RITA C. C. CRUZ
VICTOR V. TARGINO

Análise Empírica de Algoritmos de Ordenação

Natal
2021

RITA C. C. CRUZ
VICTOR V. TARGINO

Análise Empírica de Algoritmos de Ordenação

Relatório técnico apresentado à disciplina de Estrutura de Dados Básicas I, como requisito parcial para obtenção de nota referente à unidade I.

Universidade Federal do Rio Grande do Norte - UFRN
Instituto Metrópole Digital - IMD
Bacharelado em Tecnologia da Informação

Natal
2021

SUMÁRIO

1 INTRODUÇÃO

A resolução de problemas pode ser modulada por meio de um conjunto sistemático de passos ao qual chamamos de algoritmo. Nesse sentido, dois aspectos se tornam relevantes: correção e análise. O primeiro, como o nome sugere, consiste em verificar comportamento de entrada-saída do algoritmo, isto é, se para cada entrada há uma saída correta de acordo com o problema proposto a ser resolvido. O segundo atenta para a eficiência em termos de tempo de execução e memória ocupada.

Devido à sua aplicabilidade, os algoritmos de ordenação foram os primeiros a gerar uma discussão em termos de eficiência (SZWARCFITER e MARKENZON (2010)). A príncípio, é necessário compreender que massas de dados modelam situações e que escolher o melhor algoritmo representa a economia de recursos, o que direciona a uma escolha sempre inteligente e elegante. Uma vez compreendida e fixada a relevância da análise de algoritmos, cabe o destaque ao modo apropriado de testes. Nesse sentido, deve ser destacado que a depender da massa de dados, a performance de cada algoritmo varia. Um exemplo disso, é o bubble sort: apesar de a complexidade de pior caso (massa de dados a ser ordenada está em ordem reversa ao desejado) ser ruim , $O(n^2)$, este método pode ser conveniente quando a massa de dados é pequena ou quando está quase totalmente ordenada. Desta forma, a análise completa visa não apenas testar a eficiência de cada método para uma mesma massa de dados, mas também a eficiência de cada método para diversas massas de dados.

O escopo deste relatório, de modo geral, será a análise empírica de cada um dos sete algoritmos, testando seu tempo de execução para cada um dos seis casos de entrada. Discussões acerca do uso de memória também serão construídas, porém com menos destaque.

2 METODOLOGIA

2.1 Materiais utilizados

2.1.1 Computador

A tabela abaixo mostra as especificações do computador e sistema operacional usados para realizar os testes de tempo:

Inspiron One 2330		
Processador	Memória	Armazenamento
Intel(R) Core i5-3330S CPU @ 2.70GHz x 4	2 x SODIMM DDR3 1600 MHz (0,6 ns)	SSD 240GB
Ubuntu		
Versão	Kernel	Arquitetura
20.04.2 LTS	Linux 5.8.0-59-generic	x86-64

Tabela 1: Configurações do computador e sistema usados

2.1.2 Ferramentas de programação

Todos os sete algoritmos de ordenação e o programa que prepara conta o tempo dos casos foram escritos em C++14, e os gráficos foram gerados utilizando-se Python versão 3.7.11.

Os códigos foram compilados com g++ versão 9.3.0. Para executá-los é necessário utilizar a seguinte linha de código:

```
g++ -std=c++14 -Wall <arquivos .cpp> <arquivos .h> -o <executável>
```

2.2 Algoritmos

2.2.1 Insertion Sort

Neste algoritmo seleciona-se um elemento de uma lista, começando pelo segundo, e compara-se com os elementos anteriores, a partir do primeiro, ao encontrar um que seja

maior que o selecionado, eles serão trocados de posição e o próximo será selecionado. Confira abaixo um pseudocódigo mostrando seu funcionamento:

Algorithm 1 Insertion Sort

```
1: função INSERTION(A, sz)                ▷ Onde A - lista, sz - tamanho da lista
2:     para i ← 1 até sz faça
3:         para j ← i até 0 ou A[j] ≥ A[j − 1] faça
4:             troca(A[j], A[j − 1])
5:         fim para
6:     fim para
7: fim função
```

Este algoritmo é simples na sua implementação, porém no pior caso sua complexidade é quadrática, já no melhor caso ela é linear.

2.2.2 Selection Sort

Neste algoritmo há um laço externo que percorre todos os elementos, e um laço interno que começa um elemento após o externo, este laço interno seleciona o menor elemento e se ele for menor que o elemento do laço externo, ambos terão suas posições trocadas. Confira abaixo um pseudocódigo mostrando seu funcionamento:

Algorithm 2 Selection Sort

```
1: função SELECTION(A, sz)                ▷ Onde A - lista, sz - tamanho da lista
2:     para i ← 0 até sz faça
3:         min ← i
4:         para j ← i + 1 até sz faça
5:             se A[j] < A[min] então
6:                 min ← j
7:             fim se
8:         fim para
9:         se min ≠ i então
10:            troca(A[i], A[min])
11:        fim se
12:    fim para
13: fim função
```

Apesar da simples implementação o seu pior e melhor caso são quadráticos, visto que ele sempre irá percorrer todos os elementos do laço procurando o menor valor.

2.2.3 Bubble Sort

Bolhas, quando o fluido dentro delas é menos denso que o fluido do ambiente externo, flutuam de maneira proporcional ao seu tamanho, *Bubble Sort* funciona de maneira análoga: o algoritmo percorre repetidamente uma lista, compara elementos adjacentes e os troca caso não estejam em ordem.

Neste projeto foi implementado uma variante deste algoritmo nela há uma variável que armazena e último índice modificado, se não houver nenhuma modificação o algoritmo é finalizado. Confira abaixo um pseudocódigo mostrando seu funcionamento:

Algorithm 3 Bubble Sort

```
1:  função BUBBLE(A, sz)                ▷ Onde A - lista, sz - tamanho da lista
2:      i ← sz
3:      enquanto i ≠ 0 faça
4:          temp ← 0
5:          para j ← 1 até sz faça
6:              se A[j] < A[j − 1] então
7:                  troca(A[j], A[j − 1])
8:                  temp ← j
9:              fim se
10:         fim para
11:         i ← temp
12:     fim enquanto
13: fim função
```

Seu pior é quadrático e seu melhor caso é linear.

2.2.4 Shell Sort

Esse algoritmo começa comparando um par de elementos que estão há uma distância determinada até o último elemento, e essa diferença é diminuída progressivamente. E se há um troca, haverá um laço que irá verificar a ordem dos elementos anteriores mantendo a diferença.

