Algoritmos de Ordenação

Introdução

Os algoritmos de ordenação são fundamentais na ciência da computação e têm como objetivo rearranjar os elementos de uma estrutura de dados (geralmente um vetor ou lista) em uma determinada ordem, que pode ser crescente, decrescente ou seguindo algum outro critério específico. Esses algoritmos são amplamente utilizados em diversas aplicações, desde sistemas de banco de dados até interfaces de usuário, e constituem uma base importante para outros algoritmos mais complexos.

Este material apresenta os principais algoritmos de ordenação, suas características, implementações, análises de complexidade e aplicações. Compreender esses algoritmos é essencial para qualquer profissional da área de computação, pois eles ilustram conceitos fundamentais de eficiência algorítmica e técnicas de programação.

Conceitos Básicos de Ordenação

A ordenação é a operação de rearranjar os dados em uma determinada ordem. Formalmente, dado um conjunto de n elementos $\{a_1, a_2, ..., a_n\}$, o objetivo é encontrar uma permutação (reordenação) $\{a'_1, a'_2, ..., a'_n\}$ tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$.
Alguns conceitos importantes relacionados à ordenação incluem:
1. Estabilidade: Um algoritmo de ordenação é estável quando mantém a ordem relativa de elementos iguais. Isso significa que se dois elementos têm o mesmo valor e um aparece antes do outro no vetor original, após a ordenação, o elemento que estava originalmente antes continuará antes.
2. In-place: Um algoritmo é considerado in-place quando requer apenas uma quantidade constante de espaço adicional para realizar a ordenação, ou seja, ele modifica diretamente a estrutura de dados original sem necessitar de estruturas auxiliares proporcionais ao tamanho da entrada.
3. Adaptabilidade: Um algoritmo é adaptativo quando seu desempenho melhora quando os dados já estão parcialmente ordenados.
4. Comparações: A maioria dos algoritmos de ordenação baseia-se em comparações entre elementos. Alguns algoritmos, no entanto, utilizam outras propriedades dos dados para realizar a ordenação.

Classificação dos Algoritmos de Ordenação

Os algoritmos de ordenação podem ser classificados de várias formas:

1. Baseados em comparação: - Selection Sort - Bubble Sort - Insertion Sort - Merge Sort - Quick Sort - Heap Sort - Shell Sort	2. Não baseados em comparação: - Counting Sort - Radix Sort - Bucket Sort - Pigeonhole Sort	3. Algoritmos híbridos: - Tim Sort (combinação de Insertion Sort e Merge Sort) - Intro Sort (combinação de Quick Sort, Heap Sort e Insertion Sort)

Algoritmos de Ordenação Elementares

1. Bubble Sort (Ordenação por Bolha)

O Bubble Sort é um dos algoritmos de ordenação mais simples. Ele funciona comparando pares de elementos adjacentes e trocando-os se estiverem na ordem errada. Este processo é repetido até que nenhuma troca seja necessária, indicando que o vetor está ordenado.

Implementação em C:

```
void bubbleSort(int arr[], int n) {
```

```
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Troca arr[j] e arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Complexidade:
- Tempo: O(n²) no pior e caso médio, O(n) no melhor caso (quando o vetor já está ordenado)
- Espaço: O(1)

Características:
- Estável
- In-place
- Simples de implementar
- Ineficiente para grandes conjuntos de dados

2. Selection Sort (Ordenação por Seleção)

O Selection Sort funciona selecionando repetidamente o menor (ou maior) elemento da parte não ordenada do vetor e movendo-o para a parte ordenada.

Implementação em C:

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_idx = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }
        // Troca o elemento mínimo encontrado com o primeiro elemento
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Complexidade:
- Tempo: O(n²) em todos os casos
- Espaço: O(1)

Características:
- Não estável (pode ser implementado de forma estável com custo adicional)
- In-place
- Simples de implementar
- Faz menos trocas que o Bubble Sort

3. Insertion Sort (Ordenação por Inserção)

O Insertion Sort constrói a sequência ordenada um elemento por vez, inserindo cada novo elemento na posição correta entre os elementos já ordenados.

Implementação em C:

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move os elementos de arr[0..i-1] que são maiores que key
        // para uma posição à frente de sua posição atual
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Complexidade:
- Tempo: O(n²) no pior e caso médio, O(n) no melhor caso (quando o vetor já está ordenado)
- Espaço: O(1)

Características:
- Estável
- In-place
- Eficiente para pequenos conjuntos de dados
- Adaptativo (eficiente para dados parcialmente ordenados)
- Usado como parte de algoritmos mais complexos como o Shell Sort e o Tim Sort

Algoritmos de Ordenação Eficientes

1. Merge Sort (Ordenação por Intercalação)

O Merge Sort é um algoritmo baseado na técnica de divisão e conquista. Ele divide o vetor em duas metades, ordena cada metade recursivamente e depois mescla as duas metades ordenadas.

Implementação em C:

