UNIVERSIDADE FEDERAL DE OURO PRETO - MG

AVALIAÇÃO EMPÍRICA DOS ALGORITMOS: MERGE SORT, INSERTION SORT E RADIX SORT

ROGERD JÚNIOR RIBEIRO BITARÃES, 17.2.4243

OURO PRETO
2020

RESUMO

A ordenação é uma área importante de estudo na ciência da computação, devido os diversos tipos de aplicabilidade no mundo real. Contudo esse trabalho tem objetivo de realizar a implementação e comparação empírica de 3 algoritmos de ordenação: merge sort, insertion sort e radix sort. Foram realizados teste com relação ao tempo de execução de cada algoritmo, assim como também teste estatístico pareado t.
O trabalho obteve resultados esperados conforme a ordem de complexidade de cada algoritmo.

SUMÁRIO

1. Introdução

A ordenação é o processo de colocar um conjunto elementos de em alguma determinada ordem. Por exemplo, uma lista de palavras pode ser classificada em ordem alfabética ou por comprimento. Uma lista de cidades pode ser classificada por população ou área. A ordenação é uma área importante de estudo na ciência da computação, devido os diversos tipos de aplicabilidade no mundo real.

1.1. Justificativa

A ordenação está presente no cotidiano das pessoas mais do que elas imaginam, muitas vezes até sem que elas percebam. Por exemplo, quando pegam o celular e abrem o aplicativo de mensagens e visualizam as recebidas de forma ordenadas pelo horário.

Como a ordenação é área importante na computação e possui muita aplicabilidade, existem diversos algoritmos de ordenação diferentes, sendo alguns mais simples e outros mais complexos de implementar.

Ordenar um número grande de itens pode exigir uma quantidade enorme de recurso computacional, tornando a solução do problema inviável. Para pequenos conjuntos de dados, não faz muita diferença qual algoritmo utilizar. Porém quando se trata de um grande volume de dados é importante avaliar qual algoritmo irá ter um melhor desempenho.

1.2. Objetivos

Este trabalho tem como objetivo avaliar tempo de execução de 3 algoritmos de ordenação (merge sort, insertion sort e radix sort) a partir de conjuntos de instâncias de diferentes tamanhos.

1.3. Organização do trabalho

No capítulo 1, é descrito a introdução do trabalho, abordando a justificativa e objetivos. No capítulo 2, é descrito os métodos de ordenação utilizados no trabalho, assim como a sua complexidade. No capítulo 3, é mencionado a metodologia utilizada para realização do trabalho. No capítulo 4, é mostrado os resultados obtidos a partir da execução dos algoritmos sobre os diferentes conjuntos de instâncias. Por fim, no capítulo 5 é descrito a conclusão do trabalho.

2. Métodos

Neste capítulo, é descrito os algoritmos de ordenação utilizados no trabalho, assim a complexidade e estratégia de ordenação de cada algoritmo.

1.1. Insertion sort

O insertion sort é um algoritmo de ordenação simples que funciona semelhante como ordenamos cartas em jogo de baralho. Para ordenar um vetor de tamanho n de forma crescente, é necessário percorrer a lista de elementos comparando se o elemento atual for menor do que o anterior, ele deve ser comparado com os elementos anteriores, até que a condição seja falsa ou até chegar à posição inicial da lista. Quando terminar esse loop, o elemento é colocado na sua posição. Após isso é feito o mesmo processo para os outros elementos até o último elemento da lista.

1.1.1 Código

```
1  function insertionSort(arr) {
2    for (let i = 1; i < arr.length; i++) {
3      key = arr[i];
4      j = i - 1;
5      while (j ⩾ 0 && arr[j] > key) {
6        arr[j + 1] = arr[j]
7        j--;
8      }
9      arr[j + 1] = key;
10   }
11   return arr;
12 }
```

1.1.2 Complexidade

Linha	Instruções	Repetições
2	2	n
3	1	n-1
4	1	n-1

5	3	ti + 1 p/i
6	1	ti p/i
7	1	ti p/i
9	1	n-1

$$2n + 3(n-1) + \sum_{i=1}^{n-1} 3(ti+1) + \sum_{i=1}^{n-1} 2(ti) + 1$$

$$5n + 5\sum_{i=1}^{n-1} ti + 3\sum_{i=1}^{n-1} 1 - 2$$

$$5n + 5\sum_{i=1}^{n-1} ti + 3(n-1) - 2$$

$$T(n) = 8n + 5\sum_{i=1}^{n-1} ti - 5$$

A função é complexidade do algoritmo insertion sort implementado acima é $T(n) = 8n + 5\sum_{i=1}^{n-1} ti - 5$.

