Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Bacharelado em Sistemas de Informação

Um Estudo comparativo sobre os algoritmos de ordenação Tim Sort, Binary Insertion Sort e Merge Sort

Alexandre de Siqueira Melo Junior

Recife

Setembro de 2020

Dedico este trabalho ao Professor Cleviton e a minha saúde mental.

Resumo

O uso de dados em grande quantidade, pode causar um acúmulo desnecessário de dados, o que é um problema, nesse contexto os algoritmos de ordenação tem uma grande utilidade, porque eles organizam estruturas lineares. O objetivo deste estudo é avaliar por meio de experimentos, os dados gerados com a utilização de algoritmos de ordenação Binary Insertion Sort, Merge Sort e Tim Sort. Para isso, são ordenados 4 vetores diferentes, e os dados coletados foram comparados, e nessa comparação concluímos que o algoritmo Tim Sort foi o que mais se destacou.

Palavras-chave: TimSort, MergeSort ,BinaryInsertionSort, algoritmos, comparação

1. Introdução

1.1 Apresentação e Motivação

O estudo de algoritmos de ordenação é importante para conhecermos formas cada vez mais eficazes de escrever um código, gastando o mínimo de memória e processamento. Algoritmos de ordenação são usados para colocar elementos de uma sequência dada, em uma ordem específica, como os vetores pode ter uma quantidade enorme de elementos, dependendo do tamanho, também há uma quantidade enorme de possíveis posições, por isso os algoritmos de ordenação são muito usados em vetores. Por exemplo, a ordenação de uma lista com idades de alunos em ordem crescente. O objetivo deste estudo é avaliar, por meio de experimentos, em cima dos algoritmos de ordenação Binary Insertion Sort, Merge Sort e Tim Sort(algoritmo de ordenação padrão do Python), suas respectivas utilidades e eficiências. Para isso, são executados três testes com três vetores de tamanhos diferentes para cada algoritmo e os dados coletados serão comparados, o presente estudo só irá usar números inteiros nos vetores.

O procedimento é realizado com vetores de tamanho 40, 50, 1000 e 10000 onde são avaliados os tempos de execução para ordenar: Uma lista ordenada crescente, uma lista ordenada decrescente e uma lista ordenada de forma aleatória.

Com os resultados espera-se apontar vantagens e desvantagens de cada um dos algoritmos, e qual é mais vantajoso em casos específicos. Na segunda seção serão apresentados os algoritmos de ordenação que fazem parte do estudo, e seus códigos em linguagem Python. Na terceira seção será apresentado o procedimento, na quarta seção serão apresentado os dados coletados com uma pequena análise, e por fim na quinta seção serão apresentadas as conclusões baseadas na análise dos dados coletados.

2. Referencial Teórico

Outros estudos fizeram comparações de algoritmos de ordenação, como [2], [3] e [6], em [2] foi feito uma análise em cima dos algoritmos Bubble Sort, Merge Sort, Quick Sort e Shell Sort, a análise foi feita com auxílio de um aplicativo para análise comparativa do comportamento de algoritmos de ordenação, em [3], os algoritmos analisados foram Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort e Shell Sort e a análise foi em cima de dados coletados, com tabelas criadas com esses dados, e em [6] foram analisados os algoritmos Tim Sort e Merge Sort, com foco em mostrar a eficiência na melhoria do hardware em relação ao tempo de execução dos

algoritmos. O método dessa pesquisa será semelhante ao que foi feito em [3], porém incluindo o algoritmo Tim Sort e os algoritmos semelhantes a ele, que são o Binary Insertion Sort e o Merge Sort, pois o algoritmo Tim Sort é dividido em dois casos, um caso é semelhante ao Binary Insertion Sort, e o outro caso é semelhante ao Merge Sort.