```c
// Função para mesclar duas subpartes ordenadas
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    // Cria arrays temporários
    int L[n1], R[n2];

    // Copia dados para arrays temporários
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Mescla os arrays temporários de volta em arr[l..r]
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copia os elementos restantes de L[], se houver
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copia os elementos restantes de R[], se houver
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
// Função principal do Merge Sort
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2; // Ponto médio

        // Ordena primeira e segunda metades
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Mescla as metades ordenadas
        merge(arr, l, m, r);
    }
}
```

Complexidade:
- Tempo: O(n log n) em todos os casos
- Espaço: O(n)

Características:
- Estável
- Não in-place (requer espaço adicional)
- Eficiente para grandes conjuntos de dados
- Previsível (sempre O(n log n))

2. Quick Sort (Ordenação Rápida)

O Quick Sort também utiliza a técnica de divisão e conquista. Ele seleciona um elemento como pivô e particiona o vetor ao redor do pivô, de modo que elementos menores fiquem à esquerda e elementos maiores à direita. Em seguida, ordena recursivamente as duas partições.

Implementação em C:

```c
// Função para trocar dois elementos
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
// Função de particionamento
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Pivô
    int i = (low - 1); // Índice do menor elemento

    for (int j = low; j <= high - 1; j++) {
        // Se o elemento atual é menor ou igual ao pivô
        if (arr[j] <= pivot) {
            i++; // Incrementa o índice do menor elemento
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
// Função principal do Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // pi é o índice de particionamento
        int pi = partition(arr, low, high);

        // Ordena elementos antes e depois da partição
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Complexidade:
- Tempo: O(n²) no pior caso, O(n log n) no caso médio e melhor caso
- Espaço: O(log n) devido à pilha de recursão

Características:
- Não estável (pode ser implementado de forma estável com custo adicional)
- In-place
- Geralmente mais rápido na prática que outros algoritmos O(n log n)
- Sensível à escolha do pivô

3. Heap Sort (Ordenação por Heap)

O Heap Sort utiliza uma estrutura de dados chamada heap (geralmente um heap máximo) para ordenar os elementos. Ele constrói um heap a partir do vetor e, em seguida, extrai repetidamente o elemento máximo, reconstruindo o heap após cada extração.

Implementação em C:

```c
// Função para ajustar um subárvore com raiz em i
void heapify(int arr[], int n, int i) {
    int largest = i; // Inicializa o maior como raiz
    int l = 2 * i + 1; // Esquerda = 2*i + 1
    int r = 2 * i + 2; // Direita = 2*i + 2

    // Se o filho da esquerda é maior que a raiz
```

```c
// Função principal do Heap Sort
void heapSort(int arr[], int n) {
    // Constrói o heap (rearranjo do array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extrai um por um elemento do heap
    for (int i = n - 1; i > 0; i--) {
        // Move a raiz atual para o fim
```

```
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // Se o filho da direita é maior que o maior até agora
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // Se o maior não é a raiz
    if (largest != i) {
        swap(&arr[i], &arr[largest]);

        // Recursivamente heapify a subárvore afetada
        heapify(arr, n, largest);
    }
}
```

```
        swap(&arr[0], &arr[i]);

        // Chama max heapify no heap reduzido
        heapify(arr, i, 0);
    }
}
```

Complexidade:
- Tempo: O(n log n) em todos os casos
- Espaço: O(1)

Características:
- Não estável
- In-place
- Eficiente para grandes conjuntos de dados
- Previsível (sempre O(n log n))

Algoritmos de Ordenação Não Baseados em Comparação

1. Counting Sort (Ordenação por Contagem)

O Counting Sort é um algoritmo que funciona contando o número de ocorrências de cada elemento e usando essa informação para posicionar os elementos na sequência ordenada. É eficiente quando o intervalo de valores possíveis não é significativamente maior que o número de elementos a serem ordenados.

