Algoritmos e Estrutura de Dados 1 - Turma A e B

Algoritmos de Ordenação

Cauã Borges Faria (834437)
Vitor Rodrigues da Mata (831591)

UFSCar - Universidade Federal de São Carlos
São Carlos
7 de dezembro de 2024

Conteúdo

1 Códigos Utilizados 2
2 Tabela de Tempos de Execução 4
3 Gráfico de Tempos de Execução 5
4 Sobre CombSort 6

1 Códigos Utilizados

O código a seguir realiza a geração dos vetores, ordena os números e calcula o tempo de execução para análise de desempenho.

```python
# importa bibliotecas necessarias
import time
import random
import copy

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

def comb_sort(arr):
    gap = len(arr)
    shrink = 1.3
    sorted = False

    while not sorted:
        gap = int(gap / shrink)
        if gap <= 1:
            gap = 1
            sorted = True

        for i in range(len(arr) - gap):
            if arr[i] > arr[i + gap]:
                arr[i], arr[i + gap] = arr[i + gap], arr[i]
                sorted = False

```

```python
def exec_time(func, arr):
    start_time = time.time()
    func(arr)
    end_time = time.time()
    return end_time - start_time

def gerar_numeros(qtd_numeros):
    return [random.randint(0, 1000) for _ in range(qtd_numeros)]

# de 2000 ate < 50001 indo de 2000 em 2000
for tamanho in range(2000, 50001, 2000):
    vetor_original = gerar_numeros(tamanho)

    # Bubble Sort
    vetor_bubble = copy.deepcopy(vetor_original)
    tempo_bubble = exec_time(bubble_sort, vetor_bubble)
    print(f"Bubble Sort - Tamanho {tamanho}: {tempo_bubble:.4f}
        segundos")

    # Selection Sort
    vetor_selection = copy.deepcopy(vetor_original)
    tempo_selection = exec_time(selection_sort, vetor_selection)
    print(f"Selection Sort - Tamanho {tamanho}: {tempo_selection:.4
        f} segundos")

    # Insertion Sort
    vetor_insertion = copy.deepcopy(vetor_original)
    tempo_insertion = exec_time(insertion_sort, vetor_insertion)
    print(f"Insertion Sort - Tamanho {tamanho}: {tempo_insertion:.4
        f} segundos")

    # Comb Sort
    vetor_comb = copy.deepcopy(vetor_original)
    tempo_comb = exec_time(comb_sort, vetor_comb)
    print(f"Comb Sort - Tamanho {tamanho}: {tempo_comb:.4f}
        segundos")

    print("-" * 50)  # Separador para cada teste
```

Listing 1: Código Python para geração, ordenação e cálculo de tempo

2 Tabela de Tempos de Execução

Tabela 1: Tempos de execução dos algoritmos de ordenação (em segundos)

Tamanho	**Bubble Sort**	**Selection Sort**	**Insertion Sort**	**Comb Sort**
2000	0.0975	0.0386	0.0406	0.0021
4000	0.4517	0.1719	0.3231	0.0134
6000	3.0141	0.9223	0.3618	0.0064
8000	1.8289	0.7095	0.7260	0.0107
10000	5.5238	1.0489	1.0947	0.0128
12000	7.1352	1.8134	1.6271	0.0148
14000	9.0889	3.9873	3.5822	0.0206
16000	10.4996	5.4990	2.9629	0.0595
18000	14.6155	6.1338	3.5669	0.0276
20000	17.1641	6.7750	7.0006	0.0250
22000	21.8699	7.0531	7.5026	0.0901
24000	27.5707	8.9654	9.0116	0.0316
26000	29.8195	11.3057	10.1568	0.0385
28000	34.1546	13.5851	12.0644	0.1301
30000	40.5528	16.6226	16.4070	0.0523
32000	51.8460	18.9584	18.6720	0.0408
34000	53.1611	21.5385	19.2965	0.1361
36000	61.1968	25.0440	19.2368	0.0717
38000	58.8555	23.2195	21.6246	0.0925
40000	64.8101	25.0129	24.0579	0.0918
42000	70.0875	27.5498	26.8945	0.0833
44000	77.2259	30.1359	29.2495	0.0913
46000	84.8075	33.1679	33.7917	0.0969
48000	92.4779	34.0277	31.0970	0.0973
50000	96.9221	40.6714	39.7031	0.1182

