Relatório Técnico - Projeto Teoria da Complexidade

Gustavo Nogueira de Sousa[1]

[1]Disciplina de Análise e Projetos de Algoritmos
Programa de Pós-Graduação em Engenharia da Computação e Sistemas
Universidade Estadual do Maranhão (UEMA)
São Luís – MA – Brasil

{sougusta}@gmail.com

1. Repositório

- GitHub: *https://github.com/gustavons/teoria_da_complexidade_projeto*
- Google Drive: *https://drive.google.com/drive/folders/1iIYzlVUeYVDkrDkHJ9nIOwVmN9JLCeqr*

2. Algoritmos Escolhidos

Os algoritmos escolhidos foram o *Heap Sort* [hea 2018], *Merge Sort* [Cormen and Balkcom 2017] e *Intro Sort* [Karthikeyan]. Todos estes algoritmos possuem uma complexidade $O(n \log n)$ em todos os casos de execução (pior, melhor e médio). No entanto o motivo da escolha deles foi a sua utilização de memória extra na ordenação. Sendo que o *Heap Sort* utiliza *1* de memória, o*Merge Sort* utiliza *n* de memória e o *Intro Sort* utiliza $log\, n$ de memória [1], como mostrado na Figura 1. Diante disto, a motivação em escolher os algoritmos para a análise neste trabalho foi o consumo de memória na execução de cada algoritmo, apesar de terem complexidades iguais, esse parâmetro de comparação pode influenciar na escolha de um algoritmo para utilização em problemas reais.

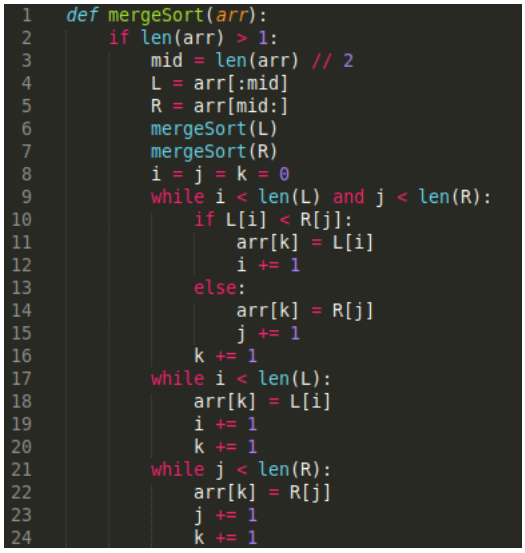
Algoritmo	Melhor Caso	Caso Média	Pior Caso	Memória
Merge sort	$n \log n$	$n \log n$	$n \log n$	n
Introsort	$n \log n$	$n \log n$	$n \log n$	$log\, n$
Heapsort	$n \log n$	$n \log n$	$n \log n$	1

Tabela 1. Performance dos algoritmos escolhidos

2.1. Complexidade - *Merge Sort*

A implementação do algoritmo *Merge Sort* foi realizada por meio do código pronto feito por [chitranayal Mayank Khanna 2 2016]. A seguir serão descritos de acordo com o trecho do código da implementação em python a complexidade deste algoritmo. Toda a complexidade foi calculada com base nos trabalhos de [kha], [chitranayal Mayank Khanna 2 2016] e [Cormen and Balkcom 2017].

[1]https://en.wikipedia.org/wiki/Sorting_algorithm



```
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]
        mergeSort(L)
        mergeSort(R)
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

Figura 1. Algoritmo de execução do *Merge Sort*

Nº linha	Complexidade	Descrição
1	*O(1)*	A tempo constante *O(1)* pois é só uma chamada
2	*O(1)*	Tempo constante *O(1)* pois apenas uma comparação é feita
3	*O(1)*	Tempo constante *O(1)* apenas operação de divisão é feita
4	*O(1)*	Tempo constante, pois apenas uma divisão da lista é feita
5	*O(1)*	Tempo constante, pois apenas uma divisão da lista é feita
6	$\frac{1}{2}n\,log\,n$	Nesta linha dó código é feita uma chamada recursiva utilizando a primeira metade da lista como parâmetro. Devido a isto, o valor é multiplicado por meio.
7	$\frac{1}{2}n\,log\,n$	Nesta linha dó código é feita uma chamada recursiva utilizando a segunda metade da lista como parâmetro. Devido a isto, o valor é multiplicado por meio.
8	3c	Tempo constante *O(1)* pois apenas atribuições são feitas
9	$\frac{1}{3}n$	Proporcionalmente, este processo irá executar um terço de *n*. Isso ocorre devido o processo de ordenação ser feito em *n* e ter outros dois pontos no código que tratam disto.
10	*O(1)*	Tempo constante *O(1)* pois apenas uma comparação é feita
11	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
12	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
13	*O(1)*	Tempo constante *O(1)* pois apenas uma comparação é feita
14	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
15	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
16	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
17	$\frac{1}{3}n$	Proporcionalmente, este processo irá executar um terço de *n*. Isso ocorre devido o processo de ordenação ser feito em *n* e ter outros dois pontos no código que tratam disto.
18	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
19	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
20	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
21	$\frac{1}{3}n$	Proporcionalmente, este processo irá executar um terço de *n*. Isso ocorre devido o processo de ordenação ser feito em *n* e ter outros dois pontos no código que tratam disto.
22	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
23	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
24	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita

Tabela 2. Descrição da complexidade do *Merge Sort*

Considerando os algoritmos apresentado na figura 1 e de acordo com os dados apresentados na Tabela 2 a seguinte expressão é formada:

$$22\textit{O(1)} * + \frac{1}{3}n + \frac{1}{3}n + \frac{1}{3}n + \frac{1}{2}n\,log\,n + \frac{1}{2}n\,log\,n$$

$$\Downarrow$$

$$22\textit{O(1)} + n + n\,log\,n$$

Ao utilizar a notação O os valores constantes são descartados. Sendo assim os valores de *23c* e *n* não farão parte da notação, ficando somente a expressão que gera o maior valor. Com isto, temos que a complexidade deste algoritmo é de $O(n\,log\,n)$.

O algoritmo de ordenação *Merge Sort* ordena os valores com base na divisão e conquista, ou seja, divide a lista de dados em sub-listas menores e as resolvem de maneira recursiva. A seguir é mostrado como se chega ao resultado de *n log n* para o *Merge Sort* com base no trabalho de [Cormen and Balkcom 2017]:

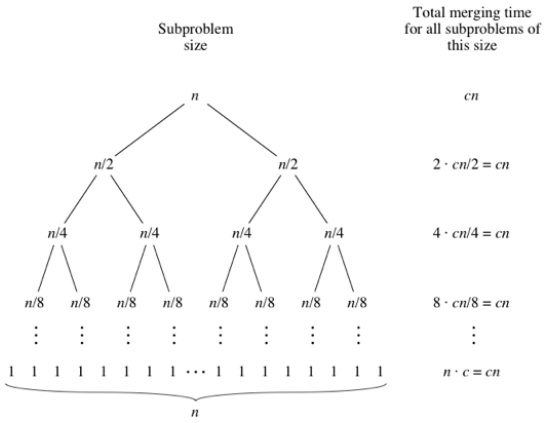


Figura 2. Arvore gerada na execução do *Merge Sort*. Imagem extraída de: https://pt.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort

De acordo com [Cormen and Balkcom 2017], a cada interação (ou nível da árvore) o número de sublistas é dobrado, na primeira interação terá uma lista *n*, na segunda duas lista de tamanho *n/2*, na terceira terá quatro lista de tamanho *n/4* e assim suscetivamente até que o tamanho da lista seja 1. Em todos os níveis da árvore o tempo requerido para a ordenação é de *cn* independente do tamanho. Se a execução recursiva da árvore gera um número *k* de níveis na árvore o tempo requerido para solução de todos os níveis da árvore será de $k * cn$. Dado que $log_2 n + 1$ pode ser usado para determinar o numero exato de niveis de uma árvore binária. Temos que $k = log_2 n + 1$ e $cn * (log_2 n + 1)$. E a complexidade do algoritmo *Merge sort* é de $O(n \log n)$.

2.2. Complexidade - *Heap Sort*

A implementação do algoritmo *Heap Sort* foi realizada por meio do código pronto feito por [Aggarwal et al.]. A seguir serão descritos de acordo com o trecho do código da implementação em python a complexidade deste algoritmo. Toda a complexidade foi calculada com base nos trabalhos de [kha], [InterviewCake], [Aggarwal et al.], [pro 2018] e [Cormen and Balkcom 2017].

