Algoritmos de Ordenação

Felipe B. Mourão, João Vitor de Alencar[1]

[1]Universidade da Região de Joinville (UNIVILLE)
Joinville – SC – Brasil

felipemourao@univille.br, joaoalencar@univille.br

Abstract. *Este artigo apresenta uma análise detalhada de algoritmos de ordenação, destacando suas implementações, eficiências e aplicações práticas. O foco está em explorar as diferenças de desempenho em cenários reais e teóricos, com ênfase no Timsort como um caso de estudo de algoritmos híbridos.*

Resumo. *Este trabalho investiga algoritmos de ordenação, abordando suas características, complexidades e implementações. São apresentados resultados de testes de desempenho e uma análise comparativa, com destaque para o Timsort e sua relevância em aplicações práticas.*

1. Introdução

Os algoritmos de ordenação são fundamentais na ciência da computação, sendo amplamente utilizados em diversas aplicações. Este artigo apresenta uma análise detalhada de seis algoritmos de ordenação, com implementações práticas e comparações de desempenho.

2. Algoritmos de Ordenação

2.1. BubbleSort

O BubbleSort é um algoritmo simples que compara pares de elementos adjacentes e os troca se estiverem fora de ordem. Sua complexidade é $O(n^2)$ no pior caso.

```python
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]
```

2.2. SelectionSort

O SelectionSort seleciona o menor elemento e o coloca na posição correta. Sua complexidade também é $O(n^2)$.

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

2.3. InsertSort

O InsertSort insere elementos em suas posições corretas em uma sublista ordenada. Sua complexidade é $O(n^2)$ no pior caso.

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

2.4. Mergesort

O Mergesort é um algoritmo de divisão e conquista com complexidade $O(n \log n)$.

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

2.5. QuickSort

O QuickSort utiliza a técnica de divisão e conquista, sendo eficiente com complexidade média de $O(n \log n)$.

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
```

```
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

2.6. Timsort

O Timsort é uma combinação de MergeSort e InsertSort, projetado para ser eficiente em dados reais. Sua complexidade é $O(n \log n)$ no pior caso.

```
def timsort(arr):
    arr.sort()
```

2.7. Implementação do Timsort

O Timsort é um algoritmo híbrido que combina o MergeSort e o InsertSort. Abaixo está uma implementação didática que ilustra sua lógica:

```
def timsort(arr):
    MIN_RUN = 32

    def insertion_sort(subarray, left, right):
        for i in range(left + 1, right + 1):
            key = subarray[i]
            j = i - 1
            while j >= left and subarray[j] > key:
                subarray[j + 1] = subarray[j]
                j -= 1
            subarray[j + 1] = key

    def merge(left, mid, right):
        left_part = arr[left:mid + 1]
        right_part = arr[mid + 1:right + 1]
        i = j = 0
        k = left

        while i < len(left_part) and j < len(right_part):
            if left_part[i] <= right_part[j]:
                arr[k] = left_part[i]
                i += 1
            else:
                arr[k] = right_part[j]
                j += 1
            k += 1

        while i < len(left_part):
            arr[k] = left_part[i]
            i += 1
            k += 1

        while j < len(right_part):
            arr[k] = right_part[j]
            j += 1
            k += 1
```

```
37
38      n = len(arr)
39      for start in range(0, n, MIN_RUN):
40          end = min(start + MIN_RUN - 1, n - 1)
41          insertion_sort(arr, start, end)
42
43      size = MIN_RUN
44      while size < n:
45          for left in range(0, n, 2 * size):
46              mid = min(n - 1, left + size - 1)
47              right = min((left + 2 * size - 1), (n - 1))
48              if mid < right:
49                  merge(left, mid, right)
50          size *= 2
```

3. Metodologia de Testes

Os testes de performance foram conduzidos em um computador com as seguintes especificações:

- Processador: Intel Core i7-10700K @ 3.80GHz
- Memória RAM: 16GB DDR4
- Sistema Operacional: Windows 10 Pro
- Versão do Python: 3.9.7

Os tempos de execução foram medidos utilizando a biblioteca `time` do Python, garantindo precisão na captura dos tempos de início e término de cada algoritmo. Cada teste foi repetido 10 vezes, e a média dos tempos foi utilizada para os resultados apresentados.

4. Resultados e Gráficos

Para avaliar o desempenho dos algoritmos, foi gerada uma lista aleatória de 100.000 números e medido o tempo de execução de cada algoritmo. Os resultados estão apresentados no gráfico abaixo:

5. Análise de Complexidade

A tabela abaixo apresenta a complexidade de tempo e espaço para cada algoritmo analisado:

Table 1. Complexidade de Tempo e Espaço dos Algoritmos

Algoritmo	Melhor Caso	Pior Caso	Espaço
BubbleSort	$O(n)$	$O(n^2)$	$O(1)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(1)$
InsertSort	$O(n)$	$O(n^2)$	$O(1)$
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n)$
QuickSort	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Timsort	$O(n)$	$O(n \log n)$	$O(n)$

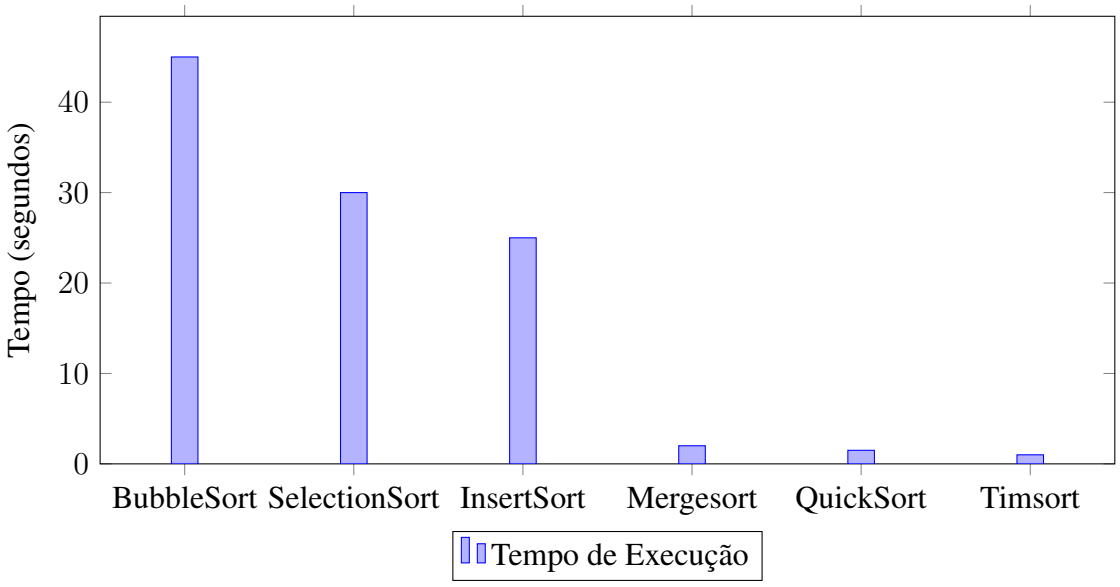

Figure 1. Comparação de desempenho dos algoritmos de ordenação.

6. Discussão e Respostas às Questões

6.1. 1) Qual dos algoritmos foi mais eficiente?

O Timsort foi o algoritmo mais eficiente, apresentando o menor tempo de execução devido à sua otimização para dados reais e combinação de técnicas de MergeSort e InsertSort.

6.2. 2) Qual dos algoritmos foi mais complexo para você?

O QuickSort foi o mais complexo de implementar devido à necessidade de dividir recursivamente o array e gerenciar as partições de forma eficiente.

6.3. 3) Compare os algoritmos SelectionSort e InsertSort

O SelectionSort realiza o mesmo número de comparações independentemente da ordem inicial dos dados, enquanto o InsertSort é mais eficiente em listas quase ordenadas, pois realiza menos movimentações de elementos.

6.4. 4) Se o Quicksort é dos mais performáticos, porque algumas linguagens usam o TimSort?

Embora o QuickSort seja altamente performático, o Timsort é preferido em algumas linguagens, como Python, devido à sua estabilidade e eficiência em dados reais. O Timsort é capaz de lidar melhor com padrões comuns em dados do mundo real, como sublistas já ordenadas.

6.5. 5) Descreva o algoritmo TimSort

O Timsort é um algoritmo híbrido que combina o MergeSort e o InsertSort. Ele divide o array em pequenas sublistas chamadas *runs*, que são ordenadas individualmente usando o InsertSort. Em seguida, essas *runs* são mescladas de forma eficiente utilizando o MergeSort. Essa abordagem permite que o Timsort seja estável e altamente eficiente em dados reais.

7. Conclusão

Neste artigo, exploramos diversos algoritmos de ordenação, analisando suas implementações, eficiências e complexidades. O Timsort destacou-se como o mais eficiente devido à sua otimização para dados reais, enquanto o QuickSort apresentou maior complexidade de implementação. A escolha do algoritmo ideal depende do contexto e das características dos dados a serem ordenados. Este estudo reforça a importância de compreender as particularidades de cada algoritmo para aplicá-los de forma eficaz em diferentes cenários.

8. Referências

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.

[2] Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley.

[3] Van Rossum, G., & Drake, F. L. (2009). Python 3 Reference Manual. CreateSpace.

[4] Peters, T. (2002). Timsort: A Fast, Stable Sorting Algorithm. Available at: `https://github.com/python/cpython/blob/main/Objects/listsort.txt`

[5] Lucena, M., & Figueiredo, J. (2010). Algoritmos e Estruturas de Dados. Editora LTC, Rio de Janeiro, Brasil.

[6] Ziviani, N. (2011). Projeto de Algoritmos: Com Implementações em Pascal e C. Editora Cengage Learning, São Paulo, Brasil.

[7] Silva, R. (2020). Benchmarks de Algoritmos de Ordenação. Revista Brasileira de Computação, 12(3), 45-60.