Esse tipo de ordenação possui diferentes formas de calcular a distância. Nesse projeto foi escolhido a versão criada por Donald E. Knuth que possuiu a seguinte fórmula geral:

$$\frac{3^k - 1}{2} \text{ e menor que } \frac{n}{3},$$

onde n é o tamanho do arranjo. Esse método de ordenação tem como complexidade $O\left(n^{\frac{3}{2}}\right)$. Confira abaixo um pseudocódigo mostrando seu funcionamento:

Algorithm 4 Shell Sort

```
1: função SHELL(A, sz)                          ▷ Onde A - lista, sz - tamanho da lista
2:     gap ← 1
3:     enquanto gap < sz faça
4:         gap ← (3 * gap) + 1                  ▷ Calcula o gap inicial
5:     fim enquanto
6:     enquanto gap > 0 faça
7:         gap ← (gap − 1)/3                    ▷ Atualiza o gap
8:         para i ← gap até i < sz faça         ▷ Laço principal onde há trocas
9:             se A[i] < A[i − gap] então
10:                troca(A[i], A[i − gap])
11:                j ← i − gap
12:                enquanto j >= 0 faça         ▷ Verifica troca de elementos anteriores
13:                    se A[gap + j] < A[j] então
14:                        troca(A[gap + j], A[j])
15:                    fim se
16:                fim enquanto
17:            fim se
18:        fim para
19:    fim enquanto
20: fim função
```

2.2.5 Merge Sort

Este algoritmo de ordenação por comparação divide o tamanho em duas metades chama-se com listas correspondendo a cada metade, e depois une enquanto ordena as duas metades em uma chamando uma função auxiliar. *Merge Sort* tem como pior caso $O(n \lg n)$. Confira abaixo um pseudocódigo mostrando seu funcionamento:

Algorithm 5 Merge Sort

função MERGE($L\_first, L\_last, R\_first, R\_last, first$) ▷ Onde L - Primeiro elemento da esquerda, L_last - Elemento após ultimo da esquerda, R - Primeiro elemento da direita, R_last - Elemento após ultimo da direita, first - Primeiro elemento da lista principal

    enquanto $L\_first \neq L\_last \wedge R\_first \neq R\_last$ **faça** ▷ Compara de 2 em 2 os elementos de R e L e os coloca em ordem a partir de $first$
        se $L\_first < R\_first$ **então**
            $first \leftarrow L\_first$
            $L\_first++$
        senão
            $first \leftarrow R\_first$
            $R\_first++$
        fim se
        $first++$
    fim enquanto
    enquanto $L\_first \neq L\_last$ **faça** ▷ Caso ainda sobre elementos de L
        $first \leftarrow L\_first$
        $L\_first++$
        $first++$
    fim enquanto
    enquanto $R\_first \neq R\_last$ **faça** ▷ Caso ainda sobre elementos de R
        $first \leftarrow R\_first$
        $R\_first++$
        $first++$
    fim enquanto
fim função

função MERGESORT($first, last$) ▷ Onde first - primeiro elemento, last - elemento após o último
    $sz \leftarrow distancia(first, last)$
    se $sz < 2$ **então**
        retorne ▷ Caso base: 1 ou 0 elementos
    fim se
    $n \leftarrow sz/2$ ▷ Acha primeira metade
    $L\_sz \leftarrow n$ ▷ Lista auxiliares
    $R\_sz \leftarrow sz - L\_sz$
    $L[L\_sz], R[R\_sz]$
    $copia(first, last, L)$ ▷ Copia do índice $[first, last)$ para L
    $copia(first, last, R)$ ▷ Copia do índice $[first, last)$ para R
    $mergesort(L, L + L\_sz)$
    $mergesort(R, R + R\_sz)$
    $merge(L, L + L\_sz, R, R + R\_sz, first)$
fim função

2.2.6 Quick Sort

Este algoritmo divide o tamanho em duas metades chama-se com listas correspondendo a cada metade, e depois une enquanto ordena as duas metades em uma chamando uma função auxiliar. Confira abaixo um pseudocódigo mostrando seu funcionamento:

Algorithm 6 Quick Sort

```
1: função PARTITION(first, last)        ▷ Onde first - Primeiro elemento da list, last - Elemento após ultimo da lista
2:     pivot ← mediana(first, last)     ▷ Move a mediana entre first, last e o valor que está na metade da lista para o final
3:     slow ← first − 1
4:     fast ← first
5:
6:     enquanto fast ≠ last faça
7:         se fast < pivot então
8:             slow + +
9:             troca(slow, fast)
10:        fim se
11:        fast + +
12:    fim enquanto
13:
14:    troca(slow + 1, last − 1) ▷ Troca final que muda o pivô para sua posição ordenada
15:    retorne slow + 1
16: fim função
17:
18: função QUICK(first, last)   ▷ Onde first - Primeiro elemento da list, last - Elemento após ultimo da lista
19:    se first < last então
20:        p ← partition(first, last)
21:        quick(first, p)
22:        quick(p + 1, last)
23:    fim se
24: fim função
```

2.2.7 Radix Sort

O *radix* ordena a partir do dígito começando pelo dígito menos significativo inserindo cada número em um linha de uma matriz de acordo com o seu valor, após isso, retorna a matriz linha por linha para o arranjo principal mantendo a ordenação e repete o procedimento até o dígito mais significativo. Sua complexidade é $O(n \log n)$, confira abaixo um pseudocódigo mostrando seu funcionamento:

Algorithm 7 Radix Sort

```
1: função RADIX(first, last)  ▷ Onde first - índice do primeiro elemento, last - índice após último elemento
2:     max ← max_digito(first, last) ▷ Retorna o número de dígitos do maior número
3:     bucket[9][ ]  ▷ Lista com todos os dígitos
4:     para i ← 0 até max faça
5:         insira(first, last, bucket, i)  ▷ Insere elementos de [first, last) em bucket de acordo com o dígito i
6:         retire(first, last, bucket)  ▷ Retira elementos em bucket, mantendo a ordem, inserindo-os em e [first, last)
7:         limpa(bucket)
8:     fim para
9: fim função
```

2.3 Medição do tempo

Nesse análise exclusivamente o tempo foi avaliado dos algoritmos, em *c++*, utilizando a biblioteca *chrono*. Os arranjos criados variam de 10^2 até 10^5 com 25 amostras, a distância de cada a amostra é de 4162 e cada uma repetia-se cinco vezes e sua média cumulativa era calculada. Inicialmente cria-se um arranjo em ordem crescente com incremento de um a cada número e a partir dele forma-se seis tipos de arranjos de acordo com sua ordenação: ordem não decrescente, ordem não crescente, aleatório até o primeiro quartil, aleatório até o segundo quartil, aleatório até o terceiro quartil, totalmente aleatório.