```
1   def heapify(arr, n, i):
2       largest = i
3       l = 2 * i + 1
4       r = 2 * i + 2
5       if l < n and arr[i] < arr[l]:
6           largest = l
7       if r < n and arr[largest] < arr[r]:
8           largest = r
9       if largest != i:
10          arr[i], arr[largest] = arr[largest], arr[i]
11          heapify(arr, n, largest)
```

Figura 3. Algoritmo de execução do *Heap Sort* método de *heapfy*

Nº linha	Complexidade	Descrição
1	*O(1)*	A tempo constante *O(1)* pois é só uma chamada
2	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
3	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
4	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
5	2*O(1)*	Tempo constante 2*O(1)* pois apenas duas comparação é feita
6	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
7	2*O(1)*	Tempo constante 2*O(1)* pois apenas duas comparação é feita
8	*O(1)*	Tempo constante *O(1)* pois apenas uma atribuição é feita
9	*O(1)*	Tempo constante *O(1)* pois apenas uma comparação é feita
10	2*O(1)*	Tempo constante 2*O(1)* pois apenas duas atribuições é feita
11	$2(log_2\, n + 1)$	Devido sua a característica de arvore binaria similar à apresentada na Figura 2, na qual o numero de cadas da arvore pode ser determinado por $log_2\, n + 1$. A interações recursivas deste algoritmo ocorrem de acordo como o número da arvore e cada interação faz no máximo duas comparações. Portanto, é possível deduzir que a complexidade desta linha é de $2(log_2\, n + 1)$.

Tabela 3. Descrição da complexidade do *Heap Sort* método de *heapfy* apresentado no Figura 3

Assim sendo, a execução do método exemplificado pela figura 3 e detalhado pela tabela 4 pode ser visto nas formulações abaixo:

$$13O(1) + 2(log_2\, n + 1)$$

$$\Downarrow$$

$$2\, log_2\, n + 13O(1) + 2$$

Com isto temos que a complexidade deste método chamado *heapfy* apresentado na figura 3 é de $2\, log_2\, n$, ou seja $O(\, log_2\, n)$.

```
1  def heapSort(arr):
2      n = len(arr)
3      for i in range(n // 2 - 1, -1, -1):
4          heapify(arr, n, i)
5      for i in range(n - 1, 0, -1):
6          arr[i], arr[0] = arr[0], arr[i]
7          heapify(arr, i, 0)
```

Figura 4. Algoritmo de execução do *Heap Sort*

Nº linha	Complexidade	Descrição
1	*O(1)*	A tempo constante *O(1)* pois é só uma chamada
2	2*O(1)*	Tempo constante 2*O(1)* pois uma contagem e uma atribuição é feita
3	$\frac{n}{2}$	Tempo é de $\frac{n}{2}$ pois é utilizado somente metade da lista
4	$n\,log_2\,n$	Como apresentado na tabela 3 uma chamada do método *heapfy* representa uma complexidade $2\,log_2\,n$. Sendo assim, essa linha terá uma complexidade de $n\,log_2\,n$ por conta de está dentro do escopo de um laço de complexidade $\frac{n}{2}$.
5	n	Tempo é de n pois a lista é percorrida no laço
6	2*O(1)*	Tempo constante 2*O(1)* pois apenas duas atribuições é feita
7	$2n\,log_2\,n$	Como apresentado na tabela 3 uma chamada do método *heapfy* representa uma complexidade $2\,log_2\,n$. Sendo assim, essa linha terá uma complexidade de $2n\,log_2\,n$ por conta de está dentro do escopo de um laço de complexidade n

Tabela 4. Descrição da complexidade do *Heap Sort* do método apresentado na figura 4

Diante disto, o algoritmo apresentado na figura 4 e detalhado na tabela 4 tem a seguinte formulação de complexidade:

$$5O(1) + nlog_2\,n + n + 2nlog_2\,n$$

$$\Downarrow$$

$$3n\,log_2\,n + n + 5O(1)\,n$$

Com isto temos que a complexidade deste método de ordenação *heapSort* apresentado na figura 4 é de $n\,log_2\,n$, ou seja $O(n\,log_2\,n)$.

2.3. Complexidade - *Intro Sort*

A implementação do algoritmo *Intro Sort* foi realizada por meio do código pronto feito por [Karthikeyan] e [Aggarwal et al.]. A seguir são descrito, com referencias a implementação em Python, a complexidade deste algoritmo. Toda a complexidade foi calculada com base nos trabalhos de [kha], [InterviewCake], [Aggarwal et al.], [pro 2018], [chitranayal Mayank Khanna 2 2016] e [Cormen and Balkcom 2017].

