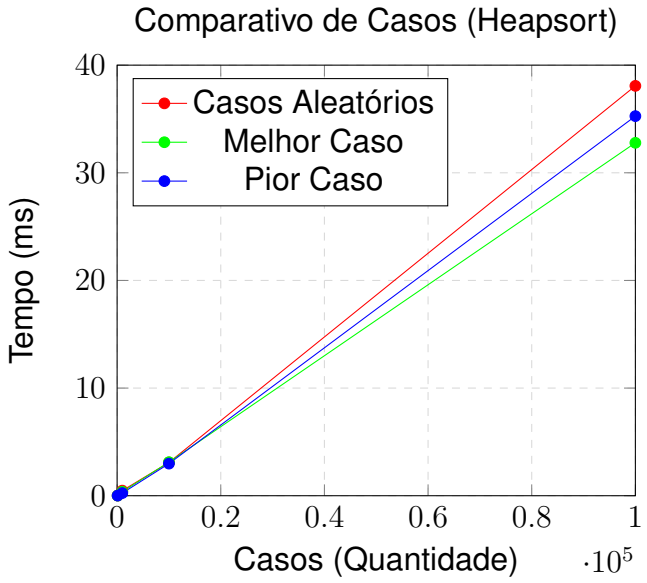


Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,000118 ms para calcular uma unidade no caso de 100 elementos. Da mesma forma, demorou 0,0002447 ms no caso de 1000 elementos, 0,0002981 ms para 10000 elementos e 0,000352634 ms para 100000 elementos. Assim, houve um aumento de 2,07 vezes quando se passou do caso de 100 para o de 1000 e, da mesma forma, 2,52 vezes do de 100 para o de 10000 e 2,98 vezes de 100 para o de 100000.

Em outras palavras, o algoritmo não se mostrou estável nos casos testes (apresentando variações significativas entre um e outro). No entanto, de modo geral, apresentou resultado compatível com o esperado. Vale acrescentar que, embora tenha se aumentado significativamente a quantidade de elementos dos casos testes, o algoritmo não apresentou padrão de complexidade do tipo quadrática.

3.2.4 Comparativo de Casos

Comparando-se os casos do acima, por meio do gráfico abaixo, contatamos que o Heapsort é um mecanismo de ordenação muito eficiente em todos os casos, mas demonstra eficiência de desempenho praticamente equivalente em todos os casos (vetores aleatórios, ordenados de forma crescente ou decrescente).

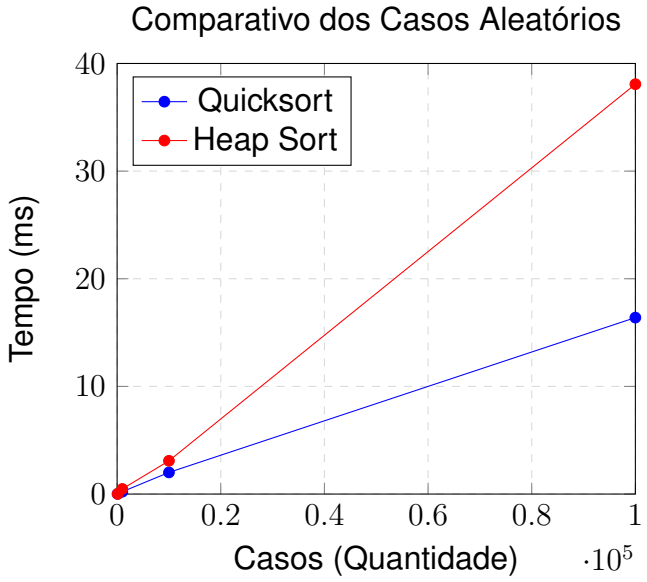


Além disso, os dados obtidos nos itens anteriores revelam uma compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório, uma vez que denotam eficiência logarítmica do tempo (O($n \log n$)).

3.3 Comparativo do Quicksort e do Heapsort

3.3.1 Comparativo dos Casos Aleatórios

Os casos de testes aleatórios do Quicksort e do Heapsort foram plotados conjuntamente no gráfico abaixo.