2.1 Binary Insertion Sort

O algoritmo insertion Sort binário ($O(n^2.log(n))$ no pior caso) percorre a lista, procurando um elemento que é menor que seu elemento à esquerda, quando este elemento é definido, o algoritmo binary search procura um lugar apropriado para o mesmo usando o método dividir e conquistar, com este lugar apropriado n, todos os elementos vão somar um ao seu index, para "abrir espaço" para o elemento ficar no seu lugar adequado. Segue abaixo um exemplo da ordenação do vetor [1, 3, 2, 5] pelo algoritmo Binary Insertion Sort e o pseudocódigo do algoritmo.

Exemplo

Index: 1

[1, **3**, 2, 5]

1 < 3, podemos continuar.

Index: 2

[1, 3, **2**, 5]

2 < 3, logo temos que achar um lugar adequado para o 2, que é entre o 1 e 3, index 1.

o index de 3, que é 1, se torna 1 + 1, para dar espaço para o número 2 no index 1.

[1, **2**, **3**, 5]

Index: 3

[1, 2, 3, **5**]

3<5, então podemos finalizar, a lista está ordenada.

Pseudocódigo

```
Binary-Search(arr,val,start,end)
1       if start = end
2               if arr[start] > val
3                       return start
4               else
5                       return start + 1
6       if start > end
7               return start
8       mid = (start + end) / 2
9       if arr[mid] < val
10              Recursão Binary-Search(arr,val,mid+1,end)
11      if arr[mid] > val
12              Recursão Binary-Search(arr,val,start,mid-1)
13      else
14              return mid
```

```
Insertion-Sort(arr)
1       for i=1 to n, onde n é a quantidade de elementos em arr
2               val = arr[i]
3               j = Binary-Search(arr, val, 0, i - 1)
4               arr = arr[0...j-1] + val + arr[j...i-1] + arr[i+1...n]
5       return arr
```

2.2 Merge Sort

O funcionamento deste algoritmo(complexidade de O(nlog(n)))[6], é baseado numa técnica chamada “dividir para conquistar”, a primeira ideia do algoritmo é dividir o vetor, em vetores menores, e depois juntar ordenando. O passo de dividir é feito usando recursão, e o passo de juntar e ordenar é feito pela função Merge. Segue abaixo

um exemplo do funcionamento do algoritmo e as figuras 1 e 2 são os pseudocódigos das funções Merge e MergeSort.

[5,7,2,8|3,0,1,6]

Dividir [5,7|2,8] [3,0|1,6]

Dividir [5|7] [2|8] [3|0] [1|6]

Dividir [5] [7] [2] [8] [3] [0] [1] [6]

Conquistar [5,7] [2,8] [0,3] [1,6]

Conquistar [2,5,7,8] [0,1,3,6]

Conquistar [0,1,2,3,5,6,7,8]

```
MERGE-SORT′(A, p, r)
1    if p < r
2        q = ⌊(p + r)/2⌋
3        spawn MERGE-SORT′(A, p, q)
4        MERGE-SORT′(A, q + 1, r)
5        sync
6        MERGE(A, p, q, r)
```

Figura 1. Pseudocódigo da função MergeSort

Fonte: Foto em [4]

```
MERGE(A, p, q, r)
1   n_1 = q − p + 1
2   n_2 = r − q
3   sejam L[1..n_1 + 1] e R[1.. n_2 + 1] novos arranjos
4   for i = 1 to n_1
5       L[i] = A[p + i − 1]
6   for j = 1 to n_2
7       R[j] = A[q + j]
8   L[n_1 + 1] = ∞
9   R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          then A[k] = L[i]
15               i = i + 1
16          else A[k] = R[j]
17               j = j + 1
```

Figura 2. Pseudocódigo da função Merge

Fonte: Foto em [4]

2.3 Timsort

TimSort (O(n.log(n)) no pior caso)[6] é uma combinação do Insertion Sort Binário e o Merge Sort, a funcionalidade do algoritmo é baseada no parâmetro opcional (OP), o valor do parâmetro determina qual algoritmo usar, entre os dois. Com o OP sendo x, o tamanho t da matriz irá decidir qual algoritmo será aplicado, se t < x, será aplicado o Binary Insertion Sort, caso contrário, será aplicado o Merge Sort.