```
1  def Partition(arr, low, high):
2
3      pivot = arr[high]
4
5      i = low - 1
6      for j in range(low, high):
7          if arr[j] <= pivot:
8              i = i + 1
9              (arr[i], arr[j]) = (arr[j], arr[i])
10     (arr[i + 1], arr[high]) = (arr[high], arr[i + 1])
11     return i + 1
```

Figura 5. Algoritmo de execução do *Intro Sort* - Etapa 5

Nº linha	Complexidade	Descrição
1	*O(1)*	Tempo constante *O(1)*, apenas a definição do método
2	-	Linha em branco
3	*O(1)*	Tempo constante *O(1)*, apenas atribuição de valor
4	-	Linha em branco
5	*O(1)*	Tempo constante *O(1)*, apenas atribuição de valor
6	n	Tempo linear (n), pois o laço irá executar o tamanho da lista
7	*O(1)*	Tempo constante *O(1)*, apenas uma comparação é feita
8	*O(1)*	Tempo constante *O(1)*, apenas atribuição de valor
9	2*O(1)*	Tempo constante 2*O(1)*, apenas atribuição de valor
10	2*O(1)*	Tempo constante 2*O(1)*, apenas atribuição de valor
11	*O(1)*	Tempo constante *O(1)*, apenas atribuição de valor

Tabela 5. Descrição da complexidade do *Intro Sort* - método *Partition*

Dado o método apresentado na figura 5 e sua descrição na tabela 5 temos que sua complexidade é:

$$7c + 4O(1)n$$

$$\Downarrow$$

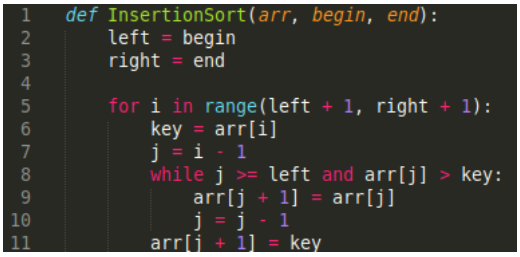
$$4O(1)n + 7O(1)$$

Diante disto, podemos definir que a complexidade do método apresentado na figura 5 é de *O(n)*.

```
1   def MedianOfThree(arr, a, b, d):
2       A = arr[a]
3       B = arr[b]
4       C = arr[d]
5       if A <= B and B <= C:
6           return b
7       if C <= B and B <= A:
8           return b
9       if B <= A and A <= C:
10          return a
11      if C <= A and A <= B:
12          return a
13      if B <= C and C <= A:
14          return d
15      if A <= C and C <= B:
16          return d
```

Figura 6. Algoritmo de execução do *Intro Sort* - Etapa 4

Na figura 6 há 12 comparações, 3 atribuições e no máximo um retorno (*return*). Diante disto podemos dizer que a complexidade é de *O(1)*



```
1   def InsertionSort(arr, begin, end):
2       left = begin
3       right = end
4
5       for i in range(left + 1, right + 1):
6           key = arr[i]
7           j = i - 1
8           while j >= left and arr[j] > key:
9               arr[j + 1] = arr[j]
10              j = j - 1
11          arr[j + 1] = key
```

Figura 7. Algoritmo de execução do *Intro Sort* - Etapa 3

Nº linha	Complexidade	Descrição
1	*O(1)*	Tempo constante *O(1)*, apenas a definição do método
2	*O(1)*	Tempo constante *O(1)*, apenas atribuição de valor
3	*O(1)*	Tempo constante *O(1)*, apenas atribuição de valor
4	-	Linha em branco
5	n	Tempo linear (n), pois o laço irá executar o tamanho da lista
6	*O(1)*	Tempo constante *O(1)*, apenas atribuição de valor
7	*O(1)*	Tempo constante *O(1)*, apenas atribuição de valor
8	n	Tempo linear (n), pois o laço irá executar o tamanho da lista
9	*O(1)*	Tempo constante *O(1)*, apenas atribuição de valor
10	*O(1)*	Tempo constante *O(1)*, apenas atribuição de valor
11	*O(1)*	Tempo constante *O(1)*, apenas atribuição de valor

Tabela 6. Descrição da complexidade do *Intro Sort* - método *InsertionSort*

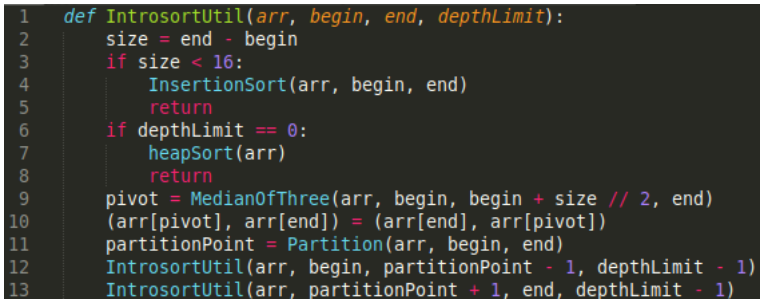
O método apresentado na figura 7 e exemplificado na tabela 6 permite a seguinte formulação de complexidade:

$$4O(1) + 4O(1)n^2$$

$$\Downarrow$$

$$4O(1)n^2 + 4O(1)$$

Diante disto, podemos definir que a complexidade do método apresentado na figura 7 é de $O(n^2)$. Porém, devido a característica do método *IntroSort* a complexidade deste método será de no máximo $O(16^2)$, pois esse método só irá ordenar lista com o *n* menor que 16.



