# Programação Básica

Prof. Ms. Eduardo Fernandes


eduardo.fernandes@idp.edu.br


@msc.eduardofernandes


www.linkedin.com/in/edufernandes89

# Busca e Ordenação

## Visão Geral

- Os algoritmos de busca e ordenação são fundamentais para a computação, pois é exatamente nesses processos que os computadores tem grande eficiência e desempenho.

- Boa parte das aplicações e dos sistemas computacionais precisam, em dado momento, procurar por alguma informação ou ordenar os dados.

# Busca e Ordenação

## Objetivos

Após estudar e realizar os exercícios propostos para este capítulo, você deverá ser capaz de:

- Escolher o melhor método de busca para os seus algoritmos;
- Escolher o melhor método de ordenação para os seus algoritmos;
- Diferenciar entre os diversos métodos de busca e classificação, e saber das vantagens e desvantagens de cada um.

idp

# Busca

# Algoritmos de Busca

## Busca linear (sequencial)

- Podemos procurar um valor desejado em um vetor por meio da pesquisa sequencial.
- Esta começa no primeiro elemento da matriz e a percorre até o final, localizando o valor escolhido.
- É a forma de busca mais simples, porém pouco eficiente.

# Algoritmos de Busca

## Exercício

1. Escreva um programa em C que busque o número 5 no seguinte vetor (utilize a busca sequencial):

Vetor[5] = {4, 3, 2, 5, 1}

# Algoritmos de Busca

## Busca linear (sequencial)

```
int buscaSequencial (int vetor[], int valor, int tamanho) {
    int i;

    for(i=0; i<tamanho;i++) {
        if (vetor[i]==valor) {
            return 1;
        }
    }
    return 0;
}
```

- Esse tipo de pesquisa pode ser ineficiente se tivermos um vetor de muitos elementos. Nesse caso, poderíamos utilizar a busca binária, como mostrado no próximo item.

# Algoritmos de Busca

## Busca Binária

- Ao contrário da busca linear, a busca binária procura otimizar o processo de procura por determinado elemento de um vetor **ordenado**, quando considera que é possível sempre dividir o conjunto de elementos em duas partes (metade).

- Como analogia, podemos citar a busca por determinada palavra em um dicionário: se buscamos a palavra “natureza”, seria uma boa estratégia abrir o dicionário no meio e decidir se a palavra procurada está antes ou depois desse ponto. No nosso caso, descartaremos a metade inferior do livro, o que é bastante eficiente.

# Algoritmos de Busca

## Busca Binária

- A seguir, tomamos a metade restante e procedemos da mesma forma, o que eliminará o último quarto de páginas do dicionário, nos trazendo mais perto da palavra buscada. Ao final de algumas iterações, localizaremos o termo “natureza”, e cumpriremos nossa missão.

- **Importante**: lembrar que o vetor tem que estar ordenado de forma crescente ou decrescente.

# Algoritmos de Busca

## Exercício

1. Escreva um programa em C que busque o número 60 no seguinte vetor (utilize a busca binária):

Vetor[10] = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90}

# Algoritmos de Busca

## Busca Binária

```
int buscaBinaria(int vetor[], int valor, int tamanho) {
    int achou = 0;
    int alto = tamanho, baixo = 0, meio;

    meio = (alto + baixo)/2;

    while((!achou) && (alto>=baixo)) {
        if(valor == vetor[meio]) {
            achou = 1;
        } else {
            if (valor < vetor[meio]) {
                alto = meio - 1;
            } else {
                baixo = meio + 1;
            }
        }
        meio = (alto+baixo)/2;
    }

    return(achou);
}
```

# Ordenação

# Algoritmos de Ordenação

## Método de ordenação BubbleSort

- Por ser o mais simples método de ordenação, o BubbleSort não é tão eficiente, e por isso é indicado para vetores de 30 elementos no máximo.

- Nesse método, os elementos do vetor são percorridos, e cada par adjacente é comparado; se necessário, o par é ordenado pela troca de posição (swap). Essa operação se repete até que o vetor esteja ordenado.

