UNIVERSIDADE FEDERAL DE RORAIMA

CENTRO DE CIÊNCIA E TECNOLOGIA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

DCC302 - ESTRUTURAS DE DADOS I - 2024.2

FILIPE DWAN PEREIRA

LEONARDO VINÍCIUS LIMA CASTRO

ARTHUR CORREIA DE OLIVEIRA RAMOS

ALGORITMOS DE ORDENAÇÃO

BOA VISTA, RR

2025

LEONARDO VINÍCIUS LIMA CASTRO

ARTHUR CORREIA DE OLIVEIRA RAMOS

ALGORITMOS DE ORDENAÇÃO

Trabalho da disciplina de Estrutura de Dados I do ano de 2024.2 apresentado à Universidade Federal de Roraima do curso de Bacharelado em Ciência da Computação.

Docente: Dr. Filipe Dwan Pereira

BOA VISTA, RR

2025

INTRODUÇÃO

Os algoritmos de ordenação e busca são essenciais para organizar e encontrar informações de forma eficiente. Alguns métodos, como Bubble Sort e Insertion Sort, são mais simples, mas lentos para grandes conjuntos de dados. Outros, como Quick Sort e Merge Sort, são mais rápidos e usados em aplicações reais. Já os algoritmos de busca, como Busca Sequencial e Busca Binária, ajudam a encontrar dados de maneira eficiente, dependendo se a lista está ordenada ou não. Cada algoritmo tem suas vantagens e desvantagens, e a escolha do melhor depende da situação.

ALGORITMOS

Selection Sort

O Selection Sort é um algoritmo de ordenação por seleção que busca repetidamente o menor (ou maior) elemento do vetor e o coloca na posição correta. Ele divide o vetor em uma parte ordenada e outra não ordenada, selecionando o menor elemento e trocando-o com o primeiro elemento não ordenado.

Abaixo será apresentado o algoritmo em C:

```
void selectionSort(int v[], int n) {

    int i, j, min, temp;

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

        min = i;

        for (j = i + 1; j < n; j++) {

            if (v[j] < v[min]) {

                min = j;

            }

        }

        temp = v[i];

        v[i] = v[min];

        v[min] = temp;

    }

}
```

Vantagens:

- Simples de implementar.
- Não requer memória extra além do vetor original.
- Útil para pequenas listas.

Desvantagens:

- Desempenho $O(n^2)$ no pior e no melhor caso.
- Ineficiente para listas grandes.
- Pouco adaptável a entradas parcialmente ordenadas.

Bubble Sort

O Bubble Sort funciona comparando elementos adjacentes e trocando-os caso estejam na ordem errada. Esse processo é repetido até que a lista esteja completamente ordenada.

Abaixo será apresentado o algoritmo em C:

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

    int i, j, temp;

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

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

            if (v[j] > v[j + 1]) {

                temp = v[j];

                v[j] = v[j + 1];

                v[j + 1] = temp;

            }

        }

    }

}
```

Vantagens:

- Simples e fácil de entender.
- Pode detectar listas já ordenadas rapidamente (com uma versão otimizada).

Desvantagens:

- Ineficiente para listas grandes, com $O(n^2)$ no pior e médio caso.
- Muitas trocas de elementos, tornando-o mais lento que Selection Sort para algumas entradas.

Radix Sort

O Radix Sort é um algoritmo de ordenação não comparativo que classifica os números por seus dígitos, utilizando contagem para ordenar em cada posição decimal. Ele é eficiente para ordenar inteiros e cadeias de caracteres de tamanho fixo.

Abaixo será apresentado o algoritmo em C:

```
#define MAX 10

void countingSort(int v[], int n, int exp) {
    int output[n], count[MAX] = {0}, i;

    for (i = 0; i < n; i++)
        count[(v[i] / exp) % 10]++;

    for (i = 1; i < MAX; i++)
        count[i] += count[i - 1];

    for (i = n - 1; i >= 0; i--) {
        output[count[(v[i] / exp) % 10] - 1] = v[i];
        count[(v[i] / exp) % 10]--;
    }

    for (i = 0; i < n; i++)
        v[i] = output[i];
}

void radixSort(int v[], int n) {
    int max = v[0], exp;
    for (int i = 1; i < n; i++)
        if (v[i] > max)
            max = v[i];
```