```
def IntrosortUtil(arr, begin, end, depthLimit):
    size = end - begin
    if size < 16:
        InsertionSort(arr, begin, end)
        return
    if depthLimit == 0:
        heapSort(arr)
        return
    pivot = MedianOfThree(arr, begin, begin + size // 2, end)
    (arr[pivot], arr[end]) = (arr[end], arr[pivot])
    partitionPoint = Partition(arr, begin, end)
    IntrosortUtil(arr, begin, partitionPoint - 1, depthLimit - 1)
    IntrosortUtil(arr, partitionPoint + 1, end, depthLimit - 1)
```

Figura 8. Algoritmo de execução do *Intro Sort* - Etapa 2

Nº linha	Complexidade	Descrição
1	*O(1)*	Tempo constante *O(1)*, apenas a definição do método
2	*O(1)*	Tempo constante *O(1)*, apenas verificação de valor
3	$O(16^2)$	Tempo $O(16^2)$,pois devido a característica deste método de ordenação *IntroSort* o *Insertion Sort* irá executar no máximo 16^2.
4	*O(1)*	Tempo constante *O(1)*, apenas um retorno
5	*O(1)*	Tempo constante *O(1)*, apenas verificação de valor
6	*O(1)*	Tempo constante *O(1)*, apenas atribuição de valor
7	$n\,log_2\,n$	Tempo constante $(n\,log_2\,n)$, pois essa é a complexidade do método *HeapSort* utilizado e é o mesmo dá Figura 4.
8	*O(1)*	Tempo constante *O(1)*, apenas um retorno
9	*O(1)*	Tempo constante *O(1)*, complexidade do método *MedianOfThree* já mostrada na Figura 6.
10	2*O(1)*	Tempo constante 2*O(1)*, apenas atribuições de valor
11	*O(n)*	Tempo constante *O(n)*, complexidade do método *Partition* já mostrado na Figura 5.
12	$\frac{1}{2}n\,log\,n$	Tempo $\frac{1}{2}n\,log\,n$, de uma chamada do *IntrosortUtil* com a metade do lista.
13	$\frac{1}{2}n\,log\,n$	Tempo $\frac{1}{2}n\,log\,n$, de uma chamada do *IntrosortUtil* com a metade do lista.

Tabela 7. Descrição da complexidade do *Intro Sort* - método *IntrosortUtil*

O método apresentado na figura 8 e exemplificado na tabela 7 permite a seguinte formulação de complexidade:

$$\frac{1}{2}n\,log\,n + \frac{1}{2}n\,log\,n + O(n) + O(1) + O(16^2) + 8O(1) + 16^2$$

$$\Downarrow$$

$$n\,log\,n + O(n) + O(1) + O(16^2) + 8O(1) + 16^2$$

Diante disto, podemos definir que a complexidade do método apresentado na figura 8 é de $O(n\,log\,n)$.