# Algoritmos de Ordenação

## Exercício

1. Escreva um programa em C que ordene o seguinte vetor (utilize o método BubbleSort):

Vetor[10] = {0, 2, 4, 5, 7, 9, 8, 5, 3, 1, 6}

# Algoritmos de Ordenação

## Método de ordenação BubbleSort

```
void bubbleSort(int vetor[], int tamanho) {
    int aux, troca, i;

    troca = 1; /*A variável "troca" será a verificação da troca em cada passada*/

    while(troca == 1) {
        troca = 0; /*Se o valor continuar 0 na próxima passada quer dizer que não houve troca e a função é encerrada.*/
        for(i = 0; i < tamanho-1; i++) {
            if(vetor[i] > vetor[i+1]) {
                aux         = vetor[i];
                vetor[i]    = vetor[i+1];
                vetor[i+1]  = aux;
                troca       = 1; /*Se houve troca, "troca" recebe 1 para continuar rodando.*/
            }
        }
    }
}
```

idp

# Algoritmos de Ordenação

## Método de ordenação QuickSort

- Utilizado em vetores com muitos elementos, o QuickSort é um método muito eficiente.
- O algoritmo QuickSort inicialmente seleciona o valor posicionado no centro da lista de elementos, ao qual chamaremos elemento central (pivô).
- Em seguida, divide o vetor em duas listas menores, separando, em uma delas, os elementos cujos valores são maiores que o valor do elemento central e, na outra lista, os elementos cujos valores são menores que o valor do elemento central.
- Entra-se, em seguida, em um processo recursivo em cada uma das listas, as quais tornam-se sempre menores, até que o vetor esteja todo ordenado.

idp

# Algoritmos de Ordenação

## Exercício

1. Escreva um programa em C que ordene o seguinte vetor (utilize o método QuickSort):

Vetor[10] = {0, 2, 4, 5, 7, 9, 8, 5, 3, 1, 6}

# Algoritmos de Ordenação

## Método de ordenação Quicksort

```
void quickSort(int *a, int left, int right) {
    int i, j, pivo, temp;

    i = left;
    j = right;
    pivo = a[(left + right) / 2];
    while(i <= j) {
        while(a[i] < pivo && i < right) {
            i++;
        }
        while(a[j] > pivo && j > left) {
            j--;
        }
        if(i <= j) {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            i++;
            j--;
        }

    }
    if(j > left) {
        quickSort(a, left, j);
    }
    if(i < right) {
        quickSort(a, i, right);
    }
}
```

# Algoritmos de Ordenação

## Outros tipos de algoritmos de ordenação

Existem outros algoritmos de ordenação, os quais apenas citaremos, tendo em vista que os dois métodos mostrados ilustram muito bem o problema de localização de elementos dentro de uma estrutura de dados tradicional.

- Método SelectionSort
- Método ShellSort
- Outros métodos...

# Algoritmos de Ordenação

## Método SelectionSort

- Parecido com o BubbleSort;
- Inicia com um elemento (em geral o primeiro) e percorre a estrutura até achar o menor dos valores, que é colocado naquela posição; seleciona, então, um segundo elemento, e busca pelo segundo menor elemento da estrutura, que é então alocado na segunda posição do vetor – e assim por diante, até que o vetor esteja ordenado.

```
void selectionSort(int num[], int tam) {
  int i, j, min, aux;
  for (i = 0; i < (tam-1); i++)
  {
      min = i;
      for (j = (i+1); j < tam; j++) {
        if(num[j] < num[min])
          min = j;
      }
      if (i != min) {
        aux = num[i];
        num[i] = num[min];
        num[min] = aux;
      }
  }
}
```

# Algoritmos de Ordenação

## Método ShellSort

- Compara elementos separados por determinada distância (gap) até que os elementos que ele identifica com a distância atual estejam ordenados.
- O algoritmo divide então o gap em dois, e o processo continua, até a ordenação completa.

```
void shellSort(int *vet, int size) {
    int i, j, value;

    int h = 1;
    while(h < size) {
        h = 3*h+1;
    }
    while (h > 0) {
        for(i = h; i < size; i++) {
            value = vet[i];
            j = i;
            while (j > h-1 && value <= vet[j - h]) {
                vet[j] = vet[j - h];
                j = j - h;
            }
            vet[j] = value;
        }
        h = h/3;
    }
}
```

INSTITUTO BRASILEIRO DE ENSINO,
DESENVOLVIMENTO E PESQUISA


eduardo.fernandes@idp.edu.br

@msc.eduardofernandes

www.linkedin.com/in/edufernandes89