```
    for (exp = 1; max / exp > 0; exp *= 10)
        countingSort(v, n, exp);
}
```

Vantagens:

- Complexidade O(nk), onde *k* é o número de dígitos do maior número, o que o torna mais rápido que algoritmos quadráticos para certos casos.
- Ideal para ordenar grandes quantidades de inteiros.

Desvantagens:

- Não funciona bem para tipos de dados que não podem ser representados como números fixos.
- Requer memória extra para os buckets.
- Depende da base escolhida para os números.

Merge Sort

O Merge Sort é um algoritmo de ordenação baseado na estratégia Dividir para Conquistar, dividindo recursivamente a lista em partes menores até que cada parte tenha apenas um elemento, e então mesclando as partes ordenadas.

Abaixo será apresentado o algoritmo em C:

```
void merge(int v[], int esq, int meio, int dir) {
    int i, j, k;
    int n1 = meio - esq + 1;
    int n2 = dir - meio;
    int E[n1], D[n2];
    for (i = 0; i < n1; i++)
        E[i] = v[esq + i];
    for (j = 0; j < n2; j++)
        D[j] = v[meio + 1 + j];
    i = 0;
    j = 0;
```

```
    k = esq;
    while (i < n1 && j < n2) {
        if (E[i] <= D[j]) {
            v[k] = E[i];
            i++;
        } else {
            v[k] = D[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        v[k] = E[i];
        i++;
        k++;
    }
    while (j < n2) {
        v[k] = D[j];
        j++;
        k++;
    }
}
void mergeSort(int v[], int esq, int dir) {
    if (esq < dir) {
        int meio = esq + (dir - esq) / 2;
```

```
        mergeSort(v, esq, meio);

        mergeSort(v, meio + 1, dir);

        merge(v, esq, meio, dir);

    }

}
```

Vantagens:

- Complexidade O(n log n) no pior, médio e melhor caso.
- Estável, ou seja, mantém a ordem relativa dos elementos iguais.
- Bom para ordenar grandes conjuntos de dados.

Desvantagens:

- Requer memória extra O(n) para armazenar os subarrays temporários.
- Pode ser mais lento que algoritmos simples para pequenas entradas devido à sobrecarga de chamadas recursivas.

Busca Sequencial

É um algoritmo de busca que recebe um array a quantidade de elementos desse array e o elemento que o usuário deseja então compara cada elemento da lista até encontrar o elemento desejado ou chegar ao final da lista.

O algoritmo em C:

```
int busca (int n, int*vet , int elem ){

     for (int i = 0: i‹n; i++)[

          if(elem == vet[i]){

               return i ;

     return -1 ;

     }
```

Vantagens:

- Não necessita de ordenação prévia da lista para ser implementado.
- É simples para implementar.

Desvantagens:

- É ruim para listas com muitos elementos por ser necessário muitas iterações no código.

- Na pior dos casos ou no caso de não ter esse elemento requisitado no array a quantidade é O(n).

Busca Binária

É um algoritmo de busca que percorre um array ordenado e recebe como parâmetros: o array, o tamanho desse array e o elemento buscado. Seu funcionamento consiste em utilizar três variáveis: maior, menor e o elemento do meio. O elemento do meio é comparado ao valor requisitado e, caso seja maior ou menor, o lado não selecionado é descartado. Esse processo se repete até que o elemento do meio seja igual ao valor buscado ou caso o elemento não seja encontrado.

O algoritmo em C:

```
int busca_binaria(int n, int* vet, int elem){

      int ini = 0;

      int fim = n - 1;

      int meio;

      while (int <= fim){

      meio = (int + fim) / 2;

      if (elem ‹ vet[meio]) {

            fim = meio- 1;

      else if (elem › vet[meio])

            ini = meio+ 1;

      else{

            return meio;

      return -1;

      }
```

Vantagens

- Velocidade: Opera em O(log n), muito mais rápido que a busca linear O(n) em grandes conjuntos de dados.
- Número reduzido de iterações\; A cada passo, elimina metade dos elementos, tornando a busca mais eficiente.

Desvantagens

- Exige ordenação prévia – O array ou estrutura deve estar ordenado para que a busca funcione corretamente.
- Não é recomendada para estruturas dinâmicas – Em listas dinâmicas, onde elementos são frequentemente inseridos/removidos, a necessidade de reordenar os dados pode tornar a busca ineficiente.
- Acesso pode ser mais lento em certas estruturas – Em árvores binárias desbalanceadas, o desempenho pode se aproximar de O(n).

Insertion Sort

O inserction sort é um algoritmo de ordenação que recebe um vetor r o reparte em duas seções a esquerda que está ordenada e a da direita que está desordenado. Ele funciona da seguinte forma. O elemento da esquerda que é o primeiro do vetor é selecionado e considera-se que ele já está ordenado então compara-se com o próximo elemento do vetor deslocando para direita todos os elementos maiores que ele e o inserindo na parte à direita. Esse processo é repetido até que o vetor esteja em ordem. O tempo necessário para a ordenação é O(n^2) nos piores casos.

O algoritmo em C:

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

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

        int chave = arr[i];

        int j = i - 1;

        if (arr[j] > chave) {

            arr[j + 1] = arr[j];

            j--;

        }


        if (j + 1 != i) {

            arr[j + 1] = chave;

        }
```

```
    }
}
```

Vantagens:

- Eficiente para pequenas listas.
- Bom desempenho para dados quase ordenados.

Desvantagens:

- Ineficiente para pequenas listas.
- Desempenho ruim em listas completamente desordenadas.

Quick Sort

É um algoritmo de ordenação que utiliza a estratégia **"Dividir para Conquistar"**, onde é escolhido um pivô. Essa escolha pode ser feita de forma aleatória, pela mediana de três ou pelo menor ou maior índice. A partir do pivô, todos os elementos são divididos em dois grupos: os menores (à esquerda) e os maiores (à direita). Esse processo gera dois subarrays, nos quais a mesma operação é repetida até que a lista esteja completamente ordenada.

Exemplo do algoritmo em C:

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void trocar(int *a, int *b) {

      int temp = *a;

      *a = *b;

      *b = temp;

}

int particionar(int arr[], int low, int high) {

      srand(time(NULL));

      int indice_pivo = low + rand() % (high - low + 1);

      trocar(&arr[indice_pivo], &arr[high]);
```

```
    int pivô = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivô) {
            i++;
            trocar(&arr[i], &arr[j]);
        }
    }
    trocar(&arr[i + 1], &arr[high]);
    return i + 1;


}

 void quickSort(int arr[], int low, int high) {

    if (low < high) {

    int pi = particionar(arr, low, high);

    quickSort(arr, low, pi - 1);