```
1  def Introsort(arr):
2      begin = 0
3      end = len(arr)-1
4
5      depthLimit = 2 * math.log2(end - begin)
6      IntrosortUtil(arr, begin, end, depthLimit)
```

Figura 9. Algoritmo de execução do *Intro Sort* - Etapa 1

Na figura 9 há 3 atribuições e uma chamada no método *IntrosortUtil*. Isso se traduz em uma complexidade de $3O(1) + O(n\,log\,n)$. Diante disto é possível definir a complexidade como sendo $O(n\,log\,n)$.

3. Testes Realizados

Todos os algoritmos foram implementados em Python 3.7 e utilizado o reuso de códigos, como já explicado anteriormente. Os testes foram realizados em um computador executando uma versão do Linux (’Linux-4.19.112+-x86_64-with-debian-9.4), com 16 Gb de memória RAM, processador Intel(R) Xeon(R) CPU @ 2.30GHz de dois núcleos.

3.1. Tempos de execução por Casos de Execução

Nesta sub-seção será discutido os tempos de execução dos algoritmos utilizados neste estudo.

3.1.1. Execução do Pior Caso

Na figura 10 é apresentado um gráfico com as execuções dos piores caso de cada algoritmos. Nele é possível observar que é difícil de notar a diferença dos tempos de execução até o tamanho da entrada de número *100 000*. Porém a partir deste valor é possível notar o quão diferente são os algoritmos, e com o tamanho da entrada de *1 000 000* vemos a diferença entre os tempos.

No tamanho da entrada de *1 000 000* é possível observar que mesmo os algoritmos tendo complexidades iguais, o *Intro Sort* se destaca com o melhor tempo, de 21 segundos. O mesmo ocorre com um tamanho da entrada de *10 000 000* com o *Intro Sort* obtendo o melhor resultado, com 240 segundos.

Em todos os tamanhos de entrada nos testes é possível observar que o mesmo padrão observado nos últimos tamanho de teste é visto nas entradas menores, com o *Heap Sort* sendo o pior caso em todos, o *Merge Sort* sendo o segundo melhor algoritmo em todos os testes e o *Intro Sort* sendo o melhor algoritmo em todos os tamanhos de entradas.

Ao analisar os resultados de cada testes com tamanhos de entrada diferentes, é possível deduzir que o tamanho da entrada não influenciou em melhora ou piora nos resultados dos algoritmos quando comparamos com os outros utilizados nos testes. Além disto, um ponto que chama a atenção é quando comparamos a ordem de crescimento dos tempos de cada algoritmo com a ordem de crescimento dos tamanhos das entradas de teste. As entrada de testes cresceram em 10 vezes o valor anterior, porém os tempos cresceram muito mais que 10 vezes o valor do tempo anterior, chegando a 25 vezes no caso do *Intro Sort*.

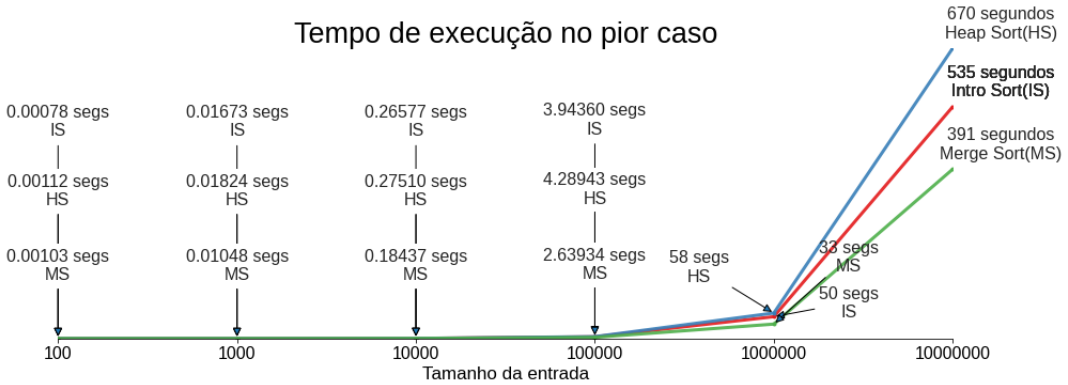


Figura 10. Tempos de execução dos piores casos bos algoritmos testados

3.1.2. Execução do Melhor Caso

Na figura 11 é apresentado um gráfico com as melhores execuções de cada algoritmo. Da mesma forma que nas execuções do pior caso, também vemos que no gráfico a diferença entre os algoritmos fica mais nítida a partir de uma entrada de tamanho *100 000*.

Nesta execução um ponto que chama a atenção é o fato de que o tamanho da entrada influenciou nos resultados dos algoritmos. Por exemplo, em uma entrada do tamanho *100* o melhor algoritmo foi o *Intro Sort*, o segundo o *Heap Sort* e o terceiro o *Merge Sort*, porém no próximo teste com a entrada de tamanho *1000* o melhor algoritmo foi o *Merge Sort*, o segundo o *Heap Sort* e o terceiro o *Intro Sort*. Com isto em mente, é possível ver que o *Heap Sort* fica com o segundo melhor caso em todos os teste com entradas menores que *1 000 000* e o *Intro Sort* e *Merge Sort* alternando entre si.