Implementação em C:

```
void countingSort(int arr[], int n, int max) {
    int output[n]; // Array de saída
    int count[max + 1]; // Array de contagem

    // Inicializa o array de contagem
    for (int i = 0; i <= max; ++i)
        count[i] = 0;

    // Armazena a contagem de cada elemento
    for (int i = 0; i < n; ++i)
```

```
        count[arr[i]]++;

    // Altera count[i] para que contenha a posição atual do elemento i no array de saída
    for (int i = 1; i <= max; ++i)
        count[i] += count[i - 1];

    // Constrói o array de saída
    for (int i = n - 1; i >= 0; --i) {
        output[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
    }

    // Copia o array de saída para arr[]
    for (int i = 0; i < n; ++i)
        arr[i] = output[i];
}
```

Complexidade:
- Tempo: O(n + k), onde k é o intervalo de valores
- Espaço: O(n + k)

Características:
- Estável
- Não in-place
- Eficiente quando o intervalo de valores é pequeno
- Não baseado em comparações

2. Radix Sort (Ordenação por Raiz)

O Radix Sort ordena os elementos processando-os dígito por dígito. Ele pode ser implementado usando qualquer algoritmo de ordenação estável como sub-rotina, mas geralmente usa o Counting Sort.

Implementação em C:

```
// Função para encontrar o maior número no array
int getMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
// Função para fazer o counting sort do array de acordo com o dígito representado por exp
void countSort(int arr[], int n, int exp) {
    int output[n]; // Array de saída
    int count[10] = {0};

    // Constrói o array de saída
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copia o array de saída para arr[]
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}
// Função principal do Radix Sort
void radixSort(int arr[], int n) {
    // Encontra o número máximo para saber o número de dígitos
    int m = getMax(arr, n);
```

```
    // Armazena a contagem de ocorrências em count[]
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Altera count[i] para que contenha a posição atual do dígito no output[]
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];
```

```
    // Faz o counting sort para cada dígito
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```

Complexidade:
- Tempo: O(d * (n + k)), onde d é o número de dígitos e k é o intervalo de valores por dígito
- Espaço: O(n + k)

Características:
- Estável
- Não in-place
- Eficiente para números com poucos dígitos
- Não baseado em comparações

3. Bucket Sort (Ordenação por Balde)

O Bucket Sort divide o intervalo de valores em um número de baldes, distribui os elementos nos baldes apropriados, ordena cada balde individualmente (geralmente com outro algoritmo de ordenação) e, em seguida, concatena os baldes.

Implementação em C (simplificada para números entre 0 e 1):

```
void bucketSort(float arr[], int n) {
    // Cria n baldes vazios
    struct Node* buckets[n];
    for (int i = 0; i < n; i++)
        buckets[i] = NULL;

    // Coloca elementos do array nos baldes
    for (int i = 0; i < n; i++) {
        int bucketIndex = n * arr[i]; // Índice do balde
        // Insere no início da lista do balde
        struct Node* current = createNode(arr[i]);
        current->next = buckets[bucketIndex];
        buckets[bucketIndex] = current;
    }
```

```
    // Ordena cada balde e coloca de volta no array
    int index = 0;
    for (int i = 0; i < n; i++) {
        // Ordena o balde (pode usar insertion sort)
        buckets[i] = insertionSortList(buckets[i]);