3 RESULTADOS

3.1 Elementos em ordem não decrescente

A figura abaixo mostra o gráfico dos tempos do arranjo em ordem não decrescente em escala logarítmica, nesse gráfico observa-se grandes variações no *selection sort* enquanto os outros algoritmos mantêm um comportamento "esperado". É possível perceber que do mais rápido para o mais lento tem-se: *bubble sort, insertion sort, merge sort, radix sort, shell sort, quick sort, selection sort.*

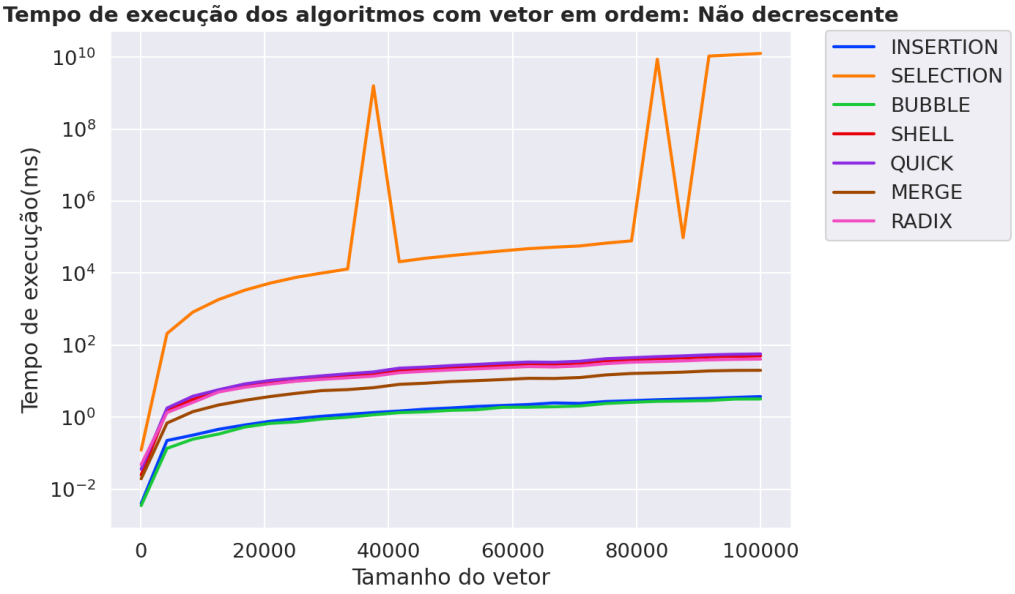

Figura 1: Arranjo em ordem não decrescente em escala logarítmica

Já nessa figura foi delimitado um limite no eixo y, para auxiliar na visualização das diferenças entre os algoritmos mais rápidos. Observa-se que *bubble sort* e *insertion sort* são muitos mais rápidos que os demais.

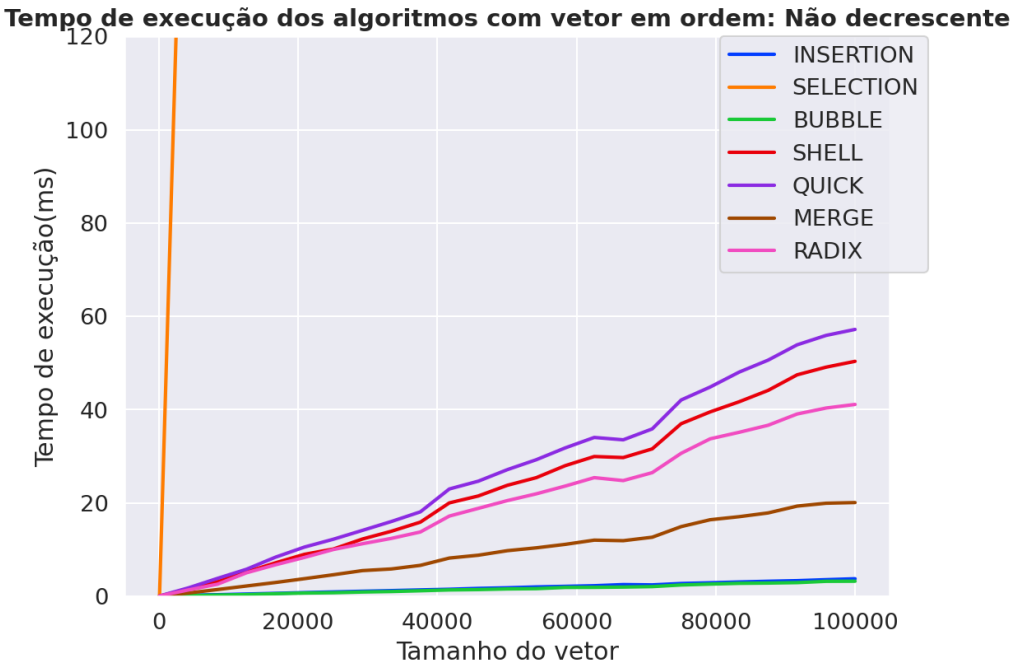

Figura 2: Arranjo em ordem não decrescente com limite no eixo y

3.2 Elementos em ordem não crescente

A figura abaixo mostra o gráfico dos tempos do arranjo em ordem não crescente em escala logarítmica, nesse gráfico observa-se grandes variações nos algoritmos *insertion sort, selection sort, bubble sort e shell sort.* É possível perceber, apesar do ruído no gráfico, que do mais rápido para o mais lento tem-se: *merge sort, radix sort, quick sort, shell sort, selection sort, insertion sort, bubble sort.*

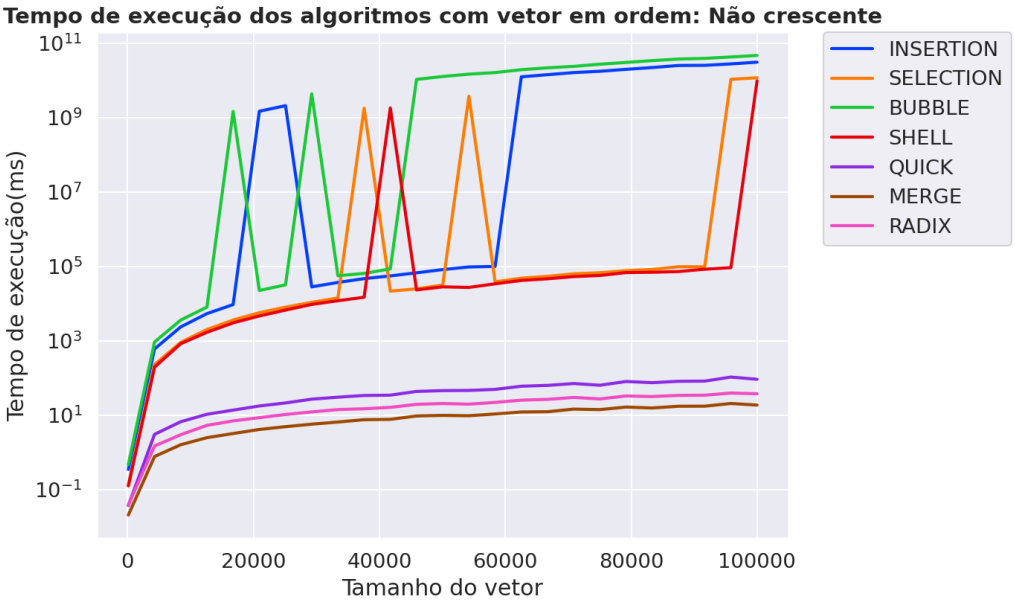

Figura 3: Arranjo em ordem não crescente em escala logarítmica

Já nessa figura foi delimitado um limite no eixo y, para auxiliar na visualização das diferenças entre os algoritmos mais rápidos.

