Danillo Melo da Fonseca
Eduarda Rodrigues Chiesa

- Análise Empírica de algoritmos de Ordenação -

Brasil
2023, v-1.0

Danillo Melo da Fonseca
Eduarda Rodrigues Chiesa

Análise Empírica de algoritmos de Ordenação

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

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

Brasil
2023, v-1.0

Sumário

1 Introdução

Problemas computacionais referem-se aos desafios que os cientistas da computação enfrentam ao resolver tarefas específicas usando algoritmos e computadores. Sempre que desejamos resolver tais problemas, é necessário, antes de tudo, descrevê-lo de forma precisa por meio de algoritmos. Um algoritmo computacional é uma sequência de passos bem definidos e organizados que descrevem como resolver um problema ou realizar uma tarefa em um computador. Ele é uma representação abstrata de um processo computacional, fornecendo uma solução passo a passo para alcançar um objetivo específico. (BACKES, 2023).

Com efeito, um algoritmo deve ser preciso, ou seja, cada passo deve ser claramente definido e sem ambiguidade. Além disso, ele deve ser eficiente, visando resolver o problema de forma rápida e utilizando recursos computacionais de maneira eficiente. Os algoritmos podem ser expressos em diferentes formas, como linguagens de programação, pseudocódigo ou diagramas de fluxo. Eles podem ser implementados em diversos ambientes de programação, desde software básico até sistemas operacionais e aplicativos específicos.

Este relatório objetiva apresentar os resultados de análises empíricas e comparações de algoritmos que objetivam ordenar, em cenários plurais de pré-ordenamento de arranjos. Implementamos sete algoritmos de ordenação: insertion sort, selection sort, bubble sort, shell sort, quick sort, merge sort e radix sort. Foi implementado um algoritmo para cada, buscando melhorar a eficiência quando possível. Os resultados foram registrados em tabelas e utilizados para a elaboração de gráficos.

2 Metodologia

2.1 Materiais Utilizados

2.1.1 Computador

- Computador Lenovo IdeaPad Gaming 3i
- Processador: Intel(R) Core(TM) i5-10300 H CPU 2.50 GHz
- RAM: 8 GB (1x8 GB) SO-DIMM DDR4 3200MHz
- GPU: NVIDIA® GeForce® GTX 1650 4 GB GDDR 6
- Armazenamento: 256 GB SSD PCIe

2.1.2 Ferramentas de Programação

No contexto descrito, os algoritmos foram implementados em C++ no Visual Studio Code e compilados usando o G++ 6.3.0 através do WSL (Windows Subsystem for Linux) no Windows 11 Pro. Foi utilizado um Makefile para compilar e executar os programas, sendo o comando "make" para compilar e "make run" para executar. O WSL permite executar um ambiente Linux dentro do Windows, proporcionando acesso ao terminal do Linux, nesse caso, ao terminal da distribuição Ubuntu.

Para cronometrar o tempo de execução dos algoritmos, foi utilizada a biblioteca chrono.

2.1.3 Algoritmos

2.1.3.1 Algoritmos de Ordenção

Para angariar respostas para o problema proposto, utilizamos a análise empírica dos algoritmos de ordenação e os colocamos em determinados cenários. Sendo assim, vamos apresentar os algoritmos elaborados:

Algoritmo 1: Bubble Sort

Input: um arranjo = {A1, A2, A3, … An}[1]
Result: arranjo ordenado

```
procedimento bubble_sort(arranjo):
    n ← tamanho(arranjo)
    trocou ← verdadeiro

    enquanto trocou seja verdadeiro faça:
        trocou ← falso
        para i de 0 até n-2 faça:
            se arranjo[i] > arranjo[i+1] então:
                trocar(arranjo[i], arranjo[i+1])
                trocou ← verdadeiro
    arranjo ← arranjo - 1

retorne arranjo
```

O algoritmo Bubble Sort é um dos algoritmos de ordenação mais conhecidos. Ele tem esse nome pois remete à ideia de bolhas flutuando em um tanque d'água em direção ao topo, até encontrarem o seu próprio nível (no caso da ordenação crescente). Tal algoritmo trabalha de forma a movimentar, uma posição por vez, o maior valor existente na porção não ordenada de um arranjo para a sua respectiva posição no arranjo que foi ordenado. Esses passos são repetidos até que todos os elementos estejam nas suas posições correspondentes.

Melhor caso: n, quando o arranjo já está ordenado.

Caso médio: n^2, quando o arranjo está aleatorizado.

Pior caso: n^2, quando o arranjo está em ordem decrescente.

[1] Na verdade, no algoritmo em C++, dois ponteiros como entrada, mas nos pseudo códigos, optamos por traduzir os ponteiros first e last como o arranjo em si.

Algoritmo 2: Insertion Sort

Input: um arranjo = {A1, A2, A3, … An}
Result: arranjo ordenado

```
procedimento insertion_sort(arranjo):
   n ← tamanho(arranjo)

   para i de 1 até n-1 faça:
      valorAtual ← arranjo[i]
      j ← i - 1

      enquanto j >= 0 e arranjo[j] > valorAtual faça:
         arranjo[j + 1] ← arranjo[j]
         j ← j - 1

      arranjo[j + 1] ← valorAtual

   retorne arranjo
```

O algoritmo insertion sort também é bastante simples. Ele tem esse nome pois se assemelha ao processo de ordenação de um conjunto de cartas de baralho com as mãos: pega-se uma carta de cada vez e a "insere" no seu devido lugar, deixando sempre as cartas da mão ordenadas. Em termos práticos, tal algoritmo possui um desempenho superior quando comparado com outros algoritmos como o bubble sort e o selection sort (vide Algoritmo 3).

O algoritmo insertion sort funciona da devida forma: percorre um arranjo e, para cada posição X, verifica se o seu valor está na posição correta. Isso é feito andando para o começo do arranjo a partir da posição X e movimentando um posição para frente os valores que são maiores do que o valor da posição X. Assim, tem-se uma posição livre para inserir o valor da posição X em seu lugar devido.

Melhor caso: n.

Caso médio: n^2.

Pior caso: n^2.

Algoritmo 3: Selection Sort:

Input: um arranjo = {A1, A2, A3, … An}
Result: arranjo ordenado

```
procedimento selection_sort(arranjo):
   n ← tamanho(arranjo)

   para i de 0 até n-1 faça:
      indiceMinimo ← i

      para j de i+1 até n faça:
         se arranjo[j] < arranjo[indiceMinimo] então:
            indiceMinimo ← j

      se indiceMinimo != i então:
         trocar(arranjo[i], arranjo[indiceMinimo])

   retorne arranjo
```

O algoritmo de ordenação por "seleção", ou selection sort, tem esse nome pois, a cada passo, "seleciona" o "melhor" elemento do arranjo (maior ou menor, dependendo do tipo de ordenação) para ocupar aquela posição. Em termos de eficiência, o selection sort possui desempenho quase sempre superior ao bubble sort.

O selection sort divide o arranjo em duas partes: a parte ordenada, à esquerda do elemento analisado, e a parte que ainda não foi ordenada, à direita do elemento. Para cada elemento do arranjo, começando do primeiro, o algoritmo procura na parte não ordenada (direita) o menor valor (ordenação crescente) e troca os dois valores de lugar. Em seguida, o algoritmo avança para a próxima posição do arranjo e repete esse processo até que o arranjo esteja completamente ordenado.

Melhor caso: n^2.

Caso médio: n^2.

Pior caso: n^2.

Algoritmo 4: Shell Sort

Input: um arranjo = {A1, A2, A3, … An}
Result: arranjo ordenado

```
procedimento shell_sort(arranjo):
   n ← tamanho(arranjo)
   intervalo ← n / 2

   enquanto intervalo > 0 faça:
      para i de intervalo até n faça:
         valor ← arranjo[i]
         j ← i

         enquanto j >= intervalo e arranjo[j - intervalo] > valor faça:
            arranjo[j] ← arranjo[j - intervalo]
            j ← j - intervalo

         arranjo[j] ← valor

      intervalo ← intervalo / 2

   retorne arranjo
```

Melhor caso: n log n.
Caso médio: a complexidade média do shell sort é considerada melhor do que $O(n^2)$, mas pior do que O(n log n).
Pior caso: n^2

Algoritmo 5: Quick Sort

Input: um arranjo = {A1, A2, A3, … An}
Result: arranjo ordenado

```
função partição(A: arranjo de ref inteiro; l: inteiro; r: inteiro):
inteiro
var x: inteiro ← A[r]
var i: inteiro ← l − 1
var j: inteiro
        para j ← l até r − 1 faça
                se compara(A[j], x) != 1 então
                i ← i + 1
                A[i] ↔ A[j]
        A[i + 1] ↔ A[r]
retorna i + 1

procedimento quicksort_aux(A: arranjo de ref inteiro; l: inteiro; r: inteiro)
se l < r então
        q ← partição(A, l, r)
        quicksort_aux(A, r, q − 1)
        quicksort_aux(A, q + 1, l)
```

O algoritmo quicksort, diferentemente dos primeiros algoritmos de ordenação apresentados neste trabalho, é recursivo e usa a ideia de “dividir para conquistar” para ordenar os dados de um arranjo. Tal algoritmo toma como base o problema da separação (no inglês, partition problem).

Esse problema resume-se em rearranjar um arranjo de modo que os valores menores que um certo “pivô” fiquem na parte esquerda do arranjo, enquanto os valores maiores do que o pivô ficam na região direita. Em geral, esse é um algoritmo muito rápido, pois parte do princípio que é mais fácil ordenar um conjunto com poucos dados do que um conjunto com vários. Todavia, em casos especiais, pode ser muito lento, como quando o arranjo está totalmente em ordem crescente ou decrescente. Nesses casos, a complexidade é quadrática.

O pior caso de desempenho ocorre quando o pivô escolhido é sempre o menor ou o maior elemento do subarranjo a ser ordenado. Existem formas de evitar o pior caso, como

usar a mediana de três ou aleatorizar totalmente a escolha do pivô, pois essas estratégias reduzem a probabilidade de escolher um pivô extremo (maior ou menor elemento da subarranjo), o que melhora o desempenho médio do algortimo. Porém, isso não exclui completamente o pior caso, apenas aumenta bastante as chances de não cair nele.

O algortimo implementado por nós não utiliza nenhuma dessas estratégias, por isso, ao receber um vetor já ordenado de forma totalmente crescente ou decrescente e com um tamanho consideravelmente grande, por exemplo, erros como segmentation fault ou stack overflow vão ocorrer eventualmente. Isso ocorre porque as chamadas recursivas excessivas que ocorrem nesse caso podem exceder o limite da pilha de execução e bagunçar as coisas.

Melhor caso: n log n.

Caso médio: n log n.

Pior caso: n^2; mas pode ser evitado, escolhendo um pivô *corretamente*. Assim, tornando-se n log n.

Algoritmo 6: Merge sort

Input: um arranjo = {A1, A2, A3, … An}
Result: arranjo ordenado

```
algoritmo mergesort_aux(ref A, l, r)
 se l < r então
        m ← (l+r)/2
        mergesort aux(A, l,m)
        mergesort aux(A,m + 1, r)
        intercale(A, l,m, r)
 fimse
 fimalgoritmo


algoritmo intercala(A, l,m, r)
 n1 ← m − l + 1
 n2 ← r − m
 defina vetores L[1, . . . , n1 + 1] e R[1, . . . , n2 + 1]
        para i = 1 até n1 faça
```

```
                L[i] ← A[l + i − 1]
fimpara
        para j = 1 até n2 faça
                R[j] ← A[m + j]
fimpara
L[n1 + 1] ← X, onde a <= X, para todo a ∈ A
R[n2 + 1] ← X, onde a <= X, para todo a ∈ A
i ← 1
j ← 1
        para k = l até r faça
                se compara(L[j],R[j])≠ 1 então
                        A[k] ← L[i]
                        i ← i + 1
                senão
                        A[k] ← R[j]
                        j ← j + 1
                fimse
        fimpara
fimalgoritmo
```

O “merge sort”, também conhecido como ordenação por “intercalação”, é um algoritmo recursivo, como o quick sort, e igualmente utiliza o conceito de “dividir para conquistar” com o intuito de ordenar os dados de um arranjo. O algoritmo divide os dados em conjuntos cada vez menores para depois ordená-los e combiná-los por meio de intercalação (merge).

Dito de outra forma, o merge sort divide, recursivamente, o arranjo em duas partes até que cada posição dele seja considerada como um arranjo de um único elemento. Após isso, o algoritmo combina dois arranjos de forma a obter um arranjo maior e ordenado. Tais combinações dos arranjos são feitas intercalando seus elementos de acordo com o sentido da ordenação (crescente ou decrescente). Esses passos vão se repetir até que exista apenas um arranjo.

Melhor caso: n log n.

Caso médio: n log n.

Pior caso: n log n.

Algoritmo 7: Radix Sort

Input: um arranjo = {A1, A2, A3, … An}
Result: arranjo ordenado

```
procedimento counting_sort(arranjo, expoente):
    tamanho ← tamanho do arranjo
    saída ← arranjo vazia
    contador ← arranjo de tamanho 10 preenchido com zeros

    Para i de 0 até tamanho - 1:
        digito ← (arranjo[i] / expoente) % 10
        contador[digito] ← contador[digito] + 1

    Para i de 1 até 9:
        contador[i] ← contador[i] + contador[i-1]

    Para i de tamanho - 1 até 0:
        digito ← (arranjo[i] / expoente) % 10
        posição ← contador[digito] - 1
        saída[posição] ← arranjo[i]
        contador[digito] ← contador[digito] - 1

    Retorne saída

procedimento radix_sort(arranjo):
    máximo ← maior valor no arranjo

    expoente ← 1
    Enquanto máximo / expoente > 0:
        lista ← counting_sort(arranjo, expoente)
```

```
    expoente ← expoente * 10

Retorne arranjo
```

O algoritmo Radix Sort é um método de ordenação não comparativo que organiza os elementos com base em suas representações numéricas, dos dígitos menos significativos aos mais significativos. Ele recebe esse nome porque classifica os elementos com base na sua "radix" ou base numérica.

O algoritmo começa identificando o número máximo de dígitos presente em todos os elementos do arranjo. Isso é feito encontrando o maior número no arranjo e contando quantos dígitos ele possui. Agora, começando o processo pelo dígito menos significativo, também conhecido como “unidade”. Cria-se 10 "baldes" para cada dígito possível, numerados de 0 a 9. O objetivo desses baldes é agrupar os elementos de acordo com o dígito atual que está sendo considerado.

Em seguida, percorre-se o arranjo original da esquerda para a direita. Para cada elemento, colocamos-o no balde correspondente ao dígito que está sendo analisado no momento. Após colocar todos os elementos nos baldes corretos, reorganizamos o arranjo original, juntando os elementos de cada balde na ordem em que foram colocados neles. Inicia-se pelo balde 0 e coloca-se todos os elementos em sequência, depois fazemos o mesmo para o balde 1, o balde 2 e assim por diante; e repete-se os passos anteriores para os próximos dígitos.

Ao terminar de processar o dígito mais significativo, o arranjo de entrada está completamente ordenado. É importante ressaltar que o radix sort preserva a ordem relativa dos elementos com dígitos iguais, garantindo a estabilidade do algoritmo. Isso significa que se dois elementos têm o mesmo dígito em uma posição específica, o elemento que ocorre primeiro no arranjo original também aparecerá primeiro no arranjo ordenado.

Melhor caso: k * n, onde "n" é o número de elementos a serem ordenados e "k" é o número de dígitos máximo entre todos os elementos.

Caso médio: k * n.

Pior caso: k * n.

2.1.4 Criação dos arranjos e obtenção dos dados

Foram considerados seis diferentes possíveis organizações para os arranjos:

1. Com seus elementos 100% aleatorizados.
2. Com seus elementos 75% aleatorizados.
3. Com seus elementos 50% aleatorizados.
4. Com seus elementos 25% aleatorizados.
5. Com seus elementos totalmente em ordem crescente.
6. Com seus elementos totalmente em ordem decrescente.

Essas diferentes organizações influenciam diretamente o tempo de execução de cada algoritmo, diferentes algoritmos podem lidar melhor ou pior com as diferentes entradas. Para obter os dados finais que foram utilizados na criação dos gráficos e tabelas, cada algoritmo foi testado para as diferentes entradas e, sempre que possível, foram realizados até 5 testes por entrada e feita a média aritmética entre eles, na tentativa de obter dados mais precisos. Nem sempre foi viável testar a quantidade de vezes desejada, uma vez que alguns algoritmos, como o bubble sort por exemplo, levam bastante tempo para terminar a execução dependendo do tamanho do arranjo, mais ainda se for considerado o arranjo que causa seu pior caso. Por isso, alguns dados referentes a piores casos foram coletados com apenas uma rodada de testes, outros, com 3 rodadas e feita a média aritmética.

Além disso, vale ressaltar que todos os testes foram realizados com um total de 25 amostras, começando com a primeira amostra, de 100 elementos apenas, indo até a última, com 99988 elementos. Os saltos entre as amostras foram de 4162 elementos, ou seja: 100, 4262, 8424, 12586…

Para aleatorizar um vetor com, digamos, 75%, dos elemento em sua ordem definitiva procedemos da seguinte forma:

1. Começamos com um arranjo V e A de dados em ordem crescente.
2. Embaralhamos os elementos do arranjo A com a função shuffle.
3. Se desejamos, por exemplo, 75% dos elementos na ordem correta, modificamos apenas 25% dos elementos originais. Para isso, fizemos um laço de i = 0 até p = [(0.25 * n)], com i sendo incrementado de 2 em 2, trocando V[A[i]] com V[A[i+1]].
4. Caso desejássemos, por exemplo, 50% dos elementos na ordem correta, apenas mudaria o valor de 0.25 para 0.50. Similarmente com 25% na ordem correta, mudaria o 0.25 por 0.75.

5. Para um arranjo totalmente aleatorizado, foi utilizada apenas a função shuffle nos elementos do arranjo inteiro.

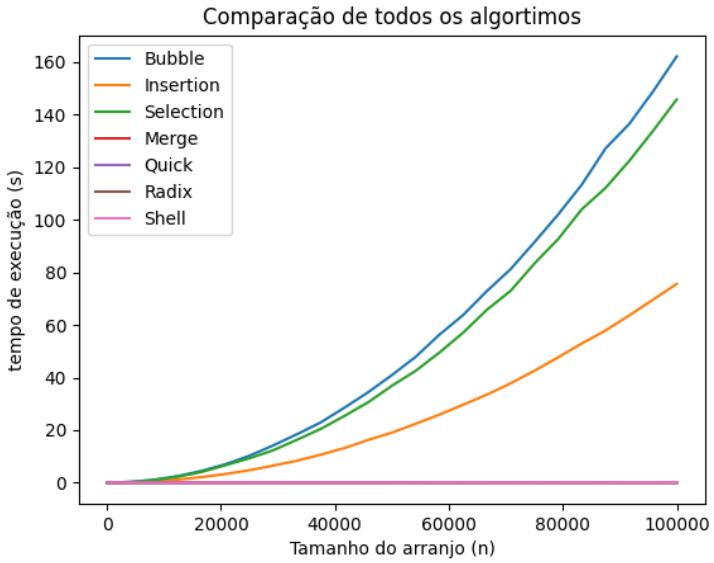
3 Resultados

Aqui serão apresentados gráficos, expondo os tempos de execução (em segundos) de diferentes algoritmos em relação ao tamanho de um arranjo e a forma como ele foi arranjado.

3.1 Gráficos

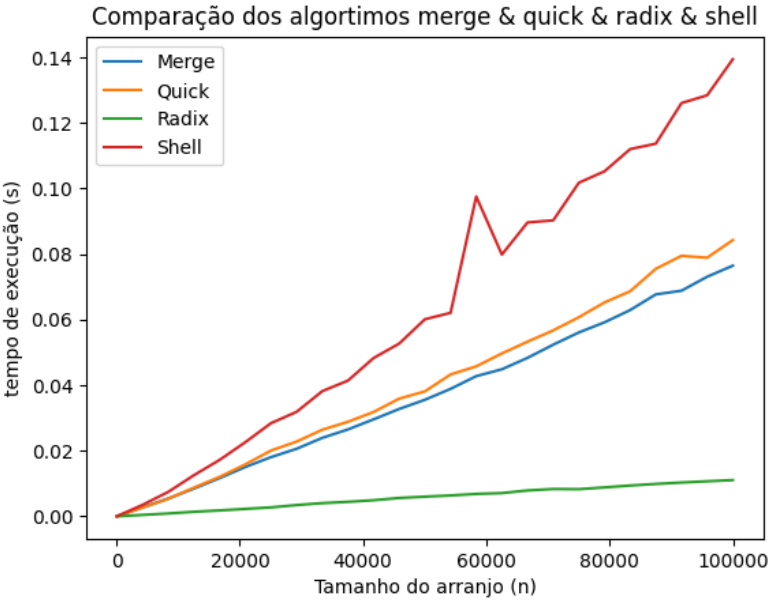
3.1.1 Arranjos 100% aleatórios

Gráfico 1 - Comparação de Todos os Algoritmos em arranjos 100% aleatórios.



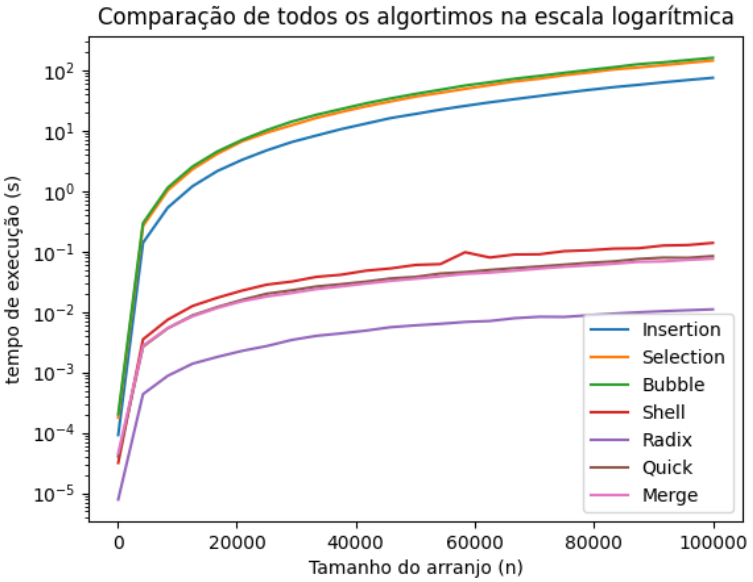
Fonte: Própria

Gráfico 2 - Comparação dos Algoritmos Merge, Quick, Radix e Shell em Arranjo 100% Aleatório



Fonte: Própria.

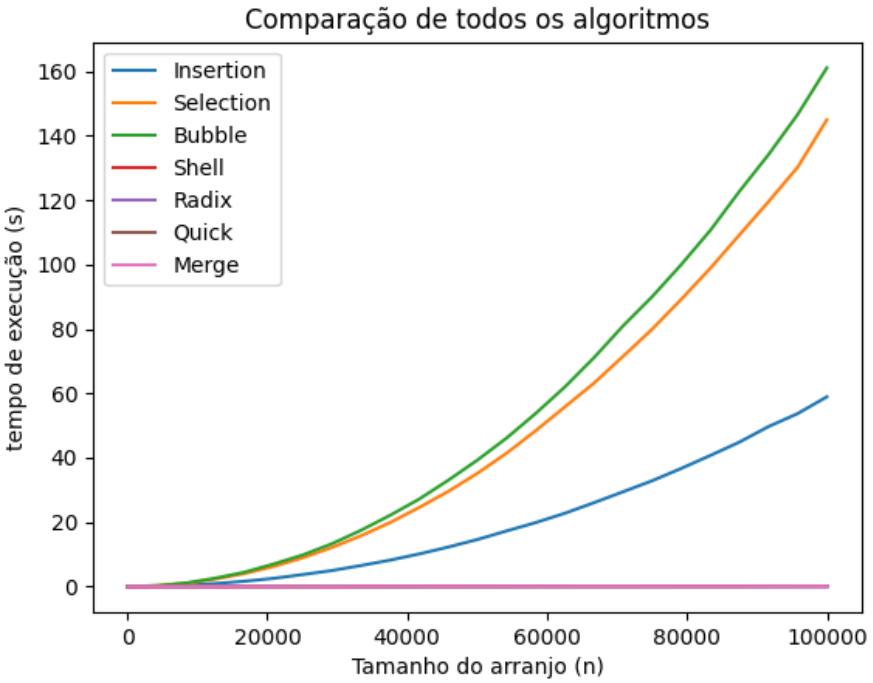
Gráfico 3 - Comparação de Todos os Algoritmos Em Arranjos 100% Aleatórios na escala logarítmica no eixo y.



Fonte: Própria.

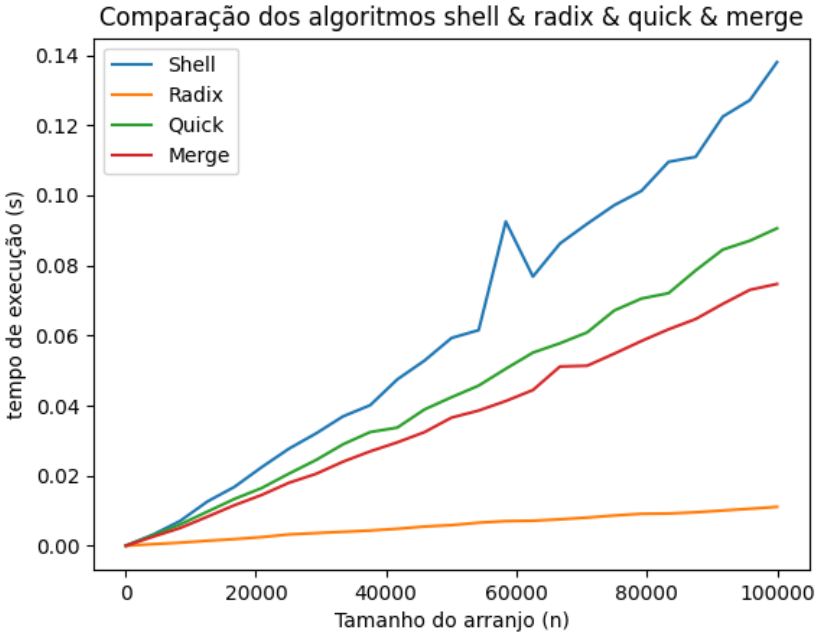
3.1.2 Arranjos 75% aleatórios

Gráfico 4 - Comparação de Todos os Algoritmos em arranjos 75% aleatórios.



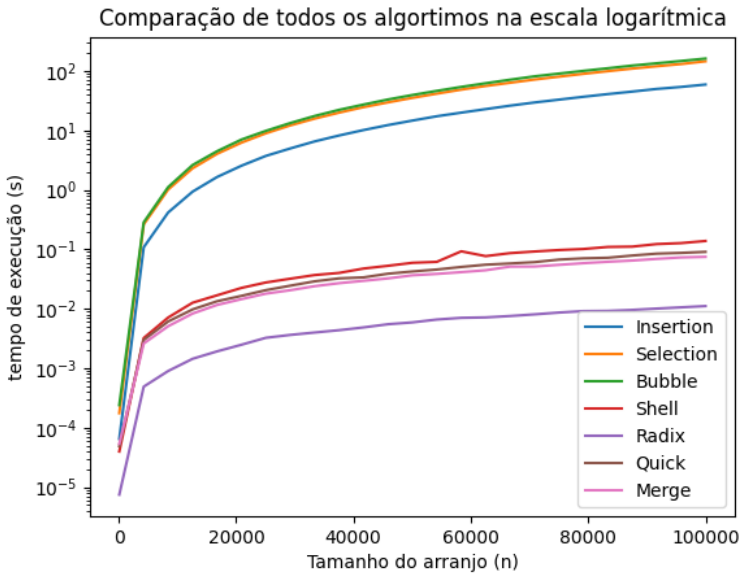
Fonte: Própria.

Gráfico 5 - Comparação dos Algoritmos Merge, Quick, Radix e Shell em Arranjo 75% Aleatório



Fonte: Própria.

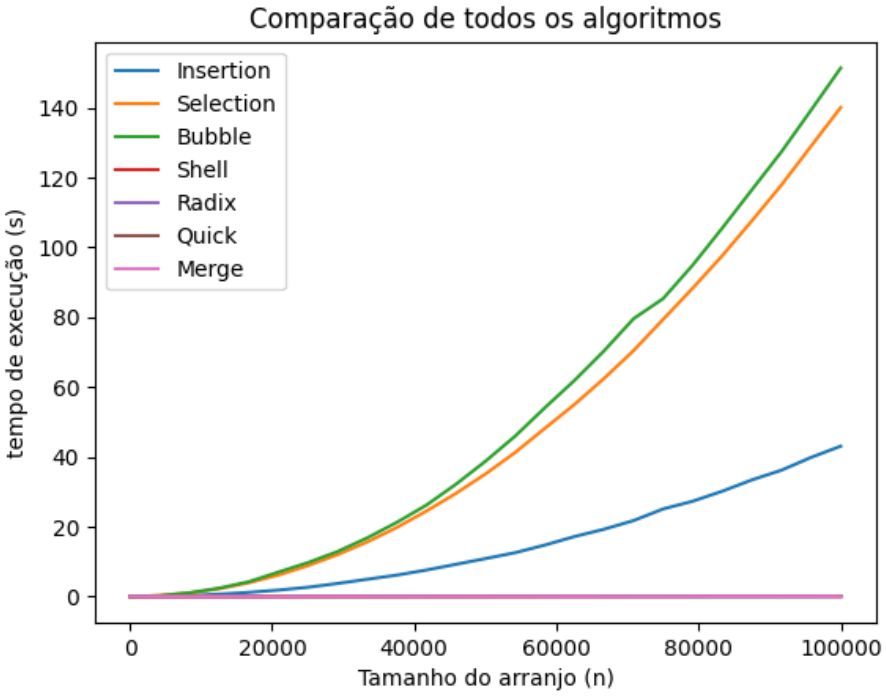
Gráfico 6 - Comparação de Todos os Algoritmos Em Arranjos 75% Aleatórios.



Fonte: Própria.

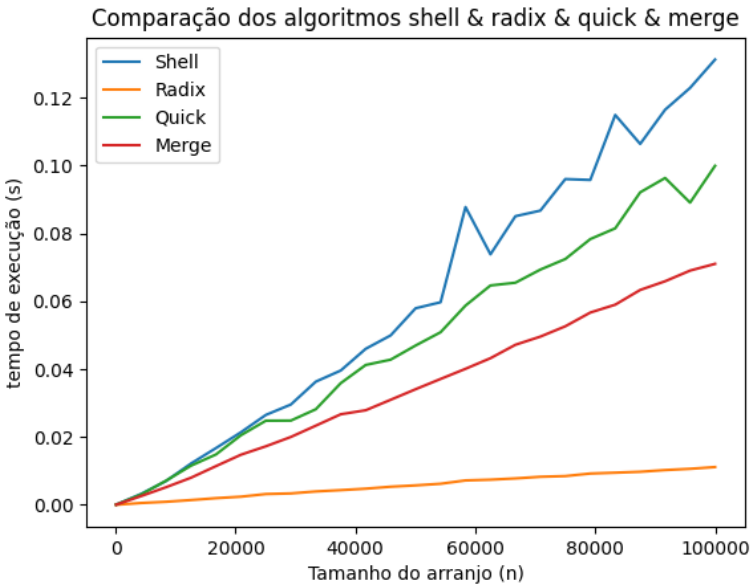
3.1.3 Arranjos 50% aleatórios

Gráfico 7 - Comparação de Todos os Algoritmos em Arranjos 50% Aleatórios.



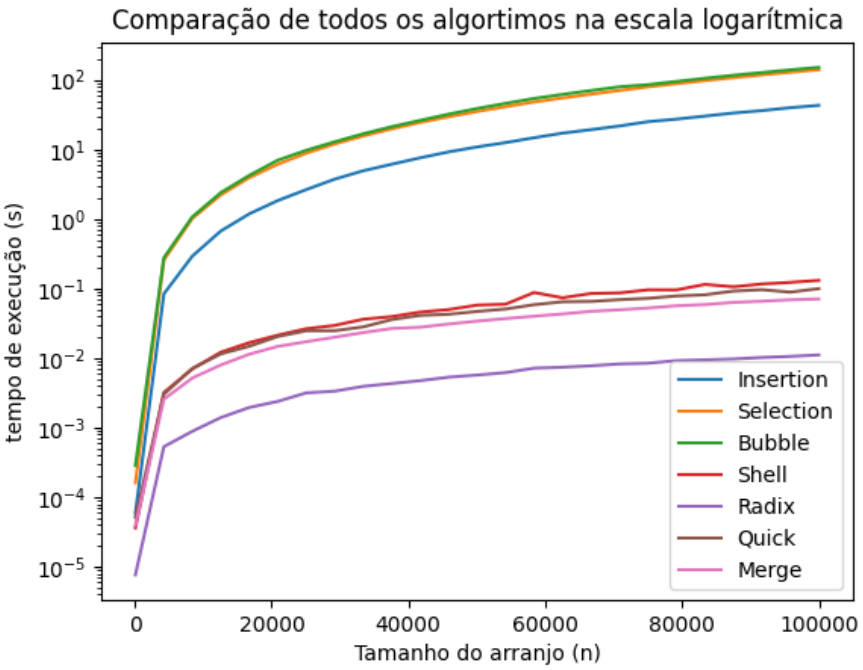
Fonte: Própria.

Gráfico 8 - Comparação dos Algoritmos Merge, Quick, Radix e Shell em Arranjo 50% Aleatório



Fonte: Própria.

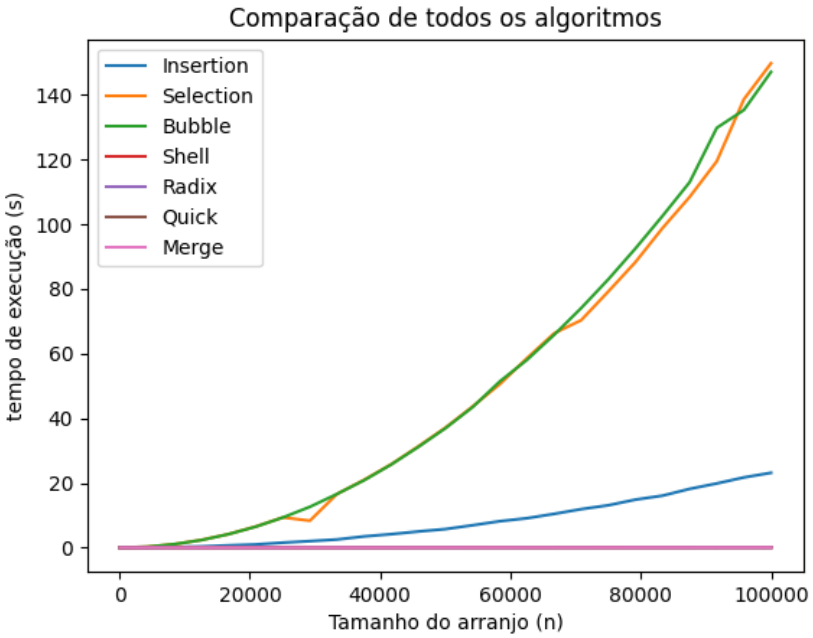
Gráfico 9 - Comparação de Todos os Algoritmos Em Arranjos 50% Aleatórios.



Fonte: Própria.

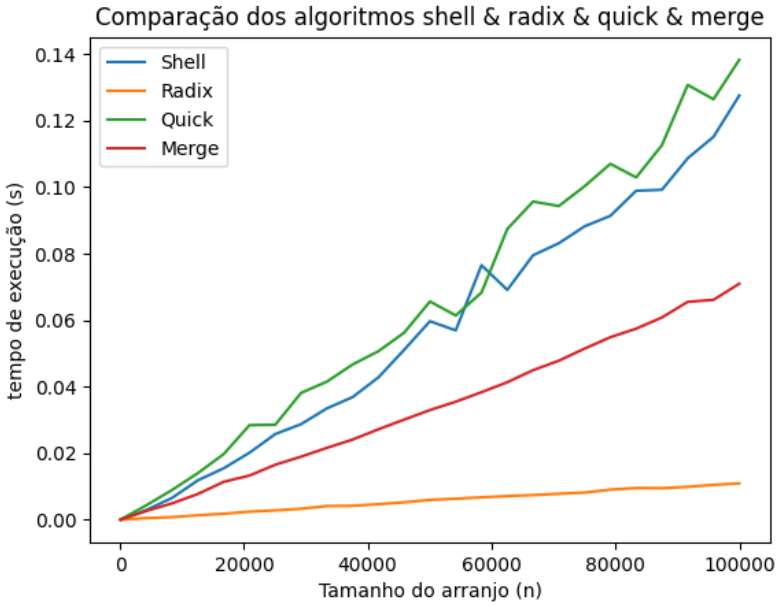
3.1.4 Arranjos 25% aleatórios

Gráfico 10 - Comparação de Todos os Algoritmos em Arranjos 25% Aleatórios.



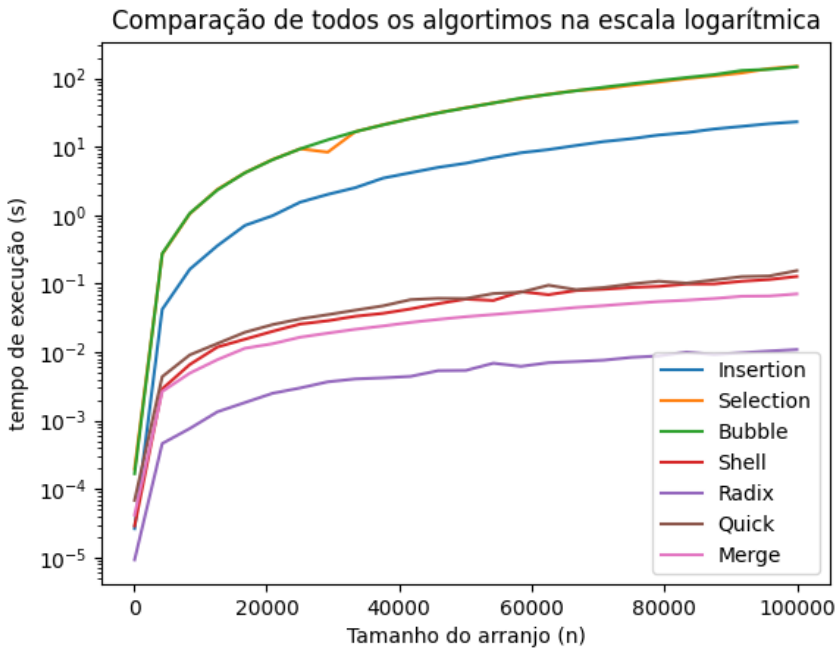
Fonte: Própria.

Gráfico 11 - Comparação dos Algoritmos Merge, Quick, Radix e Shell em Arranjo 25% Aleatório



Fonte: Própria.

Gráfico 12 - Comparação de Todos os Algoritmos Em Arranjos 25% Aleatórios.



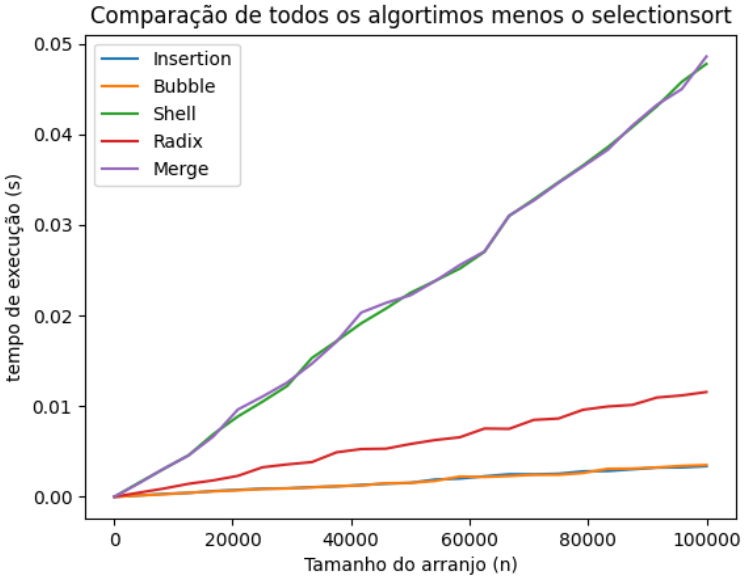
Fonte: Própria.

3.1.5 Arranjos crescentes

Gráfico 13 - Comparação de Todos os Algoritmos em Arranjos Crescentes.

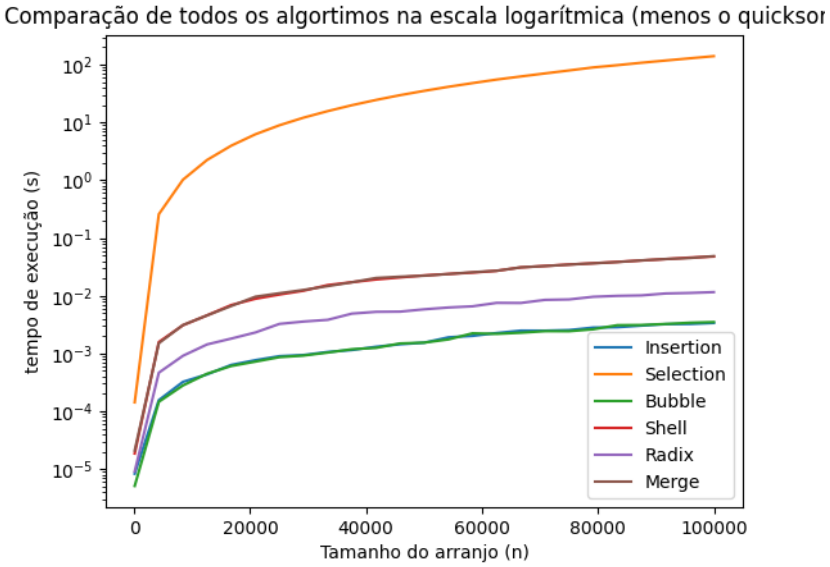
Fonte: Própria.

Gráfico 14 - Comparação dos Algoritmos Merge, Quick, Radix e Shell em Arranjo Crescente



Fonte: Própria.

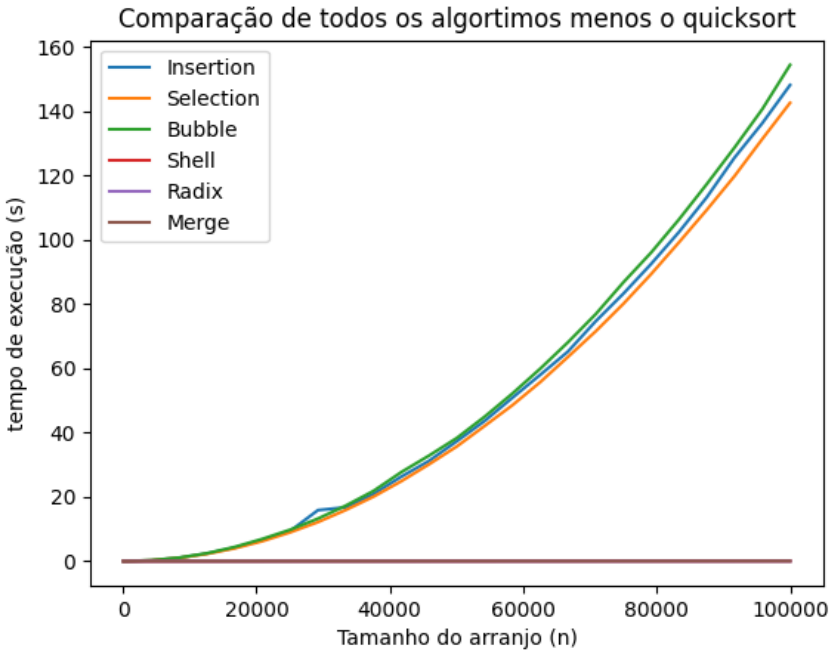
Gráfico 15 - Comparação de Todos os Algoritmos Em Arranjos Crescentes



Fonte: Própria.

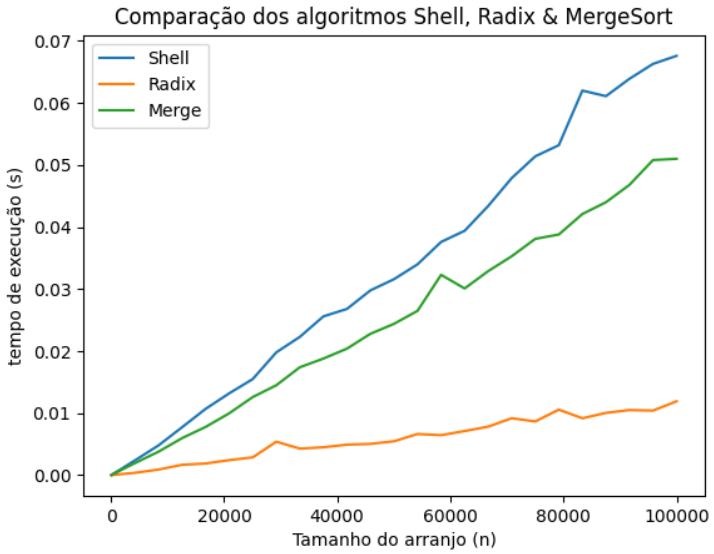
3.1.6 Arranjos decrescentes

Gráfico 16 - Comparação de Todos os Algoritmos em Arranjo Decrescente.



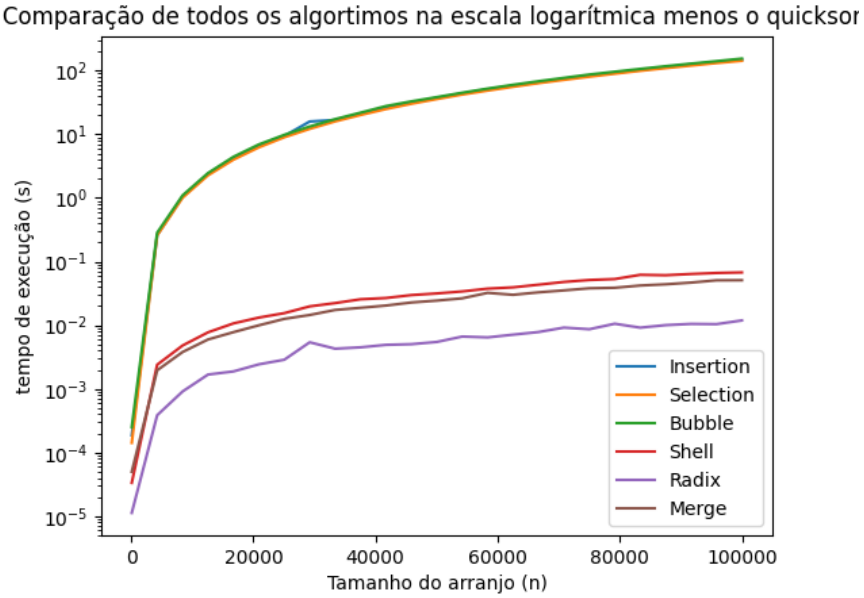
Fonte: Própria.

Gráfico 17 - Comparação dos Algoritmos Merge, Quick, Radix e Shell em Arranjo Decrescente



Fonte: Própria.

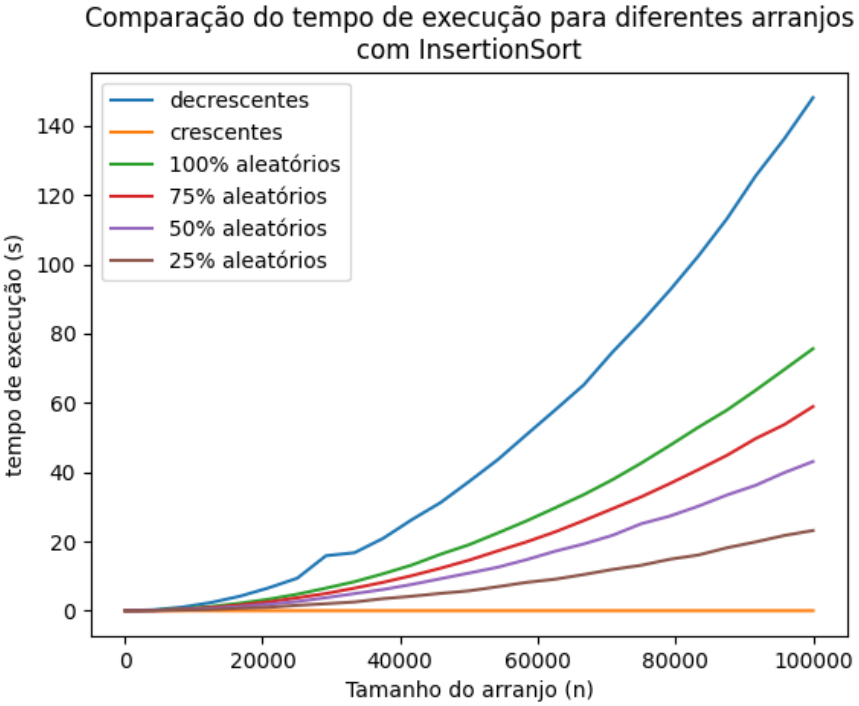
Gráfico 18 - Comparação de Todos os Algoritmos Em Arranjo Decrescente.



Fonte: Própria.

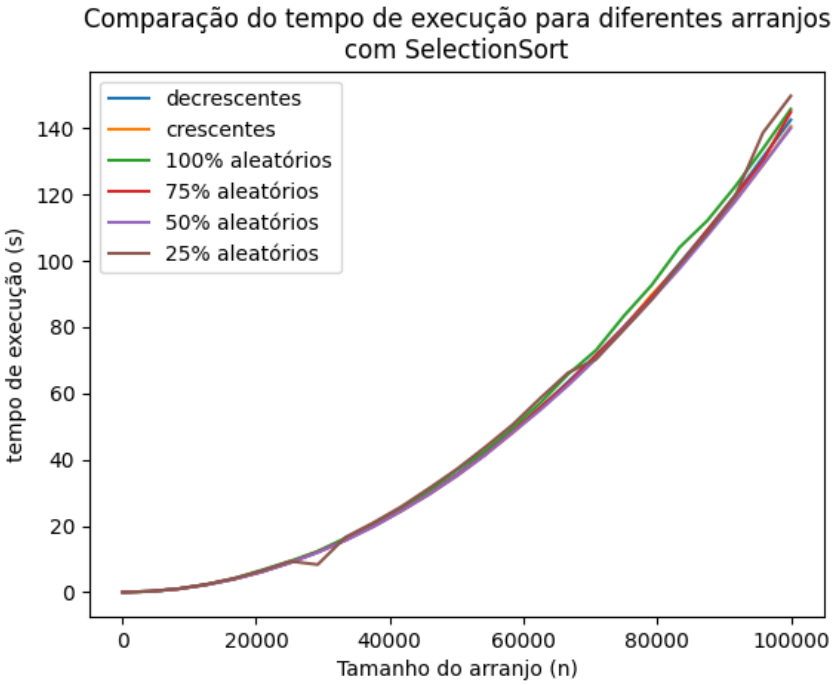
3.1.6 Comparação Progressiva do Tempo

Gráfico 19 - Comparação Progressiva do Tempo do Insertion Sort em Arranjos Diversos



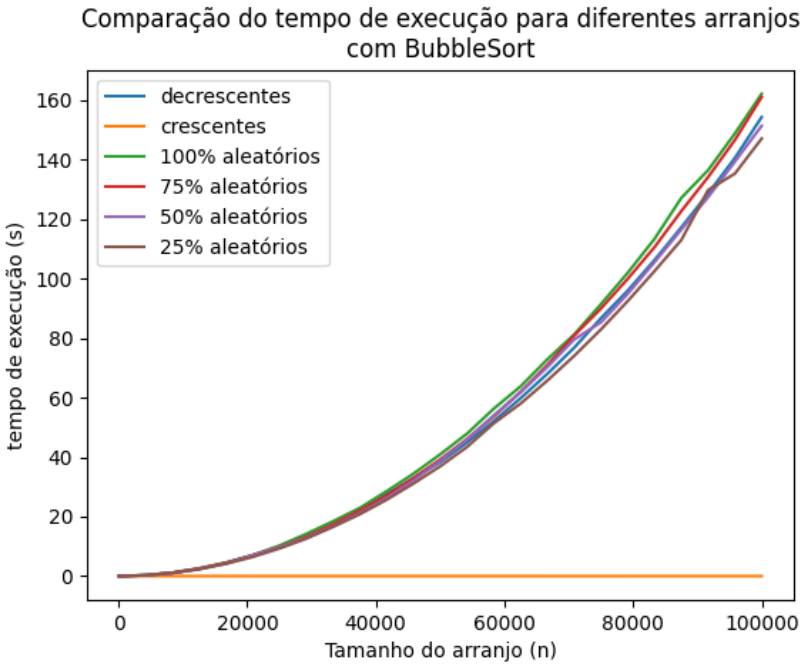
Fonte: Própria.

Gráfico 20 - Comparação Progressiva do Tempo do Selection Sort em Arranjos Diversos



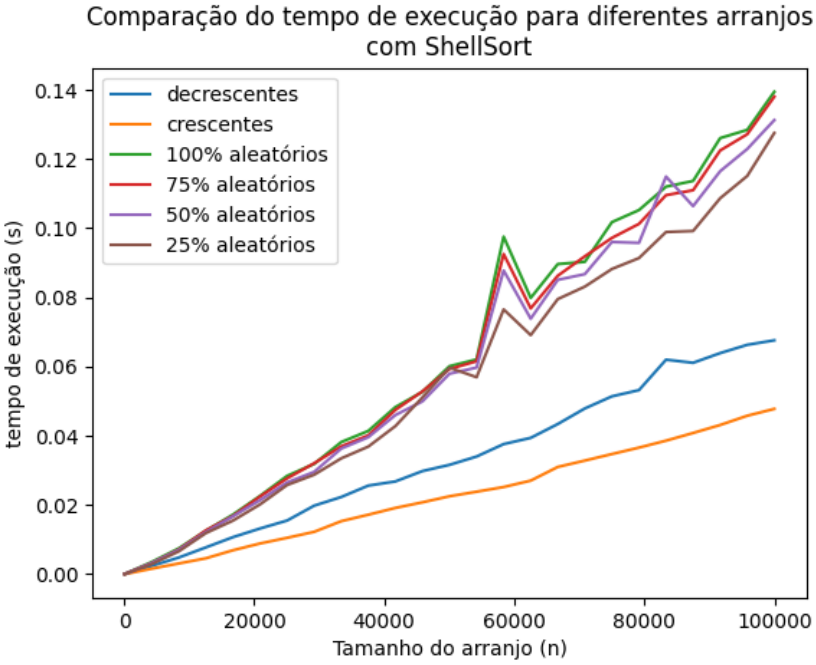
Fonte: Própria.

Gráfico 21 - Comparação Progressiva do Tempo do Bubble Sort em Arranjos Diversos



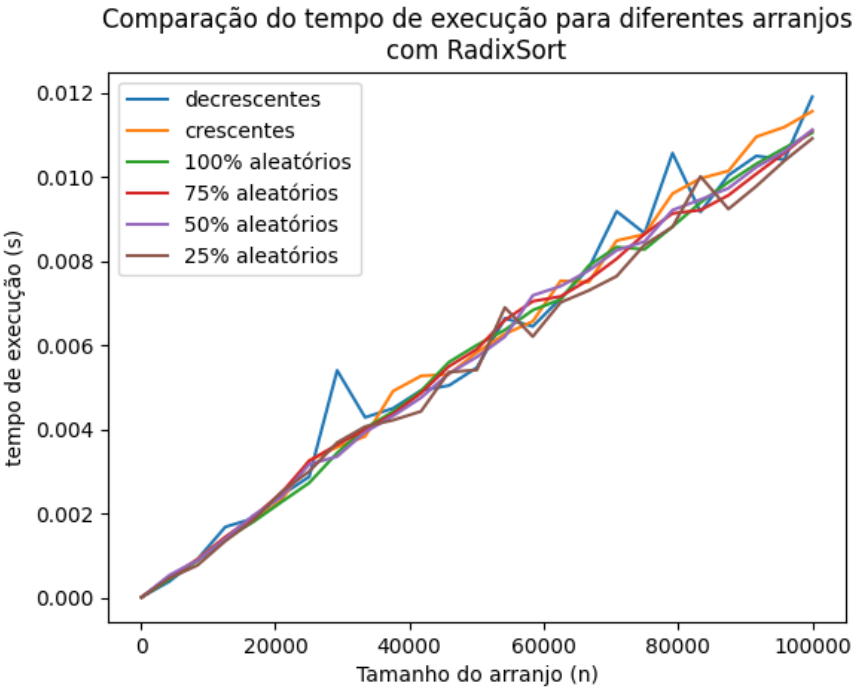
Fonte: Própria.

Gráfico 22 - Comparação Progressiva do Tempo do Shell Sort em Arranjos Diversos



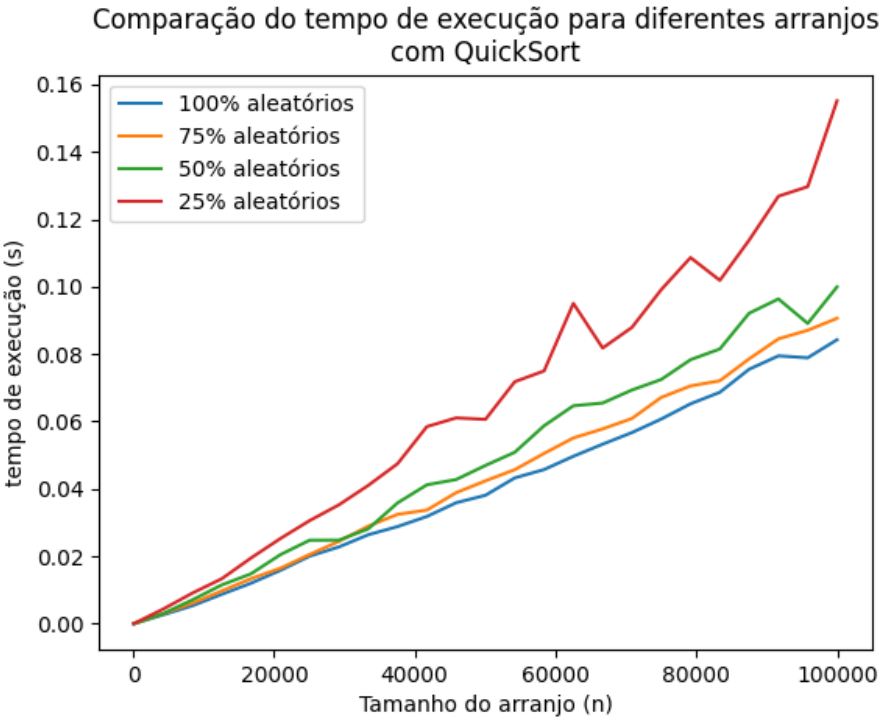
Fonte: Própria.

Gráfico 23 - Comparação Progressiva do Tempo do Radix Sort em Arranjos Diversos



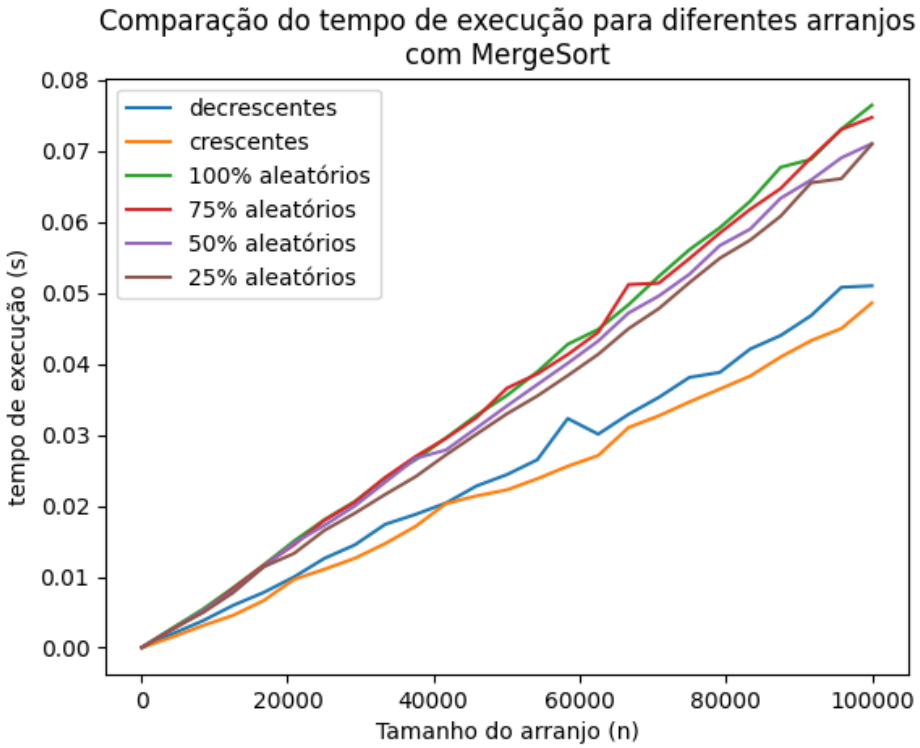
Fonte: Própria.

Gráfico 24 - Comparação Progressiva do Tempo do Quick Sort em Arranjos Diversos



Fonte: Própria.

Gráfico 25 - Comparação Progressiva do Tempo do Merge Sort em Arranjos Diversos



Fonte: Própria.

3.2 Tabelas

3.2.1 Tabelas de Tempo/Tamanho

Tabela 1 - Arranjos 100% Aleatórios

Tamanho da entrada	Insertion	Selection	Bubble	Shell	Radix	Quick	Merge
100	0.00009174	0.0001776	0.0002021	0.0000317	0.00000784	0.0000404	0.0000459
4262	0.140211	0.264671	0.296933	0.00352106	0.00043572	0.00267406	0.002763
8424	0.5361574	1.03295	1.14769	0.00746704	0.0008772	0.00537068	0.00542806
12586	1.211782	2.30702	2.56569	0.01255852	0.0013925	0.00875772	0.00847368
16748	2.159692	4.16195	4.55527	0.01721918	0.00180338	0.012049	0.01166332
20910	3.314192	6.69948	7.08389	0.02261326	0.00227036	0.01585064	0.01504652
25072	4.761754	9.32476	10.2982	0.0284123	0.00273012	0.0200716	0.018052
29234	6.518094	12.3663	14.224	0.03187564	0.0034421	0.02280782	0.02060956
33396	8.375784	16.36845	18.4676	0.03818556	0.0040284	0.02642018	0.02393048
37558	10.67868	20.5053	22.9664	0.0414052	0.00442372	0.02884058	0.0265131
41720	13.19502	25.4293	28.5841	0.0482921	0.00491718	0.03183048	0.0295821
45882	16.29548	30.6357	34.4588	0.05271002	0.00560156	0.03590984	0.03275806
50044	19.05388	36.9168	40.9815	0.0601192	0.00600652	0.03812346	0.03555812
54206	22.43252	42.6022	47.9199	0.06207172	0.00636604	0.04327726	0.03887946
58368	25.91558	49.538	56.3419	0.09756588	0.006838	0.04574512	0.04276388
62530	29.68068	57.1096	63.8213	0.0798579	0.00709394	0.04968762	0.04485718
66692	33.50426	65.7984	72.9242	0.08964436	0.00789396	0.05327988	0.04834648
70854	37.80308	73.005	81.2679	0.09028496	0.00833724	0.05672382	0.05235366
75016	42.5821	83.3395	91.4615	0.10176108	0.0082838	0.06073588	0.05609178
79178	47.68318	92.6565	102	0.1052502	0.00882392	0.06525222	0.05918326
83340	52.95158	104.029	113.402	0.112022	0.00939706	0.06866236	0.06292432
87502	57.91274	112.1435	127.16	0.1136786	0.00988298	0.07550792	0.06770724
91664	63.64608	122.4105	136.49	0.126059	0.01031764	0.07946986	0.06882226
95826	69.60452	133.827	148.817	0.128434	0.01069108	0.07890296	0.07308166
99988	75.6207	145.8025	162.161	0.1394452	0.01105762	0.08425788	0.07646332

Fonte: Própria.

Tabela 2 - Arranjos 75% Aleatórios

Tamanho da entrada	Insertion	Selection	Bubble	Shell	Radix	Quick	Merge
100	0.0000650	0.0001759666667	0.0002403	0.0000399	0.00000750	0.0000489	0.0000536402
4262	0.1083206667	0.2624613333	0.284665	0.0032212	0.00049122	0.00299318	0.002607208
8424	0.415992	1.010403667	1.11068	0.00713	0.0008991	0.00611486	0.005093558
12586	0.9316326667	2.29999	2.60991	0.01264868	0.0014429	0.0097599	0.00831075
16748	1.639906667	4.02193	4.45335	0.01683624	0.00192232	0.0133866	0.01160184
20910	2.541863333	6.212136667	6.99459	0.02243832	0.0024866	0.01648036	0.0145121
25072	3.716826667	8.917096667	9.79929	0.02768898	0.00325312	0.02057582	0.0179676
29234	4.963413333	12.1053	13.2688	0.0320506	0.00363838	0.02446452	0.0205262
33396	6.519083333	15.7546	17.4489	0.03694246	0.00399324	0.02896672	0.0240076
37558	8.19974	19.8561	22.1167	0.0401188	0.00436766	0.03245976	0.0269596
41720	10.11823333	24.5476	27.1439	0.04748314	0.00486528	0.03373448	0.02952134
45882	12.26216667	29.53343333	33.0144	0.05283724	0.00550056	0.03890062	0.0324298
50044	14.60016667	35.1542	39.26	0.0593083	0.00590826	0.04240022	0.03658228
54206	17.26016667	41.39476667	46.0961	0.06154604	0.00660018	0.04571778	0.03859718
58368	19.84946667	48.37273333	53.73	0.0925465	0.00704876	0.05052522	0.04132352
62530	22.75296667	55.7506	61.9489	0.07683024	0.00716524	0.05512638	0.04444492
66692	26.0194	63.12456667	71.03	0.08627094	0.00756244	0.0578073	0.05117992
70854	29.41823333	71.46183333	80.9144	0.09188508	0.00805836	0.0608996	0.05138558
75016	32.84553333	79.88483333	89.9907	0.0972035	0.0086496	0.06715568	0.05483264
79178	36.6804	89.1799	99.9679	0.10126132	0.00913886	0.0705503	0.05843888
83340	40.7015	98.8467	110.617	0.1095676	0.00921762	0.0720874	0.06177724
87502	44.86013333	109.1936667	122.731	0.1109832	0.00956968	0.07858882	0.06470892
91664	49.69383333	119.5443333	134.158	0.122479	0.01008068	0.08451884	0.069036
95826	53.71323333	130.2333333	146.612	0.1271768	0.01057612	0.08706248	0.07304468
99988	58.9175	144.9403333	161.062	0.1380296	0.01111282	0.09060526	0.07472176

Fonte: Própria.

Tabela 3 - Arranjos 50% Aleatórios

Tamanho da entrada	Insertion	Selection	Bubble	Shell	Radix	Quick	Merge
100	0.000059302	0.000161802	0.000287	0.0000363.	0.00000768	0.0000517	0.0000396
4262	0.0841454	0.257764	0.278343	0.003210875	0.0005347	0.0030425	0.0025716
8424	0.29312	1.01924	1.07642	0.007022225	0.00088958	0.00711062	0.0051889
12586	0.673241	2.22425	2.39769	0.012181625	0.00140256	0.01153514	0.0079906
16748	1.18743	3.93227	4.24421	0.0167241	0.00195648	0.0148462	0.011407
20910	1.84387	6.13588	7.01016	0.021392325	0.00239292	0.02051198	0.0147655
25072	2.64943	8.83978	9.72186	0.02650595	0.00317234	0.02479962	0.0172539
29234	3.74403	11.9994	12.8962	0.0295644	0.00335672	0.0248244	0.0200003
33396	4.94062	15.6228	16.7991	0.036309325	0.00393644	0.02816936	0.0233328
37558	6.13083	19.7653	21.2395	0.03959295	0.00432172	0.03586448	0.0266933
41720	7.58639	24.4713	26.1873	0.045990675	0.00475398	0.0412269	0.0278763
45882	9.22793	29.5477	32.1787	0.049948475	0.00533072	0.04279156	0.0309635
50044	10.8738	35.1275	38.7876	0.057941525	0.00572464	0.046966	0.0340773
54206	12.5561	41.268	45.9438	0.059721025	0.00620524	0.0508725	0.0371
58368	14.7338	48.1548	54.0532	0.087783	0.00718808	0.05873974	0.0400953
62530	17.1643	54.9984	61.8716	0.073847575	0.0074087	0.06466428	0.0432237
66692	19.2659	62.4468	70.3361	0.08508025	0.00777994	0.06547908	0.0471792
70854	21.7506	70.4369	79.5784	0.086736775	0.00826002	0.0693456	0.0495851
75016	25.0857	79.3906	85.341	0.096030425	0.0084698	0.07246448	0.0526289
79178	27.2964	88.3348	94.9366	0.095773975	0.0092187	0.07833392	0.0566753
83340	30.1725	97.6396	105.552	0.1149255	0.00946762	0.08150952	0.0589983
87502	33.4068	107.631	116.519	0.1063925	0.00973786	0.09212094	0.0633416
91664	36.1954	117.905	127.408	0.116488	0.01024134	0.09635314	0.065911
95826	39.8514	129.006	139.34	0.122913	0.01061124	0.08909426	0.0690508
99988	43.0611	140.078	151.386	0.131261	0.01112492	0.09993842	0.0710374

Fonte: Própria.

Tabela 4 - Arranjos 25% Aleatórios

Tamanho da entrada	Insertion	Selection	Bubble	Shell	Radix	Quick	Merge
100	0.0000265	0.000198	0.0001685	0.0000290	0.00000768	0.0000872	0.0000420
4262	0.0421771	0.26949775	0.275128	0.00291174	0.0005347	0.00437076	0.00266188
8424	0.162635	1.04867075	1.06771	0.00660354	0.00088958	0.0089828	0.00494142

12586	0.358643	2.386055	2.33598	0.01187952	0.00140256	0.01405188	0.00777072
16748	0.709941	4.1860275	4.16338	0.01548168	0.00195648	0.0197658	0.01141106
20910	0.98453	6.526725	6.46306	0.0201624	0.00239292	0.02845226	0.0132939
25072	1.55021	9.337505	9.31387	0.02576982	0.00317234	0.02856886	0.01654606
29234	2.01529	8.3354	12.6278	0.02874394	0.00335672	0.03813758	0.01896416
33396	2.52597	16.5987	16.5958	0.03347618	0.00393644	0.04151698	0.02161398
37558	3.47865	20.9846	20.8411	0.03687514	0.00432172	0.04668584	0.02411962
41720	4.17422	25.803175	25.7075	0.04280232	0.00475398	0.05066962	0.02719712
45882	5.00019	31.35635	31.1509	0.05109698	0.00533072	0.05625316	0.03010502
50044	5.74318	37.138825	36.9457	0.0596863	0.00572464	0.0656199	0.03297226
54206	6.92751	43.670675	43.451	0.05693316	0.00620524	0.06140916	0.035487
58368	8.17548	50.5227	51.4	0.0765282	0.00718808	0.06830882	0.0383574
62530	9.09651	58.57675	58.0384	0.06910186	0.0074087	0.08740146	0.0413514
66692	10.4388	66.256425	65.7745	0.07951498	0.00777994	0.09565704	0.04496628
70854	11.9037	70.324975	74.0674	0.08312276	0.00826002	0.09431884	0.04780986
75016	13.1031	79.19965	82.9596	0.08821986	0.0084698	0.10027836	0.05144858
79178	14.8471	88.28495	92.4958	0.09137312	0.0092187	0.10699572	0.0548883
83340	16.0768	98.817275	102.571	0.0989116	0.00946762	0.10292304	0.05745208
87502	18.1702	108.44475	113.007	0.09920364	0.00973786	0.1125996	0.06081426
91664	19.8498	119.477	129.801	0.108672	0.01024134	0.1307246	0.06551612
95826	21.7152	138.658	135.322	0.1151144	0.01061124	0.1264212	0.0661079
99988	23.143	149.77875	147.107	0.1275358	0.01112492	0.1382678	0.07095828

Fonte: Própria.

4 Discussão

4.1 Discussão Geral

Os algoritmos não-quadráticos (algoritmos 5, 6 e 7) são indicados para a maioria dos casos testado, talvez podendo ser utilizados para ordenar bancos de dados de grandes empresas. Todavia, dependendo da forma como os arranjos passados são ordenados, a resposta pode variar bastante. Podemos inferir que não há um algoritmo absoluto, sendo necessário um diálogo entre desenvolvedor e cliente para a escolha e implementação do algoritmo que mais satisfaça às necessidades e desafios a serem enfrentados, otimizando o trabalho e entregando um produto de maior qualidade.

É imperioso saber desenvolver algoritmos eficientes, aplicando a maior parte da energia criativa na busca de soluções eficientes, aplicando a maior parte da energia criativa na busca de soluções inteligentes para contextos específicos. Ao pensarmos na ordenação de grande quantidade de dados, algoritmos de complexidade $O(n^2)$ são ineficientes, para não se dizer *nocivos*, em questão de tempo de execução, gastando tempo em excesso quando comparado aos algoritmos não-quadráticos (vide gráfico 1), o que pode acarretar em atrasos em projetos, gasto de energia e desperdício de vida útil dos computadores. Isso tanto em escala menor, com pequenos projetos, ou mesmo em grande porte, quando pensamos no contexto de empresas. Assim, podemos inferir que a aplicação de algoritmos em contextos errados pode acarretar em considerável prejuízo.

4.2 Como o algoritmo de decomposição de chave (radix) se compara com o melhor algoritmo baseado em comparação de chaves?

O algoritmo de ordenação de decomposição de chave, também conhecido como radix sort, difere dos algoritmos de ordenação por comparação, como o quick sort ou merges ort, em sua abordagem para ordenar os elementos. Enquanto os algoritmos de comparação comparam pares de elementos para determinar sua ordem relativa, o radix sort classifica os elementos com base em dígitos individuais ou grupos de dígitos.

Com efeito, a principal diferença entre o radix sort e os algoritmos de comparação é que o radix sort não compara diretamente os elementos entre si para determinar sua ordem. Ele se baseia na análise dos dígitos ou grupos de dígitos dos elementos. Portanto, sua complexidade não depende do número total de elementos na entrada, mas sim do número de dígitos ou grupos de dígitos. Isso significa que o radix sort pode ser mais eficiente em certos casos em que o número de dígitos é menor em comparação com o número total de elementos. Que foi o ambiente de teste encontrado pelo radix sort nos experimentos desta pesquisa, logo o radix sort demonstrou uma velocidade significativa para ordenar os arranjos, podemos notar isso no gráfico 3.

4.3 Quais algoritmos são recomendados para quais cenários

Algoritmos com complexidade quadrática não são recomendados para a grande maioria dos cenários que foram testados. Apenas se o arranjo já estiver totalmente ordenado de forma crescente, o Bubble Sort e o Insertion Sort poderiam ser recomendados, em qualquer outro cenário eles perdem facilmente para outros algoritmos, tirando o Selection Sort.

O Selection Sort foi particularmente péssimo em todos os testes feitos, até mesmo com o arranjo totalmente ordenado ele não seria recomendado.

Considerando os outros cenários: arranjo totalmente decrescente, 100% aleatório, 75% aleatório, 50% aleatório e 25% aleatório. Os algoritmos: Shell Sort, Radix Sort e Merge Sort são esmagadoramente melhores que os outros. O Quick Sort também se sai bem em todos esses cenários, tirando o com o arranjo totalmente decrescente, onde, juntamente com o totalmente crescente, não pôde ser devidamente testado. Se comparados entre si, vale citar as seguintes observações:

1. Se o arranjo for bem menor, por exemplo, a escolha de um sobre o outro pode variar, porém com uma diferença de tempo minúscula. A diferença é mais evidente conforme o arranjo fica maior.
2. O Shell Sort é um pouco mais lento em relação aos outros na maioria dos cenários (mas pouco mesmo);
3. O Radix Sort possui uma enorme consistência em seu tempo de execução, sendo muito similar independente da entrada. Além disso, se destacou na

maioria dos testes, perdendo apenas em casos pontuais e por diferenças minúsculas;

4. O Merge Sort atua um pouco melhor em arranjos que já estão parcialmente ordenados, porém, novamente, por uma diferença minúscula.
5. O Quick Sort funciona um pouco melhor em arranjos menos ordenados, entretanto isso pode variar conforme a implementação e o tratamento do pior caso.

De maneira geral, a recomendação varia um pouco (ou muito) de acordo com o cenário, uma escolha razoável para quando não se sabe da organização do arranjo que se quer ordenar seria o Radix Sort, que conseguiu manter a maior consistência em diferentes cenários e com tempo de execução rápido.

4.4 Aconteceu algo inesperado nas medições? (por exemplo, picos ou vales nos gráficos) Se sim, por que?

Alguns picos e vales leves foram sim observados com a ajuda dos gráficos, porém nada muito dramático. Já era esperado que mesmo com dados retirados a partir de 5 rodadas de testes e sendo realizada a média aritmética entre eles, os gráficos gerados seriam perfeitamente iguais ao que se esperava com apenas a análise matemática. Além disso, fatores externos relacionados ao computador também podem ter influenciado nesses resultados, como o aquecimento e uso de recursos no momento que foram feitos os testes, apesar da tentativa de reservar ao máximo todo o computador unicamente para essa tarefa.

4.5 Análise Empírica vs Análise Matemática

De maneira geral, a análise empírica e matemática convergem. Todavia, como citado em tópicos anteriores, houve alguns picos inesperados, quando pensamos apenas no cálculo matemático da complexidade, nos gráficos (vide gráfico XX). É possível pensar na hipótese do computador ter sofrido um superaquecimento, pois os picos concentram-se na parte final dos gráficos. Com efeito, a aparição dessas anomalias nos gráficos reforça que não é possível se basear apenas na análise matemática para a realização de uma análise aprofundada dos algoritmos.

Apesar disso, é possível ver claramente a relação do que era esperado na análise matemática e o que foi obtido na análise empírica, tirando os pequenos picos e vales, os gráficos possuem uma enorme semelhança com o que já era esperado anteriormente.

4.6 Discussão em Foco: Quick Sort ou Merge Sort?

O merge sort foi mais eficiente que o quick sort em todos os casos em que ambos puderam ser comparados. Foi evidente também que quanto mais elementos já ordenados no arranjo, maior a diferença no tempo de execução dos dois algoritmos, favorecendo o Merge Sort nesses casos. Isso já era esperado, visto que quanto mais elementos já ordenados no arranjo, maior a chance do Quick Sort escolher pivôs extremos (maior ou menor elemento do subarranjo), o que aumenta o número de chamadas recursivas necessárias para ordenar o arranjo e consequentemente deixa o algoritmo mais lento.

Porém, vale lembrar que apesar de mais rápido que o Quick Sort, o Merge sort precisa copiar parte do arranjo, o que ocupa mais memória. Portanto, não é possível afirmar que o Merge Sort vai ser sempre uma escolha melhor em relação ao Quick Sort, uma vez que, se o fator memória for importante também, a escolha pode variar.

4.7 Estimativas de tempo de execução para um arranjo de tamanho 10^{12}

Tentamos utilizar alguns métodos diferentes, mas nada pareceu muito confiável para dar uma resposta precisa. Porém, uma estimativa não muito confiável para os algoritmos bubble, selection e insertion sem contar seus melhores casos, seria algo em torno de centenas de milênios. Já, para os algoritmos Radix, MergeSort, Quick e Shell, a estimativa gira em torno de 21 dias.

Essas estimativas foram feitas utilizando a biblioteca scipy e numpy no Python, ajustando a função para o que era esperado dos algoritmos num arranjo 100% aleatório.

5 Referências

BACKES, A. R. *Algoritmos e Estruturas de Dados em Linguagem C*. 1. ed. Rio de Janeiro: LTC, 2023.

SLIDES DA DISCIPLINA ESTRUTURA DE DADOS BÁSICAS I: *Algoritmos de Ordenação*. UFRN/IMD: 2023.

SZWARCFITER, J. L.; MARKENZON, L.. *Estruturas de Dados e seus Algoritmos*. Rio de Janeiro: LTC, 1994.

6 Apêndice

6.1 Algoritmos de Ordenação em C++

```
// INSERTION SORT
template <typename DataType, typename Compare>
void insertion(DataType *first, DataType *last, Compare cmp){
  int i, j;
  int tamanho = last - first;
  DataType aux;

  for(i = 0; i < tamanho; i++){
    aux = first[i];
    j = i - 1;

    while( (j >= 0) && ( cmp(aux,first[j]) )){
      first[j + 1] = first[j];
      j--;
    }
    first[j + 1] = aux;
  }
}

// SELECTION SORT
template <typename DataType, typename Compare>
void selection(DataType *first, DataType *last, Compare cmp){
  last = last - 1;
  DataType *i, *j, *min;

    for(i = first; i < last; i++){
        min=i;
```

```
        for(j=i+1; j<=last; j++){
            if(cmp(*j, *min)){
                min = j;
            }
        }
        if(min != i){
            DataType aux = *i;
            *i = *min;
            *min = aux;
        }
    }
}


// BUBBLE SORT
template <typename DataType, typename Compare>
void bubble(DataType *first, DataType *last, Compare cmp){
  int indice = (last - first) - 1;
  int continua = 1;
  DataType aux;

    while(continua){
        continua = 0;

        for(int i=0; i<indice; i++){

            if(cmp(first[i+1], first[i])){
                aux = first[i];
                first[i] = first[i+1];
                first[i+1] = aux;
                continua = 1;
            }
        }
        indice--;
```

```
    }
}

//SHELL SORT
template <typename DataType, typename Compare>
void shell(DataType *first, DataType *last, Compare cmp){
  int tamanho = last - first;
  int gap = tamanho / 2;

    while (gap > 0) {

        for (int i = gap; i < tamanho; i++) {
            int j = i;
            DataType aux = first[i];

            while (j >= gap && cmp(aux, first[j - gap])) {
                first[j] = first[j - gap];
                j -= gap;
            }

            first[j] = aux;
        }

        gap /= 2;
    }
}

// MERGE (para o mergesort)
template <typename DataType, typename Compare>
void merge(DataType *first, DataType *middle, DataType *last, Compare cmp) {
    int tamanho = last - first;
    DataType *aux = new DataType[tamanho];
    int i = 0, j = 0, k = 0;
```

```
    while (i < (middle - first) && j < (last - middle)) {
        if (cmp(*(first + i), *(middle + j))) {
            *(aux + k++) = *(first + i++);
        } else *(aux + k++) = *(middle + j++);
    }

    while (i < middle - first) {
        *(aux + k++) = *(first + i++);
    }

    while (j < last - middle) {
        *(aux + k++) = *(middle + j++);
    }

    for (i = 0; i < tamanho; ++i) {
        *(first + i) = *(aux + i);
    }

    delete[] aux;
}
// MERGE SORT
template <typename DataType, typename Compare>
void merge_sort(DataType *first, DataType *last, Compare cmp){
    int tamanho = last - first;
    if (tamanho > 1) {
        int *middle = first + tamanho / 2;
        merge_sort(first, middle, cmp);
        merge_sort(middle, last, cmp);
        merge(first, middle, last, cmp);
    }
}

// PARTITION (para o quicksort)
template <typename DataType, typename Compare>
```

```
int* partition(DataType *first, DataType *last, DataType *pivot, Compare comp){

    DataType *pivot_d = escolhe_pivot(first, last);
    DataType *slow = first;
    DataType *fast = first;

    while(fast != pivot_d){

        if(comp(*fast, *pivot_d)){
            std::iter_swap(slow, fast);
            slow++;
        }
        fast++;
    }

    std::iter_swap(slow, pivot_d);
    return slow;
}

// QUICK SORT
template <typename DataType, typename Compare>
void quick(DataType *first, DataType *last, Compare comp){
    int tamanho = last - first;
    if (tamanho > 1) {
        DataType* pivot = partition<DataType>(first, last, last - 1, comp);
        quick(first, pivot, comp);
        quick(pivot + 1, last, comp);
    }
}



// RADIX SORT
/*!
 * This function implements the Radix Sorting Algorithm based on the **less
```

```
 * significant digit** (LSD).
 *
 * @note There is no need for a comparison function to be passed as argument.
 *
 * @param first Pointer/iterator to the beginning of the range we wish to sort.
 * @param last Pointer/iterator to the location just past the last valid value
 * of the range we wish to sort.
 * @tparam FwrdIt A forward iterator to the range we need to sort.
 * @tparam Comparator A Comparator type function tha returns true if first
 * argument is less than the second argument.
 */
template <typename DataType, typename Comparator>
void radix(DataType *first, DataType *last, Comparator cmp){
   int tamanho = last - first;

   DataType max = first[0];
   for (int i = 1; i < tamanho; i++) {
      if (cmp(max, first[i])) max = first[i];
   }

   for (int exp = 1; max / exp > 0; exp *= 10){

      const int RANGE = 10;
      DataType output[tamanho];
      DataType count[RANGE] = {0};

   for (int i = 0; i < tamanho; i++)
      count[(first[i] / exp) % 10]++;

   for (int i = 1; i < RANGE; i++)
      count[i] += count[i - 1];

   for (int i = tamanho - 1; i >= 0; i--) {
      output[count[(first[i] / exp) % 10] - 1] = first[i];
```

```
        count[(first[i] / exp) % 10]--;
    }

    for (int i = 0; i < tamanho; i++)
        first[i] = output[i];
    }
}
```