Análise Comparativa de Algoritmos de Ordenação

Este trabalho tem como objetivo aplicar e comparar os pricipais tipos de Algoritmos de ordenação

Selection Sort

Consiste em um algoritmo de classificação baseado em comparação. Ele classifica uma matriz por meio da repetida troca de posição do menor elemento não classificado pela menor posição disponível.

```
[7, 5,'1', 8, 3] // Menor elemento
['1', 5, 7, 8, 3] // Menor elemento -> índice 0
[1,'3', 7, 8, 5] // Segundo menor -> índice 1
[1, 3, '5', 8, 7] // ...
[1, 3, 5, '7', 8] // ...
```

Implementação do Selection Sort

- Percorra toda a lista e descubra o **`menor elemento`**
- Troque de posição o **`menor elemento`** com o **`primeiro elemento`**
- Encontre o **`menor elemento (segundo menor)`** entre os elementos restantes e trocamos com o **`segundo elemento`** do vetor
- Continue o processo até que todo os elementos estejam classificados corretamente

```
void selectionSort(vector<int> &arr) {
    int n = arr.size();

    for (int i = 0; i < n - 1; ++i) {

        int min_idx = i;

        // Intera através da posição não classificada
        // para encontrar o mínimo atual
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[min_idx]) {

                // Atualiza o minimo se for menor
                // para o elemento encontrado
                min_idx = j;
            }
        }

        // Move o mínimo para a posição correta
        swap(arr[i], arr[min_idx]);
    }
}
```

Complexidade

`Complexidade de tempo: O(n²) -> Dois loops aninhados`

- Loop para selecionar o **menor elemento do array = O(n)**
- Outro loop para **comparar esse elemento com os demais = O(n)**
- **Compelxidade de tempo: O(n) * O(n) = O(n²)**

Complexidade Espacial

`O(1)` é a **única memoria extra utilizada** para variáveis temporárias

Vantagens

- `Facil implementação`
- `Requer apens um espaço de memoria extra O(1)`
- `Requer um número baixo de trocas se comparado com outros algoritmos de seleção`

Desvantagens

- **Possui um tempo de complexidade alto** `O(n²)` se comparado a algoritmos como `Quick Sort` e `Merge Sort`
- `Não é estável` - Não mantém ordem relativa de elementos iguais

Aplicações

Recomendado para conjunto de `dados pequenos` e em situações de recursos de `memória limitados`

Isertion Sort

Consiste na classificação por meio da inserção de forma interativa de cada elemento de uma lista não classificada em sua posição correte em uma paerte classificadas da lista

```
[23, 1, 10, 5, 2]
[23, '1', 10, 5, 2] // Elemento inicial
[23, '1', 10, 5, 2] // Parte classificada
[*23*, '1', 10, 5, 2] // Compara 1 com 23
[*1, 23*, 10, 5, 2] // Parte classificada
[*1, 23*, '10', 5, 2] // Compara 10 com 1 e 23. Insere 10 entre 1 e 23
[*1, 10, 23*, 5, 2] // Parte classificada
[*1, 10, 23*, '5', 2] // Compara 5 com a parte classificada e insere entre
1 e 5
[*1, 5, 10, 23*, 2] // Parte classificada
[*1, 5, 10, 23*, '2'] // Compara 2 com a parte classificada e insere entre
1 e 5
[*1, 2, 5, 10, 23*] // Array classificado
```

Implementação

- Começa com o segundo elemento da matriz, considerando o primeiro elemento como classificado
- Compara o segundo elemento com o primeiro, se o segundo for menor, troque-os de posição
- Compare o terceiro elemento com os anteriores e coloque-o na posição correta
- Repita até que toda a amtriz esteja classifcada

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

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Complexidade do Insertion Sort

- **`Melhor caso: O(n)`**, Lista ordenada
- **`Caso médio: O(n²)`**, Lista ordenada aleatoriamente
- **`Pior caso: O(n²)`**, lista na **`ordem inversa`**

