# ED2 - Aula 4.pdf - Rafael Manfrim

## Busca Binária

### (1) Implemente em C/C++ o algoritmo da Busca Binária de acordo com a explicação anterior. O algoritmo tradicional verifica se o elemento central é igual ao elemento buscado. Mas teremos um algoritmo mais eficiente (não no sentido assintótico) se evitarmos essa comparação até o fim.

```
#include <iostream>

using namespace std;

int busca_binaria(int arr[], int tamanho, int elemento) {
    int inicio = 0;
    int fim = tamanho - 1;

    while (inicio <= fim) {
        int meio = inicio + (fim - inicio) / 2;

        if (arr[meio] == elemento) {
            return meio;
        }
        if (arr[meio] < elemento) {
            inicio = meio + 1;
        } else {
            fim = meio - 1;
        }
    }

    return -1;
}

int main() {
    int arr[] = {2, 3, 4, 10, 15, 18, 40};
    int tamanho = 7;
    int elemento = 3;
    int resultado = busca_binaria(arr, tamanho, elemento);

    if (resultado == -1) {
        cout << "Elemento não encontrado no array" << endl;
    } else {
        cout << "Elemento encontrado no índice: " << resultado << endl;
    }
    return 0;
}
```

## (2) Implemente em C/C++ o algoritmo da Busca Binária realizando essa verificação apenas quando sobrar um único elemento.

```
#include <iostream>

using namespace std;

int busca_binaria(int arr[], int tamanho, int elemento) {
    int inicio = 0;
    int fim = tamanho - 1;

    while (inicio < fim) {
        int meio = inicio + (fim - inicio) / 2;

        if (arr[meio] < elemento) {
            inicio = meio + 1;
        } else {
            fim = meio;
        }
    }

    if (inicio == fim && arr[inicio] == elemento) {
        return inicio;
    }

    return -1;
}

int main() {
    int arr[] = {2, 3, 4, 10, 15, 18, 40};
    int tamanho = 7;
    int elemento = 18;
    int resultado = busca_binaria(arr, tamanho, elemento);

    if (resultado == -1) {
        cout << "Elemento não encontrado no array." << endl;
    } else {
        cout << "Elemento encontrado no índice: " << resultado << endl;
    }
    return 0;
}
```

## (3) Demonstre que a Busca Binária executa em O(log2 n).

Na matemática, a operação de potenciação de caracteriza por realizar uma sequência de multiplicações, sua operação contrária é o logarítmo, que tem como característica, a realização de uma sequência de divisões.

No contexto de busca binária, a cada tentativa de busca, se é realizado uma divisão por 2 da quantidade de elementos restante no vetor, o que caracteriza uma sequência de divisões até a lista ser reduzida a um único elemento. Para demonstrar

que a Busca Binária executa em O(log2 n), partimos dessa premissa.
  n / 2^k = 1, onde n é o número total de elementos e k é o número de iterações
  n = 2^k, isolamos n
  log2(n) = log2(2^k), aplicamos log2 dos dois lados
  log2(n) = k, simplificamos e assim provamos que o número de iterações k é log2(n).

# Insertion Sort

### (1) Determine a sua complexidade de tempo no pior caso utilizando a notação-O.

  O pior caso se daria por N verificações e N - 1 trocas, ou seja N * (N - 1) / 2 seguindo a fórmula da soma de uma P.A. Fazendo com que Soma = (N² - N) / 2, ignorando os termos de ordem inferior, a complexidade de tempo no pior caso é O(N²).

### (1.1) Qual seria a propriedade da instância onde teríamos o pior caso?

  O pior caso ocorre quando o vetor está ordenado de forma decrescente e o algoritmo faz a troca de todos os elementos com todos os elementos anteriores.

### (2) Determine a sua complexidade de tempo no melhor caso utilizando a notação-O.

  No melhor caso, o algoritmo faz N - 1 comparações e nenhuma troca, ou seja, O(N), o que é um ótimo resultado, mas não possui sentido prático, pois em raríssimos casos o algoritmo iria executar com o vetor já ordenado.

### (2.1) Qual seria a propriedade da instância onde teríamos o melhor caso?

  O melhor caso ocorre quando o vetor já está ordenado em ordem crescente, não sendo necessário fazer nenhuma troca.

# Merge Sort

### (1) A animação representa perfeitamente o algoritmo apresentado do Merge Sort? Justifique sua resposta.

  Não, pois a animação não considera corretamente a ordem de execução no Heap do Sistema Operacional, na chamada de funções recursivas. Por exemplo, os 4 elementos

"6", "5", "3" e "1" seriam ordenados primeiro, antes do código dar sequência para os próximos números.

### (2) Explique em muito detalhe o funcionamento da implementação fornecida do Merge Sort.

A implementação do Merge Sort se dá realizando divisões de um vetor em vários subvetores menores por meio de recursão, até que um mesmo fique com apenas com 1 item, em seguida ele "volta" pelo Heap do S.O. realizando a junção desses vetores em vetores maiores novamente (merge), durante a criação desses vetores mergeados é realizada a ordenação.

A lógica de ordenação é a seguinte: identificar no vetor principal o as posições onde será realizado o merge, criar dois subvetores temporários para os dois subvetores onde será mergeado, copiar os itens para esses subvetores temporários, enquanto houverem elementos não mergeados andar pelos vetores temporários e verificar o menor, enviando para o vetor principal. Por fim, copiar todos os elementos restantes para suas posições no vetor principal.

## Bubble Sort

### (3) O Bubble Sort é um algoritmo simples de ordenação que percorre uma lista, comparando pares de elementos adjacentes e trocando-os se estiverem na ordem errada. O processo é repetido até que a lista esteja ordenada. Implemente o algoritmo Bubble Sort. Mais detalhes do algoritmo no último slide.

```
#include <iostream>

using namespace std;

void bubble_sort(int vetor[], int tamanho) {
    for(int i = 0; i < tamanho - 1; i++) {
        for(int j = 0; j < tamanho - i - 1; j++) {
            if(vetor[j] > vetor[j+1]) {
                int temp = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = temp;
            }
        }
    }
}

void imprime_vetor(int vetor[], int tamanho) {
    for(int i = 0; i < tamanho; i++) {
        cout << vetor[i] << " ";
    }
}
```

```
int main() {
    int vetor[] = {5, 2, 9, 1, 4, 6};
    int tamanho_vetor = 6;

    imprime_vetor(vetor, tamanho_vetor);
    cout << endl;
    bubble_sort(vetor, tamanho_vetor);
    imprime_vetor(vetor, tamanho_vetor);

    return 0;
}
```

### (4) Determine a complexidade de tempo do Bubble Sort no pior caso e no melhor caso utilizando a notação-O.

Assim como no Insertion Sort, o Bubble Sort também tem seu pior caso como O($n^2$), pois precisará andar por todo o vetor e realizar N - 1 trocas, ou seja, N(N - 1) / 2, resultando em ($N^2$ - N) / 2. O melhor caso também é O(n), quando não for necessário fazer nenhuma troca.

### (5) Qual o melhor caso para o Bubble Sort? Por quê? Qual o pior caso? Por quê?

O melhor caso seria quando o vetor já estivesse ordenado, pois seria necessário apenas percorrer ele fazendo verificações, sem realizar trocas, ou seja O(n). Já o pior caso ocorre quando o vetor está ordenado em ordem inversa, pois seria necessário realizar N - 1 trocas para cada N.

## Comparações

### (6) Por que o Bubble Sort geralmente é considerado menos eficiente que o Insertion Sort?

O Bubble Sort tem como característica sempre realizar as comparações, e realizar trocas à mais do que o Insertion Sort, pois após posicionar no local correto, o Insertion Sort não verifica e realiza trocas com os elementos que ele sabe que já estão posicionados na posição correta.

### (7) Em quais situações o Merge Sort é preferível ao Insertion Sort e ao Bubble Sort?

Em grandes quantidades de dados e principalmente quando estão completamente desordenados, o Merge Sort será mais eficiente, pois executa em O(n log n) já os

outros outros tem o caso médio $O(n^2)$. O Insertion Sort pode ser preferível apenas quando os dados já estão quase ordenados, executando em O(n).

**(8) Considere as listas de números abaixo. Para cada lista, aplique o Insertion Sort, Bubble Sort e Merge Sort e conte o número de comparações realizadas por cada algoritmo.**

1. [5, 2, 9, 1, 5, 6]
2. [1, 2, 3, 4, 5, 6] (Lista já ordenada)
3. [6, 5, 4, 3, 2, 1] (Lista em ordem inversa)

**Qual dos três algoritmos realizou o maior número de comparações em cada caso? Explique por que alguns algoritmos realizaram menos comparações em vetores já ordenados ou quase ordenadas.**

```
Insertion Sort:
        Lista Aleatória: 9 Comparações
        Lista Ordenada: 5 Comparações
        LiSta Inversa: 15 Comparações

Merge Sort:
        Lista Aleatória: 11 Comparações
        Lista Ordenada: 9 Comparações
        Lista Inversa: 7 Comparações

Bubble Sort:
        Lista Aleatória: 15 Comparações
        Lista Ordenada: 15 Comparações
        LiSta Inversa: 15 Comparações

Em todas as listas, o Bubble Sort mostrou-se menos eficiente na quantidade de
comparações necessárias, realizando 15 comparações em todas, no caso da lista
inversamente ordenada o Insertion Sort também realizou 15 comparações. Em listas
ordenadas, o Insertion Sort e o Merge Sort se beneficiam do fato de não precisarem
comparar todas as vezes, pois o Insertion Sort consegue colocar um número na
posição correta e não precisar verificar com os restantes, já o Merge Sort consegue
ordenar e copiar os elementos restantes de um vetor temporário.
```

**(9) Considere um algoritmo que precisa ordenar uma lista onde os primeiros 50% dos elementos já estão ordenados, mas os últimos 50% estão em ordem inversa. Qual dos três algoritmos (Insertion Sort, Bubble Sort ou Merge Sort) você escolheria para esse caso? Justifique sua escolha baseando-se no comportamento de cada algoritmo.**

Nesse caso, o Merge Sort continuaria executando em O(n log n), o Bubble Sort ainda executaria próximo de O($n^2$) e o Insertion Sort já poderia ter um desempenho próximo de O(n), então para esse caso, é necessário tomar uma decisão entre a garantia de o Merge Sort ou a possibilidade de um Insertion Sort ser mais perfomático, considerando também a dificuldade de implementação e manutenção de cada algorítmo e o uso de memória.

Em meu caso, escolheria o Insertion Sort, pois no quesito uso de memória ele também pode ser muito vantajoso em relação ao Merge Sort, visto que não cria vetores temporários necessitando alocar espaço, possui uma implementação mais fácil e pode ser mais rápido no caso apresentado.

## (10) Implemente os três algoritmos (Insertion Sort, Merge Sort e Bubble Sort) e modifique o código para contar o número total de trocas e comparações realizadas durante a execução. Execute-os em vetores de diferentes tamanhos e ordens (aleatória, já ordenada, inversa, etc.).

```
#include <iostream>

using namespace std;

void insertion_sort(int vetor[], int tamanho, int &trocas, int &comparacoes) {
    for (int i = 0; i < tamanho - 1; i++) {
        int chave = vetor[i + 1];

        int j = i;

        bool entrou = false;

        while(j >= 0 && vetor[j] > chave) {
            entrou = true;
            comparacoes++;
            vetor[j + 1] = vetor[j];
            trocas++;
            j--;
        }
        if (!entrou)
            comparacoes++;

        if (j > 0 && entrou)
            comparacoes++;

        vetor[j + 1] = chave;
    }
}

int main() {
    int tamanho = 6;
```

```
    int vetor_aleatorio[] = {5, 2, 9, 1, 5, 6};
    int vetor_ordenado[] = {1, 2, 3, 4, 5, 6};
    int vetor_inversamente_ordenado[] = {6, 5, 4, 3, 2, 1};

    int trocas = 0;
    int comparacoes = 0;

    insertion_sort(vetor_aleatorio, tamanho, trocas, comparacoes);
    cout << "Vetor Aleatório: " << trocas << " Trocas e " << comparacoes << "
Comparações" << endl;

    trocas = 0;
    comparacoes = 0;

    insertion_sort(vetor_ordenado, tamanho, trocas, comparacoes);
    cout << "Vetor Ordenado: " << trocas << " Trocas e " << comparacoes << "
Comparações" << endl;

    trocas = 0;
    comparacoes = 0;

    insertion_sort(vetor_inversamente_ordenado, tamanho, trocas, comparacoes);
    cout << "Vetor Inversamente Ordenado: " << trocas << " Trocas e " <<
comparacoes << " Comparações" << endl;

    return 0;
}
```

```
#include <iostream>

using namespace std;

void Merge(int V[], int inicio, int meio, int fim, int &trocas, int &comparacoes) {
    int tamEsq = meio - inicio + 1; // Comprimento de V[inicio ... meio]
    int tamDir = fim - meio; // Comprimento de V[meio + 1 ... fim]
    int E[tamEsq], D[tamDir]; // Cria novos vetores E[] e D[]

    for(int i = 0; i < tamEsq; i++) // Copia dados para os vetores temporários E[]
e D[] */
        E[i] = V[inicio + i];

    for(int j = 0; j < tamDir; j++) // Copia V[q + 1 ... r] para D[0 ... tamDir -
1]
        D[j] = V[meio + 1 + j];

    int i = 0, j = 0;

    int k = inicio; // Índice inicial do subvetor mesclado

    bool entrou = false;

    while(i < tamEsq && j < tamDir) { // Enquanto L e R tiverem elementos não
```

```
mesclados
        entrou = true;
        comparacoes++;
        if(E[i] <= D[j]) {
            V[k] = E[i];
            i++;
        } else {
            trocas++;
            V[k] = D[j];
            j++;
        }        k++;
    }
    if (!entrou)
        comparacoes++;

    while(i < tamEsq) { // Copia o restante de L, se houver
        V[k] = E[i];
        i++;
        k++;
    }
    while(j < tamDir) { // Copia o restante de R, se houver
        V[k] = D[j];
        j++;
        k++;
    }}

void MergeSort(int V[], int inicio, int fim, int &trocas, int &comparecoes) {
    if(inicio < fim) { // Se há mais de um elemento
        int meio = (inicio + fim) / 2;
        MergeSort(V, inicio, meio, trocas, comparecoes); // Ordena recursivamente o
primeiro subvetor
        MergeSort(V, meio + 1, fim, trocas, comparecoes); // Ordena recursivamente
o segundo subvetor
        Merge(V, inicio, meio, fim, trocas, comparecoes); // Combina os subvetores
ordenados
    }
}

int main() {
    int vetor_aleatorio[] = {5, 2, 9, 1, 5, 6};
    int vetor_ordenado[] = {1, 2, 3, 4, 5, 6};
    int vetor_inversamente_ordenado[] = {6, 5, 4, 3, 2, 1};

    int tamanho = 6;
    int trocas = 0;
    int comparacoes = 0;

    MergeSort(vetor_aleatorio, 0, tamanho - 1, trocas, comparacoes);
    cout << "Vetor Aleatório: " << trocas << " Trocas e " << comparacoes << "
Comparações" << endl;

    trocas = 0;
    comparacoes = 0;
```

```
    MergeSort(vetor_ordenado, 0, tamanho - 1, trocas, comparacoes);
    cout << "Vetor Ordenado: " << trocas << " Trocas e " << comparacoes << "
Comparações" << endl;

    trocas = 0;
    comparacoes = 0;

    MergeSort(vetor_inversamente_ordenado, 0, tamanho - 1, trocas, comparacoes);
    cout << "Vetor Inversamente Ordenado: " << trocas << " Trocas e " <<
comparacoes << " Comparações" << endl;

    return 0;
}
```

```
#include <iostream>

using namespace std;

void bubble_sort(int vetor[], int tamanho, int &trocas, int &comparacoes) {
    for(int i = 0; i < tamanho - 1; i++) {
        for(int j = 0; j < tamanho - i - 1; j++) {
            comparacoes++;
            if(vetor[j] > vetor[j+1]) {
                trocas++;
                int temp = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = temp;
            }
        }
    }
}


int main() {
    int vetor_aleatorio[] = {5, 2, 9, 1, 5, 6};
    int vetor_ordenado[] = {1, 2, 3, 4, 5, 6};
    int vetor_inversamente_ordenado[] = {6, 5, 4, 3, 2, 1};

    int trocas = 0;
    int comparacoes = 0;
    int tamanho = 6;

    bubble_sort(vetor_aleatorio, tamanho, trocas, comparacoes);
    cout << "Vetor Aleatório: " << trocas << " Trocas e " << comparacoes << "
Comparações" << endl;

    trocas = 0;
    comparacoes = 0;

    bubble_sort(vetor_ordenado, tamanho, trocas, comparacoes);
    cout << "Vetor Ordenado: " << trocas << " Trocas e " << comparacoes << "
```

```
Comparações" << endl;

    trocas = 0;
    comparacoes = 0;

    bubble_sort(vetor_inversamente_ordenado, tamanho, trocas, comparacoes);
    cout << "Vetor Inversamente Ordenado: " << trocas << " Trocas e " <<
comparacoes << " Comparações" << endl;

    return 0;
}
```

```
Insertion Sort:
        Lista Aleatória: 9 Comparações e 6 Trocas
        Lista Ordenada: 5 Comparações e 0 Trocas
        Lista Inversa: 15 Comparações e 15 Trocas

Merge Sort:
        Lista Aleatória: 11 Comparações e 4 Trocas
        Lista Ordenada: 9 Comparações e 0 Trocas
        Lista Inversa: 7 Comparações e 7 Trocas

Bubble Sort:
        Lista Aleatória: 15 Comparações e 6 Trocas
        Lista Ordenada: 15 Comparações e 0 Trocas
        Lista Inversa: 15 Comparações e 15 Trocas
```

## (10.1) Como esses números variam entre os algoritmos e diferentes tipos de listas?

```
O Insertion Sort e o Bubble Sort realizam a mesma quantidade de trocas, com a
vantagem do Insertion Sort não precisar realizar sempre todas as comparações. Já o
Merge Sort tem a vantagem de poder realizar menos trocas, e em listas parcialmente
ordenadas menos comparações, já em listas aleatórias precisa realiar algumas
comparações à mais.
```