Melhor caso:
Para demonstrar o melhor caso, basta substituir o *ti* por 0, pois no melhor caso o vetor já está ordenado, com isso as linhas 6 e 7 não serão executadas, com isso *ti* = 0.
Logo, no melhor caso:
$T(n) = 8n - 5$
Ordem de complexidade: $\Theta(n)$

Pior caso:
Para demonstrar o pior caso, basta substituir o *ti* por *i-1*, pois no pior caso o estará ordenado no sentido contrário, com isso as linhas 6 e 7 serão executadas i - 1 vezes, com isso *ti* = 0.
Logo, no melhor caso:

$$T(n) = 8n - 5 + 5\sum_{i=1}^{n-1} i - 1$$

$$T(n) = 8n - 5 + 5\sum_{i=0}^{n-2} i$$

$$T(n) = 8n + 5(n-1)(n-2)/2 - 5$$

$$T(n) = 8n + (5n-5)(n-2)/2 - 5$$

$$T(n) = 8n + (5n^2 - 15n - 10)/2 - 5$$

$$T(n) = (5n^2 + n - 20) / 2$$

Ordem de complexidade: $\Theta(n^2)$

Caso médio:
Para demonstrar o caso médio, é necessário calcular o *ti* médio. Assim, temos

$$ti\ médio\ = \frac{1}{i}\sum_{i=0}^{i-1} j\ \ = \frac{1}{i}i(i-1)/2\ =\ (i-1)/2$$

Substituindo na função de complexidade, temos:

$$T(n)\ =\ 8n\ -5\ \ +\ 5\sum_{i=1}^{n-1}(i-1)/2$$

$$T(n)\ =\ 8n\ -5\ \ +\ \frac{5}{2}\sum_{i=1}^{n-1}(i-1)$$

$$T(n)\ =\ 8n\ -5\ \ +\ \frac{5}{2}\sum_{i=0}^{n-2} i$$

$$T(n)\ =\ 8n\ -5\ \ +\ 5(n-1)(n-2)/4$$

$$T(n)\ =\ 8n\ -5\ \ +\ (5n^2-15n\ -10)/4$$

$$T(n)\ =\ (5n^2\ +17n\ -30)\ /\ 4$$

Ordem de complexidade: $\Theta(n^2)$

1.2. Merge sort

O algoritmo de ordenação merge sort é baseado na técnica de divisão e conquista, com isso ele divide a lista de elementos em sub-problemas e quando é resolvido a solução para cada subproblema, elas são combinadas de forma que a lista fique ordenada. Contudo, o merge sort é divido em 3 etapas:
1 - Se a lista tiver apenas um elemento ela já está ordenada.
2- Divida a lista recursivamente até que não possa ser mais dívida.
3- Combine as listas menores em uma nova lista na de forma ordenada.

1.2.1. Código

```
1  function mergeSort(arr) {
2    if (arr.length < 2) return arr;
3    const middle = Math.floor(arr.length / 2);
4    const left = arr.slice(0, middle);
5    const right = arr.slice(middle);
6    return merge(mergeSort(left), mergeSort(right));
7  }
```

```
function merge(left, right) {
  let arr = [];
  while (left.length && right.length) {
    if (left[0] < right[0])
      arr.push(left.shift());
    else
      arr.push(right.shift());
  }
  return arr.concat(left.slice().concat(right.slice()));
}
```

1.1.2 Complexidade

A função mergeSorte faz duas chamadas recursivas passando como parâmetro um array de tamanho n/2. E a função merge combina as soluções dos subproblemas. Desse modo, a equação de recorrência do merge sort é:

$$T(n) = \begin{cases} 2T(n/2) + \theta(n), & para\ n > 1 \\ O(1), & para\ n \leq 1 \end{cases}$$

O termo 2T(n/2) é o tempo para ordenar dois vetores de tamanho n/2, já o termo Θ(n) é o tempo para fundir/intercalar esses vetores, isto é, é o tempo da função merge.
Podemos resolver a relação de recorrência do merge sort pelo teorema mestre. Contudo, temos:
$log_2 2 = 1$, portando a ordem de complexidade do merge sort é $\theta(n\ log\ n)$

Melhor caso: $\theta(n\ log\ n)$
Pior caso: $\theta(n\ log\ n)$
Caso médio: $\theta(n\ log\ n)$

1.3. Radix sort

Radix sort é um algoritmo de ordenação que ordena inteiros processando dígitos individuais. Como os inteiros podem representar strings compostas de caracteres e pontos flutuantes especialmente formatados, radix sort não é limitado somente a inteiros.