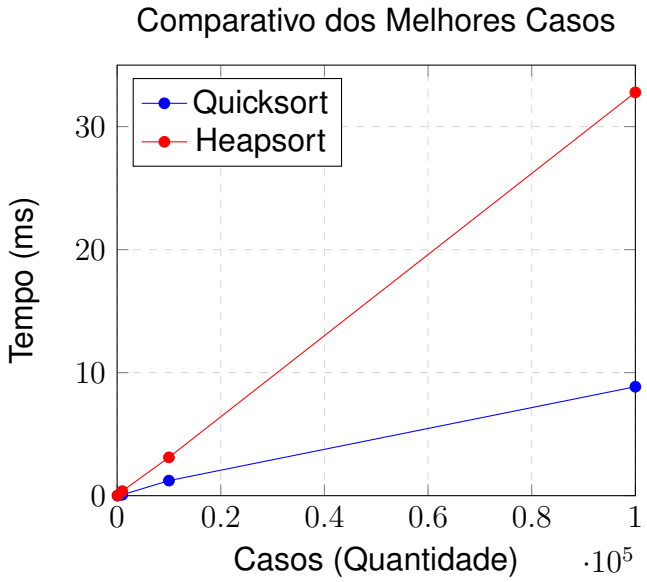


Tendo em vista essas informações, a comparação revela que não há diferença significativa entre a eficiência do Quick Sort e do Heap Sort para vetores pequenos e intermediários (até 1.000 elementos). Porém, a partir deste limite, quanto maior é a entrada, mais eficiente é o

Quicksort em relação ao Heapsort. Nesse contexto, os dados do gráfico mostram compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório.

3.3.2 Comparativo dos Melhores Casos

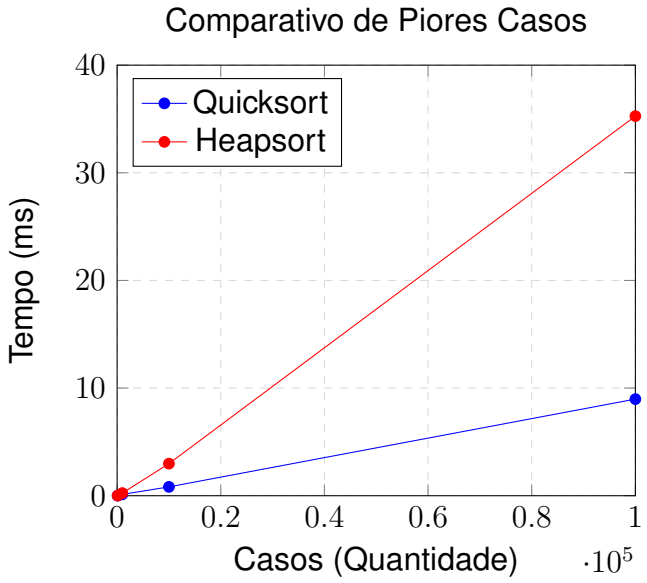
Os melhores casos testes do Quicksort e do Heapsort foram plotados conjuntamente no gráfico abaixo.



Tendo em vista essas informações, a comparação revela que existe uma pequena diferença, porém não significativa, entre a eficiência do Quicksort e do Heapsort para vetores pequenos (até 1.000 elementos). No entanto, a partir desta quantidade de entrada, quanto maior for a entrada, mais eficiente será o Quicksort em relação ao Heapsort. Nesse contexto, os dados do gráfico mostram compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório.

3.3.3 Comparativo dos Piores Casos

Os piores casos testes do Quicksort e do Heapsort foram plotados conjuntamente no gráfico abaixo.

