# Ordenação

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

O problema de ordenação consiste em, dado uma sequência de $n$ números $\langle a_1, a_2, \ldots, a_n \rangle$, determinar uma permutação (reordenação) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ da sequência de entrada tal que, $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

# Projeto de algoritmos incremental

A ideia de um algoritmo incremental é:

- Iniciar com a solução para um problema trivial;
- Estender a solução iterativamente para um problema maior até obter a solução do problema que queremos resolver;

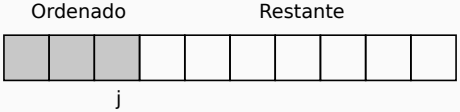
Como projetar um algoritmo incremental para somar os elementos de um arranjo?

- Começamos com a soma do arranjo vazio que é 0;
- Estendemos a soma adicionando um elemento do arranjo por vez até que todos os elementos tenham sido somados.

# Ordenação incremental

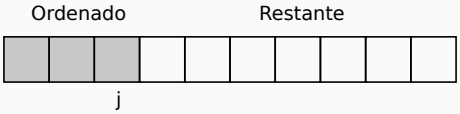
Como projetar um algoritmo incremental para ordenar os elementos de um arranjo?

- Iniciamos com um subarranjo vazio já ordenado;
- Estendemos o subarranjo já ordenado com um elemento da parte restante por vez até que todos os elementos tenham sido selecionados.




Temos que tomar duas decisões para tornar o processo concreto:

- Como selecionar o próximo elemento?
- Como estender o subarranjo ordenado?




Como selecionar o próximo elemento?

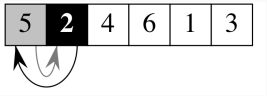
- Pegamos o primeiro elemento do restante.

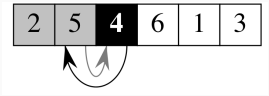
Como estender o subarranjo ordenado?

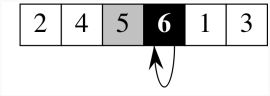
- Inserindo o elemento selecionado na parte ordenada.

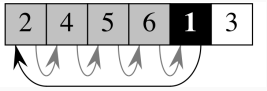
Este algoritmo é conhecido como **ordenação por inserção** (*insertion sort*).

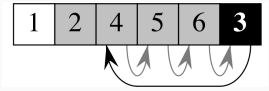
# Ordenação por inserção

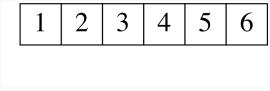


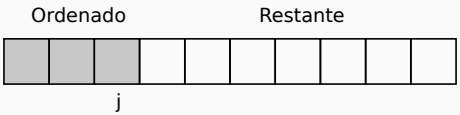
# Ordenação por inserção

Projete uma função que implemente o algoritmo de ordenação por inserção.

```
def ordena_insercao(lst: list[int]):
    '''
    Ordena *lst* em ordem não decrescente usando o algoritmo de ordenação por inserção.
    Exemplos
    >>> lst = [5, 2, 4, 6, 1, 3]
    >>> ordena_insercao(lst)
    >>> lst
    [1, 2, 3, 4, 5, 6]
    '''

    for i in range(1, len(lst)):
        j = i
        while j > 0 and lst[j - 1] > lst[j]:
            lst[j - 1], lst[j] = lst[j], lst[j - 1]
            j -= 1
```

# Ordenação por seleção




Como selecionar o próximo elemento?

- Pegamos o menor elemento do restante.

Como estender o subarranjo ordenado?

- Trocando de posição o menor elemento com o primeiro do restante.

Este algoritmo é conhecido como **ordenação por seleção** (*selection sort*).