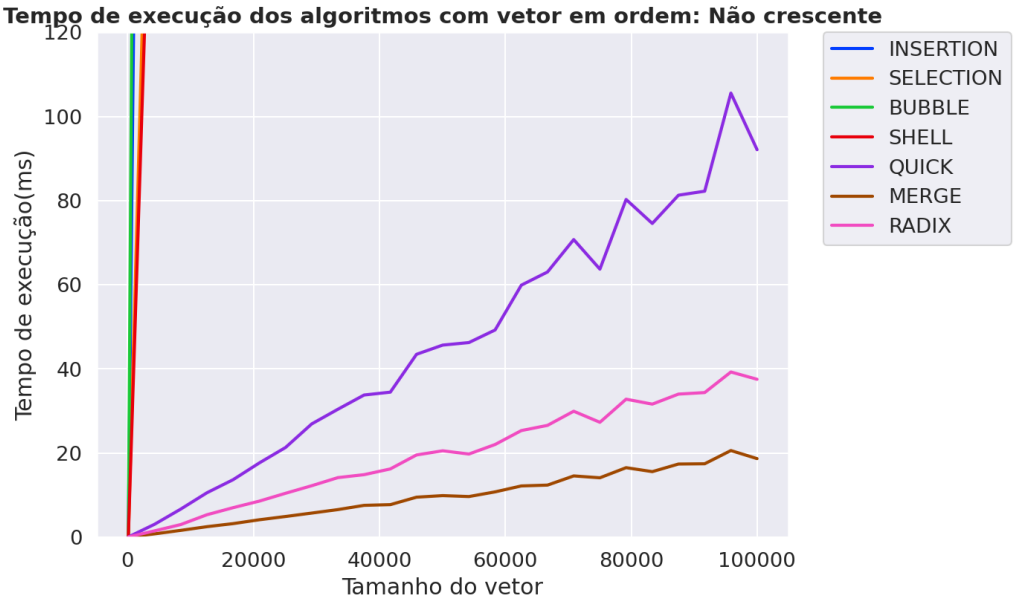

Figura 4: Arranjo em ordem não crescente com limite no eixo y

3.3 75% ordenado

A figura abaixo mostra o gráfico dos tempos do arranjo a partir do primeiro quartil ordenados em escala logarítmica, nesse gráfico observa-se grandes variações nos algoritmos

insertion sort, selection sort, bubble sort. É possível perceber, apesar do ruído no gráfico, que do mais rápido para o mais lento tem-se: *merge sort, radix sort, quick sort, selection sort, shell sort, bubble sort,insertion sort.*

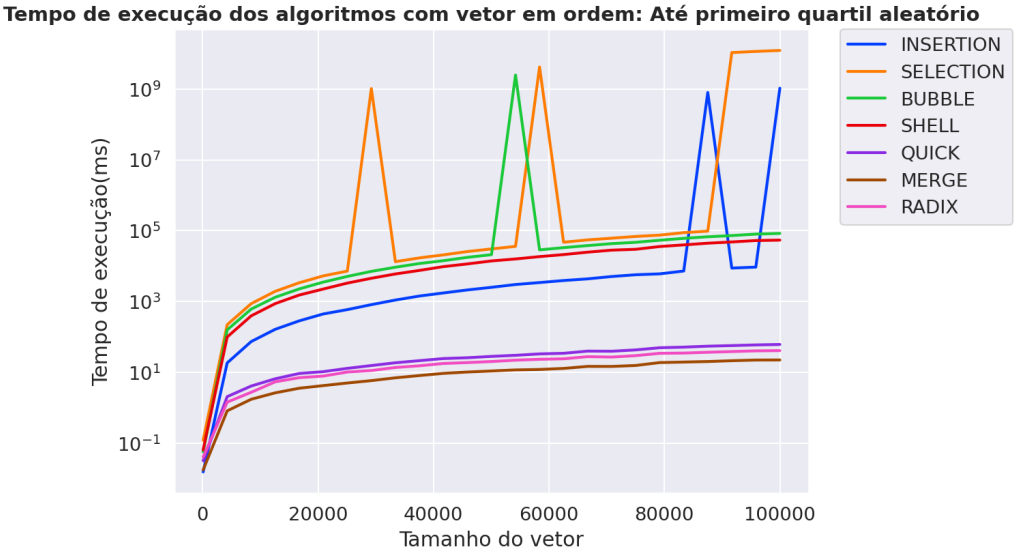

Figura 5: Arranjo com 75% ordenado em escala logarítmica

Já nessa figura foi delimitado um limite no eixo y, para auxiliar na visualização das diferenças entre os algoritmos mais rápidos.