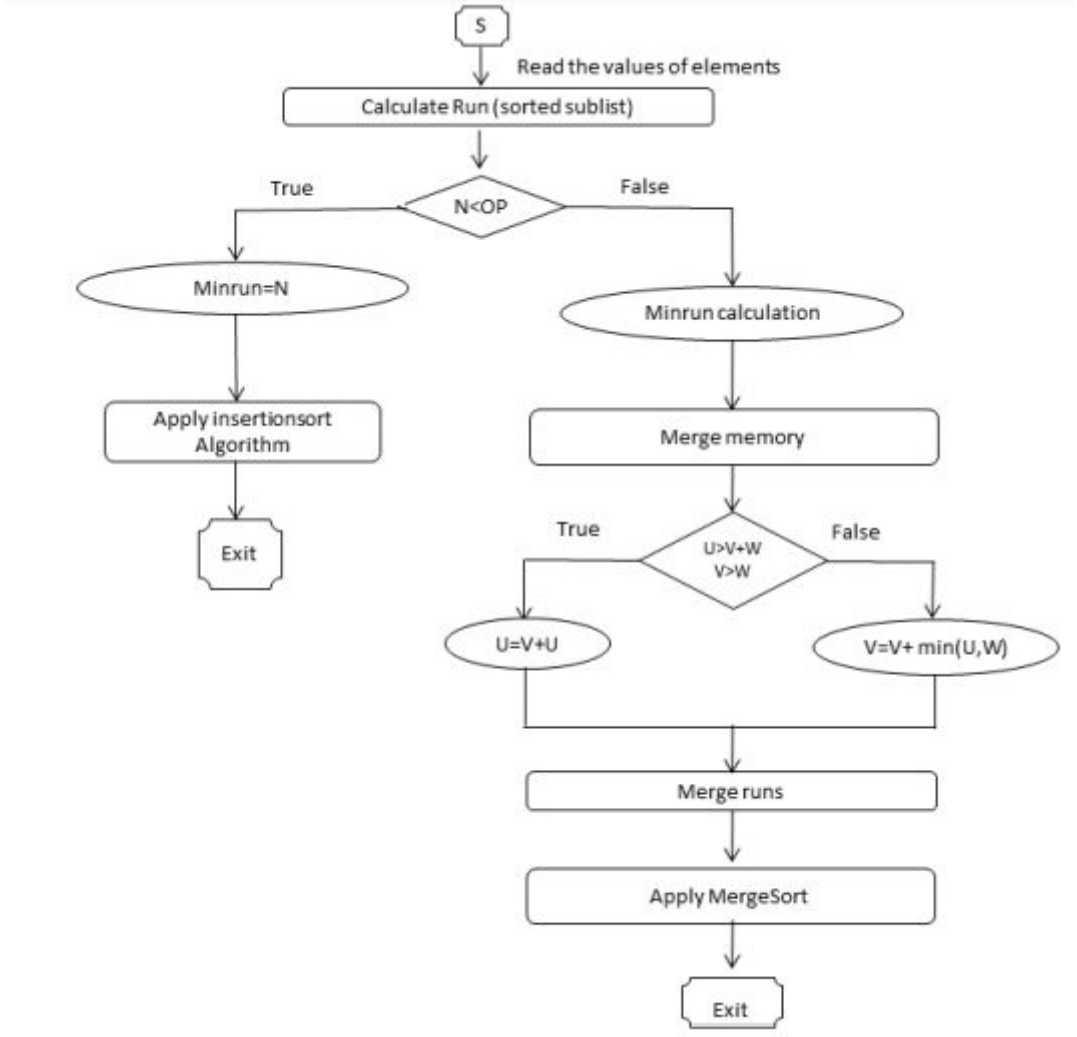

Figura 3. Passos para aplicação do algoritmo TimSort

Fonte: Foto original em [6]

"**Passo Calculate Run** : No vetor, há subvetores que já estão ordenados, então esses subvetores são temporariamente extraídos e guardado em listas temporárias chamadas run.