Complexidade Espacial

`O(1)`, o Insertion Sort requer apenas **O(1)** espaço adicional, semelhantemente ao **`Selection Sort`**

Vantagens

- **`Facil implementação`**
- **`Estável`**
- **`Baixa Complexidade Espacial`**, algoritmo in-place
- **`Número de inversões diretamente proporcional ao número de swaps`**

Desvantagens

- Ineficiente para elevados números de dados
- Não é tão eficiente quando outros algoritmos como Merge Sort e Quick Sort para a maioria dos casos

Bubble Sort

Consiste na troca repetidamente dos elementos adjacentes se eles estiverem na ordem errada. Esse algoritmo não é recomendado para grandes volumes de dados, pois a compelxidade de tempo média e para

o pior caso é alta

Implementação

- Classifiuqe

```
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    bool swapped;

    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }

        // Se não houver dois elementos trocados, break
        if (!swapped)
            break;
    }
}
```

Complexidade

- **`Melhor caso: O(n)`**, Lista ordenada
- **`Caso médio: O(n²)`**, Lista ordenada aleatoriamente
- **`Pior caso: O(n²)`**, lista na **`ordem decrescente`**

Complexidade Espacial

`O(1)`, o Bubble Sort requer apenas **O(1)** espaço adicional, semelhantemente ao **`Selection Sort`** e ao **`Insertion Sort`**. Porém o **`Bubble Sort`** precisa de um quantidade constante de espaço adicional

Vantagens do Bubble Sort

- **`Facil implementação`**
- **`Estável`**
- **`Baixa Complexidade Espacial`**

Desvantagens

- **`Ruim para grande conjunto de dados - Complexidade média O(n²)`**

Quick Sort

Consiste em um algoritmo de classificação baseada no **Dividir para conquistar** que escolhe um elemento como um pivô e particioa a matriz fornecida em torno do pivô escolhido, colocando o pivô em sua posição correta na matriz classificada

Implementação

- Escolha um pivô
- Partição da matriz: Reorganiza a matriz ao redor do pivô. Após o particionamento, todos os elementos menores que o pivô estarão à sua esquerda e os menores à sua direita. O pivô fica na posição correta e obtemos o índice do pivô
- Chame recurssivamete: Aplique recurssivamente o mesmo processo às matrizes particionadas
- Caso base: A recursão para quando resta apenas um elemento na submatriz, pois um único element já está classificado

```
int partition(vector<int>& arr, int low, int high) {

    // Choose the pivot
    int pivot = arr[high];

    // Index of smaller element and indicates
    // the right position of pivot found so far
    int i = low - 1;

    // Traverse arr[low..high] and move all smaller
    // elements on left side. Elements from low to
    // i are smaller after every iteration
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    // Move pivot after smaller elements and
    // return its position
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

// The QuickSort function implementation
void quickSort(vector<int>& arr, int low, int high) {

    if (low < high) {

        // pi is the partition return index of pivot
        int pi = partition(arr, low, high);

        // Recursion calls for smaller elements
        // and greater or equals elements
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
```

```
        }
    }
```

Complexidade

- `Melhor caso: O(log n)`, Partições balanceadas - Cada partição com n/2 elementos
- `Caso médio: O(n log n)`, O pivô divide a matriz em duas partes, mas não necessariamente iguais
- `Pior caso: O(n²)`, Ocorre quando o menor ou o maior elemento é sempre escolhido como pivô (vetor classificado)

COmplexidade espacial

`O(n)`, devido a pilha de chamadas recursivas

Vantagens

- Eficiente paara grande volume de dados
- Sobrecarga baixa
- Não precisa de matriz auxiliar
- Algorimo de uso geral mais rápido quando a estabilidade não é necessária

Desvantagens

- Complexidade alta no piro caso
- Não é uma boa escolha para algoritmos pequenos
- Não é estável, o que significa que se dois elementos tiverem a mesma chave, sua ordem relativa não será preservada na saída classificada em caso de classificação rápida, porque aqui estamos trocando elementos de acordo com a posição do pivô (sem considerar seu original posições)