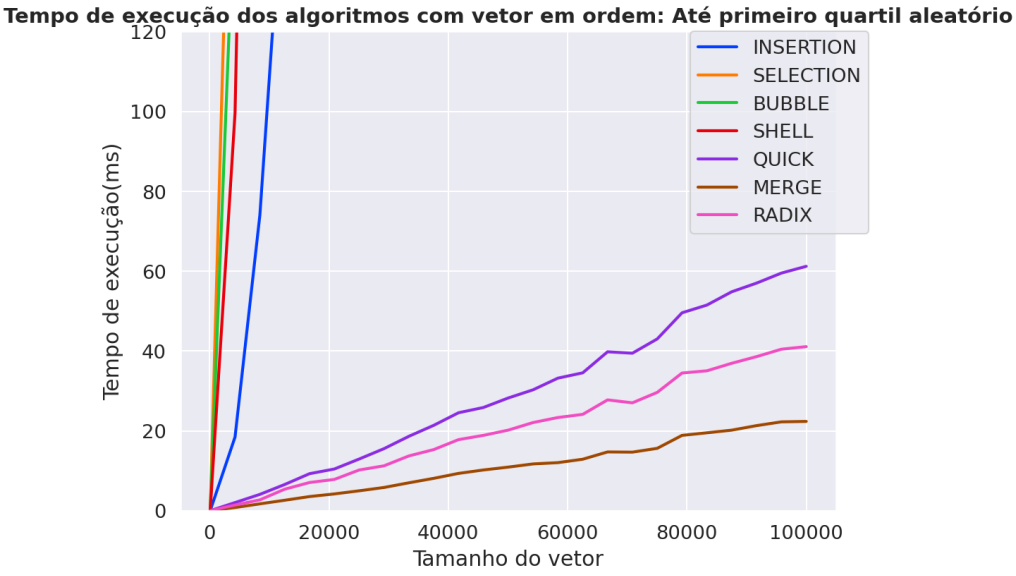

Figura 6: Arranjo com 75% ordenado com limite no eixo y

3.4 50% ordenado

A figura abaixo mostra o gráfico dos tempos do arranjo a partir do segundo quartil ordenados em escala logarítmica, nesse gráfico observa-se grandes variações nos algoritmos *insertion sort, selection sort, bubble sort.* É possível perceber, apesar do ruído no gráfico, que do mais rápido para o mais lento tem-se: *merge sort, radix sort, quick sort, selection sort, shell sort, bubble sort,insertion sort.*

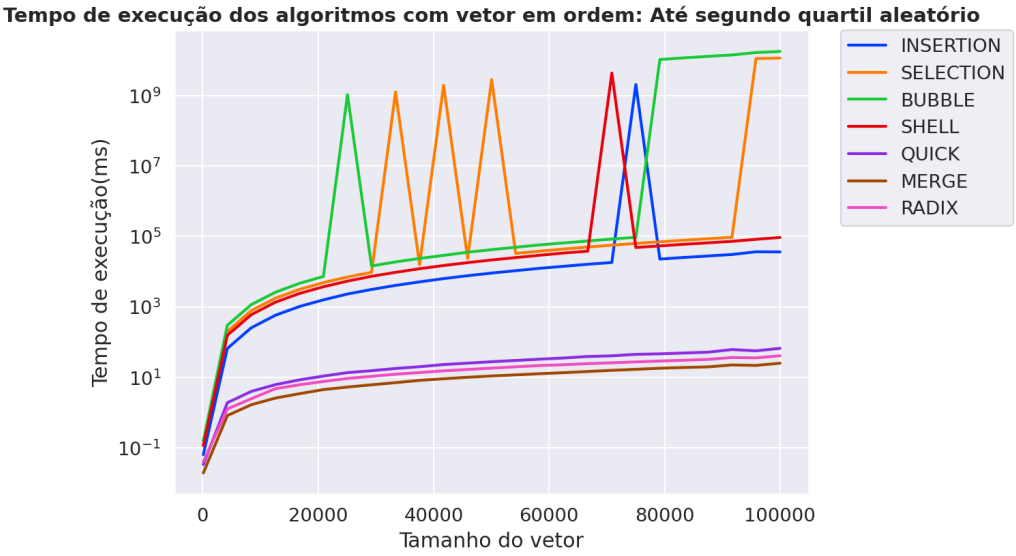

Figura 7: Arranjo com 50% ordenado em escala logarítmica

Já nessa figura foi delimitado um limite no eixo y, para auxiliar na visualização das diferenças entre os algoritmos mais rápidos.

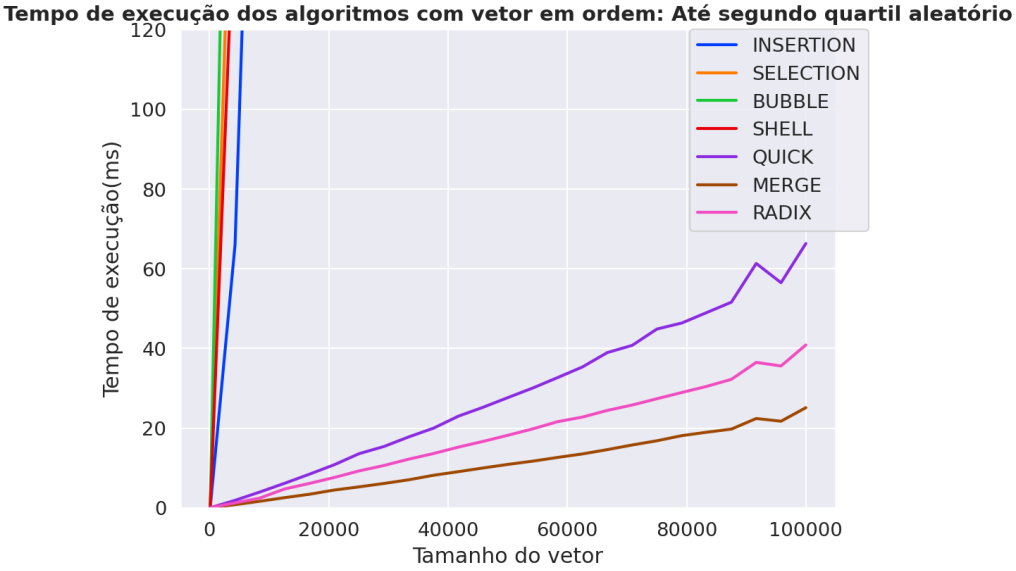

Figura 8: Arranjo com 50% ordenado com limite no eixo y

3.5 25% ordenado

A figura abaixo mostra o gráfico dos tempos do arranjo a partir do terceiro quartil ordenados em escala logarítmica, nesse gráfico observa-se grandes variações nos algoritmos *insertion sort, selection sort, bubble sort, shell sort.* É possível perceber, apesar do ruído no gráfico, que do mais rápido para o mais lento tem-se: *merge sort, radix sort, quick sort, insertion sort, shell sort, selection sort, bubble sort.*