        // Coloca os elementos do balde de volta no array
        struct Node* current = buckets[i];
        while (current != NULL) {
            arr[index++] = current->data;
            current = current->next;
        }
    }
}
```

Complexidade:
- Tempo: O(n²) no pior caso, O(n + k) no caso médio (onde k é o número de baldes)
- Espaço: O(n + k)

Características:
- Estável (dependendo do algoritmo usado para ordenar os baldes)
- Não in-place
- Eficiente para distribuições uniformes
- Pode usar diferentes algoritmos para ordenar cada balde

Comparação entre Algoritmos de Ordenação

A escolha do algoritmo de ordenação adequado depende de vários fatores, como o tamanho dos dados, a distribuição dos valores, a estabilidade necessária e as restrições de memória. A tabela a seguir resume as características dos principais algoritmos:

Algoritmo	**Melhor Caso**	**Caso Médio**	**Pior Caso**	**Espaço**	**Estável**	**In-place**
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)	Sim	Sim
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)	Não	Sim
Insertion Sort	O(n)	O(n²)	O(n²)	O(1)	Sim	Sim
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Sim	Não
Quick Sort	O(n log n)	O(n log n)	O(n²)	O(log n)	Não	Sim
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	Não	Sim
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(n + k)	Sim	Não
Radix Sort	O(d(n + k))	O(d(n + k))	O(d(n + k))	O(n + k)	Sim	Não
Bucket Sort	O(n + k)	O(n + k)	O(n²)	O(n + k)	Sim	Não

Aplicações dos Algoritmos de Ordenação

Os algoritmos de ordenação têm inúmeras aplicações em diversas áreas da computação:
1. Bancos de dados: Para indexação e consultas eficientes.
2. Processamento de texto: Para ordenar palavras alfabeticamente.
3. Computação gráfica: Para ordenar objetos por profundidade (z-buffer).
4. Compressão de dados: Alguns algoritmos de compressão usam ordenação como pré-processamento.
5. Análise de dados: Para facilitar a busca e visualização de informações.
6. Sistemas operacionais: Para gerenciamento de processos e recursos.
7. Redes de computadores: Para roteamento e gerenciamento de pacotes.

Boas Práticas na Implementação de Algoritmos de Ordenação

1. Escolha o algoritmo adequado para o problema específico, considerando o tamanho dos dados, a distribuição dos valores e as restrições de memória.
2. Para pequenos conjuntos de dados (n < 50), algoritmos simples como Insertion Sort podem ser mais eficientes devido à baixa sobrecarga.
3. Para grandes conjuntos de dados, prefira algoritmos com complexidade O(n log n) como Merge Sort, Quick Sort ou Heap Sort.
4. Se a estabilidade for importante, escolha algoritmos estáveis como Merge Sort ou Insertion Sort.
5. Se o espaço for uma restrição, prefira algoritmos in-place como Quick Sort ou Heap Sort.
6. Para dados com intervalo limitado, considere algoritmos não baseados em comparação como Counting Sort ou Radix Sort.
7. Implemente otimizações específicas para o seu caso de uso, como a escolha do pivô no Quick Sort ou o tamanho dos baldes no Bucket Sort.

Ordenação em Memória Secundária

Quando os dados são muito grandes para caber na memória RAM, é necessário utilizar técnicas de ordenação em memória secundária, como discos rígidos ou SSDs.

External Sorting

O External Sorting permite ordenar grandes volumes de dados dividindo-os em blocos menores, que são ordenados separadamente na memória primária e depois combinados de forma eficiente.

Merge Sort Externo

O Merge Sort Externo é uma variação do Merge Sort adaptada para dados armazenados em disco. Ele divide o conjunto de dados em partes que cabem na memória, ordena essas partes e as mescla progressivamente.
Exemplo: Suponha que temos 10 GB de dados para ordenar e apenas 1 GB de memória disponível. O Merge Sort Externo divide os dados em 10 blocos de 1 GB, ordena cada um na memória e depois mescla os blocos em uma única sequência ordenada.
Aplicação: Usado em bancos de dados, sistemas de arquivos e grandes data centers, onde a ordenação precisa ser feita em discos rígidos devido ao tamanho dos dados.

Conclusão

