Algoritmos de Ordenação

Os algoritmos de ordenação são fundamentais na ciência da computação e frequentemente aparecem em maratonas de programação. Vou explicar detalhadamente os principais algoritmos, suas características, implementações e análises de complexidade.

1. Bubble Sort

Descrição: O Bubble Sort compara pares adjacentes de elementos e os troca se estiverem na ordem errada. O processo é repetido até que não sejam necessárias mais trocas.

Implementação:

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        # Flag para otimização - para se nenhuma troca for feita
        swapped = False

        # Última i iterações já colocaram os i maiores elementos nas posições corretas
        for j in range(0, n - i - 1):
            # Troca se o elemento encontrado for maior que o próximo
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        # Se nenhuma troca aconteceu nesta passagem, o array está ordenado
        if not swapped:
            break

    return arr
```

Complexidade:

- **Tempo:**
  - Melhor caso: O(n) - quando o array já está ordenado
  - Caso médio: $O(n^2)$
  - Pior caso: $O(n^2)$
- **Espaço:** O(1) - ordenação in-place

Características:

- Algoritmo estável (não altera a ordem relativa de elementos iguais)
- Simples de implementar
- Ineficiente para grandes conjuntos de dados
- Usado principalmente para fins educacionais

2. Selection Sort

Descrição: O Selection Sort divide o array em uma parte ordenada e outra não ordenada. A cada iteração, busca o menor elemento na parte não ordenada e o move para o final da parte ordenada.

Implementação:

```python
def selection_sort(arr):
    n = len(arr)

    for i in range(n):
        # Encontra o índice do menor elemento na parte não ordenada
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j

        # Troca o menor elemento encontrado com o primeiro elemento não ordenado
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

    return arr
```

Complexidade:

- **Tempo:**
  - Melhor caso: $O(n^2)$
  - Caso médio: $O(n^2)$
  - Pior caso: $O(n^2)$
- **Espaço:** O(1) - ordenação in-place

Características:

- Não é estável na implementação padrão

- Número de trocas é no máximo O(n), menor que outros algoritmos $O(n^2)$
- Consistentemente ruim, independente da ordem inicial dos dados
- Simples de entender e implementar

3. Insertion Sort

Descrição: O Insertion Sort constrói a sequência ordenada um elemento de cada vez. Para cada elemento, encontra sua posição correta na parte já ordenada e o insere lá.

Implementação:

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        # Move elementos maiores que key uma posição à frente
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        # Coloca key na posição correta
        arr[j + 1] = key

    return arr
```

Complexidade:

- **Tempo:**
  - Melhor caso: O(n) - quando o array já está ordenado
  - Caso médio: $O(n^2)$
  - Pior caso: $O(n^2)$ - quando o array está ordenado em ordem reversa
- **Espaço:** O(1) - ordenação in-place

Características:

- Algoritmo estável
- Eficiente para pequenos conjuntos de dados
- Adaptativo: eficiência melhora se o array estiver parcialmente ordenado

- Usado como subrotina em algoritmos mais avançados como o Timsort
- Eficiente para inserção contínua de novos elementos

4. Merge Sort

Descrição: O Merge Sort é um algoritmo de divisão e conquista que divide o array em metades, ordena cada metade recursivamente e depois mescla as metades ordenadas.

Implementação:

python

```python
def merge_sort(arr):
    if len(arr) > 1:
        # Divide o array ao meio
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Ordenação recursiva das duas metades
        merge_sort(left_half)
        merge_sort(right_half)

        # Mesclagem das duas metades ordenadas
        i = j = k = 0

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

        # Verifica se há elementos restantes
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

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

    return arr
```

Complexidade:

- **Tempo:**
  - Melhor caso: O(n log n)
  - Caso médio: O(n log n)

    - Pior caso: O(n log n)
- **Espaço:** O(n) - requer espaço adicional para o processo de mesclagem

Características:

- Algoritmo estável
- Desempenho consistente independente da distribuição dos dados
- Eficiente para grandes conjuntos de dados
- Ideal para ordenação de listas ligadas (com adaptações)
- Paralelizável

5. Quick Sort

Descrição: O Quick Sort é um algoritmo de divisão e conquista que seleciona um elemento 'pivô' e particiona o array ao redor desse pivô, colocando elementos menores à esquerda e maiores à direita.