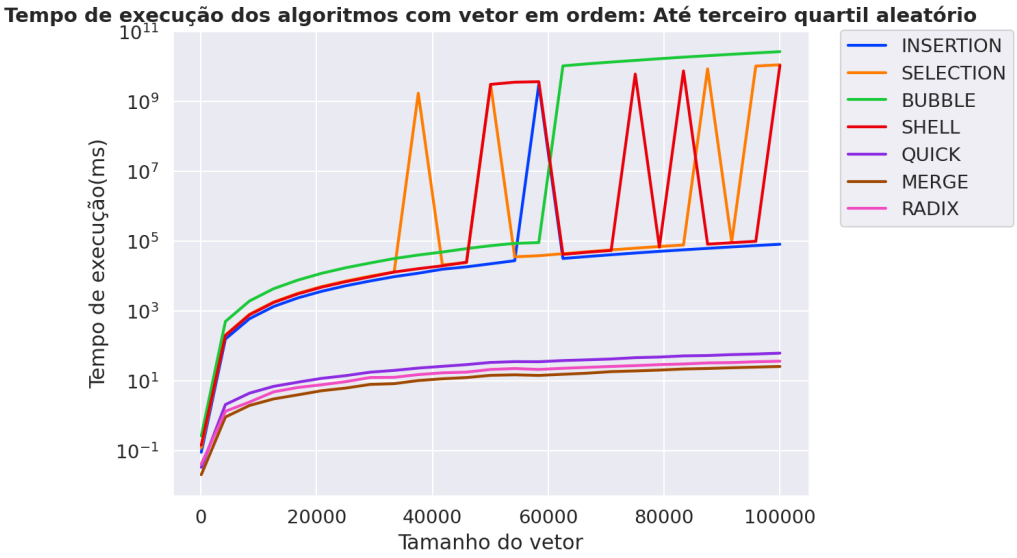

Figura 9: Arranjo com 25% ordenado em escala logarítmica

Já nessa figura foi delimitado um limite no eixo y, para auxiliar na visualização das diferenças entre os algoritmos mais rápidos.

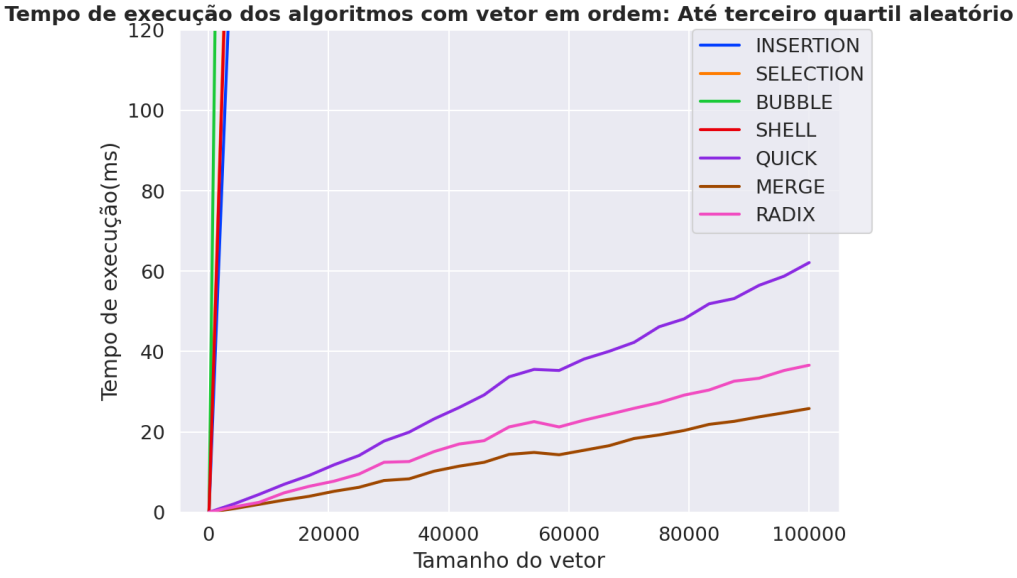

Figura 10: Arranjo com 25% ordenado com limite no eixo y

3.6 100% aleatório

A figura abaixo mostra o gráfico dos tempos do arranjo totalmente aleatório em escala logarítmica, nesse gráfico observa-se grandes variações nos algoritmos *insertion sort, selection sort, bubble sort e shell sort.* É possível perceber, apesar do ruído no gráfico, que do mais rápido para o mais lento tem-se: *merge sort, radix sort, quick sort, selection sort, shell sort, bubble sort,insertion sort.*

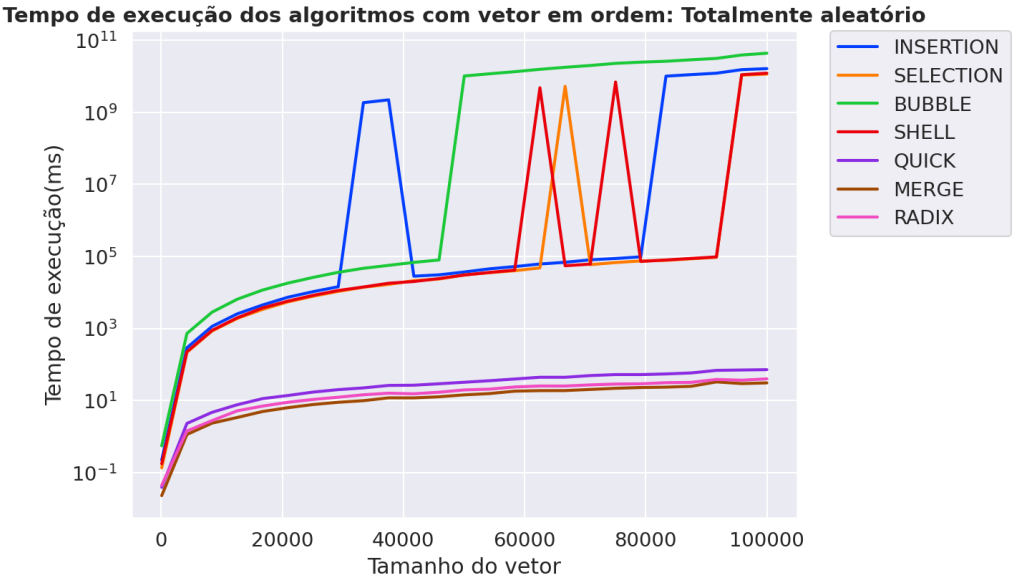

Figura 11: Arranjo em ordem totalmente aleatória em escala logarítmica

Já nessa figura foi delimitado um limite no eixo y, para auxiliar na visualização das diferenças entre os algoritmos mais rápidos.