    quickSort(arr, pi + 1, high); }

    }
```

Obs.: Nesse exemplo o pivô está sendo escolhido de forma aleatória.

Vantagens:

- Alto desempenho em grande maioria dos códigos.
- Bom desempenho para grande quantidade de dados.

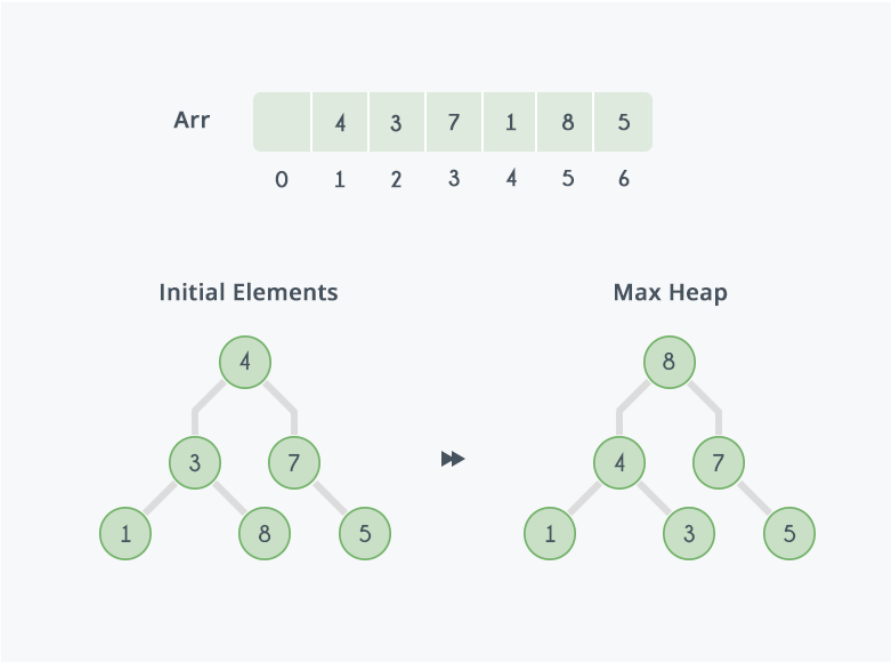
Desvantagens:

- Caso o pivô seja mal escolhido o tempo de execução é muito maior.
- Pode acabar alterando a ordem de elementos iguais.

Heap Sort

Consiste na criação de uma árvore binária onde cada nó é pai é maior que o filho então o maior elemento (primeiro da lista) é retirado e trocado pelo último esse processo é repetido até que só sobre um nó.

Segue a baixo a figura da busca binária:



Algoritmo em C de heap sort :

```
void heapify(int arr[], int n, int i) {
    int maior = i;
    int esquerda = 2 * i + 1;
    int direita = 2 * i + 2;

    if (esquerda < n && arr[esquerda] > arr[maior])
        maior = esquerda;

    if (direita < n && arr[direita] > arr[maior])
        maior = direita;

     if (maior != i) {
        int temp = arr[i];
        arr[i] = arr[maior];
        arr[maior] = temp;

        heapify(arr, n, maior);
```

```
    }
}

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        heapify(arr, i, 0);
    }
}
```

Vantagens:

- Velocidade: Graças à estrutura de heap (árvore binária), o algoritmo sempre opera em O(n log n) no pior, melhor e caso médio.
- Bom para fluxos contínuos de dados: Como o heap mantém os elementos parcialmente ordenados, facilita a inserção e remoção de novos dados de forma eficiente.

Desvantagens:

- Não é estável: Se houver elementos iguais, a ordem relativa entre eles pode ser alterada durante a ordenação.
- Acesso não sequencial: O algoritmo realiza muitas operações de acesso à memória, o que pode prejudicar o desempenho em relação à eficiência do cache.

CONCLUSÃO

O estudo desses algoritmos mostra que não existe uma única solução ideal, mas sim a mais adequada para cada caso. Métodos simples funcionam bem para listas pequenas, enquanto algoritmos mais avançados são melhores para grandes volumes de dados. A busca eficiente depende da organização dos dados, tornando a ordenação muitas vezes um passo necessário. Conhecer essas técnicas ajuda a melhorar o desempenho dos programas e a tomar melhores decisões na hora de processar informações.

REFERÊNCIAS

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. ***Introdução a algoritmos.*** **4. ed. Cambridge: MIT Press, 2022.**

CELES, W.; CERQUEIRA, R.; RANGEL, J. L. ***Introdução a estruturas de dados com técnicas de programação em C.*** **4. tiragem. Rio de Janeiro: Elsevier, 2004.**

BHARGAVA, A. Y. ***Entendendo algoritmos: um guia ilustrado para programadores e outros curiosos.*** **São Paulo: Novatec Editora, 2017.**