1.3.1. Implementação

```
1   function radixSort(arr) {
2     const max = getMax(arr);
3     for (let i = 0; i < max; i++) {
4       let buckets = Array.from({ length: 10 }, () => [ ])
5       for (let j = 0; j < arr.length; j++) {
6         buckets[getPosition(arr[ j ], i)].push(arr[ j ]);
7       }
8       arr = [ ].concat( ... buckets);
9     }
10    return arr
11  }
```

```
13  function getPosition(num, place){
14    return  Math.floor(Math.abs(num)/Math.pow(10,place))% 10
15  }
16
17  function getMax(arr) {
18    let max = 0;
19    for (let num of arr) {
20      if (max < num.toString().length) {
21        max = num.toString().length
22      }
23    }
24    return max
25  }
```

1.3.2. Complexidade

Na linha 2, a função radixSort chama a função getMax, que percorre todo o array de elementos para pegar o valor de maior comprimento. Como a função percorre todo o array, ela tem a ordem de complexidade $\theta(n)$. Entre linha 3 a 9 da função radixSort ela possui 2 laços de repetição, o primeiro correspondente ao comprimento do maior valor retornado pela função getMax. Após isso, para cada repetição desse laço, na linha 5, possui outro laço de repetição que executado sobre o comprimento do array que deve ser ordenado. Tendo assim esse bloco de código, uma ordem de complexidade $\theta(n * k)$, onde k é o comprimento do maior valor retornado pela função getMax.

Desse modo, a ordem de complexidade do radixSort é:

$\theta(n) + \theta(n * k) = \theta(n * k)$

Melhor caso: $\theta(n * k)$
Pior caso: $\theta(n * k)$
Caso médio: $\theta(n * k)$

3. Metodologia

O trabalho foi desenvolvido utilizando o Node.js que é uma plataforma que permite executar código javascript no lado do servidor. Além disso, foi utilizando HTML e a biblioteca chart.js para realização de uma interface interativa para comparar o tempo de execução de cada algoritmo considerando o melhor tempo, pior tempo e tempo médio da execução dos algoritmos.

A execução dos algoritmos foram realizadas em uma máquina com as seguintes configurações:
Marca: Dell
Modelo: Inspiron 14 - 5457
Processador: Intel Core i7-6500U CPU 2.50GHz
RAM: 16GB
HD: 1 TB

A primeira etapa do trabalho foi a realização do estudo sobre os algoritmos de ordenação merge sort, insertion sort e radix sort. Após isso, foi realizado a implementação dos algoritmos, assim como o estudo da complexidade de cada algoritmo, conforme mostra o capítulo 2.

Para realização do teste de análise de desempenho dos algoritmos foram criadas 5 coleções de instâncias sendo que cada coleção possui 20 instâncias de um mesmo tamanho. Todas essas instâncias foram geradas de forma aleatória, de acordo com seguinte algoritmo:

```
1  function generateInstance(length){
2    let array = [];
3    for(let i=0; i<length; i++){
4      array.push(Math.floor(Math.random()*i)+1);
5    }
6    return array;
7  }
```

Desse modo, a função recebe como entrada o comprimento do vetor que deve ser retornado, e retorna o vetor de como que o valor década posição i seja menor ou igual a i+1. No total, foram geradas e ordenadas usando cada algoritmo:
20 instâncias de 100 elementos
20 instâncias de 1000 elementos
20 instâncias de 10000 elementos
20 instâncias de 100000 elementos