O *Merge Sort* chama a atenção nos testes com entradas iguais ou superiores que *1 000 000*, ficando sempre como melhor resultado. Já o *Heap Sort* que com entradas inferiores a *1 000 000* sempre foi o segundo melhor, em valores superiores ficou sendo o pior caso. Isto indica um ponto negativo para o *Heap Sort*, pois mostra que o algoritmo não lida muito bem com valores de entrada grandes.

Outro ponto que chama a atenção é a ordem de crescimento, que da mesma forma que nas execução do pior caso ficou bem acima de 10 vezes. Além disto, na entrada de tamanho *10 000 000* é possível observar que a diferença do melhor algoritmo para o pior é quase o dobro da quantidade de segundo.

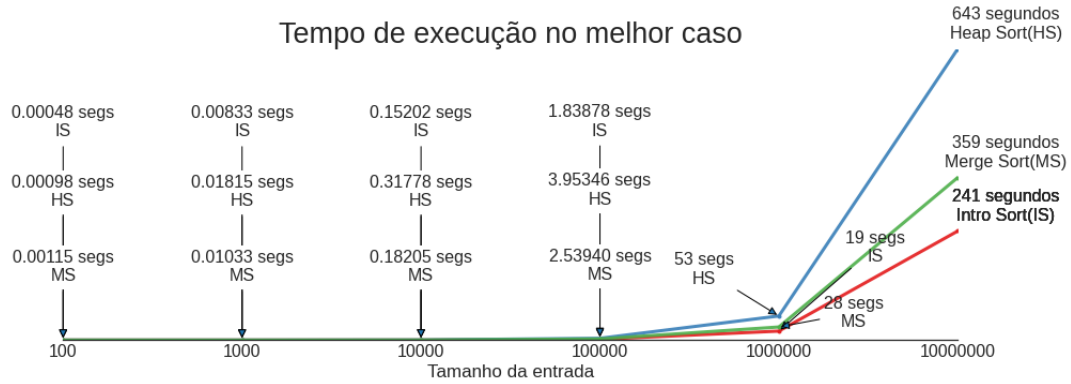


Figura 11. Tempos de execução dos melhores casos nos algoritmos testados

3.1.3. Execução do Caso Médio

Na figura 12 é apresentado um gráfico com os casos médios das execuções de cada algoritmo. Da mesma forma que nas execuções do pior e melhor caso, também vemos que no gráfico a diferença entre os algoritmos fica mais nítida a partir de uma entrada de tamanho *100 000*. Porém, em resultados, as execuções com valores menores, revelam a sensibilidade dos algoritmos ao tamanho das entradas, com exceção do *Heap Sort* que foi o pior em todos os casos. Os algoritmos *Merge Sort* e *Heap Sort* em execuções com valores menores que *100 000* alternaram-se as posições, hora um era melhor e o outro era pior.

Em todos os casos é possível observar que o *Heap Sort* foi o que teve o pior resultado, sendo que em todos teve seu tempo de execução com mais que o dobro do melhor resultado. Da mesma que o pior e o melhor caso médio também teve uma variação maior que 10 vezes de uma execução para a outra.

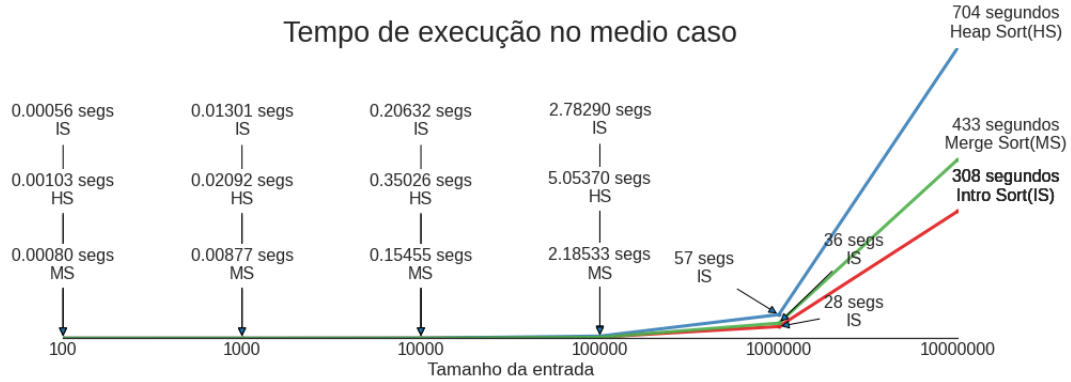


Figura 12. Tempos de execução dos casos médios nos algoritmos testados

3.2. Tempos de execução por Algoritmos

3.2.1. Tempos - Merge Sort

No *Merge Sort* os tempos entres os casos testados tiveram pouca variação. Na figura 13 contém a comparação dos três casos testados com diferentes valores de entrada. Os

tempos de execução tiveram pouca variação entre os casos, pois em todos os casos o *Merge Sort* tem complexidade *n log n*.

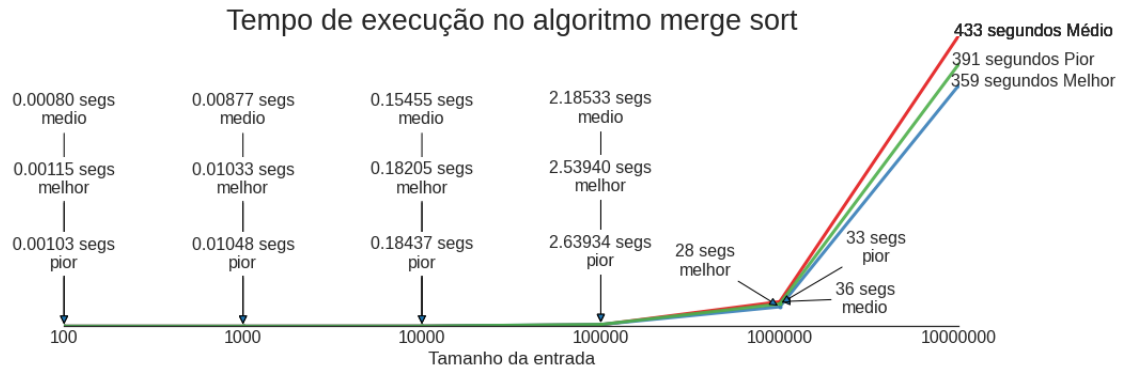


Figura 13. Tempos de execução dos casos do algoritmo Merge Sort

3.2.2. Tempos - Heap Sort

No *Heap Sort*, assim como no *Heap Sort*, os tempos tiveram poucas variações. Na figura 14 contém a comparação dos três casos testados com diferentes valores de entrada.

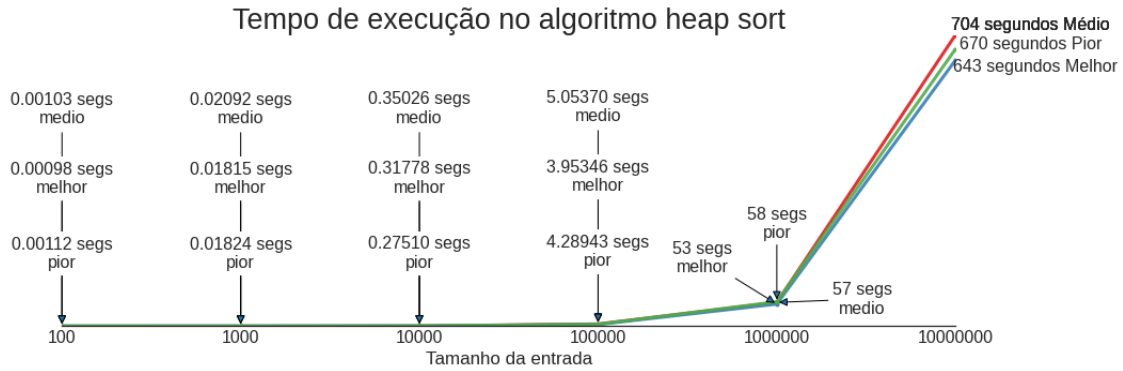


Figura 14. Tempos de execução dos casos do algoritmo Heap Sort

3.2.3. Tempos - Intro Sort

No *Intro Sort* os tempos tiveram uma variação considerável entre os casos testados. Na figura 15 contém a comparação dos três casos testados com diferentes valores de entrada. Nele vemos que o tempo de execução do pior caso foi muito maior que os tempos dos outros casos. A variação deste tempo pode ter ocorrido devido a característica híbrida deste algoritmos.

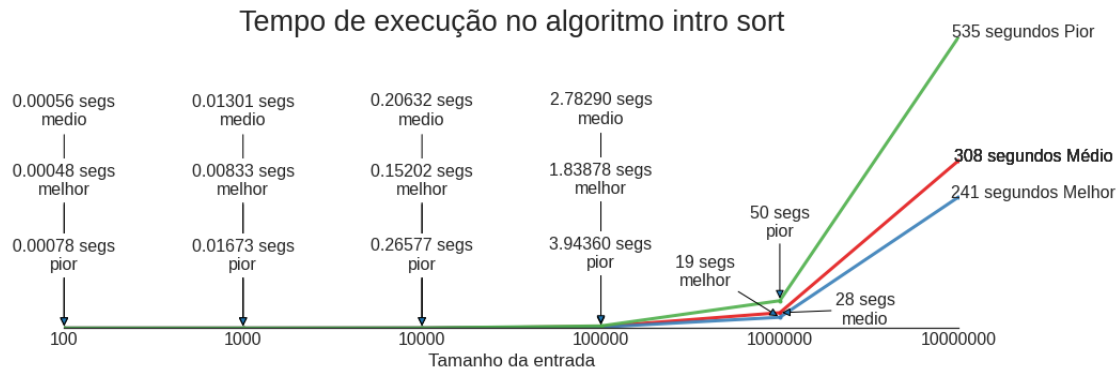