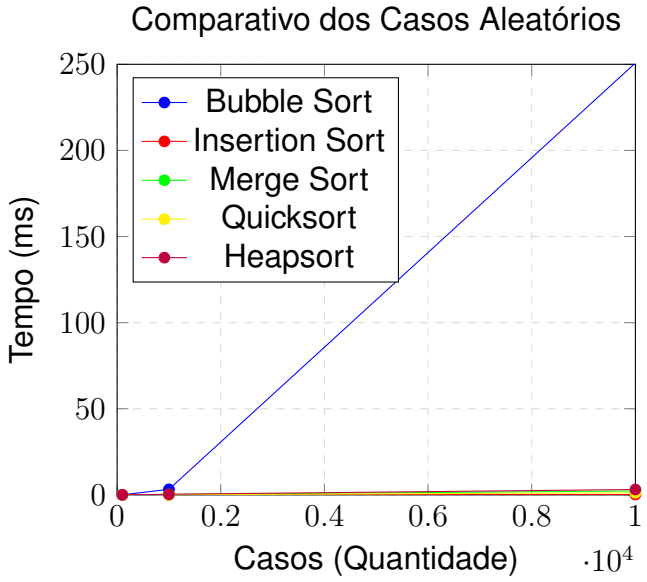


Tendo em vista essas informações, a comparação revela que não há diferença significativa entre a eficiência do Quicksort e do Heapsort para vetores pequenos (até 1.000 elementos). Porém, a partir desta quantidade, quanto maior for a entrada, mais eficiente será o Quicksort em relação ao Heapsort. Nesse contexto, os dados do gráfico mostram compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório.

3.4 Comparativo: Bubble Sort, Insertion Sort, Merge Sort, Quicksort e Heapsort

3.4.1 Comparativo dos Casos Aleatórios

Os casos testes dos melhores casos do Bubble Sort, Insertion Sort, Merge Sort, Quicksort e Heapsort foram plotados conjuntamente no gráfico abaixo.

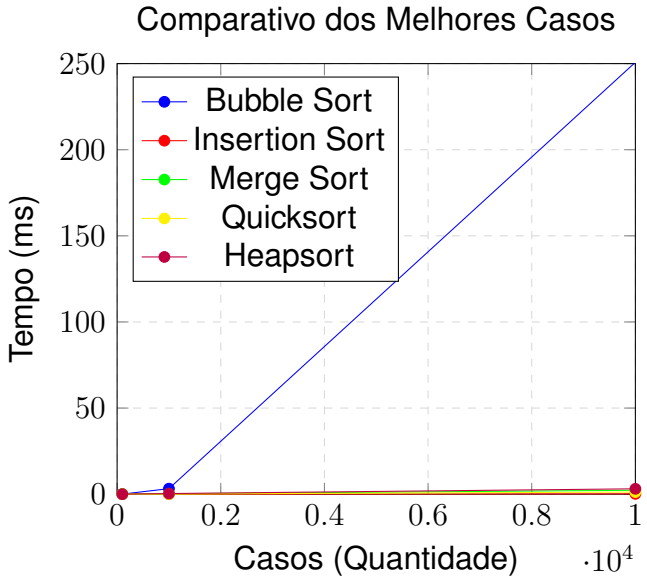


Tendo em vista as informações acima, a comparação revela que não há diferença significativa entre a eficiência do Insertion Sort, Merge Sort, Quicksort e Heapsort para a ordenação de casos aleatórios em toda a extensão dos testes (tanto que as linhas estão praticamente sobrepostas no gráfico). No entanto, o Bubble Sort passa a apresentar diferenças de performance a partir de vetores com 100 elementos, sendo certo que exibe um aumento elevado no tempo de execução especialmente após o caso teste acima de 1.000 elementos.

Nesse contexto, os dados do gráfico mostram compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório.

3.4.2 Comparativo dos Melhores Casos

Os casos testes dos melhores casos do Bubble Sort, Insertion Sort, Merge Sort, Quicksort e Heapsort foram plotados conjuntamente no gráfico abaixo.