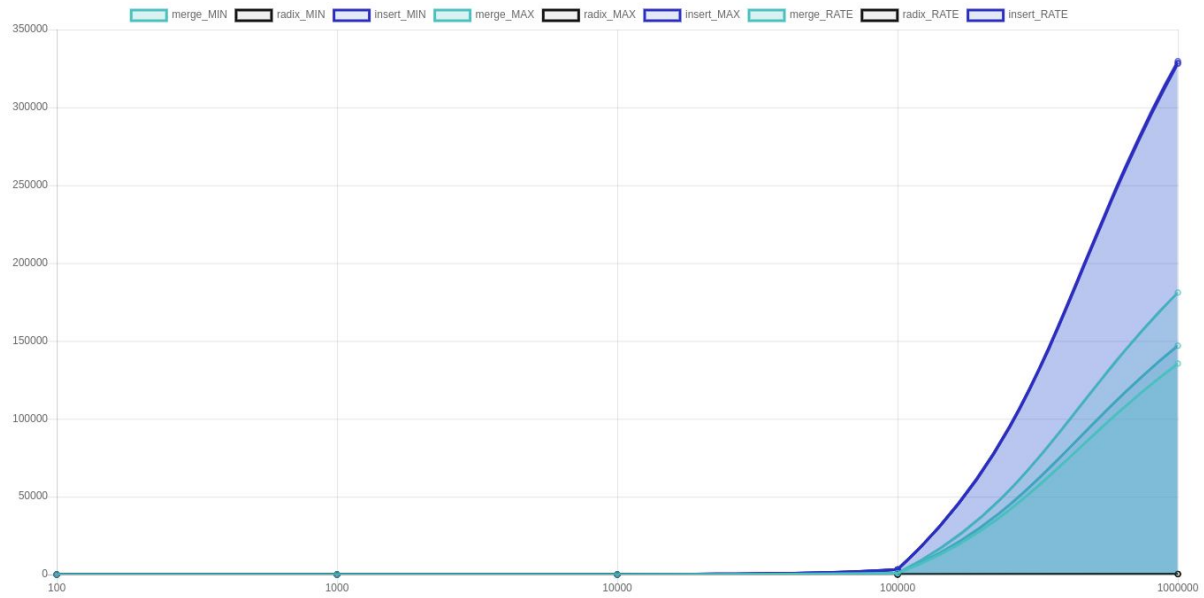
20 instâncias de 1000000 elementos

A métrica utilizada para comparação dos algoritmos foi o tempo de execução (milissegundos) de cada algoritmo gasto para ordenar uma instância. Como cada coleção possui 20 instâncias, foram realizadas análises sobre o pior tempo, tempo médio e melhor tempo de execução dos algoritmos. Sendo que o pior tempo é o maior tempo de execução entre as 20 instâncias ordenadas de um algoritmo, o melhor tempo é o menor tempo de execução e o tempo médio é a soma do tempo de execução das 20 instâncias dividido por 20. Além disso também foi realizado o teste t pareado com 95% de confiança para informar qual algoritmo obteve o melhor desempenho ou se houve empate estatístico.

4. Resultados

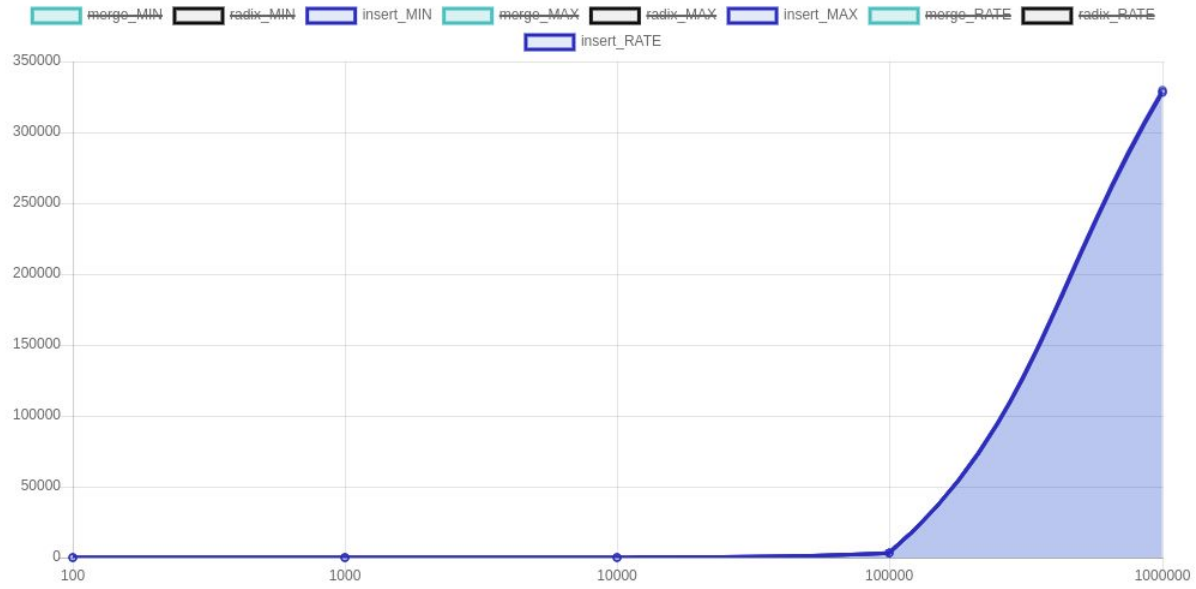
Neste capítulo é abordado os resultados obtidos a partir da realização dos testes de comparação entre os algoritmos implementados.