Implementação:

python

```python
def quick_sort(arr, low=None, high=None):
    if low is None:
        low = 0
    if high is None:
        high = len(arr) - 1

    if low < high:
        # Encontra o índice de partição
        pi = partition(arr, low, high)

        # Recursivamente ordena elementos antes e depois da partição
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

    return arr

def partition(arr, low, high):
    # Escolhe o pivô (neste caso, o último elemento)
    pivot = arr[high]

    # Índice do menor elemento
    i = low - 1

    for j in range(low, high):
        # Se o elemento atual for menor ou igual ao pivô
        if arr[j] <= pivot:
            # Incrementa o índice do menor elemento
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    # Coloca o pivô na posição correta
    arr[i + 1], arr[high] = arr[high], arr[i + 1]

    return i + 1
```

Complexidade:

- **Tempo:**
  - Melhor caso: O(n log n)
  - Caso médio: O(n log n)

- Pior caso: $O(n^2)$ - quando o pivô é sempre o menor ou maior elemento

- **Espaço:**
  - O(log n) - espaço da pilha de recursão no caso médio
  - O(n) - no pior caso

Características:

- Não é estável na implementação típica
- Geralmente mais rápido na prática que outros algoritmos O(n log n)
- Comportamento pode depender da escolha do pivô
- Existem variações como o Quick Sort com três vias e o Quick Sort aleatório
- Usado como base nos algoritmos de ordenação de muitas linguagens de programação

Otimizações do Quick Sort:

1. **Escolha do pivô:**

   python

   ```python
   def median_of_three(arr, low, high):
       mid = (low + high) // 2
       # Ordena arr[low], arr[mid], arr[high]
       if arr[low] > arr[mid]:
           arr[low], arr[mid] = arr[mid], arr[low]
       if arr[low] > arr[high]:
           arr[low], arr[high] = arr[high], arr[low]
       if arr[mid] > arr[high]:
           arr[mid], arr[high] = arr[high], arr[mid]
       # Retorna o valor médio
       return mid
   ```

2. **Quick Sort com Insertion Sort para partições pequenas:**

python

```python
def optimized_quick_sort(arr, low=None, high=None):
    if low is None:
        low = 0
    if high is None:
        high = len(arr) - 1

    # Usa insertion sort para partições pequenas
    if high - low + 1 < 10:
        insertion_sort_range(arr, low, high)
        return arr

    if low < high:
        # Usa método da mediana de três para escolha do pivô
        mid = median_of_three(arr, low, high)
        arr[mid], arr[high] = arr[high], arr[mid]

        pi = partition(arr, low, high)
        optimized_quick_sort(arr, low, pi - 1)
        optimized_quick_sort(arr, pi + 1, high)

    return arr
```

6. Heap Sort

Descrição: O Heap Sort usa uma estrutura de dados heap para ordenar elementos. Primeiro, constrói um max-heap a partir do array, depois repetidamente extrai o elemento máximo.

Implementação:

python

```python
def heapify(arr, n, i):
    largest = i  # Inicializa o maior como raiz
    left = 2 * i + 1
    right = 2 * i + 2

    # Verifica se o filho da esquerda existe e é maior que a raiz
    if left < n and arr[left] > arr[largest]:
        largest = left

    # Verifica se o filho da direita existe e é maior que o maior até agora
    if right < n and arr[right] > arr[largest]:
        largest = right

    # Muda a raiz se necessário
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        # Heapify a subárvore afetada
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

    # Constrói um max-heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extrai elementos um a um
    for i in range(n - 1, 0, -1):
        # Move a raiz atual para o final
        arr[i], arr[0] = arr[0], arr[i]
        # Chama max heapify no heap reduzido
        heapify(arr, i, 0)

    return arr
```

Complexidade:

- **Tempo:**
  - Melhor caso: O(n log n)
  - Caso médio: O(n log n)

    - Pior caso: O(n log n)
- **Espaço:** O(1) - ordenação in-place

Características:

- Não é estável na implementação padrão
- Desempenho consistente independente da distribuição dos dados
- Não requer espaço adicional além do array original
- Eficiente para encontrar os k maiores/menores elementos
- Menos eficiente na prática do que Quick Sort bem implementado

7. Counting Sort

Descrição: O Counting Sort é um algoritmo de ordenação não-comparativo que funciona contando o número de ocorrências de cada elemento e usando essa informação para posicioná-los.

Implementação:

python

```python
def counting_sort(arr):
    if not arr:
        return arr

    # Encontra o valor máximo no array
    max_val = max(arr)
    min_val = min(arr)
    range_val = max_val - min_val + 1

    # Inicializa o array de contagem e o array de saída
    count = [0] * range_val
    output = [0] * len(arr)

    # Conta ocorrências de cada elemento
    for num in arr:
        count[num - min_val] += 1

    # Modifica o array de contagem para conter as posições reais
    for i in range(1, len(count)):
        count[i] += count[i - 1]

    # Constrói o array de saída
    for i in range(len(arr) - 1, -1, -1):
        output[count[arr[i] - min_val] - 1] = arr[i]
        count[arr[i] - min_val] -= 1

    return output
```

Complexidade:

- **Tempo:** O(n + k), onde k é o tamanho do intervalo de valores (max - min + 1)
- **Espaço:** O(n + k)

Características:

- Algoritmo estável
- Eficiente quando o intervalo de valores não é muito maior que o número de elementos
- Não é baseado em comparações
- Limitado a dados inteiros ou que possam ser mapeados para inteiros
- Ineficiente se o intervalo de valores for muito grande

8. Radix Sort

Descrição: O Radix Sort ordena números processando dígito por dígito, começando pelo menos significativo até o mais significativo.

Implementação:

python

```python
def counting_sort_digit(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10  # 0-9

    # Conta ocorrências do dígito atual
    for i in range(n):
        idx = arr[i] // exp % 10
        count[idx] += 1

    # Acumula contagens para obter posições
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Constrói o array de saída
    for i in range(n - 1, -1, -1):
        idx = arr[i] // exp % 10
        output[count[idx] - 1] = arr[i]
        count[idx] -= 1

    # Copia de volta para o array original
    for i in range(n):
        arr[i] = output[i]

def radix_sort(arr):
    if not arr:
        return arr

    # Encontra o número máximo para saber quantos dígitos
    max_val = max(arr)

    # Faz counting sort para cada dígito
    exp = 1
    while max_val // exp > 0:
        counting_sort_digit(arr, exp)
        exp *= 10

    return arr
```

Complexidade:

- **Tempo:** O(d * (n + k)), onde d é o número de dígitos e k é a base (10 para decimal)

- **Espaço:** O(n + k)

Características:

- Algoritmo estável
- Eficiente para números inteiros com tamanho limitado
- Não utiliza comparações
- Desempenho independe da distribuição dos dados
- Pode ser estendido para strings e outros tipos de dados

9. Bucket Sort

Descrição: O Bucket Sort divide o intervalo em "baldes" iguais, distribui os elementos nos baldes e então ordena cada balde individualmente.

Implementação:

python

```python
def bucket_sort(arr, num_buckets=10):
    if not arr:
        return arr

    # Encontra o valor máximo e mínimo
    max_val, min_val = max(arr), min(arr)

    # Define o tamanho do intervalo para cada balde
    bucket_range = (max_val - min_val) / num_buckets

    # Cria os baldes
    buckets = [[] for _ in range(num_buckets)]

    # Distribui elementos nos baldes
    for num in arr:
        # Calcula o índice do balde
        index = min(int((num - min_val) / bucket_range), num_buckets - 1)
        buckets[index].append(num)

    # Ordena cada balde (usando insertion sort)
    for i in range(num_buckets):
        # Ordenação simples aqui - pode ser substituída por qualquer algoritmo
        buckets[i].sort()

    # Concatena os baldes ordenados
    result = []
    for bucket in buckets:
        result.extend(bucket)

    return result
```

Complexidade:

- **Tempo:**
  - Melhor caso: O(n + k), onde k é o número de baldes
  - Caso médio: O(n + k)
  - Pior caso: O(n^2) - quando todos os elementos caem em um único balde
- **Espaço:** O(n + k)

Características:

- Algoritmo estável se o algoritmo usado para ordenar os baldes for estável
- Eficiente quando os dados estão uniformemente distribuídos
- Paralelizável - cada balde pode ser ordenado independentemente
- O desempenho depende da função de distribuição dos baldes
- Funciona bem com arrays de números reais em [0,1)

10. Tim Sort

Descrição: Tim Sort é um algoritmo híbrido derivado do Merge Sort e do Insertion Sort, utilizado por padrão no Python e Java.

Implementação (simplificada):

```
python
```

```python
def insertion_sort_range(arr, left, right):
    for i in range(left + 1, right + 1):
        key = arr[i]
        j = i - 1
        while j >= left and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

def merge(arr, left, mid, right):
    len1, len2 = mid - left + 1, right - mid
    left_arr, right_arr = arr[left:mid+1], arr[mid+1:right+1]

    i, j, k = 0, 0, left

    while i < len1 and j < len2:
        if left_arr[i] <= right_arr[j]:
            arr[k] = left_arr[i]
            i += 1
        else:
            arr[k] = right_arr[j]
            j += 1
        k += 1

    while i < len1:
        arr[k] = left_arr[i]
        i += 1
        k += 1

    while j < len2:
        arr[k] = right_arr[j]
        j += 1
        k += 1

def tim_sort(arr):
    n = len(arr)
    min_run = 32  # Tamanho mínimo para runs

    # Ordena runs individuais usando insertion sort
    for i in range(0, n, min_run):
        insertion_sort_range(arr, i, min(i + min_run - 1, n - 1))

    # Mescla runs
```

```
    size = min_run
    while size < n:
        for left in range(0, n, 2 * size):
            mid = min(n - 1, left + size - 1)
            right = min(left + 2 * size - 1, n - 1)
            if mid < right:
                merge(arr, left, mid, right)
        size *= 2

    return arr
```