# Ordenação por seleção

Projete uma função que implemente o algoritmo de ordenação por seleção.

```
def ordena_selecao(lst: list[int]):
    '''
    Ordena *lst* em ordem não decrescente usando o algoritmo de ordenação por inserção.
    Exemplos
    >>> lst = [5, 2, 4, 6, 1, 3]
    >>> ordena_insercao(lst)
    >>> lst
    [1, 2, 3, 4, 5, 6]
    '''

    for i in range(len(lst)):
        m = i # índice do menor em lst[i:]
        for j in range(i + 1, len(lst)):
            if lst[j] < lst[m]:
                m = j
        lst[i], lst[m] = lst[m], lst[i]
```

A ideia de um algoritmo divisão e conquista é:

- Resolver o problema diretamente se ele for trivial, senão **dividir** o problema em dois ou mais subproblemas do mesmo tipo;
- **Conquistar** os subproblemas resolvendo-os recursivamente;
- **Combinar** as soluções dos subproblemas para obter a solução do problema original

Como projetar um algoritmo de divisão e conquista para somar os elementos de um arranjo? (Note que esse algoritmo não traz nenhum vantagem, é apenas uma ilustração)

- Se o arranjo for vazio, a soma é 0. Senão dividir o arranjo na metade e calcular a soma de cada metade recursivamente;
- Obter a soma do arranjo somando o resultado de cada metade;

# Ordenação incremental

Como projetar um algoritmo de divisão e conquista para ordenar os elementos de um arranjo?

- Se o arranjo tiver mais que um elemento, separamos os elementos em dois subarranjos;
- Ordenamos cada subarranjo recursivamente;
- Combinamos os dois subarranjos para obter a ordenação do arranjo inicial.

Temos que tomar duas decisões para tornar o processo concreto:

- Como separar os elementos em dois subarranjos?
- Como combinar os dois subarranjos ordenados?

# Ordenação por intercalação

Como separar os elementos em dois subarranjos?

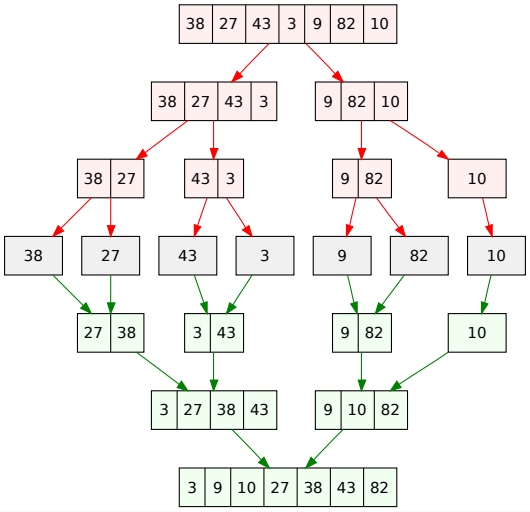
- Dividindo o arranjo ao meio;

Como combinar os dois subarranjos ordenados?

- Fazendo a intercalação em ordem dos elementos dos subarranjos;

Este algoritmo é conhecido como **ordenação por intercalação** (*merge sort*).

# Ordenação por intercalação



# Ordenação por intercalação

Projete uma função que implemente o algoritmo de ordenação por intercalação.

```
def ordena_intercalacao(lst: list[int]):
    # Se o problema não é trivial
    if len(lst) > 1:

        # Divide em dois subproblemas
        m = len(lst) // 2
        a = lst[:m]
        b = lst[m:]

        # Conquista recursivamente
        ordena_intercalacao(a)
        ordena_intercalacao(b)

        # Combina as soluções
        intercala(lst, a, b)
```

Projete uma função que implemente a intercalação.

```
def intercala(lst: list[int], a: list[int], b: list[int]):
    '''
    Faz a intercalação em ordem não decrescente dos
    elementos de *a* e *b* em *lst*.
    Requer que len(lst) = len(a) + len(b).
    Requer que a e b estejam em ordem não decrescente.

    Exemplos
    >>> lst = [0, 0, 0, 0, 0, 0, 0]
    >>> intercala(lst, [1, 6], [3, 5, 6, 8, 10])
    >>> lst
    [1, 3, 5, 6, 6, 8, 10]
    >>> intercala(lst, [3, 5, 6, 8, 10], [1, 6])
    >>> lst
    [1, 3, 5, 6, 6, 8, 10]
    '''
```

# Ordenação por intercalação

Projete uma função que implemente o algoritmo de ordenação por intercalação.

```python
def ordena_intercalacao(lst: list[int]):
    # Se o problema não é trivial
    if len(lst) > 1:

        # Divide em dois subproblemas
        m = len(lst) // 2
        a = lst[:m]
        b = lst[m:]

        # Conquista recursivamente
        ordena_intercalacao(a)
        ordena_intercalacao(b)

        # Combina as soluções
        intercala(lst, a, b)
```

```python
def intercala(lst: list[int], a: list[int], b: list[int]):
    assert len(lst) == len(a) + len(b)
    i, j, k = 0, 0, 0
    while i < len(a) and j < len(b):
        if a[i] <= b[j]:
            lst[k] = a[i]
            i += 1
        else:
            lst[k] = b[j]
            j += 1
        k += 1
    while i < len(a): # Copia o restante de a
        lst[k] = a[i]
        i += 1
        k += 1
    while j < len(b): # Copia o restante de b
        lst[k] = b[j]
        j += 1
        k += 1
```

# Ordenação por intercalação

Projete uma função que implemente o algoritmo de ordenação por intercalação.

```
def ordena_intercalacao(lst: list[int]):
    # Se o problema não é trivial
    if len(lst) > 1:

        # Divide em dois subproblemas
        m = len(lst) // 2
        a = lst[:m]
        b = lst[m:]

        # Conquista recursivamente
        ordena_intercalacao(a)
        ordena_intercalacao(b)

        # Combina as soluções
        intercala(lst, a, b)
```

# Ordenação por intercalação

A forma mais comum de implementar a ordenação por intercalação é utilizando índices para informar o intervalo de ordenação / intercalação.

Note que nessa versão a cópia dos subarranjos geralmente é feita na função `intercala` e não em `ordena_intercala` como fizemos anteriormente.

```python
def ordena_intercalacao(lst: list[int], ini: int, fim: int):
    '''
    Ordena o subarranjo lst[ini:fim] em ordem não
    decrescente.
    Requer que 0 <= i <= fim <= len(lst).
    '''

    # Se o problema não é trivial
    if ini < fim - 1:

        # Divide em dois subproblemas
        meio = (ini + fim) // 2

        # Conquista recursivamente
        ordena_intercalacao(lst, ini, meio)
        ordena_intercalacao(lst, meio, fim)

        # Combina as soluções
        intercala(lst, ini, meio, fim)
```

Projete essa versão de `intercala`.

No projeto de um algoritmo de divisão e conquista para ordenar um arranjo temos que tomar duas decisões:

- Como separar os elementos em dois subarranjos?
- Como combinar os dois subarranjos ordenados?

Como combinar dois arranjos ordenados sem precisar passar por todos os elementos?

Supondo que o arranjo de entrada `lst[0:n]` seja dividido em `lst[0:p]` e `lst[p:n]`, o que é necessário para que após a ordenação de `lst[0:p]` e `lst[p:n]` o arranjo `lst[0:n]` fique ordenado sem precisarmos fazer mais nada?

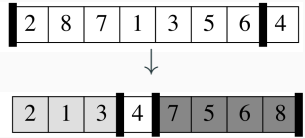
Os elementos de `lst[0:p]` devem ser menores ou iguais aos elementos de `lst[p:n]`!

# Particionamento

Então, o que precisamos fazer?

Projetar uma função que **particione** um arranjo em duas partes, uma com os “menores” e outra com os demais elementos (“maiores”).

Para isso precisamos de um “pivô” para determinar em que parte cada elemento deve ficar.




Faça a especificação da função que faz o particionamento de um arranjo. A função deve devolver o índice que separa as duas partes.

```
def particiona(lst: list[int], ini: int, fim: int) -> int:
    '''
    Reorganiza os elementos de lst[ini:fim] e devolve um
    índice p de maneira que os elementos de lst[ini:p]
    são menores ou iguais que lst[p:fim].

    Exemplos
    >>> lst = [2, 8, 7, 1, 3, 5, 6, 4]
    >>> particiona(lst, 0, len(lst))
    3
    >>> lst
    [2, 1, 3, 4, 8, 7, 5, 6]
    '''
```

# Ordenação por particionamento

O algoritmo de ordenação de divisão e conquista baseado na função de particionamento é chamado de **ordenação por particionamento** ou **quick sort**.

Implemente a ordenação por particionamento.

# Ordenação por partição

```python
def ordena_particionamento(lst: list[int],
                           ini: int,
                           fim: int):
    '''
    Ordena o subarranjo lst[ini:fim] em ordem
    não decrescente.

    Requer que 0 <= i <= fim <= len(lst).
    '''

    # Se o problema não é trivial
    if ini < fim - 1:

        # Divide em dois subproblemas
        p = particiona(lst, ini, fim)

        # Conquista recursivamente
        ordena_particionamento(lst, ini, p)
        ordena_particionamento(lst, p, fim)

        # As soluções já estão combinadas!
```

```python
def ordena_intercalacao(lst: list[int],
                        ini: int,
                        fim: int):

    # Se o problema não é trivial
    if ini < fim - 1:

        # Divide em dois subproblemas
        meio = (ini + fim) // 2

        # Conquista recursivamente
        ordena_intercalacao(lst, ini, meio)
        ordena_intercalacao(lst, meio, fim)

        # Combina as soluções
        intercala(lst, ini, meio, fim)
```

# Ordenação por partição

```
def ordena_particionamento(lst: list[int],
                           ini: int,
                           fim: int):

    # Se o problema não é trivial
    if ini < fim - 1:

        # Divide em dois subproblemas
        p = particiona(lst, ini, fim)

        # Conquista recursivamente
        ordena_particionamento(lst, ini, p)
        ordena_particionamento(lst, p, fim)

        # As soluções já estão combinadas!
```

# Particionamento

Como podemos implementar a função `particiona`?

Uma forma simples é usar arranjos auxiliares para armazenar as duas partições enquanto elas são construídas.

```
def particiona(lst: list[int], ini: int, fim: int) -> int:
    pivo = lst[fim - 1]
    menores = []
    maiores_iguais = []
    for i in range(ini, fim - 1):
        if lst[i] < pivo:
            menores.append(lst[i])
        else:
            maiores_iguais.append(lst[i])
    # Copia os menores do que o pivo para lst
    for i in range(len(menores)):
        lst[i] = menores[i]
    # Copia o pivo para lst
    p = len(menores)
    lst[p] = pivo
    # Copias os maiores ou iguais ao pivo para lst
    for j in range(len(maiores_iguais)):
        lst[p + j + 1] = maiores_iguais[j]
    return p # Retorna o índice do pivo
```

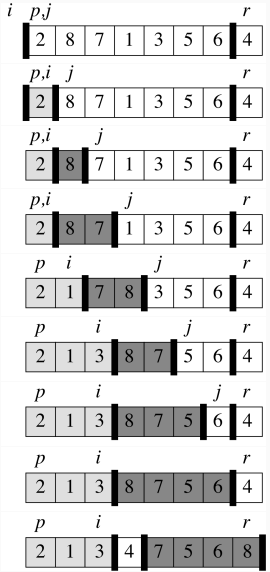
# Particionamento in-loco

As duas formas mais comum de fazer o particionamento in-loco são:

A forma sugerida por Tony Hoare, criador do quick sort, é manter dois índices, um para a partição do início do arranjo com os menores, e outro para a partição no final com os maiores. Os índices movem em direção ao meio e os elementos são trocados de lugar quando necessário.

A outra forma é o particionamento de Lomuto. Nesse esquema tanta a partição dos menores fica no início do arranjo e a dos maior logo em seguida.

# Particionamento de Lomuto




```python
def particiona(lst: list[int], ini: int, fim: int) -> int:
    pivo = lst[fim - 1]
    i = ini - 1
    for j in range(ini, fim - 1):
        if lst[j] <= pivo:
            i += 1
            lst[i], lst[j] = lst[j], lst[i]
    lst[i + 1], lst[fim - 1] = lst[fim - 1], lst[i + 1]
    return i + 1
```

# Referências

Thomas H. Cormen et al. Introduction to Algorithms. 3rd edition. Capítulos 6, 7 e 8.