3 Gráfico de Tempos de Execução

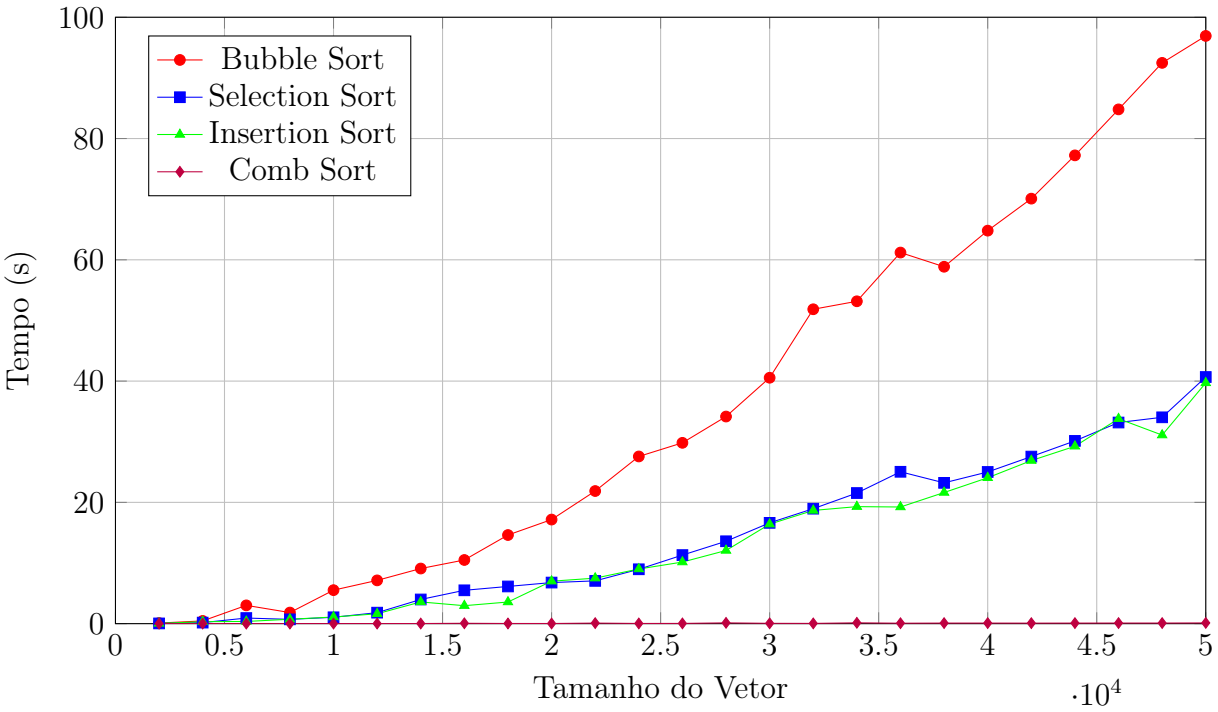

Figura 1: Gráfico comparativo dos tempos de execução dos algoritmos de ordenação.

4 Sobre CombSort

Comb Sort e Bubble Sort: Diferenças

O Comb Sort é uma versão aprimorada do Bubble Sort. Enquanto o Bubble Sort compara e troca elementos adjacentes, o Comb Sort utiliza gaps maiores inicialmente, permitindo que elementos distantes sejam ordenados primeiro. Isso resolve rapidamente os "coelhos"(elementos grandes no final da lista) e reduz o tempo total de execução.

Comb Sort e Selection Sort: Diferenças

O Selection Sort busca o menor elemento da lista inteira e o posiciona corretamente em cada iteração, enquanto o Comb Sort reduz gradualmente o tamanho do gap e utiliza comparações em múltiplas posições ao longo da lista.

Comb Sort e Insertion Sort: Diferenças

O Insertion Sort ordena elementos conforme os insere na posição correta, trabalhando localmente, enquanto o Comb Sort atua inicialmente em elementos distantes antes de refinar a ordenação com elementos adjacentes.

Complexidade do Comb Sort

- **Melhor caso**: $O(n \log n)$ — devido à redução exponencial do gap.
- **Pior caso**: $O(n^2)$ — similar ao Bubble Sort em listas quase ordenadas.