Os algoritmos de ordenação são ferramentas fundamentais na ciência da computação, com aplicações em praticamente todas as áreas que envolvem processamento de dados. Compreender as características, vantagens e limitações de cada algoritmo permite escolher a solução mais adequada para cada problema específico.

Embora existam algoritmos com complexidade teórica ótima de O(n log n) para ordenação baseada em comparações, não existe um "melhor" algoritmo universal. A escolha depende das características dos dados e dos requisitos específicos da aplicação.
O estudo dos algoritmos de ordenação também fornece insights valiosos sobre técnicas de projeto de algoritmos, como divisão e conquista, uso de estruturas de dados auxiliares e análise de complexidade, que são aplicáveis a uma ampla gama de problemas computacionais.

Referências

- 📖 Livros e Apostilas
  - CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Introduction to Algorithms. MIT Press.
  - SEDGEWICK, R.; WAYNE, K. Algorithms. Addison-Wesley Professional.
  - KNUTH, D. E. The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley.
  - COELHO, H.; FÉLIX, N. Métodos de Ordenação: Selection, Insertion, Bubble, Merge (Sort). UFG.
  - MENEZES, P. B. Matemática Discreta para Computação e Informática. Bookman.
- 🌐 Recursos Online
  - GeeksforGeeks - Sorting Algorithms - https://www.geeksforgeeks.org/sorting-algorithms/
  - VisuAlgo - Sorting - https://visualgo.net/en/sorting
  - IME-USP - Correção e desempenho de algoritmos básicos de ordenação - https://www.ime.usp.br/~pf/algoritmos/aulas/ordena.html
  - Khan Academy - Algoritmos de Ordenação
  - Sorting Algorithms Animations - https://www.toptal.com/developers/sorting-algorithms
- 📽️ Vídeos e Cursos
  - 15 Sorting Algorithms in 6 Minutes - https://www.youtube.com/watch?v=kPRA0W1kECg
  - Algoritmo SELECTION SORT | Algoritmos de Ordenação | Algoritmos #3

Anexo - Resumo Estruturado/Mapa Mental Gerado por IA

📌 Introdução

- Reorganização de elementos em uma ordem definida (crescente, decrescente, etc.)
- Base para outros algoritmos complexos
- Usados em: bancos de dados, sistemas operacionais, análise de dados, interfaces de usuário

---

🔑 Conceitos Básicos

- **Estabilidade** → mantém ordem relativa de elementos iguais
- **In-place** → usa pouca memória extra
- **Adaptabilidade** → desempenho melhora em dados parcialmente ordenados
- **Comparações** → maioria dos algoritmos depende de comparações

---

🗂️ Classificação dos Algoritmos

1. **Baseados em comparação**
    - Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Shell Sort
2. **Não baseados em comparação**
    - Counting Sort, Radix Sort, Bucket Sort, Pigeonhole Sort
3. **Híbridos**
    - Tim Sort (Insertion + Merge)
    - Intro Sort (Quick + Heap + Insertion)

🧩 Algoritmos Elementares

- **Bubble Sort** → Simples, estável, $O(n^2)$
- **Selection Sort** → Poucas trocas, não estável, $O(n^2)$
- **Insertion Sort** → Bom para pequenos ou parcialmente ordenados, estável, $O(n^2)$

⚡ Algoritmos Eficientes

- **Merge Sort** → Divisão e conquista, estável, $O(n \log n)$, não in-place
- **Quick Sort** → Pivô, muito usado na prática, $O(n \log n)$ médio, $O(n^2)$ pior caso
- **Heap Sort** → Usa heap, $O(n \log n)$, in-place, não estável

---

📊 Não Baseados em Comparação

- **Counting Sort** → Usa contagem, $O(n + k)$, eficiente se intervalo pequeno
- **Radix Sort** → Ordena por dígitos, usa Counting Sort como sub-rotina

- **Bucket Sort** → Distribui em baldes, eficiente para distribuições uniformes

---

⚖️ Comparação entre Algoritmos

- **Tempo de execução** (melhor, médio, pior caso)
- **Uso de memória**
- **Estabilidade**
- **Adequação ao tamanho e distribuição dos dados**

🛠️ Boas Práticas

- Escolher algoritmo conforme:
  - Tamanho do conjunto
  - Distribuição dos dados
  - Estabilidade necessária
  - Restrições de memória
- Usar híbridos em cenários práticos
- Testar desempenho em diferentes cenários

💾 Ordenação em Memória Secundária

- **External Sorting** → divide dados em blocos menores e mescla depois
- **Merge Sort Externo** → usado em bancos de dados e grandes data centers

---

🎯 Conclusão

- Ordenação é fundamental para computação
- Não existe um "melhor" algoritmo universal
- Escolha depende dos dados e da aplicação
- Fornece base para compreender eficiência algorítmica e técnicas como divisão e conquista