Passo para seleção do algoritmo: Insertion Sort será selecionado se N for menor que OP; caso contrário, MergeSort é considerado (onde N é o tamanho da matriz).

Passo para calcular o Minrun: Minrun é o comprimento mínimo de execuções que depende do comprimento da matriz de entrada. minrun = N se o algoritmo Insertion Sort é selecionado, se o Merge Sort for o algoritmo selecionado, $\frac{N}{Minrun} < 2^N$.

Passo Merge Memory: Essas execuções precisam ser combinadas para obter a matriz ordenada final. TimSort mescla apenas as execuções consecutivas. A Figura 6

representa três comprimentos de execuções (U, V, W) onde as duas condições devem ser satisfeitas: (a) U> V + W; (b) V> W. Se a primeira regra não for satisfeita, então V é fundido com o mínimo de U ou W. Este passo é repetido até que as condições (a) e (b) sejam satisfeitas

Passo Merge runs: Duas execuções adjacentes são mescladas aplicando as seguintes etapas: (i) Os elementos da execução de tamanho mínimo são copiados para uma execução temporária

(ii) Os elementos da maior execução são comparados à execução temporária em que o menor valor é movido em uma nova matriz. O ponteiro é atualizado para a matriz da qual o elemento foi obtido

(iii) Repita as etapas anteriores até o final de uma das matrizes.

Passo Apply MergeSort: MergeSort é aplicado para classificar o resto dos elementos na matriz."[6]

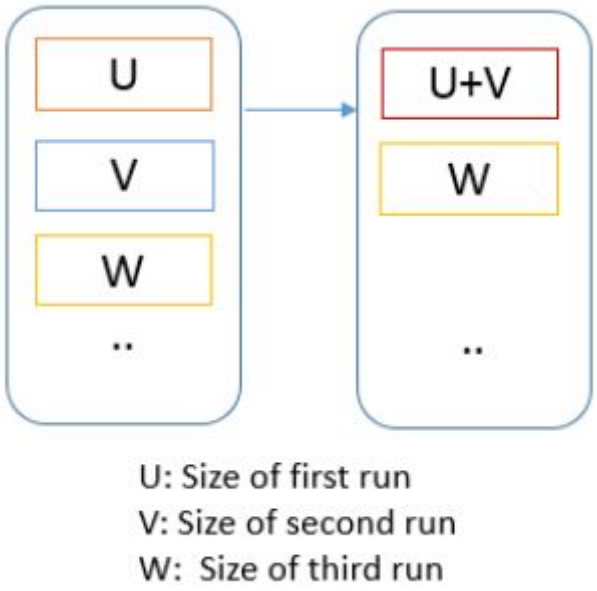

Figura 4. Representação do passo Merge Memory

Fonte: Foto em [6]

Segue abaixo um exemplo do funcionamento de um algoritmo com um vetor menor e maior que o OP, nesse exemplo OP = 10 para facilitar o entendimento.

Exemplo len(x) < OP:

x = [4,5,6,3]

len(x) < 10 ? SIM

então, apenas aplicar o algoritmo Insertion Sort.

Aplicação do Insertion Sort:

[4,5,6,3]

O Insertion Sort começa a analisar pelo segundo elemento da lista, ou seja, x[1], e ignora todos o elementos a frente dele.