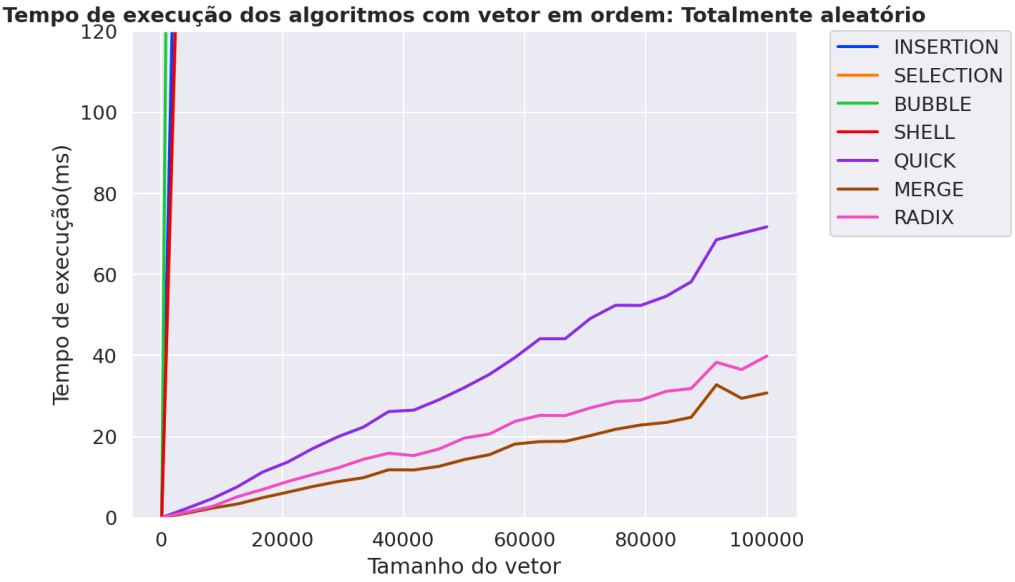

Figura 12: Arranjo em ordem totalmente aleatória com limite no eixo y

4 DISCUSSÃO

Na maior parte dos casos, as curvas que registram o melhor comportamento de runtime são *quicksort*, *mergesort* e *radix*. Para estes resultados, os cenários são elementos em ordem não crescente, 100% dos elementos em ordem aleatória, 75% dos elementos em ordem aleatória, 50% dos elementos em ordem aleatória e 25% dos elementos em ordem aleatória. Partindo disso, é importante destacar que a fase de ordenação de uma massa de dados é um pré-processamento na maior parte das vezes, isto é, consiste em um aprimorar de dados para que estes possam ser utilizados posteriormente; é um objetivo intermediário, não final. Nesse sentido, talvez o teste que mais se aproxima de um modelo frequentemente encontrado seja o de entrada aleatória. Isso porque entradas quase ordenadas, de qualquer que seja o modo (não crescente ou não decrescente), são cenários de menor probabilidade. Logo, o melhor algoritmo, considerando este aspecto, é o *mergesort*, que obteve melhor desempenho que os outros dois candidatos (*quicksort* e *radix*).

Outro caso frequente é aquele cujos dados estão ordenados, porém houve alguma inserção a esta estrutura. Deste modo, é natural que tal inserção seja feita no fim da estrutura, o que pode ser compreendido como os cenários 25% dos elementos em ordem aleatória, 50% dos elementos em ordem aleatória ou 75% dos elementos em ordem aleatória. Para estes casos, o algoritmo com melhor desempenho é também o *mergesort*.

Por fim, um dos resultados esperados no estudo advinha da literatura de Jayme Luiz Szwarcfiter e Lilian Markenzon, autores do livro *Estruturas de Dados e Seus Algoritmos*. Szwarcfiter e Markenzon alertaram sobre o comportamento do bubble sort e do insertion sort: a complexidade de pior caso é ruim, $O(n^2)$ para os dois métodos, porém, devido a otimizações de código, estes algoritmos são convenientes para o caso de uma entrada já ordenada, o que representa o melhor caso e tem complexidade temporal $O(n)$.

4.1 Melhores algoritmos por caso

Analisando os resultados, percebe-se que o *merge sort* foi o melhor em quase todos os cenários, com o arranjo em ordem não decrescente o *bubble sort* foi o mais eficiente. Entretanto, por usar recursão o *merge sort* possui um alto consumo de memória, assunto

que não está no escopo de discussão desse relatório. A figura abaixo apresenta a menor soma de tempo em cada tipo de ordenação.

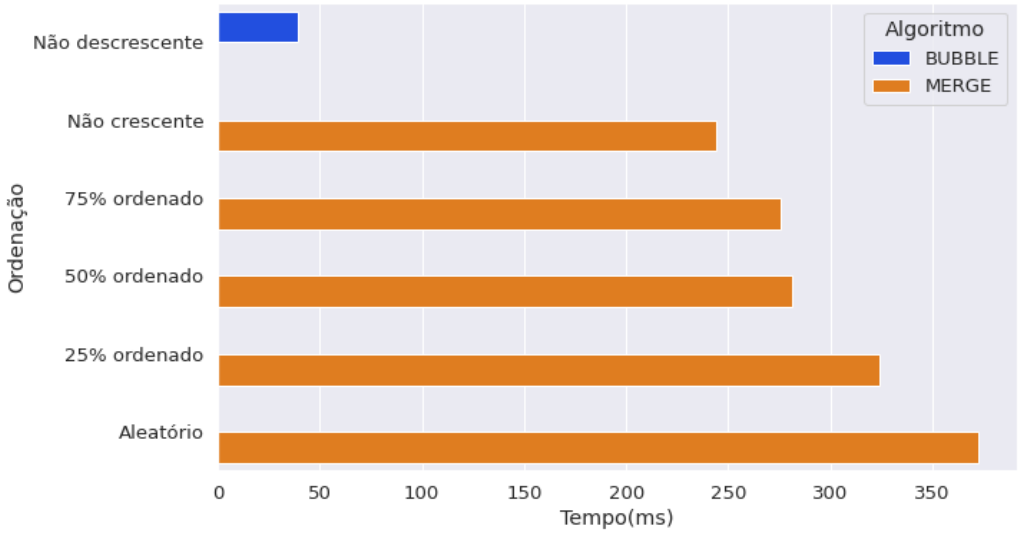

Figura 13: Comparação das menores somas dos tempos dos algoritmos em cada tipo de arranjo

4.2 Comparação e decomposição de chaves

Algoritmos de ordenação do tipo *comparação* aplicam uma função comparativa sobre seus elementos para determinar a ordem final da estrutura de dados, enquanto algoritmos de ordenação que *não utilizam comparação* procedem utilizando algum tipo de análise de chave que determina a posição final do elemento sem a necessidade de comparar com os demais. Nesta análise, apenas o radix não utiliza comparação, a ordenação é realizada pela decomposição de cada número em seus dígitos, conforme exposto na seção 2.2.7.

Uma vez fixadas as definições, o melhor algoritmo de comparação dentre os seis testados é o *mergesort*, de complexidade temporal $O(n \lg n)$. Sobre o *radix*, sua complexidade é $O(n)$. Esses resultados teóricos atribuem maior eficiência ao *radix*, entretanto o *mergesort* foi mais eficiente para cada entrada de cada um dos cenários propostos na análise empírica realizada. Uma possível explicação é que, para cada entrada, o *radix* precisa fazer uma busca pelo maior elemento daquele arranjo antes de iniciar a ordenação. A discussão aprofundada e rigorosa sobre este resultado fica como objetivo para futuros estudos.

4.3 *Quicksort* e *mergesort*

Apesar das grandes variações que houveram nos testes os dois algoritmos não tiveram grande outliers, portanto é certo dizer que o *merge sort* se demonstrou melhor que o *quick sort* em todos os casos de teste, sendo melhor que o *quick*.

4.4 Comportamento anômalo

Os algoritmos que apresentaram algum comportamento anômalo foram: *insertion sort, selection sort , bubble sort e shell sort*. O que eles têm em comum é sua complexidade espacial, ou seja, o espaço de memória necessário para resolver o problema da ordenação em função do tamanho de arranjo de entrada, que é de $O(1)$, assim o algoritmo gasta a mesma quantidade de memória. Porém, não é possível definir nenhum causador para esses outliers visto que há notáveis limitações na medição dos testes como o tempo calculado é baseado na média os valores extremos provenientes de problemas do computador e do sistema de quem o está rodando podem ter grande impacto medição. Assim se faz necessária uma investigação mais profunda a respeito desse problema.

4.5 Estimativa para 10^{12} elementos

Nas estimativas para 10^{12} elementos, será considerado o cenário de arranjo com elementos totalmente aleatórios. A estimativa é feita com base na entrada de tamanho 99988. Portanto, haverá um aumento de aproximadamente 10^7 vezes em cada arranjo.

4.5.1 Algoritmos de complexidade $O(n)$

O único algoritmo de complexidade linear testado foi o *radix*. De sua complexidade temporal ser $O(n)$, decorre que o tempo de execução será diretamente proporcional à entrada. Pela análise empírica, 39.8711ms foi o tempo necessário para ordenar uma entrada de 99988 elementos. Assim, estima-se um aumento de 10^7 neste tempo, resultando em 398711000ms, equivalente a aproximadamente 110 horas.

4.5.2 Algoritmos de complexidade $O(n \lg n)$

O único algoritmo de complexidade linearítmica testado foi o *mergesort*. Pela análise empírica, 30.765899988ms foi o tempo necessário para ordenar uma entrada de 99988 elementos. Com um aumento de 10^7 vezes no tamanho da entrada, estima-se um aumento no tempo de resposta em $10^7 \cdot \lg 10^7$ vezes, isto é, 70000000, resultando em 2153612999.16ms, equivalente a aproximadamente 598 horas.

4.5.3 Algoritmos de complexidade $O\left(n^{\frac{3}{2}}\right)$

O único algoritmo de complexidade $O\left(n^{\frac{3}{2}}\right)$ testado foi o *shellsort*. Pela análise empírica, 12243699988ms foi o tempo necessário para ordenar uma entrada de 99988 elementos. Com um aumento de 10^7 vezes, estima-se um aumento no tempo de resposta em $10^{\frac{21}{2}}$ vezes, resultando em 3871797894985625410 51.74300905984ms, equivalente a aproximadamente 122773905 séculos.

4.5.4 Algoritmos de complexidade $O(n^2)$

Para os algoritmos quadráticos, estima-se um aumento em aproximadamente 10^{14} vezes no tempo de resposta. A tabela a seguir ilustra os tempos estimados para a entrada de tamanho 10^{12} com base na entrada de tamanho 99988.

	Runtime (ms)	
Algoritmo	99988 elementos	10^{12} elementos
Selection	11544099988	$11544099988 \cdot 10^{14}$
Insertion	16361899988	$16361899988 \cdot 10^{14}$
Quick	71.727399988	$71727399988 \cdot 10^5$
Bubble	43864099988	$4386409998 \cdot 10^{14}$

Tabela 2: Estimativa para entrada de tamanho 10^{12}

Os resultados parecem absurdos, mas é necessário ter em mente que 10^{12} é um número extremamente grande. Para se ter ideia, segundo a Agência Espacial Europeia, 10^{12} é a quantidade estimada de galáxias no Universo.

4.6 Análise empírica e análise matemática

Curve fitting é um tipo de otimização que busca os parâmetros que melhor definem uma função utilizando o método dos mínimos quadrados não lineares para o conjunto

de dados observados. Os resultados matemáticos a seguir foram descritos utilizando a biblioteca *SciPy* do *Python*.

Para o *radix*, existe a função $an+b$, onde $a = 0.0003821192606897709$ e $b = 0.3886484098165662$. A medida que n tende ao infinito, as constantes a e b deixam de ser significativas, fazendo com que o valor da expressão dependa apenas de n – comportamento linear.

Para o *mergesort*, existe a função $an \lg(nc+d)+b$, onde $a = 6.045548487858693 \cdot 10^{-5}$, $b = 0.4289070765032858$, $c = 0.9886377113244929$ e $d = -98.863767402566$. Quando n tende ao infinito, todas as constantes passam a não expressar alterações significativas, de modo que os termos relevantes são $n \lg n$ – comportamento linearítmico.

Para o *bubble*, existe a função $a(nc + d)^2 + b$, tal que $a = 1.7601557572319748$, $b = -868349569.2267942$, $c = 1.7998715855768297$ e $d = -22784.54695992194$. Quando n tende ao infinito, todas as constantes passam a não expressar alterações significativas, de modo que o termo relevante é n^2 – comportamento quadrático.

Para o *quicksort*, existe a função $a(nc+d)^2+b$, tal que $a = 4.276410202182439 \cdot 10^{-9}$, $b = -74.07924849038874$, $c = 0.5252957121014648$ e $d = 131995.6627810443$. Quando n tende ao infinito, todas as constantes passam a não expressar alterações significativas, de modo que o termo relevante é n^2 – comportamento quadrático.

Para o *insertion*, existe a função $a(nc + d)^2 + b$, tal que $a = 2.7891606824216777$, $b = -1538211928.4811208$, $c = 1.1486554841510028$ e $d = -38613.91304667779$. Quando n tende ao infinito, todas as constantes passam a não expressar alterações significativas, de modo que o termo relevante é n^2 – comportamento quadrático.

Para o *selection*, existe a função $a(nc + d)^2 + b$, tal que $a = 1.6292248820444857$, $b = -836916393.0362043$, $c = 1.0306933885940988$ e $d = -36285.028168465804$. Quando n tende ao infinito, todas as constantes passam a não expressar alterações significativas, de modo que o termo relevante é n^2 – comportamento quadrático.

Para o *shell*, existe a função $a(nc + d)^{\frac{3}{2}} + b$, tal que $a = 8.242216370161819$, $b = -1004714821.6122259$, $c = 8.097950420559217$ e $d = -809.7683255007822$. Quando n tende ao infinito, todas as constantes passam a não expressar alterações significativas, de modo que o termo relevante é $n^{\frac{3}{4}}$.

Portanto, as análises matemáticas estão todas de acordo com as análises empíricas, comprovando que para entradas cada vez maiores (n tendendo ao infinito) os algoritmos de melhor complexidade temporal são radix e mergesort.

REFERÊNCIAS

SZWARCFITER; MARKENZON. Estruturas de dados e seus algoritmos. **LTC**, 2010.