O imagem a seguir, mostra a interface da aplicação realizada para visualizar a comparação dos algoritmos implementados. Na parte superior é possível selecionar o resultado dos algoritmos que devem ser mostrado no gráfico, que são os algoritmos merge sort, insertion sort e radix sort, todos no melhor tempo, pior tempo e tempo médio. O eixo y do gráfico indica o tempo gasto para a execução do algoritmo em milissegundos, o eixo x indica a tamanho da instância.



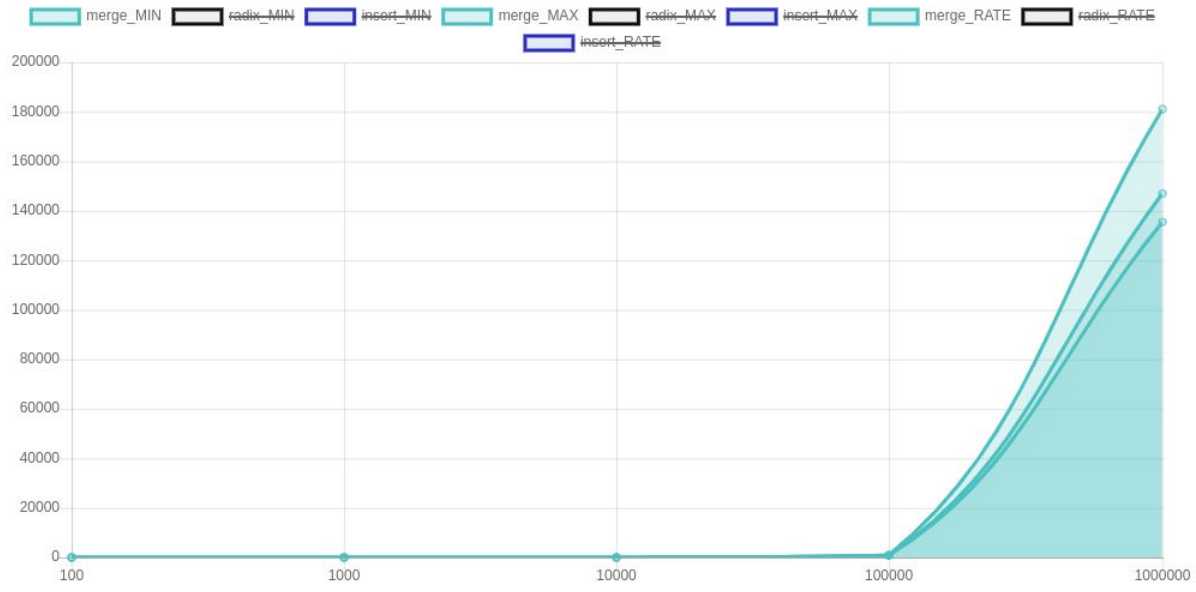
Insertion sort

A figura abaixo mostra o gráfico do tempo de execução em relação ao tamanho da instância do insertion sort no melhor tempo, tempo médio e pior tempo. Note que o algoritmo obteve uma variação muito pequena de tempo de execução para as instâncias, pois as as linhas do gráfico estão muito próximas.