5 > 4 ? SIM, logo não há o que fazer, vamos para o próximo elemento, x[2],

6 > 5 ? SIM, logo não há o que fazer, vamos para o próximo elemento, x[3],

3 > 6 ? NÃO, logo jogamos o 6 uma posição para frente, e continuamos a comparação,

[4,5,6,6]

3 > 5 ? NÃO, logo jogamos o 5 uma posição para frente, e continuamos a comparação,

[4,5,5,6]

3 > 4 ? NÃO, logo jogamos o 5 uma posição para frente, e colocamos o 3 no seu lugar adequado, pois não há mais números para comparar.

[3,4,5,6]

A lista está ordenada.

Exemplo len(x) > OP:

[12,11,10,9,8,7,6,5,4,3,2,1]

len(x) > OP ? SIM,

então iremos dividir a lista em tamanho OP,

[12,11,10,9,8,7,6,5,4,3], [2,1]

Aplicar Insertion Sort em cada uma (O funcionamento do Insertion Sort está no exemplo anterior)

[3,4,5,6,7,8,9,10,11,12], [1,2]

e para finalizar, aplicamos a função Merge, (O funcionamento do Merge está em 2.2)

[1,2,3,4,5,6,7,8,9,10,11,12]

3. Método ou Procedimento

1. O primeiro passo para produzir este trabalho foi escolher os algoritmos de ordenação, escolhi os três já citados com foco no Tim Sort, pois ele usa métodos dos outros dois algoritmos no funcionamento.
2. O segundo passo foi implementá-los e pesquisar outros estudos ou bibliotecas que me auxiliem a conseguir dados para compará-los, a partir disso, decidir como será minha base de dados para a análise e escolher em qual linguagem e plataforma desenvolver.
3. O terceiro passo, foi usar as plataformas escolhidas, neste estudo a base de dados será organizada em duas tabelas, uma com vetores de tamanho 40 e 50, e a outra com vetores de tamanho 1000 e 10000, para cada tamanho de vetor, teremos um vetor ordenado da forma crescente, um ordenado na forma decrescente, e um desordenado, e o tempo de execução para ordenar cada vetor na forma crescente e a quantidade de trocas. A linguagem Python foi definida como meio de codificação, os ambientes de desenvolvimentos foram o Visual Studio Code para desenvolver os algoritmos, e o Google Colab para a análise dos dados coletados e medição do tempo de execução, as bibliotecas usadas foram timeit para medir o tempo de execução, e pandas para organização dos dados. Os códigos estão no apêndice

Quanto ao hardware, a máquina de teste tem as seguintes configurações:

Processador: Intel(R) Core(TM) i3-6006U CPU @2.00GHz 2.00 GHz

Memória instalada (RAM): 4,00 GB

Tipo de sistema: Sistema Operacional de 64 bits, processador com base em x64, Windows 10 Pro.

A análise será feita em cima do tempo de execução de cada algoritmo, será colocado vetores menores e maiores que o valor da variável OP = 64 do TimSort, com a

ajuda da biblioteca timeit, mediremos o tempo de execução para ordenar os vetores com 8000 repetições, e tiramos a média.

4. Resultados

4.1 Vetores de tamanho 1000 e 10000

A tabela 1 mostra o tempo de execução dos algoritmos ordenando vetores ordenados de forma crescente, decrescente, e aleatória.

Tabela 1. Valores médios do tempo de execução para um vetores de tamanho 1000

Fonte: Criação própria

	Tamanho 1000					
	Tempo	Trocas	Tempo	Trocas	Tempo	Trocas
Lista	Crescente		Decrescente		Aleatório	
Merge Sort	0.00294888	10975	0.00291999	10975	0.00478026	10975
Binary Insertion Sort	0.0108234	0	0.00931151	999	0.0102205	991
Tim Sort	0.00135898	4360	0.00343133	19709	0.00358097	11974

Tabela 2. Valores médios do tempo de execução para um vetores de tamanho 10000

Fonte: Criação própria

	Tamanho 10000					
	Tempo	Trocas	Tempo	Trocas	Tempo	Trocas
Lista	Crescente		Decrescente		Aleatório	
Merge Sort	0.0381209	143615	0.0394855	143615	0.0676233	143615
Binary Insertion Sort	0.850504	0	0.721897	9999	0.891714	9986
Tim Sort	0.0242489	9361	0.0436558	198045	0.0567424	142296