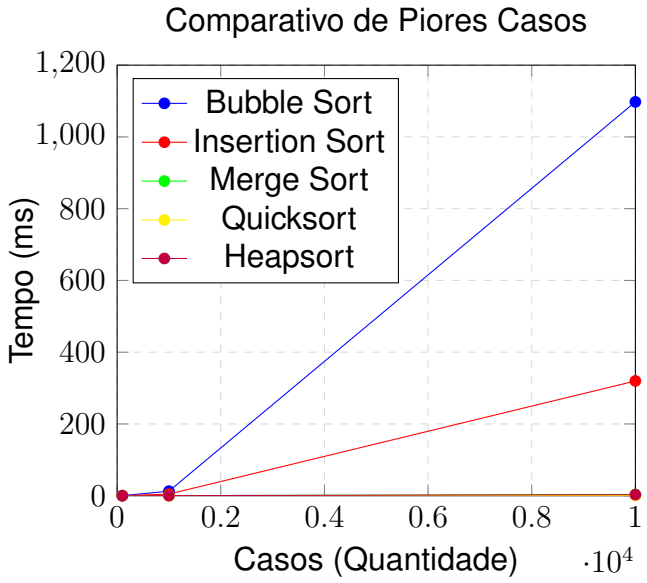


Tendo em vista as informações acima, a comparação revela que não há diferença significativa entre a eficiência do Insertion Sort, Merge Sort, Quicksort e Heapsort para a ordenação de casos aleatórios com vetores de até 10.000 elementos (tanto que as linhas estão sobrepostas no gráfico). Porém, em situação diversa, temos o Bubble Sort, que apresenta elevado tempo de execução após casos acima de 1000 elementos.

Nesse contexto, os dados do gráfico mostram compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório.

3.4.3 Comparativo dos Piores Casos

Os casos testes dos melhores casos do Bubble Sort, Insertion Sort, Merge Sort, Quicksort e Heapsort foram plotados conjuntamente no gráfico abaixo.



Tendo em vista as informações acima, a comparação revela que não há diferença significativa entre a eficiência do Merge Sort, Quicksort e Heapsort para a ordenação de casos aleatórios com vetores de até 10.000 elementos (tanto que as linhas estão sobrepostas no gráfico). Porém, em situação diversa, temos o Insertion Sort e o Bubble Sort, os quais apresentam elevado tempo de execução após casos acima de 1000 elementos, sendo ainda certo mencionar que o Insertion Sort é mais eficiente que o Bubble Sort.

Nesse contexto, os dados do gráfico mostram compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório.

4 Conclusão

Embora tenha havido pequenas variações de performance em casos específicos, concluimos o relatório confirmando as expectativas reunidas inicialmente. Nesse sentido, comprovou-se, por meio de dados empíricos, as informações fornecidas pela literatura especializada e pelas aulas de laboratório. Isto é, mostrou-se, por meio da análise assintótica dos algoritmos de ordenação, que os mecanismos Quicksort e o Heapsort possuem ordem logarítmica $\Theta(n.\log n)$).

5 Referências Bibliográficas

CORMEN, Thomas et alii. Algoritmos: Teoria e Prática. Rio de Janeiro: Editora Elsevier, 2002.

FEOFILOFF, Paulo. HeapSort. Projeto de Algoritmos. Acesso em:
<https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html#peneira-desempenho>.
Data de Acesso: 04/11/2021.

LAFETA, Fernando. Algoritmos de Ordenação - Heap Sort. HEAPSORT, in GITHUB. Acesso:
<https://github.com/Fernando-Lafeta/Algoritmos-de-ordenacao/blob/master/heapsort.c>.
Data de acesso: 03/11/2021.

LEMOS, Carlos Filipe de Castro. 01 - Relatório - Eficiência de Algoritmos de Busca e
Ordenação. USP, ICMC, Laboratório de Introdução a Ciência da Computação II,
São Carlos: 2021.

QUICKSORT, in Wikipedia: a Enciclopedia Livre. Acesso:
<https://pt.wikipedia.org/wiki/Quicksort>. Data de acesso: 03/11/2021.

REZENDE, P. J. de et alii. MC458 - Projeto e Análise de Algorítmos I. Acesso:
<https://www.ic.unicamp.br/~rezende/ensino/mc458/2018s1/MC458-Parte3.pdf>.
Data de acesso: 03/11/2021.

SANCHES, Carlos Alberto Alonso. Estruturas de Dados, Análise de Algoritmos e
Complexidade Estrutural. Acesso em:
<http://www.comp.ita.br/~alonso/ensino/CT234/CT234-Cap06.pdf>
Data de acesso: 03/11/2021.