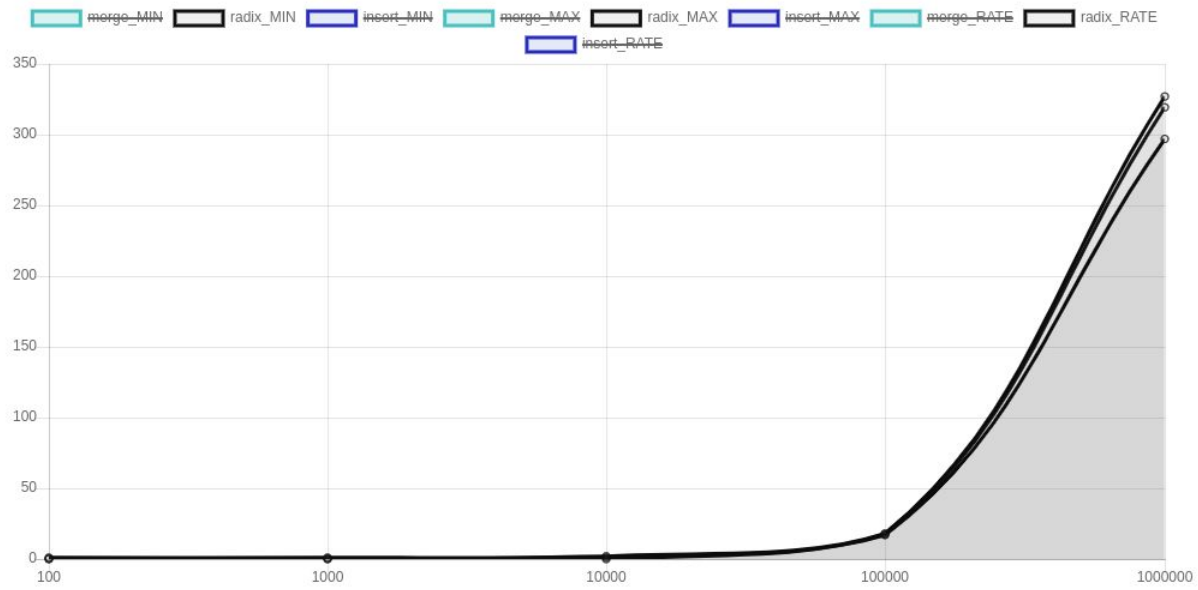
Merge sort:

A figura abaixo mostra o gráfico do tempo de execução em relação ao tamanho da instância do merge sort no melhor tempo, tempo médio e pior tempo. Diferente do insertion sort, o merge sort obteve uma maior variação entre o melhor tempo, tempo médio e pior tempo.