Figura 15. Tempos de execução dos casos do algoritmo Intro Sort

3.3. Uso de memória por Casos de Execução

As Figura 16, 17 e 18 apresentam as medições do uso de memória por cada um dos algoritmos. Um ponto que chama muita atenção é que nos três tipos de testes houve nada ou muita pouca variação, os tamanhos das entradas nos três casos pouco afetou a memória, por exemplo o *Merge Sort* no pior, melhor e no caso médio sempre utilizou aproximadamente 160 Mb de memórias.

Dos algoritmos testados, o único que teve variações significativas no uso das memórias foi o *Intro Sort* que em sua complexidade utilizada $log\, n$ de memória. O *Heap Sort* utiliza sempre *1* de memória, e isso pode ser observado nos resultados, em que em todas as execução a quantidade de memória utilizada ficou em todos de *0.00300* Mb, se pondo como o algoritmo mais eficiente em termos de utilização de memória. O *Merge Sort* foi o algoritmo que consumiu a maior quantidade de memória, e seu consumo de memória é dado em *n*, este algoritmo chegou a utilizar aproximadamente *160* Mb na entrada de tamanho *10 000 000* enquanto o *Heap Sort* utilizou somente *0.00300* Mb.

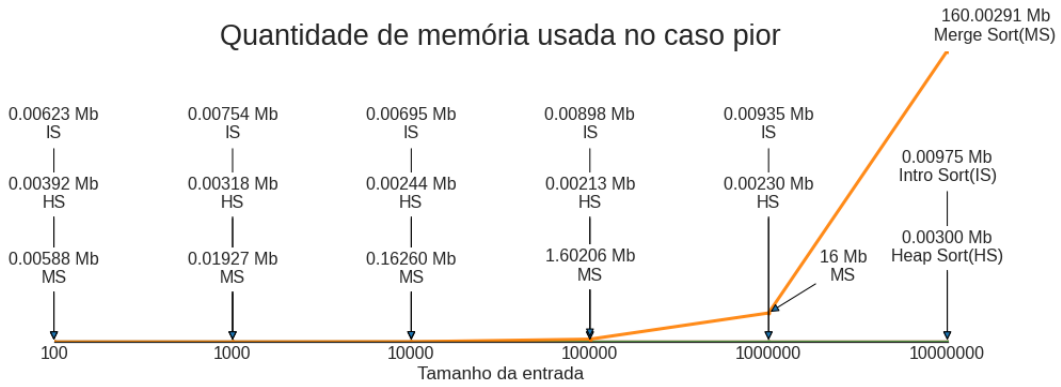


Figura 16. Uso da memória dos piores casos nos algoritmos testados

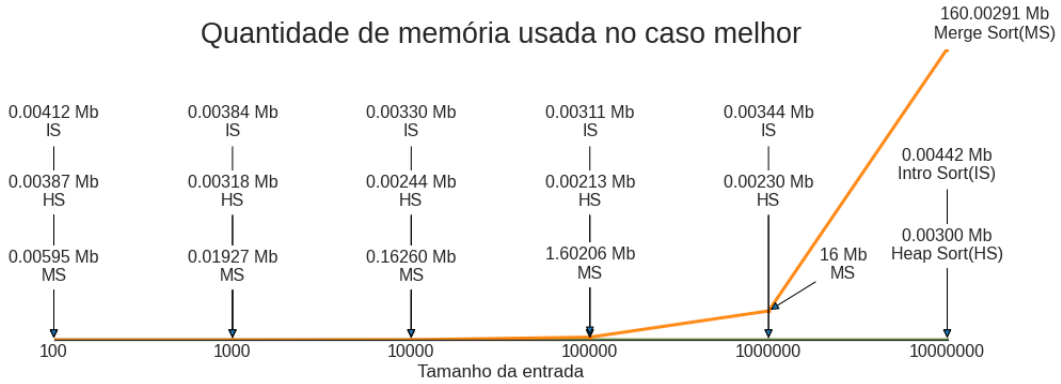


Figura 17. Uso da memória dos melhores casos nos algoritmos testados

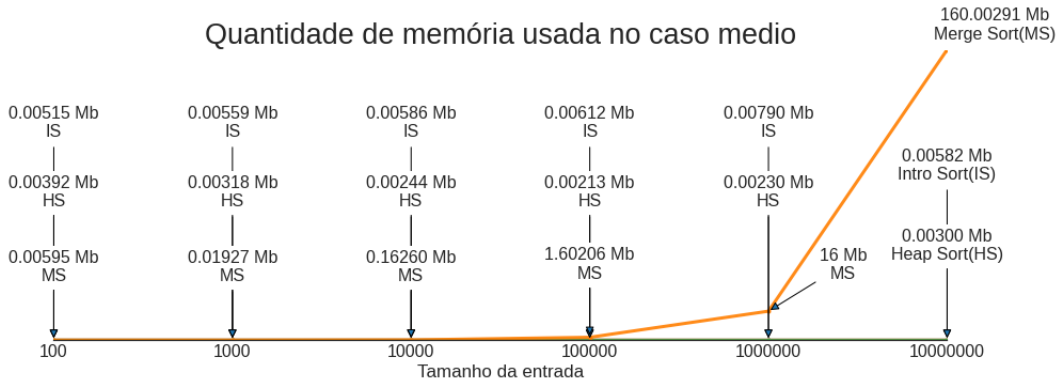


Figura 18. Uso da memória dos casos médios nos algoritmos testados

3.4. Uso de memória por Algoritmos

3.4.1. Memória - Merge Sort

Na figura 19 é apresentado a quantidade de memória utilizada nos testes realizados com o *Merge Sort*. A partir do tamanho da entrada de tamanho *10000* é possível observar que a utilização da memória aumentou cerca de 10 vezes no teste seguinte.

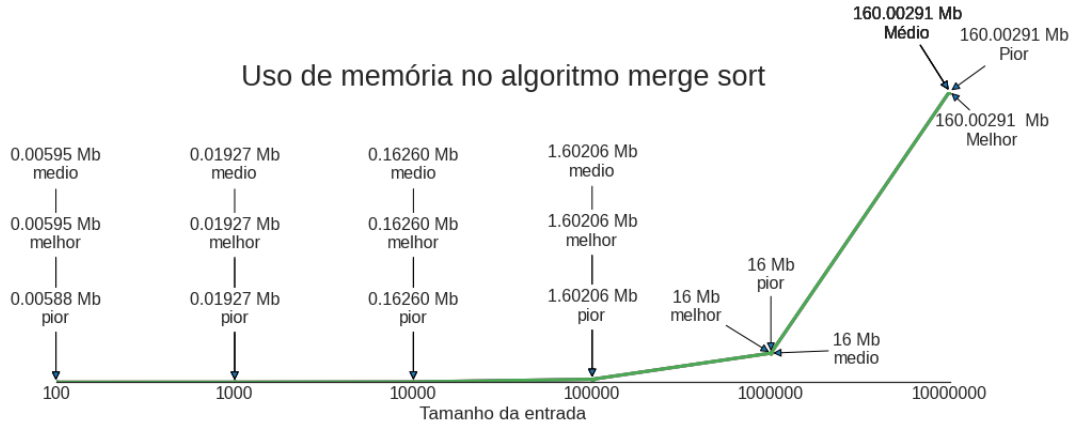


Figura 19. Memória de execução dos casos do algoritmo Merge Sort

3.4.2. Memória - Heap Sort

Na figura 20 é apresentado a quantidade de memória utilizada nos testes realizados com o *Heap Sort*. É possível observar que a utilização de memória é muito baixa em todos dos testes. Na entrada de tamanho *100* a utilização de memória foi maior que nos demais testes, porém os valores são muito pequenos e próximos uns dos outros.

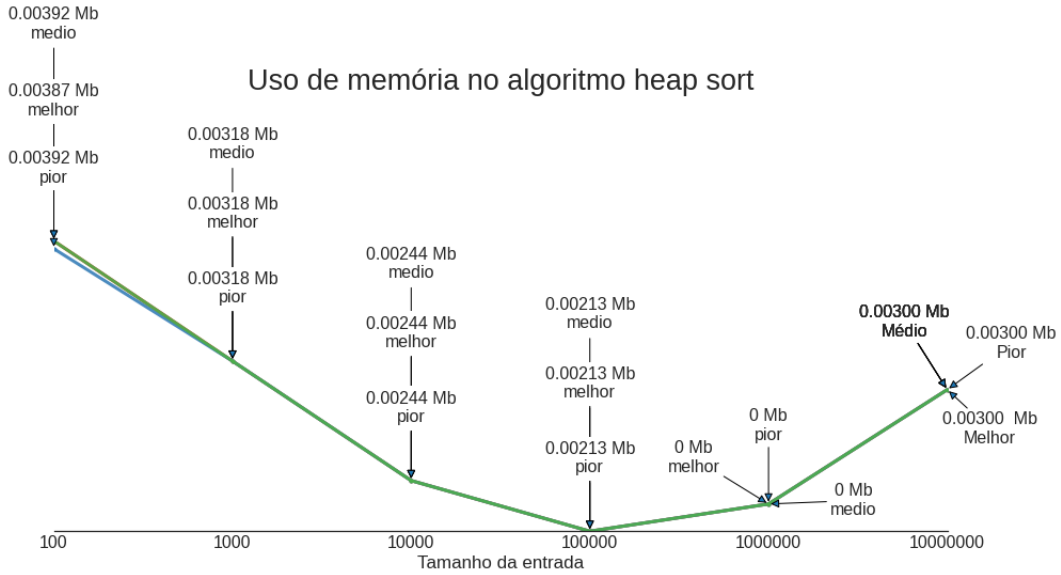


Figura 20. Memória de execução dos casos do algoritmo Heap Sort

3.4.3. Memória - Intro Sort

Na figura 21 é apresentado a quantidade de memória utilizada nos testes realizados com o *Intro Sort*. Nele é possível observar que a quantidade de memória utilizada foi muito pequena em todos os casos. Além disto, podemos observar que os casos representaram literalmente a utilização de memória com seu nome, pois o pior caso utilizou mais memória em todos os casos, o caso médio teve a segunda maior utilização de memória e o melhor caso utilizou menos memórias que todos os outros casos.

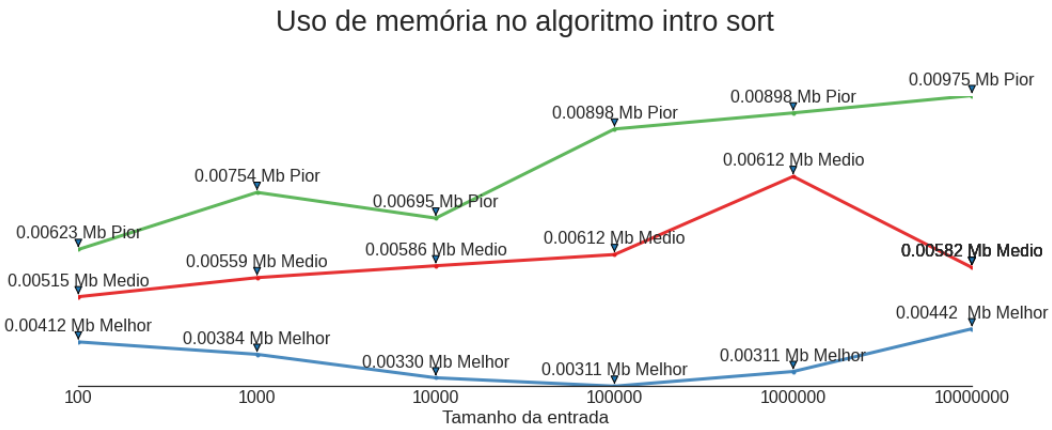


Figura 21. Memória de execução dos casos do algoritmo Intro Sort

4. Discussão dos resultados

A seguir segue alguma considerações importantes a respeito dos resultados obtidos :

- Os resultados com entrada menores que *100 000* foram imperceptíveis ao analisar as linhas dos gráficos com os resultados temporais;
- As diferenças só ficaram mais nítidas em entradas superiores a *100 000* nos resultados temporais;