4.2 Vetores de tamanho 40 e 50

A tabela 2 mostra o tempo de execução dos algoritmos ordenando vetores ordenados de forma crescente, decrescente, e aleatória. A notação e-05 é equivalente a multiplicar o número por 10 elevado a -5, por exemplo, o número 2.3e-5 é equivalente a 0.000023

Tabela 3. Valores médios do tempo de execução para um vetores de tamanho 40

Fonte: Criação própria

Tamanho 40						
	Tempo	Trocas	Tempo	Trocas	Tempo	Trocas
Lista	Crescente		Decrescente		Aleatório	
Merge Sort	7.61515e-05	255	8.13064e-05	255	0.000106109	255
Binary Insertion Sort	9.16627e-05	0	7.9877e-05	39	8.29903e-05	36
Tim Sort	2.36606e-05	32	7.58395e-05	532	5.77925e-05	326

Tabela 4. Valores médios do tempo de execução para um vetores de tamanho 50

Fonte: Criação própria

Tamanho 50						
	Tempo	Trocas	Tempo	Trocas	Tempo	Trocas
Lista	Crescente		Decrescente		Aleatório	
Merge Sort	0.000102666	335	0.000108015	335	0.00014146	335
Binary Insertion Sort	0.000117862	0	0.000109511	49	0.000115333	46
Tim Sort	2.52158e-05	32	9.88136e-05	667	7.33419e-05	417

Podemos observar que, nos testes com os vetores já ordenados de forma crescente, o Tim Sort foi o mais eficiente na questão do tempo de execução com todos os vetores analisados, porém o tempo de execução dos 3 algoritmos foram bastante semelhantes nos vetores menores que 64, com base na complexidade de cada algoritmo, era esperado que o Binary Insertion Sort tivesse o pior tempo de execução. Nos vetores de tamanho 40 e 50, o Binary Insertion Sort e o Tim Sort funcionam de forma semelhante, o melhor desempenho do Tim Sort pode ter vindo da economia de tempo que o passo Calculate Run trás. Nos vetores de tamanho 1000 e 10000 o rendimento do Binary Insertion Sort foi muito pior em relação aos outros, baseado neste resultado e na complexidade dos algoritmos, podemos conjecturar que quanto maior o vetor, pior o

rendimento do Binary Insertion Sort comparado aos outros dois algoritmos,, o que pode ter sido levado em consideração para a estratégia do Tim Sort em deixar de lado o Insertion Sort para vetores de tamanho maiores que 64.

Nas listas com ordenação decrescente, o tempo de execução nos vetores de tamanho 40 e 50 de cada algoritmo foi muito semelhante, ao ponto do Binary Insertion Sort ter o menor tempo de execução no vetor de tamanho 40, nos vetores de tamanho 1000 e 10000, o desempenho do Binary Insertion Sort se distanciou do desempenho dos outros algoritmos, como dito anteriormente. O melhor desempenho em vetores maiores foi o merge sort, porém foi muito semelhante ao Tim Sort.

Nas listas ordenadas de forma aleatória, nos vetores menores que 64, o tempo de execução do Merge Sort e Binary Insertion sort foram muito semelhantes e o melhor desempenho foi o Tim Sort com uma diferença notável, já nos vetores maiores que 64, analogamente aos casos comentados anteriormente, o desempenho do Merge Sort e Tim Sort foram semelhantes e o Binary Insertion Sort teve uma notável diferença.

5. Conclusão

Como dito anteriormente, o objetivo do trabalho é comparar a eficiência de três algoritmos de ordenação (Tim Sort, Binary Insertion Sort, Merge Sort) a partir da medição do tempo de execução de vetores de tamanho 40, 50, 1000 e 10000, pois como o Tim sort funciona de forma diferente dependendo do tamanho do vetor, temos dois casos para cada caminho do Tim Sort.

