Algoritmos de Ordenação de Dados Aplicados na Linguagem C: Comparação e Análise

Luis Henrique da Silva Resende[1]

[1] Departamento de Ciência da Computação
Universidade Federal Tecnológica do Paraná (UTFPR)
Santa Helena, Paraná - Brasil

luisresende@alunos.utfpr.edu.br

Abstract. *This article addresses the comparative analysis of various sorting algorithms implemented in the C programming language, aiming to identify the most effective method in terms of performance and efficiency. In this study, eight widely used sorting algorithms were selected: Bubble Sort, Selection Sort, Insertion Sort, Shell Sort, Quick Sort, Merge Sort, Heap Sort and Radix Sort. Each of these algorithms was implemented in the C language and tested on the same datasets with five different sizes and three types of order (ascending, descending, and random).*

Resumo. *Este artigo aborda a análise comparativa de diversos algoritmos de ordenação implementados na linguagem C, com o objetivo de identificar o método mais eficaz em termos de desempenho e eficiência. Neste estudo, foram selecionados 8 algoritmos amplamente utilizados para ordenação de dados: Bubble Sort, Selection Sort, Insertion Sort, Shell Sort, Quick Sort, Merge Sort, Heap Sort e Radix Sort. Cada um desses algoritmos foi implementado em linguagem C e testado nas mesmas bases de dados, com 5 tamanhos de dados e 3 tipos de ordenação (ascendente, descendente e aleatória).*

1. Introdução

O objetivo primordial deste projeto é conduzir uma análise abrangente de diversos algoritmos de ordenação de dados, com a utilização de diferentes tamanhos e ordens nas listas de dados. Esta pesquisa visa aprofundar a compreensão dos algoritmos de ordenação e suas eficácias em uma variedade de cenários. A ordenação de dados é uma tarefa fundamental em ciência da computação, com implicações críticas em várias aplicações, desde otimização de bancos de dados até aprimoramento de algoritmos de busca. Através deste estudo, buscamos identificar quais algoritmos se destacam em termos de desempenho, considerando uma ampla gama de fatores, incluindo o tamanho dos conjuntos de dados e os diferentes tipos de ordenação.

1.1. Especificações da máquina utilizada

Para garantir a igualdade de condições entre os métodos, foi utilizado a mesma configuração de máquina para a execução dos algoritmos. A máquina possui as seguintes especificações: placa mãe Asus TUF Gaming B460M-Plus, processador Intel(R) Core (TM) i510400-F, 16GB Memória RAM DDR4 2666Mhz, SSD M2 Nvme 1TB com leitura a 2400 Mb/s e escrita a 1700 Mb/s e placa de vídeo GTX 1650 Super PNY NVIDIA.

1.2 Medição do tempo de execução dos algoritmos

A medição do tempo de execução é uma parte fundamental na avaliação do desempenho dos métodos. O cálculo desse tempo é realizado por meio da função clock() da biblioteca nativa da linguagem C, chamada "time".

```
clock_t start_time = clock();

//Função a ser executada

clock_t end_time = clock();

double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;
```

Figura 1. Código de obtenção do tempo de execução

A figura 1 apresenta o código que ilustra como é feita essa medição do tempo de execução. A abordagem envolve o uso de duas variáveis cruciais: start_time e end_time. A primeira é obtida no momento de início da execução do algoritmo referente ao método utilizado, enquanto a segunda é registrada ao final da execução.

Para calcular o tempo decorrido, subtrai-se o valor de start_time de end_time. O resultado é então dividido pela constante CLOCKS_PER_SEC, a qual representa a quantidade de "tics" de relógio que ocorrem em um segundo no sistema.

2. Métodos de ordenação

Nesta sessão, abordaremos sobre os métodos de ordenação, seus códigos, funcionamentos e resultados medidos em segundos.

2.1. Bubble Sort

O algoritmo conhecido como Bubble Sort realiza várias iterações em uma lista de elementos para ordená-la. Durante cada iteração, o algoritmo compara pares de elementos adjacentes e troca aqueles que estão fora de ordem. Gradualmente, à medida que as iterações continuam, os maiores elementos "flutuam" para suas posições finais, assim como bolhas que sobem à superfície da água. Esse processo se repete até que a lista inteira esteja ordenada. Em resumo, o Bubble Sort é um método de classificação no qual os elementos se movem gradualmente para suas posições corretas.

```
int aux;
for(int x = count-1; x >= 1; x--){
    for(int y = 0; y <= x-1; y++){
        if(numbers[y] > numbers[y+1]){
            aux = numbers[y];
            numbers[y] = numbers[y+1];
            numbers[y+1] = aux;
        }
    }
}
```

Figura 2. Código Bubble Sort

Bubble Sort					
	100000	200000	300000	500000	1000000
Ordenado	9.36	36.979	83.297	231.142	926.012
Invertido	15.852	63.211	142.018	395.812	1576.31
Aleatório	8.935	98.504	221.719	615.574	2473.32

Figura 3. Resultados Bubble Sort

Figura 4. Gráfico Bubble Sort

2.2. Insertion Sort

O algoritmo de ordenação por inserção é uma técnica que organiza uma lista, passo a passo, comparando cada elemento da lista com seu elemento adjacente.

Este algoritmo utiliza um índice que aponta para o elemento atual, indicando assim a posição atual na ordenação. No início do processo de ordenação, quando o índice está definido como zero, o valor atual é comparado com o valor imediatamente à esquerda. Se o valor atual for maior ou igual ao valor à esquerda, nenhuma modificação é feita na lista. O mesmo acontece se o valor atual for igual ao valor à esquerda.

No entanto, se o valor à esquerda do valor atual for menor, o algoritmo move a posição do valor à esquerda para a direita, continuando a mover até que encontre um valor menor à esquerda ou alcance a extremidade da lista ordenada.

```
int aux, atual, index;
for(int i = 1; i < count; i++){
    atual = numbers[i];
    index = i-1;

    while(index >= 0 && atual < numbers[index]){
        numbers[index + 1] = numbers[index];
        index--;
    }
    numbers[index+1] = atual;
}
```

Figura 5. Código Insertion Sort

Insertion Sort					
	100000	200000	300000	500000	1000000
Ordenado	1E-07	0.001	0.001	0.001	0.003
Invertido	11.179	44.414	99.932	277.713	1109.58
Aleatório	5.543	22.29	50.104	138.33	554.828

Figura 6. Resultados Insertion Sort

Figura 7. Gráfico Insertion Sort

2.3. Selection Sort

O Selection Sort é um algoritmo de ordenação que opera selecionando repetidamente o menor (ou maior, dependendo da ordem requerida) elemento da lista e o movendo para a primeira posição não ordenada. Isso é feito através de comparações sequenciais e trocas de elementos. O processo continua até que todos os elementos estejam em suas posições corretas, resultando em uma lista ordenada.

```
int i, j, menor, troca;
for (i = 0; i < count; i++){
    menor = i;
    for (j = i+1; j < count; j++){
        if(numbers[j] < numbers[menor]){
            menor = j;
        }
    }
    troca = numbers[i];
    numbers[i] = numbers[menor];
    numbers[menor] = troca;
}
```

Figura 8. Código Selection Sort

Selection Sort					
	100000	200000	300000	500000	1000000
Ordenado	10.001	39.871	89.164	247.677	990.52
Invertido	9.511	38.032	85.183	236.443	946.19
Aleatório	9.971	39.821	89.321	247.726	991.186

Figura 9. Resultados Selection Sort

Figura 10. Gráfico Selection Sort

2.4. Shell Sort

O Shell Sort, também conhecido como "ordenação por incrementos diminutos", otimiza o processo de ordenação por inserção dividindo a lista em subconjuntos menores. Esses subconjuntos são então ordenados usando a ordenação por inserção. A característica distintiva do Shell Sort é a forma como esses subconjuntos são escolhidos. Em vez de criar subconjuntos com elementos adjacentes, o Shell Sort usa um incremento, chamado de "gap", para selecionar elementos que estão a uma distância fixa um do outro na lista original.

Essa técnica de incremento variável torna o Shell Sort eficaz ao lidar com diferentes tamanhos de gaps, tornando a reorganização dos elementos mais rápida e eficiente. A escolha adequada dos incrementos é essencial para o desempenho eficiente do algoritmo.

```
int h;
for(h = 1; h < count; h = 3*h+1);

while(h > 0){
    h = (h - 1) / 3;
    for(int i = h; i < count; i++){
        int aux = numbers[i];
        int j = i;

        while(numbers[j-h] > aux){
            numbers[j] = numbers[j-h];
            j -= h;
            if(j < h){
                break;
            }
        }
        numbers[j] = aux;
    }
}
```

Figura 11. Código Shell Sort

Shell Sort					
	100000	200000	300000	500000	1000000
Ordenado	0.003	0.006	0.01	0.017	0.037
Invertido	0.005	0.01	0.015	0.026	0.053
Aleatório	0.019	0.044	0.071	0.129	0.296

Figura 12. Resultados Shell Sort

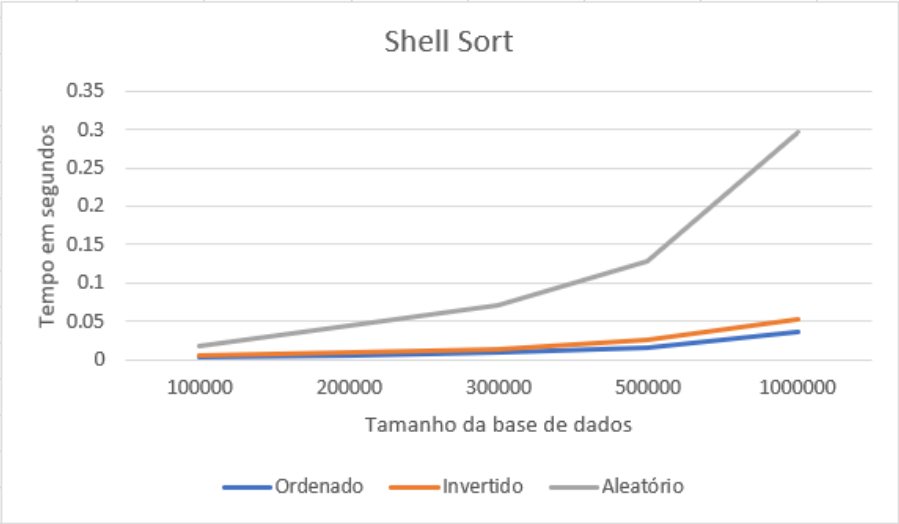

Figura 13. Gráfico Shell Sort

2.5. Quick Sort

O Quicksort é amplamente reconhecido como um dos algoritmos mais eficiente para ordenação por comparação. Sua abordagem consiste em selecionar um elemento pivô e reorganizar a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores. Ao final deste processo, o elemento pivô ocupa sua posição final na lista. Os dois grupos resultantes, um à esquerda e outro à direita do pivô, são então recursivamente submetidos ao mesmo processo, até que toda a lista esteja completamente ordenada.

A escolha do pivô é determinante no desempenho do algoritmo, e nas comparações realizadas, o pivô foi escolhida de forma aleatória.

```
void quick_sort_execute(int *numbers, int esq, int dir){
    int i;
    if(dir > esq){
        i = particione(numbers, esq, dir);
        quick_sort_execute(numbers, esq, i-1);
        quick_sort_execute(numbers, i+1, dir);
    }
}
```

Figura 14. Código Quick Sort Parte 1

```
int particione(int *numbers, int esq, int dir){
    int random_index = rand() % (dir - esq + 1) + esq;
    int x = numbers[random_index];

    numbers[random_index] = numbers[esq];
    numbers[esq] = x;

    int up = dir;
    int down = esq;


    while(down < up){
        while(down < dir && numbers[down] <= x){
            down++;
        }
        while(numbers[up] > x){
            up--;
        }
        if(down < up){
            int aux = numbers[down];
            numbers[down] = numbers[up];
            numbers[up] = aux;
        }
    }

    numbers[esq] = numbers[up];
    numbers[up] = x;
    return up;
}
```

Figura 15. Código Quick Sort Parte 2

Quick Sort					
	100000	200000	300000	500000	1000000
Ordenado	0.013	0.014	0.023	0.043	0.125
Invertido	0.007	0.016	0.022	0.058	0.155
Aleatório	0.007	0.026	0.04	0.07	0.149

Figura 16. Resultados Quick Sort

Figura 17. Gráfico Quick Sort

2.6. Merge Sort

A ideia fundamental por trás do Merge Sort é criar uma sequência ordenada a partir de duas sequências igualmente ordenadas. O algoritmo Merge Sort começa dividindo a sequência original em pares de elementos, reorganizando esses pares em ordem, e, em seguida, mesclando as sequências de pares ordenados para formar uma nova sequência ordenada com quatro elementos. Esse processo é repetido de forma recursiva até que toda a sequência esteja completamente ordenada.

```
void merge_sort_execute(int *numbers, int esq, int dir){
    if(esq < dir){
        int meio = (esq + dir) / 2;
        merge_sort_execute(numbers, esq, meio);
        merge_sort_execute(numbers, meio+1, dir);
        merge(numbers, esq, meio, dir);
    }
}
```

Figura 18. Código Merge Sort Parte 1

```
void merge(int *numbers, int p, int q, int r){
    int i, j, k;
    int *w = (int *)malloc((r - p + 1) * sizeof(int));

    i = p;
    j = q + 1;
    k = 0;

    while (i <= q && j <= r) {
        if (numbers[i] <= numbers[j]){
            w[k++] = numbers[i++];
        }else{
            w[k++] = numbers[j++];
        }

    }

    while (i <= q){
        w[k++] = numbers[i++];
    }

    while (j <= r){
        w[k++] = numbers[j++];
    }

    for (i = p, k = 0; i <= r; i++, k++) {
        numbers[i] = w[k];
    }
    free(w);
}
```

Figura 19. Código Merge Sort Parte 2

Merge Sort					
	100000	200000	300000	500000	1000000
Ordenado	0.021	0.025	0.047	0.078	0.147
Invertido	0.013	0.026	0.047	0.062	0.142
Aleatório	0.012	0.047	0.062	0.11	0.223

Figura 20. Resultados Merge Sort

Figura 21. Gráfico Merge Sort

2.7. Heap Sort

O Heap Sort é um que utiliza a estrutura de dados conhecida como heap binária. Esta estrutura é uma árvore binária especial em que o valor de cada nó é maior (no caso de um heap máximo) ou menor (no caso de um heap mínimo) do que os valores de seus filhos. A principal vantagem do Heap Sort é sua eficiência e, portanto, ele é frequentemente utilizado para classificar grandes conjuntos de dados.

O processo de ordenação Heap Sort começa por criar um heap a partir da lista de elementos a serem classificados. Isso é feito construindo um heap máximo, onde o maior elemento está na raiz da árvore binária. Uma vez que o heap é construído, o maior elemento (localizado na raiz) é extraído e colocado na posição correta na parte classificada da lista. Essa extração é realizada repetidamente até que todos os elementos tenham sido inseridos na parte classificada, transformando gradualmente o heap inicial em uma lista ordenada.

```
for(int i = count / 2 - 1; i >= 0; i--){
    heap_sort_execute(numbers, count, i);
}

for (int i = count - 1; i >= 0; i--) {
  trocar(&numbers[0], &numbers[i]);

  heap_sort_execute(numbers, i, 0);
}
```

Figura 22. Código Heap Sort Parte 1

```
void trocar(int *a, int *b) {
    int tempvar = *a;
    *a = *b;
    *b = tempvar;
  }

void heap_sort_execute(int *numbers, int n, int i){
    int greatest = i;
    int leftSide = 2 * i + 1;
    int rightSide = 2 * i + 2;

    if (leftSide < n && numbers[leftSide] > numbers[greatest])
      greatest = leftSide;

    if (rightSide < n && numbers[rightSide] > numbers[greatest])
      greatest = rightSide;

    if (greatest != i) {
      trocar(&numbers[i], &numbers[greatest]);
      heap_sort_execute(numbers, n, greatest);
    }
}
```

Figura 23. Código Heap Sort Parte 2

Heap Sort					
	100000	200000	300000	500000	1000000
Ordenado	0.017	0.036	0.057	0.099	0.214
Invertido	0.022	0.05	0.08	0.14	0.308
Aleatório	0.017	0.035	0.055	0.097	0.21

Figura 24. Resultados Heap Sort

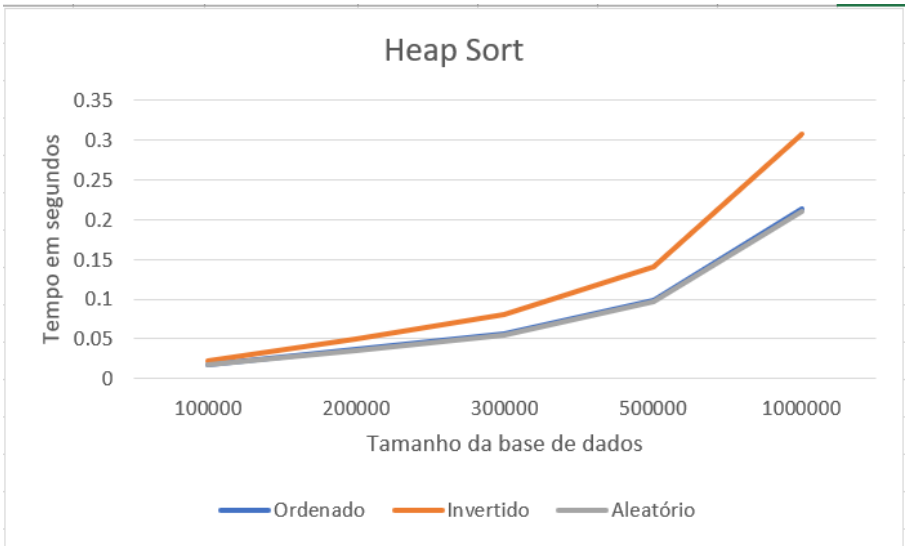

Figura 25. Gráfico Heap Sort

2.8. Radix Sort

O Radix Sort é um algoritmo de ordenação que se distingue pelo seu caráter não comparativo. O termo "radix" refere-se, por definição, à base ou ao número de dígitos únicos utilizados para representar números. Nesse contexto, o Radix Sort classifica elementos por meio de um processo que envolve múltiplas passagens, dígito por dígito.

A cada passagem, os elementos são ordenados de acordo com os dígitos de um determinado valor posicional, utilizando um algoritmo de ordenação estável, geralmente o counting sort, como uma sub-rotina. O algoritmo pode ser implementado para iniciar a ordenação a partir do dígito menos significativo (LSD) ou do dígito mais significativo (MSD), dependendo dos requisitos específicos.

Uma característica distintiva do Radix Sort é que o número de passagens necessárias para ordenar completamente os elementos é igual ao número de valores posicionais (dígitos) presentes no maior elemento entre todos os elementos de entrada. Essas passagens continuam até que todos os elementos estejam classificados.

```
int max = getMax(numbers, count);

for (int place = 1; max / place > 0; place *= 10)
    radix_sort_execute(numbers, count, place);
```

Figura 26. Código Radix Sort Parte 1

```
int getMax(int array[], int n) {
    int max = array[0];
    for (int i = 1; i < n; i++)
        if (array[i] > max)
            max = array[i];
    return max;
}
```

Figura 27. Código Radix Sort Parte 2

```
void radix_sort_execute(int numbers[], int n, int place) {
      int output[n + 1];
      int max = (numbers[0] / place) % 10;

      for (int i = 1; i < n; i++) {
        if (((numbers[i] / place) % 10) > max)
          max = numbers[i];
      }
      int count[max + 1];

      for (int i = 0; i < max; ++i)
        count[i] = 0;

      for (int i = 0; i < n; i++)
        count[(numbers[i] / place) % 10]++;

      for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

      for (int i = n - 1; i >= 0; i--) {
        output[count[(numbers[i] / place) % 10] - 1] = numbers[i];
        count[(numbers[i] / place) % 10]--;
      }

      for (int i = 0; i < n; i++)
        numbers[i] = output[i];
}
```

Figura 28. Código Radix Sort Parte 3

Radix Sort					
	100000	200000	300000	500000	1000000
Ordenado	0.017	0.036	0.056	0.1	0.213
Invertido	0.0161	0.035	0.055	0.097	0.206
Aleatório	0.024	0.05	0.079	0.139	0.304

Figura 29. Resultados Radix Sort

Figura 30. Gráfico Radix Sort

3. Comparações entre os métodos

Nesta seção, será realizada análises comparativas entre os métodos de ordenação, com o objetivo de determinar suas respectivas eficácias em diferentes cenários.

3.1. Bubble Sort e Insertion Sort

Ao compararmos os métodos Bubble Sort e Insertion Sort nas figuras 4 e 7, torna-se evidente que o Insertion Sort apresenta um desempenho superior em todas as situações. Para uma melhor visualização e compreensão dessa diferença, é fornecido a seguir um gráfico comparativo.

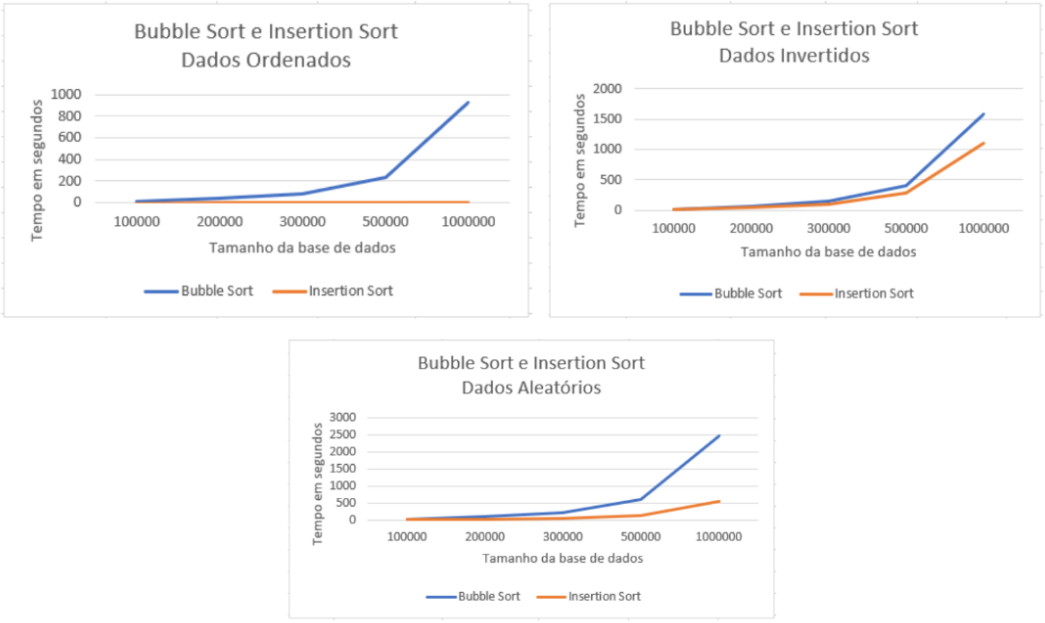

Figura 31. Gráficos Bubble Sort e Insertion Sort

O resultado mais impactante se manifesta quando consideramos conjuntos de dados já ordenados. Nesse cenário, o Insertion Sort conclui a ordenação em menos de 1 segundo, enquanto o Bubble Sort demanda mais de 15 minutos na maior base de dados. Essa discrepância é atribuída ao fato de que o algoritmo Bubble Sort, conforme implementado, segue o seu desenvolvimento original e realiza um grande número de operações mesmo em listas já ordenadas. No entanto, é importante destacar que uma melhoria substancial no desempenho do Bubble Sort pode ser alcançada com a simples adição de uma cláusula "if" no código, a qual verifica se a lista já está ordenada. Isso aproxima significativamente os resultados do Bubble Sort dos alcançados pelo Insertion Sort em listas ordenadas, embora, em contrapartida, possa resultar em pior desempenho em outros tipos de listas.

Esses resultados eram esperados, dado que o algoritmo e a lógica por trás do Bubble Sort são relativamente mais simples em comparação com o Insertion Sort. Essa simplicidade tende a ter um impacto significativo nos resultados de desempenho.

3.2. Selection Sort e Shell Sort

Observando as figuras que representam os métodos Selection Sort e Shell Sort (Figuras 9, 10, 12 e 13), podemos discernir claramente que o Shell Sort exibe um desempenho notavelmente superior em comparação ao Selection Sort. O Selection Sort opera de maneira relativamente simples, realizando a mesma operação em todos os elementos da lista, independentemente de sua ordenação ou do tamanho dos dados. Esse comportamento uniforme é evidente na Figura 9, onde o tempo de execução permanece notavelmente constante para diferentes tipos de ordenação, desde que o tamanho da lista seja mantido constante.

Em contraste, o Shell Sort utiliza uma abordagem mais sofisticada. Ele divide a lista em subconjuntos menores, aplicando o algoritmo de ordenação por inserção em cada um desses subconjuntos e, em seguida, realiza fusões progressivas desses subconjuntos para obter a ordenação completa. Essa técnica de divisão e conquista resulta em uma considerável melhoria no desempenho, particularmente em listas desordenadas ou parcialmente ordenadas. O Shell Sort se beneficia da capacidade de ajustar o tamanho dos subconjuntos (também conhecido como "incremento") de acordo com a natureza dos dados, adaptando-se de forma mais eficiente às condições específicas de ordenação.

Em resumo, o Shell Sort oferece um desempenho superior devido à sua estratégia de divisão e conquista adaptativa, enquanto o Selection Sort mantém uma abordagem simples, mas menos eficiente, que não se ajusta às características dos dados.

Figura 32. Gráficos Selection Sort e Shell Sort

3.3. Bubble Sort, Insertion Sort, Selection Sort e Shell Sort

Ao considerarmos as análises dos métodos Bubble Sort, Insertion Sort, Selection Sort e Shell Sort, podemos criar um gráfico que apresenta uma comparação abrangente entre esses algoritmos de ordenação. Esse gráfico nos permite visualizar de forma clara e concisa o desempenho relativo de cada método em diferentes cenários e configurações de dados.

O gráfico comparativo a seguir encapsula as descobertas até o momento, permitindo uma visão geral das forças e fraquezas de cada método de ordenação em diversos contextos:

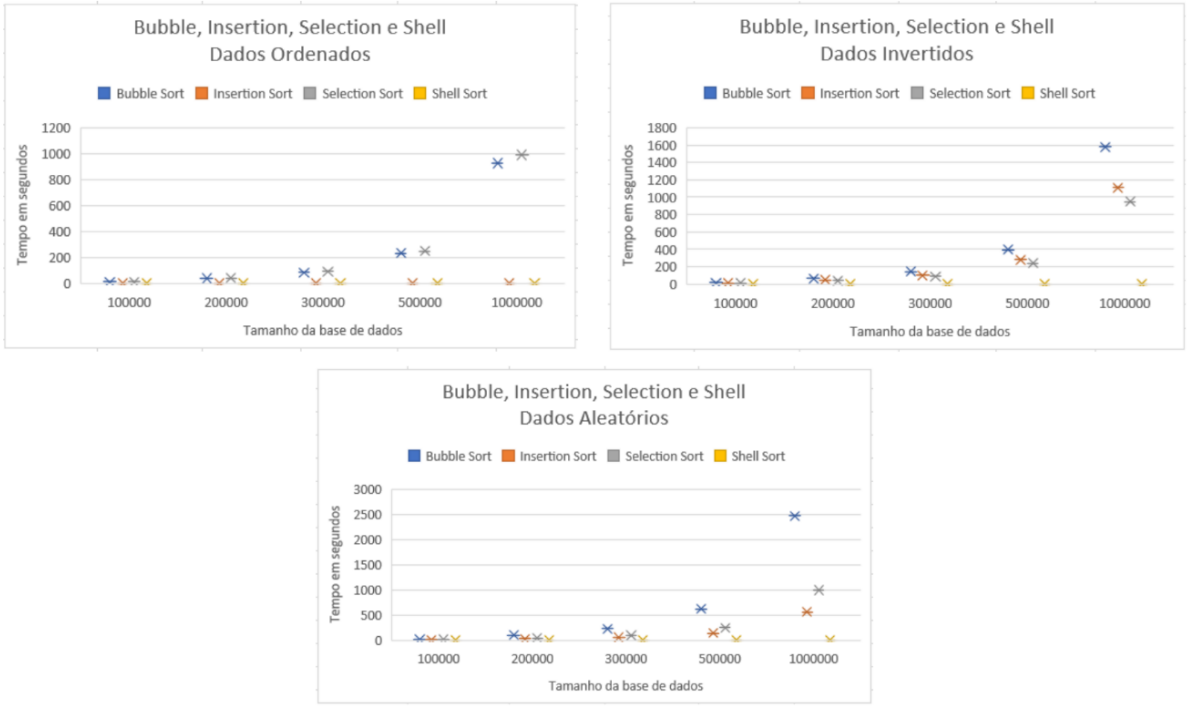

Figura 33. Gráficos Bubble Sort, Insertion Sort, Selection Sort e Shell Sort

3.4. Quick Sort e Merge Sort

A competição entre os métodos Quick Sort e Merge Sort é intensa, pois a diferença de desempenho entre eles é notável, muitas vezes na ordem de poucos milésimos de segundo. Em todos os casos analisados, o Quick Sort apresentou um desempenho superior. No entanto, é relevante destacar uma observação intrigante: houve um aumento significativo no tempo de execução quando lidamos com uma lista de tamanho 1.000.000, e nesse ponto, o Merge Sort superou o Quick Sort. Isso levanta uma interessante perspectiva, sugerindo que o Merge Sort pode se destacar ainda mais em listas de tamanhos consideravelmente maiores.

Essa descoberta enfatiza a complexidade das escolhas algorítmicas e como o desempenho de um método pode variar com base no tamanho e na estrutura dos dados. A decisão entre Quick Sort e Merge Sort dependerá, portanto, das particularidades do seu conjunto de dados e das necessidades específicas do seu projeto. Ambos os métodos têm suas vantagens e desvantagens, e compreender esses detalhes é essencial para fazer a escolha certa ao enfrentar o desafio de ordenação de dados em diferentes cenários.

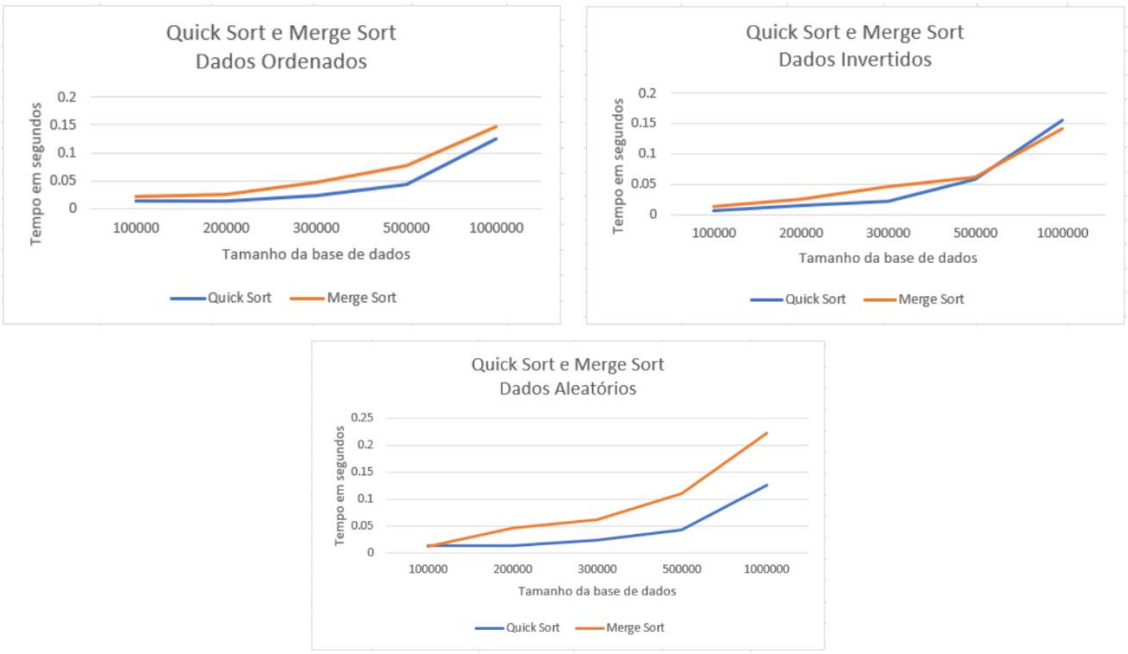

Figura 34. Gráficos Quick Sort e Merge Sort

3.5. Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Quick Sort e Merge Sort

Agora, podemos ter uma breve visualização sobre os 6 primeiros métodos apresentados.

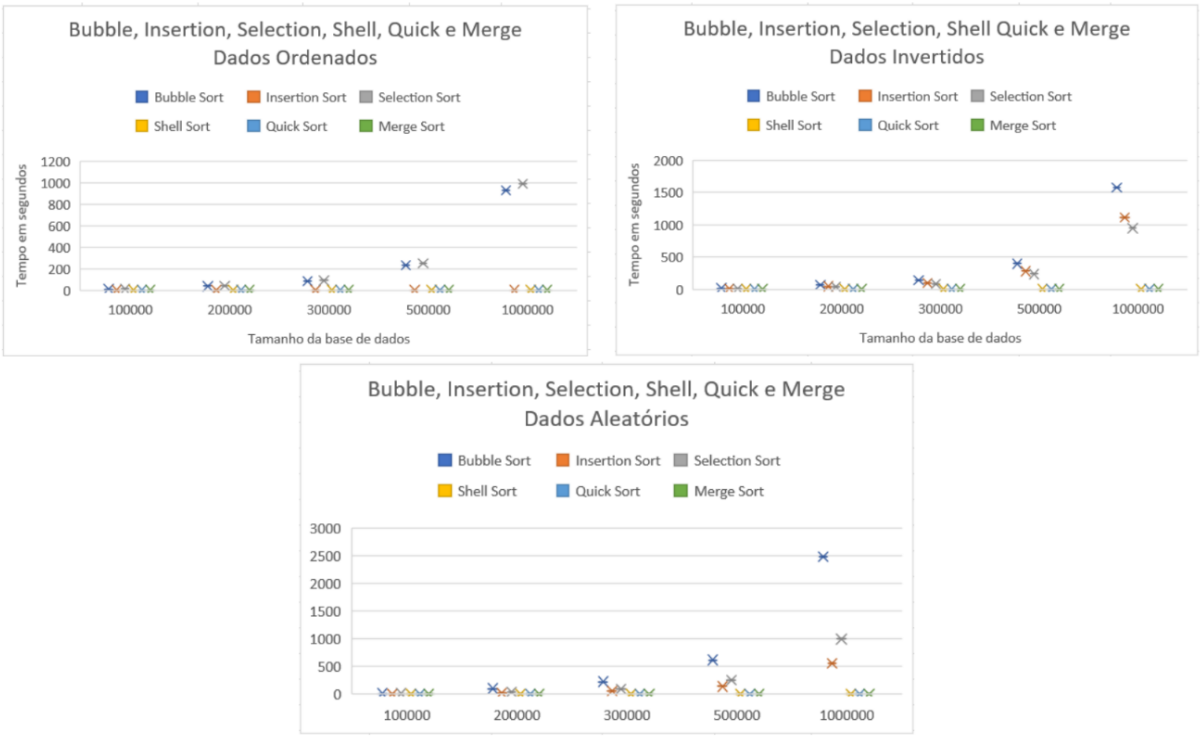

Figura 35. Gráficos Bubble, Insertion, Selection, Shell, Quick e Merge

No entanto, é importante observar que, devido ao baixo tempo de execução demonstrado por muitos dos métodos, a diferença entre os resultados mais eficientes torna-se praticamente imperceptível ao plotar o gráfico. Nesse sentido, é apresentado a seguir um ranking destacando os métodos mais eficazes para cada tipo de dado.

	Dados Ordenados
1º	Insertion Sort
2º	Shell Sort
3º	Quick Sort
4º	Merge Sort
5º	Bubble Sort
6º	Selection Sort

	Dados Invertidos
1º	Shell Sort
2º	Quick Sort
3º	Merge Sort
4º	Selection Sort
5º	Insertion Sort
6º	Bubble Sort

	Dados Aleatórios
1º	Quick Sort
2º	Merge Sort
3º	Shell Sort
4º	Insertion Sort
5º	Selection Sort
6º	Bubble Sort

Figura 36. Bubble, Insertion, Selection, Shell, Quick e Merge

Essa abordagem nos permite visualizar rapidamente quais métodos se destacam em diferentes cenários.

3.6. Heap Sort e Radix Sort

O método Heap Sort demonstrou uma notável consistência em uma variedade de cenários, independentemente do tamanho da base de dados ou do tipo de ordenação. Sua abordagem de divisão e conquista contribui para essa uniformidade, resultando em tempos de execução consistentemente baixos.

Por outro lado, o Radix Sort exibiu um desempenho mais variável, dependendo da natureza dos dados. Em particular, o Radix Sort mostrou-se mais adaptativo quando os dados estão ordenados de forma aleatória. Isso se deve à sua estratégia de classificação baseada em dígitos, que pode se beneficiar de algumas preordens ou ordenações parciais nos dados de entrada.

Essas observações destacam as diferenças fundamentais entre os dois algoritmos. O Heap Sort oferece consistência em todos os cenários, enquanto o Radix Sort pode se destacar em situações em que a natureza dos dados é aleatória.

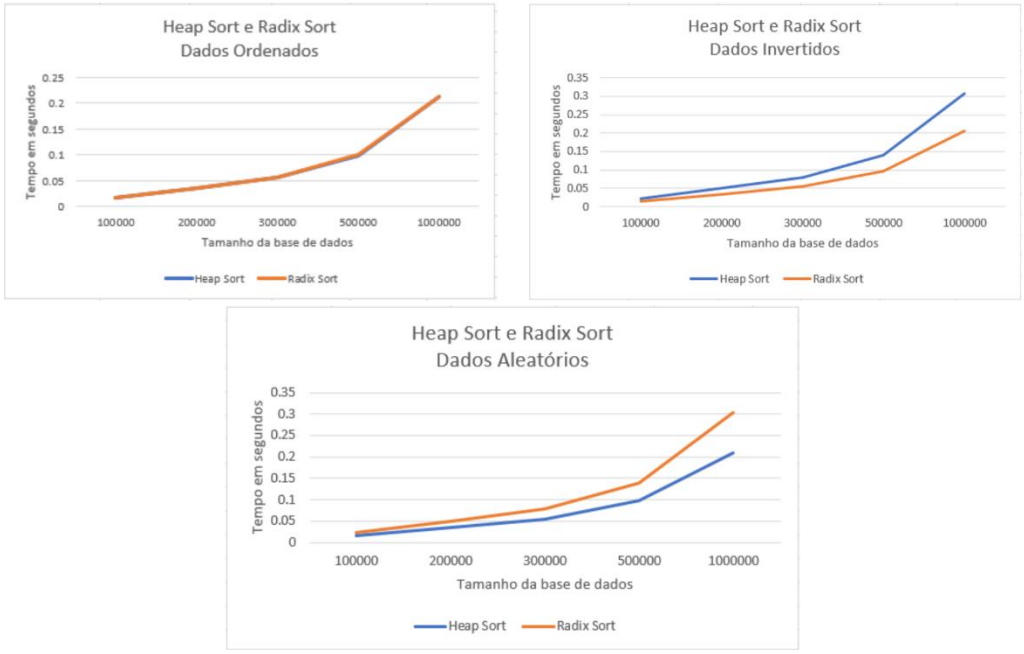

Figura 37. Gráficos Selection Sort e Shell Sort

3.7. Shell Sort, Quick Sort, Merge Sort, Heap Sort e Radix Sort

Nesta sessão poderemos obter uma análise comparativa gráfica entre os métodos mais eficientes, que levaram menos de 1 segundos para realizar as suas ordenações em todos os casos.

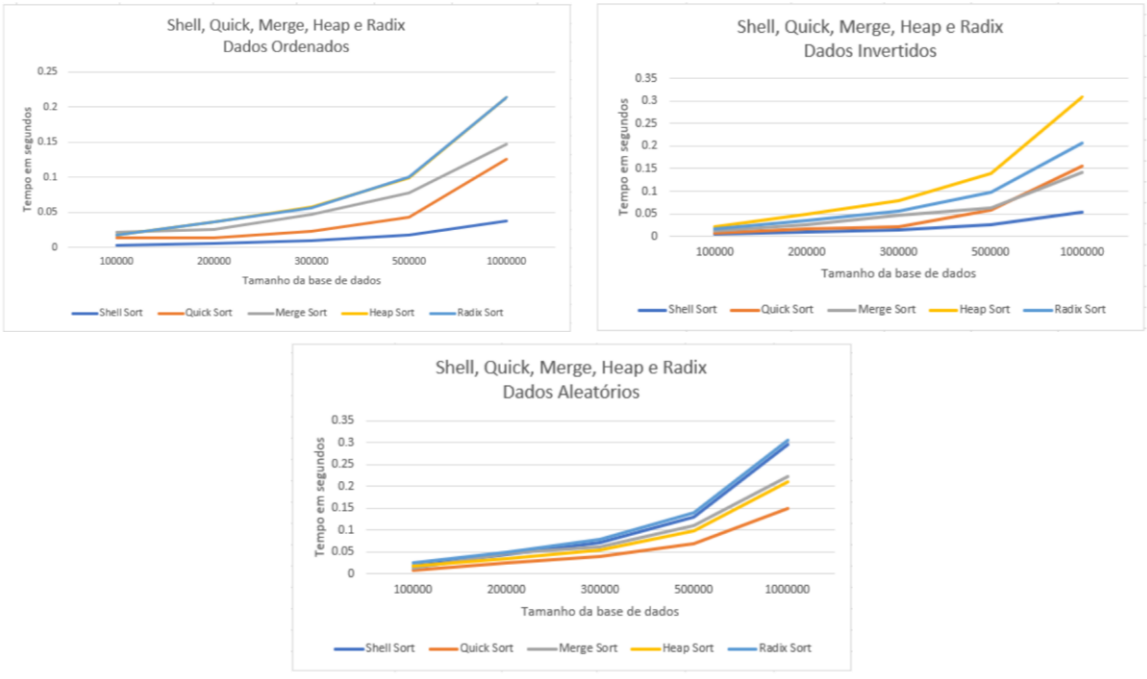

Figura 38. Gráficos Métodos Mais Eficazes

4. Conclusão

As análises revelaram as características distintas de cada método de ordenação e ofereceram insights valiosos para a escolha do algoritmo mais adequado em diferentes cenários. O Bubble Sort, embora simples, mostrou-se menos eficiente, principalmente em listas extensas, devido à sua natureza de iteração repetitiva.

O Insertion Sort destacou-se como uma escolha sólida, apresentando desempenho notavelmente rápido, especialmente quando confrontado com listas parcialmente ordenadas. Sua estratégia de inserção gradual provou ser eficaz nesses casos.

O Selection Sort, embora simples e previsível, revelou um desempenho moderado e constante, realizando uma quantidade fixa de operações, independentemente da ordenação inicial dos dados. No entanto, sua simplicidade pode ser uma desvantagem em comparação com métodos mais eficientes, especialmente em cenários mais complexos.

O Shell Sort emergiu como um dos melhores métodos, demonstrando alta eficiência. Sua estratégia de divisão e conquista, adaptada às características dos dados, resultou em tempos de execução notavelmente melhores, especialmente em listas desordenadas ou parcialmente ordenadas.

Além disso, é importante notar que o Quick Sort e o Merge Sort também desempenharam papéis importantes na análise. Na classificação de dados ordenados, o Quick Sort se destacou como uma escolha sólida, seguido pelo Merge Sort. No entanto, em listas invertidas, o Shell Sort se sobressaiu, seguido pelo Quick Sort e pelo Merge Sort. Em dados aleatórios, o Quick Sort foi o método mais eficiente, com o Merge Sort e o Shell Sort logo atrás.

Além disso, o método Heap Sort demonstrou uma notável consistência em uma variedade de cenários, independentemente do tamanho da base de dados ou do tipo de ordenação. Sua abordagem de divisão e conquista contribui para essa uniformidade, resultando em tempos de execução consistentemente baixos.

Por outro lado, o Radix Sort exibiu um desempenho mais variável, dependendo da natureza dos dados. Em particular, o Radix Sort mostrou-se mais adaptativo quando os dados estão ordenados de forma aleatória. Isso se deve à sua estratégia de classificação baseada em dígitos, que pode se beneficiar de algumas preordens ou ordenações parciais nos dados de entrada.

Em última análise, a seleção do método de ordenação ideal dependerá das características específicas do seu conjunto de dados e dos requisitos de desempenho de seu projeto. Cada método tem suas forças e fraquezas, e compreender esses detalhes é essencial para tomar decisões informadas sobre como classificar eficientemente os dados em uma variedade de cenários.

Referências

Honorato, Bruno de Almeida. Algoritmos de Ordenação: Análise e Comparação. 1. Disponível em: <https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261>. Acesso em: 14 set. 2023.

Farias, Ricardo. Estrutura de Dados e Algoritmos. Disponível em: <https://www.cos.ufrj.br/~rfarias/cos121/aula_07.html>. Acesso em: 14 set. 2023.

Garcia, Islene Calciolari. Islene Calciolari Garcia. Disponível em: https://www.ic.unicamp.br/~islene/. Acesso em: 14 set. 2023.

INSERTION Sort Explained–A Data Scientists Algorithm Guide | NVIDIA Technical Blog. Disponível em: <https://developer.nvidia.com/blog/insertion-sort-explained-a-data-scientists-algorithm-guide/>. Acesso em: 14 set. 2023.

MANSI. Heap Sort Program in C. Disponível em: <https://www.scaler.com/topics/heap-sort-program-in-c/> Acesso em: 29 set. 2023.

Owen Astrachan, "Bubble sort: An archaeological algorithmic analysis", Proc. ACM Technol. Symp. Comput. Sci. Educ. (SIGCSE), pp. 1-2, 2003.

PROGRAMIZ. Radix Sort Algorithm. Disponível em: <https://www.programiz.com/dsa/radix-sort>. Acesso em: 29 set. 2023.

Resolução de Problemas com Algoritmos e Estruturas de Dados usando Python — Resolução de Problemas Usando Python. Disponível em: <https://panda.ime.usp.br/panda/static/pythonds_pt/>. Acesso em: 13 set. 2023.

Sepulveda, Gloria Patrica Lopez. (2023). Pesquisa e Ordenação de Dados - Métodos de Ordenação de Dados - Parte I, II e III. Universidade Tecnologica Federal do Paraná, Campus - Santa Helena.