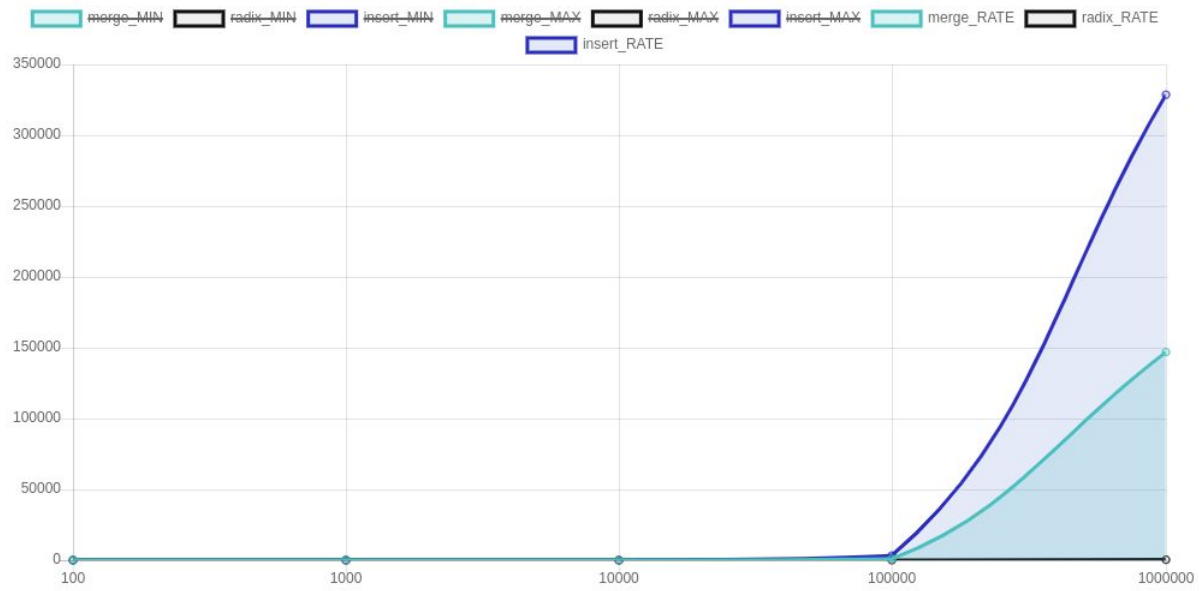
Radix Sort

A figura abaixo mostra o gráfico do tempo de execução em relação ao tamanho da instância do radix sort no melhor tempo, tempo médio e pior tempo. A variação do tempo de execução no melhor tempo, pior tempo e tempo médio do radix sort foi maior que o insertion sort e menor que o merge sort.



Tempo médio: Insertion vs Merge vs Radix

O gráfico abaixo compara o insertion sort, merge sort e radix sort, todos em relação ao tempo médio de gasto na ordenação de cada instância. Conforme mostra o gráfico, para as instâncias menores, todos algoritmos tem um bom desempenho, tendo pouca variação entre o tempo de execução. Porém quando o número de instância é muito grande, o insertion sort possui um tempo de execução muito maior do que o radix sort e merge sort. Já o radix sort, obteve uma menor curva de crescimento no tempo de execução em relação ao crescimento da instância. Tendo o melhor desempenho entre os algoritmos comparados.



Teste T pareado

A tabela a seguir mostra o resultado do teste t pareado com 95% de confiança. Na primeira coluna, é mostrado o tamanho da instância avaliada em questão. A segunda coluna é correspondente a comparação entre o insertion sort e merge sort. A terceira coluna é a comparação do insertion sort e radix sort. Por fim, a última coluna mostra o resultado da comparação entre o merge sort e radix sort. Os possíveis resultados para a tabela são:

merge: indica que o merge teve um melhor desempenho em comparação ao algoritmo concorrente.
radix: indica que o radix teve um melhor desempenho em comparação ao algoritmo concorrente.
insertion: indica que o insertion teve um melhor desempenho em comparação ao algoritmo concorrente.
EMPATE: indica que houve um empate estatístico.

length	insert_X_merge	insert_X_radix	merge_X_radix
100	'EMPATE'	'EMPATE'	'EMPATE'
1000	'merge'	'EMPATE'	'radix'
10000	'merge'	'radix'	'radix'
100000	'merge'	'radix'	'radix'
1000000	'merge'	'radix'	'radix'

5. Conclusão

Conforme foi mostrado nos gráficos no capítulo anterior, o algoritmo que teve o melhor resultado para a maioria das instâncias foi o radix sort, em segundo lugar foi o merge sort e por fim o insertion sort. Porém, nem para todos os conjuntos de instâncias foram o desempenho foi nessa ordem. Além disso, para instâncias menores, não teve uma diferença muito grande de desempenho entre os algoritmos, como mostrou o teste pareado t, houve empate estatístico em todas comparações das instâncias de 100 elementos e houve um empate estatístico também na instância de 1000 elementos.

Conforme aponta do teste pareado t e também os gráficos de comparação entre os algoritmos, quanto maior o número da instância de entrada, maior fica a diferença de desempenho entre os algoritmos.

É importante ressaltar também que a comparação do desempenho do tempo gasto pela execução dos algoritmos foram conforme o esperado, devido a suas ordens de complexidade.

6. Referências

https://runestone.academy/runestone/books/published/pythonds/SortSearch/sorting.html

https://www.geeksforgeeks.org/insertion-sort/

https://medium.com/@henriquebraga_18075/algoritmos-de-ordena%C3%A7%C3%A3o-iii-insertion-sort-bfade66c6bf1

http://www2.dcc.ufmg.br/disciplinas/aeds2_turmaA1/radixsort.pdf

https://www.programiz.com/dsa/merge-sort

https://www.chartjs.org/docs/latest/

https://brilliant.org/wiki/radix-sort/#:~:text=Radix%20sort%20is%20an%20integer,sort%20an%20array%20of%20numbers.

https://gist.github.com/bellbind/801c5de20aa4f310062363f3f95a099e