Os resultados alcançados, analisando os dados coletados, o Tim Sort teve os melhores resultados (dentro dos limites da máquina de teste usada), é possível ver também que o Insertion Sort têm melhores resultados nos vetores de tamanho menores, mas os outros dois algoritmos não ficaram pra trás, já nos vetores maiores, a eficiência do Insertion Sort foi muito pior que os outros, e a eficiência do Tim Sort e Merge Sort foram muito semelhantes, com Tim Sort um pouco melhor, com base nisso, podemos dizer que o Tim Sort usa uma estratégia muito eficiente, pelo fato de usar o Insertion em vetores menores, e a junção do Merge e Insertion nos maiores.

Um futuro trabalho poderia ser uma coleta de dados análoga, com uma máquina de teste mais potente, ou um estudo em cima do cálculo da complexidade dos algoritmos Tim Sort e Merge Sort, para analisar a causa da pequena diferença de eficiência, que pode ser por exemplo a base do log.

Referências Bibliográficas

[1] Pereira, S. L. (2010). "Algoritmos e Lógica de Programação em C: uma abordagem didática." 1ª edição. São Paulo. Érica.

[2] Folador, J. P., Neto, L. N. P., & Jorge, D. C. (2014). Aplicativo para análise comparativa do comportamento de algoritmos de ordenação. *Revista Brasileira de Computação Aplicada*, *6*(2), 76-86.

[3]Souza, J. E., Ricarte, J. V. G., & de Almeida Lima, N. C. (2017). Algoritmos de Ordenação: Um estudo comparativo. Anais do Encontro de Computação do Oeste Potiguar ECOP/UFERSA (ISSN 2526-7574), (1).

[4]Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2002). Algoritmos: teoria e prática. *Editora Campus*, *2*, 296.

[5]Auger, N., Jugé, V., Nicaud, C., & Pivoteau, C. (2018). On the worst-case complexity of TimSort. *arXiv preprint arXiv:1805.08612*.

[6]Jmaa, Y. B., Ali, K. M., Duvivier, D., Jemaa, M. B., & Atitallah, R. B. (2017, July). An efficient hardware implementation of timsort and mergesort algorithms using high level synthesis. In *2017 International Conference on High Performance Computing & Simulation (HPCS)* (pp. 580-587). IEEE.

Apêndices

Neste tópico terão os códigos implementados para esta pesquisa na linguagem python.

Tim Sort

```
def insertion_sort(array, left=0, right=None):
    if right is None:
        right = len(array) - 1

    for i in range(left + 1, right + 1):

        key_item = array[i]


        j = i - 1
```

```
        while j >= left and array[j] > key_item:

            array[j + 1] = array[j]
            j -= 1


        array[j + 1] = key_item

    return array


def merge(left, right):

    if len(left) == 0:
        return right


    if len(right) == 0:
        return left

    result = []
    index_left = index_right = 0


    while len(result) < len(left) + len(right):

        if left[index_left] <= right[index_right]:
            result.append(left[index_left])
            index_left += 1
        else:
            result.append(right[index_right])
            index_right += 1


        if index_right == len(right):
            result += left[index_left:]
            break

        if index_left == len(left):
            result += right[index_right:]
            break

    return result



def timsort(array):
    min_run = 32
    n = len(array)
```

```
    for i in range(0, n, min_run):
        insertion_sort(array, i, min((i + min_run - 1), n - 1))


    size = min_run
    while size < n:

        for start in range(0, n, size * 2):

            midpoint = start + size - 1
            end = min((start + size * 2 - 1), (n-1))


            merged_array = merge(
                left=array[start:midpoint + 1],
                right=array[midpoint + 1:end + 1])


            array[start:start + len(merged_array)] = merged_array


        size *= 2

    return array
```