Complexidade:

- **Tempo:**
  - Melhor caso: O(n)
  - Caso médio: O(n log n)
  - Pior caso: O(n log n)
- **Espaço:** O(n)

Características:

- Algoritmo estável
- Adaptativo - aproveita ordenação parcial no array
- Eficiente para dados do mundo real
- Tem bom desempenho com diversos tipos de dados
- Algoritmo bastante complexo na implementação completa
- Default em muitas linguagens como Python e Java

11. Shell Sort

Descrição: Shell Sort é uma extensão do Insertion Sort que permite a troca de elementos distantes entre si, reduzindo gradualmente o intervalo entre os elementos até ordenar os dados.

Implementação:

python

```python
def shell_sort(arr):
    n = len(arr)
    gap = n // 2

    while gap > 0:
        for i in range(gap, n):
            # Insertion sort com gap
            temp = arr[i]
            j = i

            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap

            arr[j] = temp

        gap //= 2

    return arr
```

Complexidade:

- **Tempo:** Depende da sequência de gaps
  - Melhor caso: O(n log n) - com sequência de gaps otimizada
  - Caso médio: O(n^1.3) - com sequência de gaps otimizada
  - Pior caso: $O(n^2)$
- **Espaço:** O(1) - ordenação in-place

Características:

- Não é estável
- Melhora substancialmente o Insertion Sort
- Desempenho depende da sequência de gaps escolhida
- Simples de implementar
- Eficiente para arrays de tamanho médio

Comparação e Quando Usar Cada Algoritmo

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Estável	In-Place	Quando Usar
Bubble Sort	O(n)	O(n^2)	O(n^2)	Sim	Sim	Conjuntos muito pequenos ou quase ordenados
Selection Sort	O(n^2)	O(n^2)	O(n^2)	Não	Sim	Quando o custo de troca é alto, para conjuntos pequenos
Insertion Sort	O(n)	O(n^2)	O(n^2)	Sim	Sim	Pequenos conjuntos, arrays quase ordenados
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Sim	Não	Grandes conjuntos, quando estabilidade é necessária
Quick Sort	O(n log n)	O(n log n)	O(n^2)	Não	Sim*	Propósito geral, grande volume de dados
Heap Sort	O(n log n)	O(n log n)	O(n log n)	Não	Sim	Quando é necessário garantir O(n log n) e espaço constante
Counting Sort	O(n + k)	O(n + k)	O(n + k)	Sim	Não	Pequeno intervalo de valores inteiros
Radix Sort	O(d*(n + k))	O(d*(n + k))	O(d*(n + k))	Sim	Não	Valores inteiros com número limitado de dígitos
Bucket Sort	O(n + k)	O(n + k)	O(n^2)	Sim*	Não	Dados uniformemente distribuídos
Tim Sort	O(n)	O(n log n)	O(n log n)	Sim	Não	Algoritmo de propósito geral, dados do mundo real
Shell Sort	O(n log n)*	O(n^1.3)*	O(n^2)	Não	Sim	Arrays de tamanho médio, implementação simples

* Depende da implementação ou da sequência de gaps

Considerações para Maratonas de Programação

1. **Use os algoritmos da biblioteca padrão:**
   - Em Python: `sorted()` ou `list.sort()` (Timsort)
   - Em C++: `std::sort()` (IntroSort - híbrido de QuickSort, HeapSort e InsertionSort)
   - Em Java: `Arrays.sort()` ou `Collections.sort()` (TimSort para objetos, quicksort para primitivos)
2. **Implementação manual:**
   - Priorize algoritmos O(n log n) como Merge Sort ou Quick Sort para problemas gerais
   - Use Counting Sort ou Radix Sort quando aplicável (valores inteiros pequenos)

- Insertion Sort para arrays muito pequenos (n < 20)

3. **Considerações especiais:**
   - Se a estabilidade for importante, use Merge Sort
   - Se o espaço adicional for limitado, considere Heap Sort ou Quick Sort
   - Para ordenação parcial (top k elementos), considere um heap
4. **Otimizações comuns:**
   - Combine algoritmos (como no TimSort)
   - Use Insertion Sort para partições pequenas no Quick Sort
   - Escolha inteligente de pivô no Quick Sort (mediana de três)
   - Implemente versões não-recursivas para evitar estouro de pilha

Dominando esses algoritmos e sabendo quando aplicar cada um, você estará bem preparado para resolver problemas de ordenação em maratonas de programação.