- Ao executar o pior caso de cada algoritmo o *Intro Sort* foi o que se saiu melhor no seu pior caso, com todas as quantidades de entrada nos resultados temporais;
- A execução do pior caso de cada algoritmo mostrou que, para este caso o tamanho da entrada pouco determina se um algoritmo vai ser pior ou melhor que os demais;
- Em todos os casos testados (pior, médio e melhor) a ordem de crescimento dos tempos de execução foi maior que 10 vezes nos resultados temporais;
- Em todos os casos testados (pior, médio e melhor) o crescimento no consumo de memória permaneceu estável com pouca variância;
- Nos resultados temporais da execução do melhor caso foi verificado que, o tamanho da entrada influencia bastante no resultado do algoritmo *Merge Sort e Intro Sort*;
- Nos resultados temporais do melhor caso o *Merge sort* se mostrou que lida melhor com grande quantidade de dados, enquanto o *Heap Sort* não lida muito bem com essas entradas;
- Nos caso médio dos resultados temporais o *Heap Sort*, apesar ter a mesma complexidade que os outros, teve um resultado muito pior que os demais;
- Em termos de uso de memória o *Merge Sort* foi o que teve o maior consumo de memória, enquanto o *Heap Sort* se manteve praticamente constante em todos os tamanhos de entrada.
- O consumo de memória do *Heap Sort* e do *Merge Sort* permaneceram constantes em todos os casos e tamanhos de entradas testadas. Por outro lado, o *Intro Sort* teve uma variação no consumo de memória nos casos em um mesmo tamanho de entrada.

Diante dos resultados, foi possível observar que não existe um algoritmos perfeito dentre os comparados neste estudo, no qual todos tem seus pontos fracos e seus pontos

fortes. O que exige que o problema que necessita do uso de um algoritmo de ordenação precise ser analisado e entendido afundo para que seja utilizado o melhor algoritmo de ordenação. Em outra palavras, não existe resposta fácil, pois se ganhamos em performance podemos perder em memória e vice e versa.

Referências

Algoritmos — Ciência da Computação — Khan Academy. Disponível em https://pt.khanacademy.org/computing/computer-science/algorithmsmerge-sort.

(2018). Heap Sort Algorithm. Disponivel em https://www.studytonight.com/data-structures/heap-sort https://www.programiz.com/dsa/heap-sort. Acesso: 08/07/2020.

(2018). Heap Sort Algorithm. Disponivel em https://www.studytonight.com/data-structures/heap-sort https://www.programiz.com/dsa/heap-sort. Acesso: 08/07/2020.

Aggarwal, S., Rai, A., Advani, R., Gupta, V., Kushjaing, and Rishiraj. HeapSort - GeeksforGeeks. Disponivel em https://www.geeksforgeeks.org/heap-sort/. Acesso: 08/07/2020.

chitranayal Mayank Khanna 2 (2016). Merge Sort - GeeksforGeeks. Disponivel em https://www.geeksforgeeks.org/merge-sort/?ref=rp https://www.geeksforgeeks.org/merge-sort/. Acesso: 04/07/2020.

Cormen, T. and Balkcom, D. (2017). Análise do merge sort (artigo) — Algoritmos — Khan Academy. Disponivel em https://pt.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort. Acesso: 05/07/2020.

InterviewCake. Heapsort Algorithm — Interview Cake. Disponível em https://www.interviewcake.com/concept/java/heapsort. Acesso: 08/07/2020.

Karthikeyan, L. IntroSort or Introspective sort - GeeksforGeeks. Disponivel em https://www.geeksforgeeks.org/introsort-or-introspective-sort/. Acesso: 08/07/2020.