Segue agora o código modificado para a contagem das trocas

```
from random import shuffle
def insertion_sort(array, left=0, right=None):
    contador = 0
    if right is None:
        right = len(array) - 1

    for i in range(left + 1, right + 1):

        key_item = array[i]


        j = i - 1


        while j >= left and array[j] > key_item:

            array[j + 1] = array[j]
            j -= 1
```

```
            contador += 1


        array[j + 1] = key_item

    return [array,contador]


def merge(left, right):
    contador = 0
    if len(left) == 0:
        return right


    if len(right) == 0:
        return left

    result = []
    index_left = index_right = 0


    while len(result) < len(left) + len(right):

        if left[index_left] <= right[index_right]:
            result.append(left[index_left])
            contador += 1
            index_left += 1
        else:
            result.append(right[index_right])
            index_right += 1
            contador += 1


        if index_right == len(right):
            result += left[index_left:]
            break

        if index_left == len(left):
            result += right[index_right:]
            break

    return [result,contador]



def timsort(array):
    contador = 0
    min_run = 32
    n = len(array)
```

```
    for i in range(0, n, min_run):
        contador += insertion_sort(array, i, min((i + min_run - 1), n - 1))[1]


    size = min_run
    while size < n:

        for start in range(0, n, size * 2):

            midpoint = start + size - 1
            end = min((start + size * 2 - 1), (n-1))


            merged_array = merge(
                left=array[start:midpoint + 1],
                right=array[midpoint + 1:end + 1])
            contador += merged_array[1]


            #array[start:start + len(merged_array)] = merged_array[0]


        size *= 2

    return [array,contador]
```

Merge Sort

Neste código, basta o usar a função merge já implementada no Tim Sort, junto com a função merge sort (a modificação para a contagem de trocas na função merge foi idêntica na modificação do TimSort)

```
def merge_sort(array):

    if len(array) < 2:
        return array

    midpoint = len(array) // 2

    return merge(
        left=merge_sort(array[:midpoint]),
        right=merge_sort(array[midpoint:]))
```

código para contagem de trocas:

```
def mergesort(lista, inicio=0, fim=None):
    contador = 0
    if fim is None:
        fim = len(lista)
    if(fim - inicio > 1):
        meio = (fim + inicio)//2
        contador += 1
        x = mergesort(lista, inicio, meio)
        contador += x
        y = mergesort(lista, meio, fim)
        contador += y
        z = merge(lista, inicio, meio, fim)
        contador += z

    return contador
```

Binary Insertion Sort

```
def binary_search(arr, val, start, end):

    if start == end:
        if arr[start] > val:
            return start
        else:
            return start+1


    if start > end:
        return start

    mid = int((start+end)/2)
    if arr[mid] < val:
        return binary_search(arr, val, mid+1, end)
    elif arr[mid] > val:
        return binary_search(arr, val, start, mid-1)
    else:
        return mid

def insertion_sort(arr):
    for i in range(1, len(arr)):
        val = arr[i]
```

```
        j = binary_search(arr, val, 0, i-1)
        arr = arr[:j] + [val] + arr[j:i] + arr[i+1:]
    return arr
```

O código para calcular a quantidade de trocas

```
def binary_search(arr, val, start, end):

    if start == end:
        if arr[start] > val:
            return start
        else:
            return start+1


    if start > end:
        return start

    mid = int((start+end)/2)
    if arr[mid] < val:
        return binary_search(arr, val, mid+1, end)
    elif arr[mid] > val:
        return binary_search(arr, val, start, mid-1)
    else:
        return mid

def insertion_sort(arr):
    contador = 0
    for i in range(1, len(arr)):
        val = arr[i]
        j = binary_search(arr, val, 0, i-1)
        arr = arr[:j] + [val] + arr[j:i] + arr[i+1:]
        if i != j:
            contador += 1
    return contador
```

Construção da tabela

O código do Google colab está neste link: