# Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

## Aula 21 – Algoritmos de Ordenação I

# Agenda

- Introdução
- Bubble Sort
- Merge Sort

- **PS:** Parte do conteúdo retirado do material do Prof. Flávio B. Gonzaga

# Introdução

- A ordenação de dados é, sem dúvidas, uma das operações mais essenciais dentro de Ciência da Computação.

- Desde nosso sistemas de arquivos, passando por planilhas até chegar aos *sites* que visitamos, a ordenação de dados está presente em todas estas aplicações.

# Introdução

- São alguns dos motivos de usarmos algoritmos de ordenação:
  i. Facilitar a busca
  ii. Melhorar a visualização e interpretação
  iii. Pré-processamento para outros algoritmos
  iv. Aplicações em problemas do mundo real (como banco de dados, motores de busca, ordenação de produtos em um *e-commerce*, etc)
  v. Entendimento de complexidade algorítmica

# Introdução

- Podemos classificar os algoritmos de ordenação em três tipos, sendo eles **simples**, **eficientes** e **especiais**.

- São exemplos de algoritmos de ordenação: *Bubble Sort*, *Insertion Sort*, *Selection Sort*, *Quick Sort*, *Merge Sort*, *Heap Sort*, *Counting Sort* e *Radix Sort*.

# Introdução

- Chamamos de **simples** os algoritmos cuja lógica é direta, fácil de implementar e entender, mesmo por iniciantes.
  - Eles geralmente têm **custo computacional maior** – normalmente O ($n^2$) – e não são indicados para grandes volumes de dados.
  - São usados principalmente para fins didáticos ou quando a entrada é muito pequena.
  - Exemplos: *Bubble Sort*, *Insertion Sort* e *Selection Sort*.

# Introdução

- Algoritmos **eficientes** são mais sofisticados e têm melhor desempenho assintótico, especialmente em conjuntos de dados grandes.

- Muitos deles usam estratégias como divisão e conquista e conseguem complexidade média *O(n logn)*. São amplamente utilizados na prática.
  - Exemplos: *Quick Sort*, *Merge Sort* e *Heap Sort*.

# Introdução

- Algoritmos **especiais** são projetados para casos específicos, geralmente quando há conhecimento prévio sobre os dados, como limites numéricos ou tipo de entrada.

- Eles não usam comparações e podem ter complexidade linear *O(n)*, o que os torna muito rápidos para esses casos.
  - Mas não funcionam com qualquer tipo de dado!

# Introdução

- São exemplos de algoritmos especiais de ordenação (que não serão abordados na disciplina):
  - **Counting Sort**: Conta ocorrências de cada valor e monta o vetor ordenado. Funciona para inteiros em faixa conhecida.
  - **Radix Sort**: Ordena números dígito a dígito, usando algoritmos estáveis como base (geralmente *Counting Sort*).

# Bubble Sort

- O algoritmo mais básico nesta área, conhecido como *Bubble Sort*, possui registros de ter sido primeiramente descrito em 1956, em um artigo chamado *Sorting on electronic computer systems* de autoria do matemático e atuário Edward Harry Friend.

# Bubble Sort

- O *Bubble Sort* é um algoritmo de ordenação simples e intuitivo que funciona comparando pares de elementos adjacentes em uma lista e trocando-os de posição sempre que estiverem fora de ordem.

- Esse processo é repetido várias vezes, fazendo com que os maiores valores "borbulhem" para o final da lista a cada passagem.

# Bubble Sort

- Apesar de sua fácil implementação e compreensão, o *Bubble Sort* não é eficiente para grandes volumes de dados, pois sua complexidade de tempo no pior caso é $O(n^2)$.

- Por isso, é mais utilizado em contextos educacionais para introduzir conceitos básicos de algoritmos de ordenação.

# Bubble Sort

- O passo-a-passo do *Bubble Sort* pode ser descrito como:
  1. Percorra a lista do primeiro ao último elemento.
  2. Para cada par de elementos adjacentes, compare-os. Se o elemento da esquerda for maior que o da direita, troque-os de posição.
  3. Continue esse processo até o final da lista. Após uma passada, o maior elemento estará na última posição.

# Bubble Sort

- O passo-a-passo do *Bubble Sort* pode ser descrito como (*cont.*):
    4. Repita o processo para as próximas posições da lista, sem incluir o último elemento a cada passada (pois já está ordenado).
    5. Pare quando uma passada completa não fizer nenhuma troca, indicando que a lista está ordenada.

# Bubble Sort


**An example of bubble sort.**

Disponível em: <https://en.wikipedia.org/wiki/Bubble_sort#/media/File:Bubble-sort-example-300px.gif>

# Bubble Sort

**Bubble sort with Hungarian, folk dance.**

Disponível em: <https://www.youtube.com/watch?v=Iv3vgjM8Pv4>

# Bubble Sort

- A função *bubbleSort* recebe como parâmetro o vetor a ser ordenado (*arr*) e o seu tamanho (*n*).

- A função utiliza dois laços for aninhados para percorrer repetidamente o vetor.

```c
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

# Bubble Sort

- O primeiro laço controla o número de passagens pelo vetor, enquanto o segundo compara elementos adjacentes: *arr[j]* com *arr[j + 1]*.

```
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

- Se os elementos estiverem fora de ordem (ou seja, se o elemento da esquerda for maior que o da direita), eles são trocados de posição usando a variável auxiliar *temp*.

# Bubble Sort

- A cada passagem, o maior elemento restante é empurrado para o final da área não ordenada do vetor, o que reduz a quantidade de comparações necessárias nas próximas iterações.

```c
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

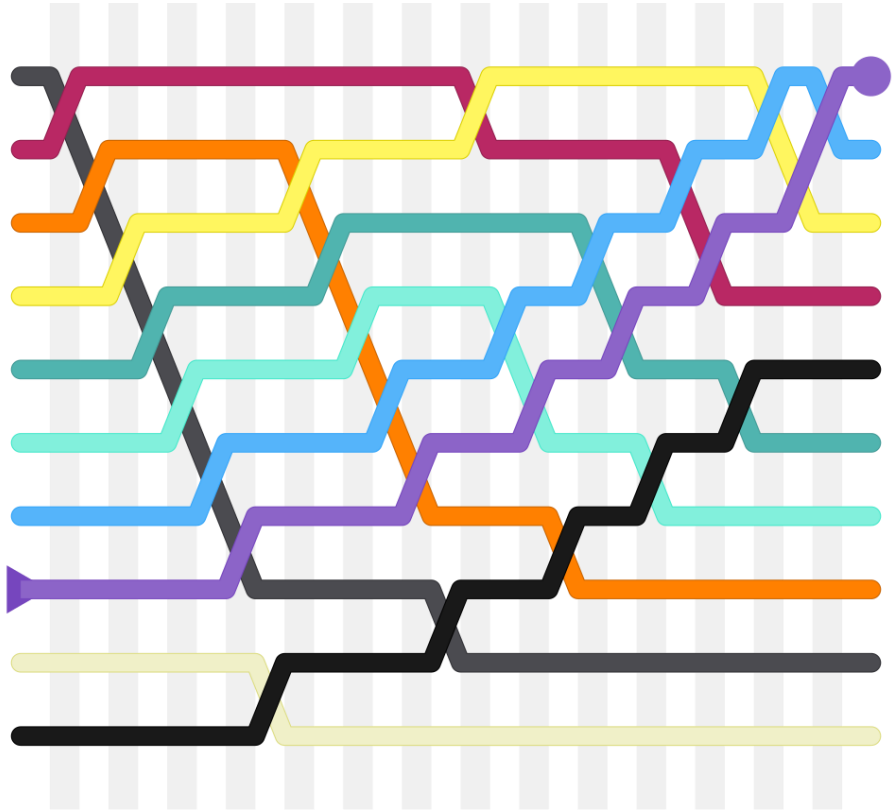
- Ao final da execução, todos os elementos estarão ordenados em ordem crescente.

# Bubble Sort

- Com relação à performance (complexidade de tempo nas comparações e trocas), temos:

| Casos | Complexidade de Tempo (Comparações) | Complexidade de Tempo (Trocas) |
|---|---|---|
| Pior Caso | $O(n^2)$ | $O(n^2)$ |
| Caso Médio | $O(n^2)$ | $O(n^2)$ |
| Melhor Caso | $O(n)$ | $O(1)$ |

# Bubble Sort



Visualização estática do *Bubble Sort*.

# Merge Sort

- Um dos algoritmos clássicos, conhecido como Merge Sort, foi proposto por John Von Neumann em 1945.

- John Von Neumann foi professor na Universidade de Princeton e um dos construtores do ENIAC. Faleceu aos 53 anos (1957).

# Merge Sort

- O passo-a-passo do *Merge Sort* pode ser descrito como:
  1. Divida a lista não ordenada em n sublistas, cada uma contendo um elemento (uma lista contendo um elemento é considerada ordenada).
  2. Repita operações mesclando as sublistas, de modo a se produzir novas sub-listas ordenadas, até que se alcance uma única lista novamente. Esta lista estará, portanto, ordenada.

# Merge Sort

- O Merge Sort é um algoritmo de ordenação baseado na técnica de divisão e conquista, que divide recursivamente o vetor em partes menores, ordena essas partes e depois as combina de forma ordenada.

- Ele funciona dividindo o vetor ao meio até que cada subvetor contenha um elemento (trivialmente ordenado), e em seguida realiza a intercalação dessas partes de forma ordenada, reconstruindo o vetor original.

# Merge Sort

- Com complexidade de tempo *O(n logn)* **em todos os casos**, o Merge Sort é eficiente e estável, sendo especialmente útil em situações que exigem desempenho consistente, mesmo com grandes volumes de dados.

- Apesar de sua eficiência, ele exige espaço adicional proporcional ao tamanho da entrada, o que pode ser uma limitação em ambientes com restrições de memória.

# Merge Sort


**Exemplo de execução do Merge Sort**.

Disponível em: <https://pt.wikipedia.org/wiki/Merge_sort#/media/Ficheiro:Merge-sort-example-300px.gif>

# Merge Sort

**Merge-sort with Transylvanian-saxon (German) folk dance**

Disponível em: <https://www.youtube.com/watch?v=XaqR3G_NVoo>

# Merge Sort

- A função *merge* é responsável por intercalar dois subvetores ordenados dentro de um vetor principal.
- Ela recebe como parâmetros o vetor *arr* e três índices: *l*, *m* e *r*, que indicam respectivamente o *início*, o *meio* e o fim do intervalo que será processado.
- A ideia é que o vetor esteja dividido em duas partes já ordenadas: da posição *l* até *m*, e de *m* + *1* até *r*.

```c
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = malloc(n1 * sizeof(int));
    int *R = malloc(n2 * sizeof(int));

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    free(L);
    free(R);
}
```

# Merge Sort

- Para realizar a intercalação, a função calcula os tamanhos dessas duas partes e cria dois vetores auxiliares, $L$ e $R$, que vão armazenar temporariamente os elementos das duas metades.

- Em seguida, esses elementos são copiados do vetor original para os vetores auxiliares.

```c
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = malloc(n1 * sizeof(int));
    int *R = malloc(n2 * sizeof(int));

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    free(L);
    free(R);
}
```

# Merge Sort

- Depois disso, três índices são usados: *i* para percorrer o vetor *L*, *j* para o vetor *R*, e *k* para escrever no vetor original.
- A função entra em um laço que compara os elementos atuais de *L* e *R*, copiando para o vetor principal aquele que for menor ou igual.
- Esse processo continua até que um dos vetores auxiliares seja totalmente percorrido.

```c
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = malloc(n1 * sizeof(int));
    int *R = malloc(n2 * sizeof(int));

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    free(L);
    free(R);
}
```

# Merge Sort

- Quando isso acontece, o laço termina, mas pode ainda haver elementos restantes no vetor L ou R.

- Por isso, dois laços adicionais garantem que todos os elementos restantes sejam copiados de volta para o vetor original.

```c
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = malloc(n1 * sizeof(int));
    int *R = malloc(n2 * sizeof(int));

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    free(L);
    free(R);
}
```

# Merge Sort

- Ao final da função, os dois vetores auxiliares são liberados com free, encerrando o processo de forma limpa.

- O resultado final é que o intervalo de arr[l] até arr[r] estará completamente ordenado.

```c
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = malloc(n1 * sizeof(int));
    int *R = malloc(n2 * sizeof(int));

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    free(L);
    free(R);
}
```

# Merge Sort

- A execução começa verificando se *l* é menor que *r*, o que significa que ainda há mais de um elemento a ser ordenado naquele intervalo.
- Se essa condição for verdadeira, o próximo passo é calcular a posição do meio do intervalo, usando a expressão *l + (r - l) / 2*, que evita problemas de estouro de inteiros.

```
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

# Merge Sort

- Com o índice do meio definido, a função se chama recursivamente duas vezes: uma para ordenar a metade esquerda do vetor, do índice *l* até *m*, e outra para ordenar a metade direita, do índice *m* + *1* até *r*.

```
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

# Merge Sort

- Depois que essas duas chamadas recursivas retornam, indicando que ambas as metades já estão ordenadas, a função merge é chamada para intercalar essas duas partes em uma única sequência ordenada.

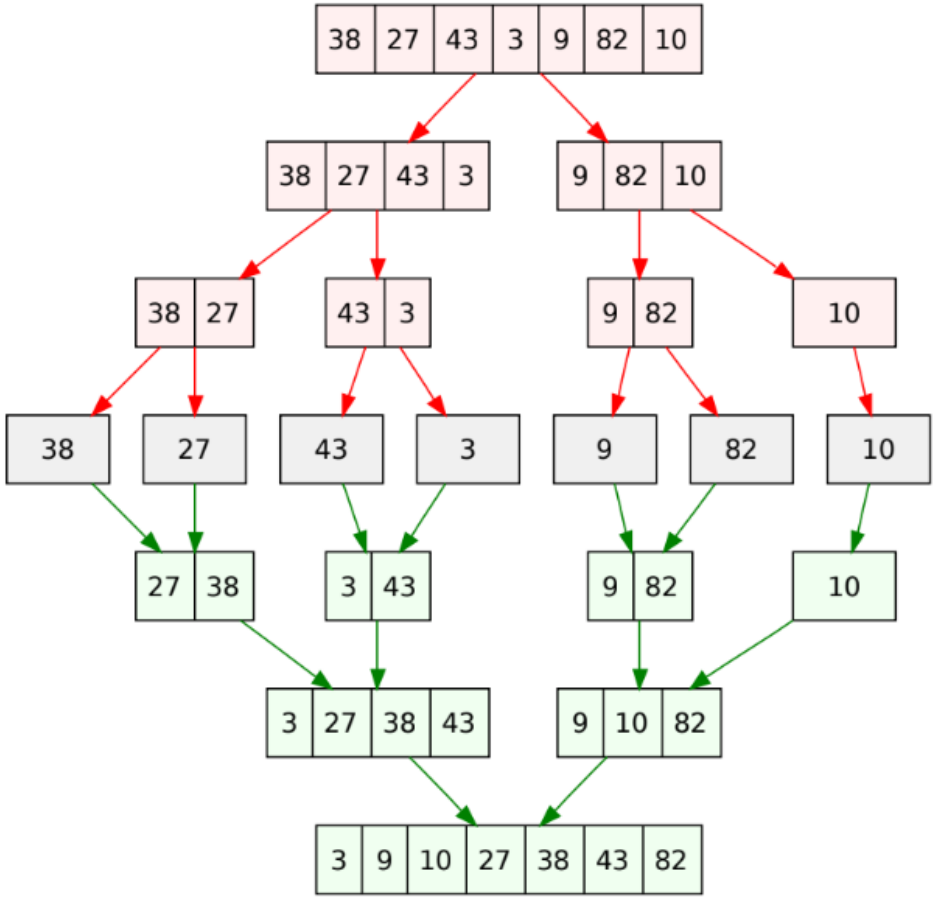
```
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

- O processo se repete até que todo o vetor original esteja ordenado, seguindo a lógica de dividir e depois conquistar a solução.

# Merge Sort




Visualização do *Merge Sort*.

# Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

## Aula 21 – Algoritmos de Ordenação I