Aula 13: Algoritmos de Ordenação MergeSort e QuickSort

Luís Felipe

UFF

10 de Novembro de 2025

Luís Felipe

10/11/25

Algoritmos de ordenação

- Em aulas passadas estudamos alguns algoritmos de ordenação.
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
- Cada um desses algoritmos possui complexidade quadrática em função do tamanho da entrada no pior caso.
- Será que dá para fazer melhor?
  - Veremos que sim!

Luís Felipe

10/11/25

MergeSort

Problema:

- Temos uma lista v de inteiros de tamanho n.
- Objetivo: ordenar v de forma crescente.

Técnica utilizada: Dividir e Conquistar com recursão.

Etapas:

- Dividir: quebrar P em subproblemas menores.
- Resolver: subproblemas são resolvidos recursivamente.
- Conquistar: unir soluções dos subproblemas para resolver P.

Luís Felipe

10/11/25

MergeSort: Ideia Geral

Dividir e conquistar

- Dividir: separar a lista em duas sublistas de tamanhos $\approx n/2$.
- Recursão: ordenar cada sublista.
- Conquistar: intercalar as sublistas ordenadas.

Luís Felipe

10/11/25

Função mergeSort

```
1  def mergeSort(v, ini, fim, aux):
2      meio = (fim + ini) // 2
3      if ini < fim:                          # lista tem pelo menos 2 elementos
4          mergeSort(v, ini, meio, aux)
5          mergeSort(v, meio+1, fim, aux)
6          merge(v, ini, meio, fim, aux)
```

Luís Felipe

10/11/25

Função merge (Fusão)

Recebe duas listas ordenadas e devolve uma lista ordenada contendo todos os elementos.

```
1  def merge(a, b):
2      i = 0; j = 0
3      c = []
4      while i < len(a) and j < len(b):
5          if a[i] <= b[j]:
6              c.append(a[i])
7              i += 1
8          else:
9              c.append(b[j])
10             j += 1
11     while i < len(a):
12         c.append(a[i])
13         i += 1
14     while j < len(b):
15         c.append(b[j])
16         j += 1
17     return c
```

Luís Felipe

10/11/25

Merge para sublistas

```
1  def merge(v, ini, meio, fim, aux):
2      i = ini; j = meio+1; k = 0
3      while i <= meio and j <= fim:
4          if v[i] <= v[j]:
5              aux[k] = v[i]; i += 1
6          else:
7              aux[k] = v[j]; j += 1
8          k += 1
9      while i <= meio:
10         aux[k] = v[i]; i += 1; k += 1
11     while j <= fim:
12         aux[k] = v[j]; j += 1; k += 1
13     i = ini; k = 0
14     while i <= fim:
15         v[i] = aux[k]
16         i += 1; k += 1
```

Luís Felipe

10/11/25

Exemplo de uso do MergeSort

```
1  def main():
2      v = [10, 8, 5, 3, 7, 13, 1, 6]
3      aux = [0 for _ in range(len(v))]
4      print("Lista original:", v)
5      mergeSort(v, 0, len(v)-1, aux)
6      print("Lista ordenada:", v)
7
8  main()
```

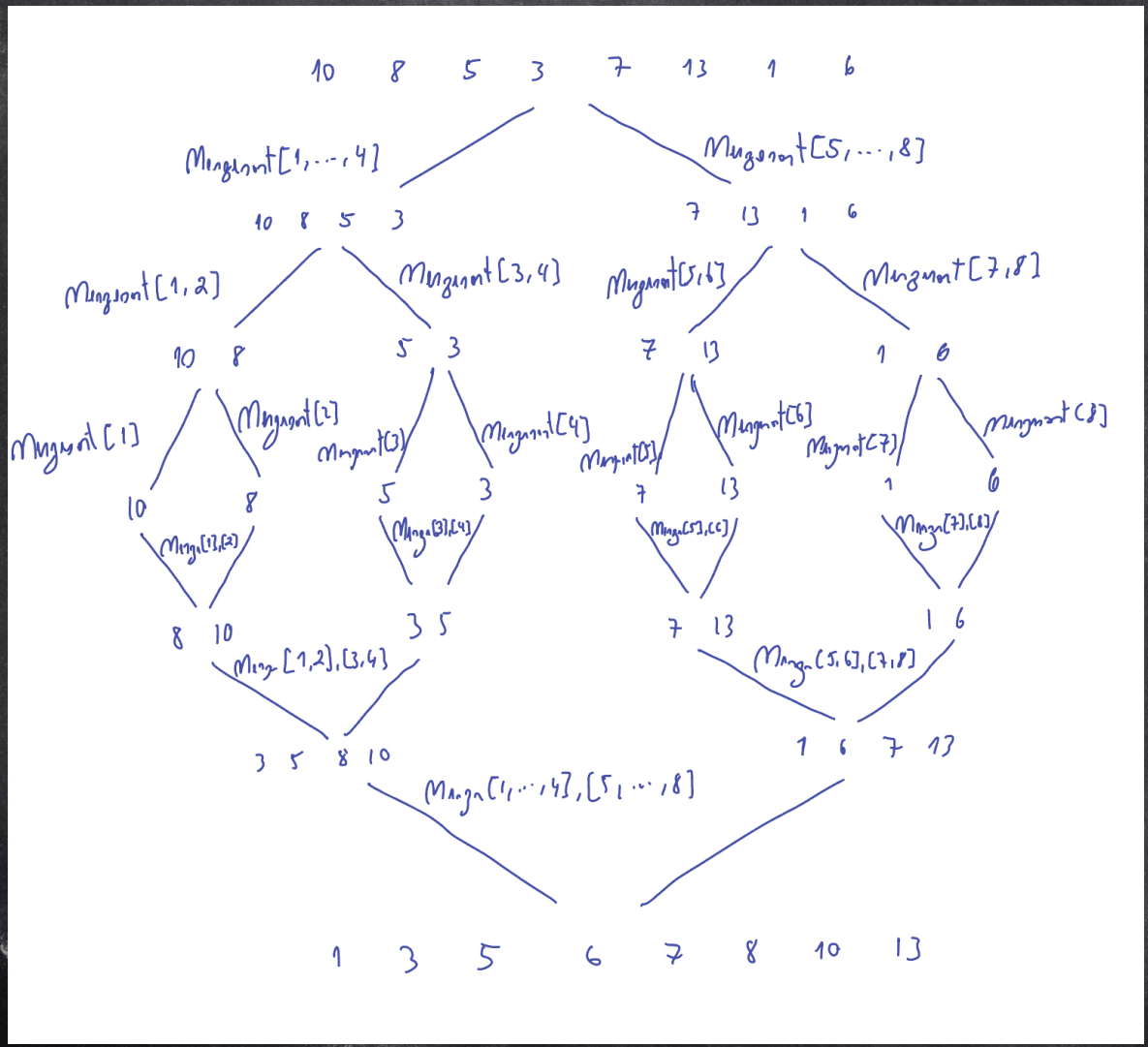
Apenas duas listas são criadas:

- v: a lista a ser ordenada
- aux: lista auxiliar com o mesmo tamanho de v

Luís Felipe

10/11/25

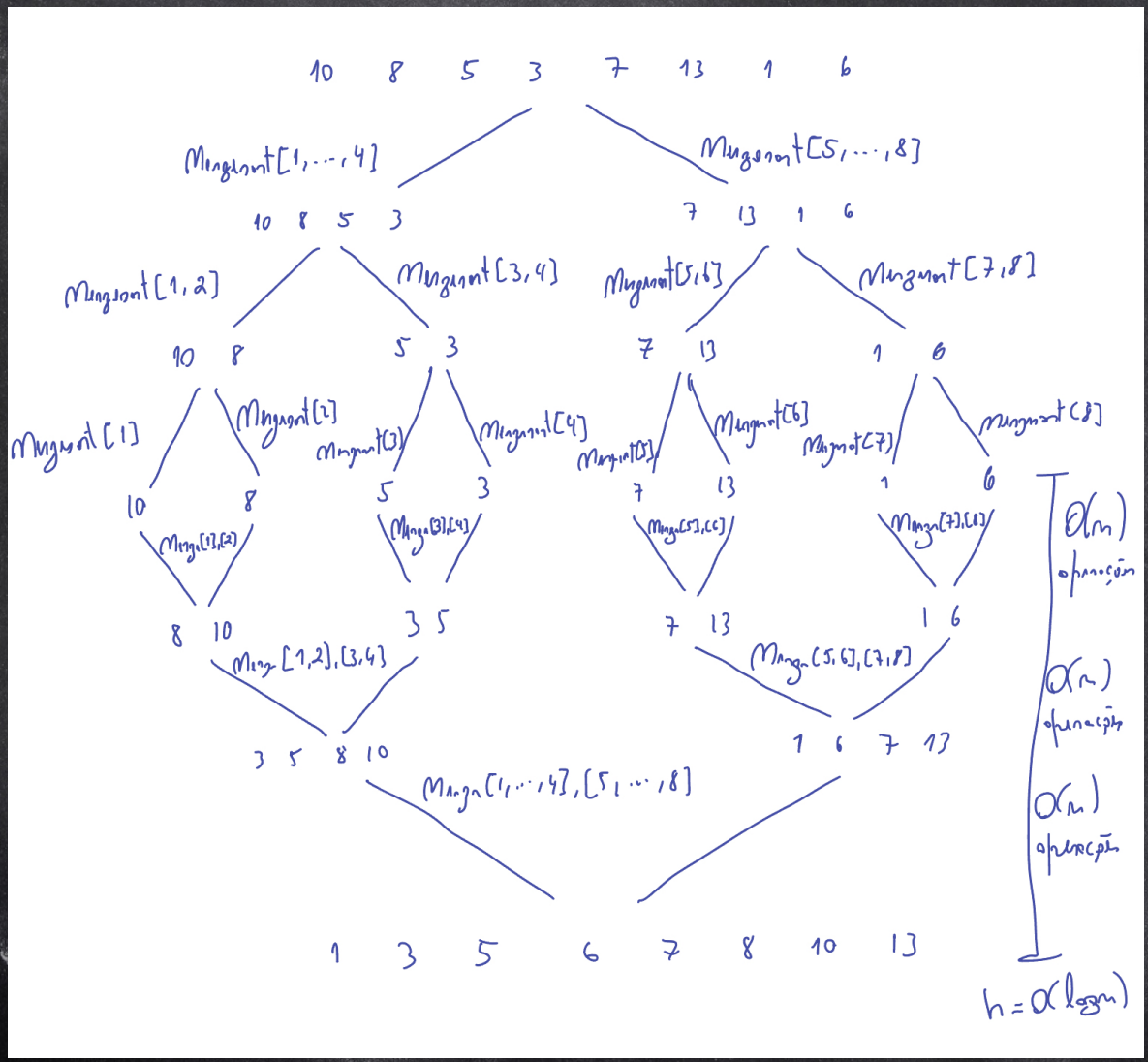
Exemplo



Luís Felipe

10/11/25

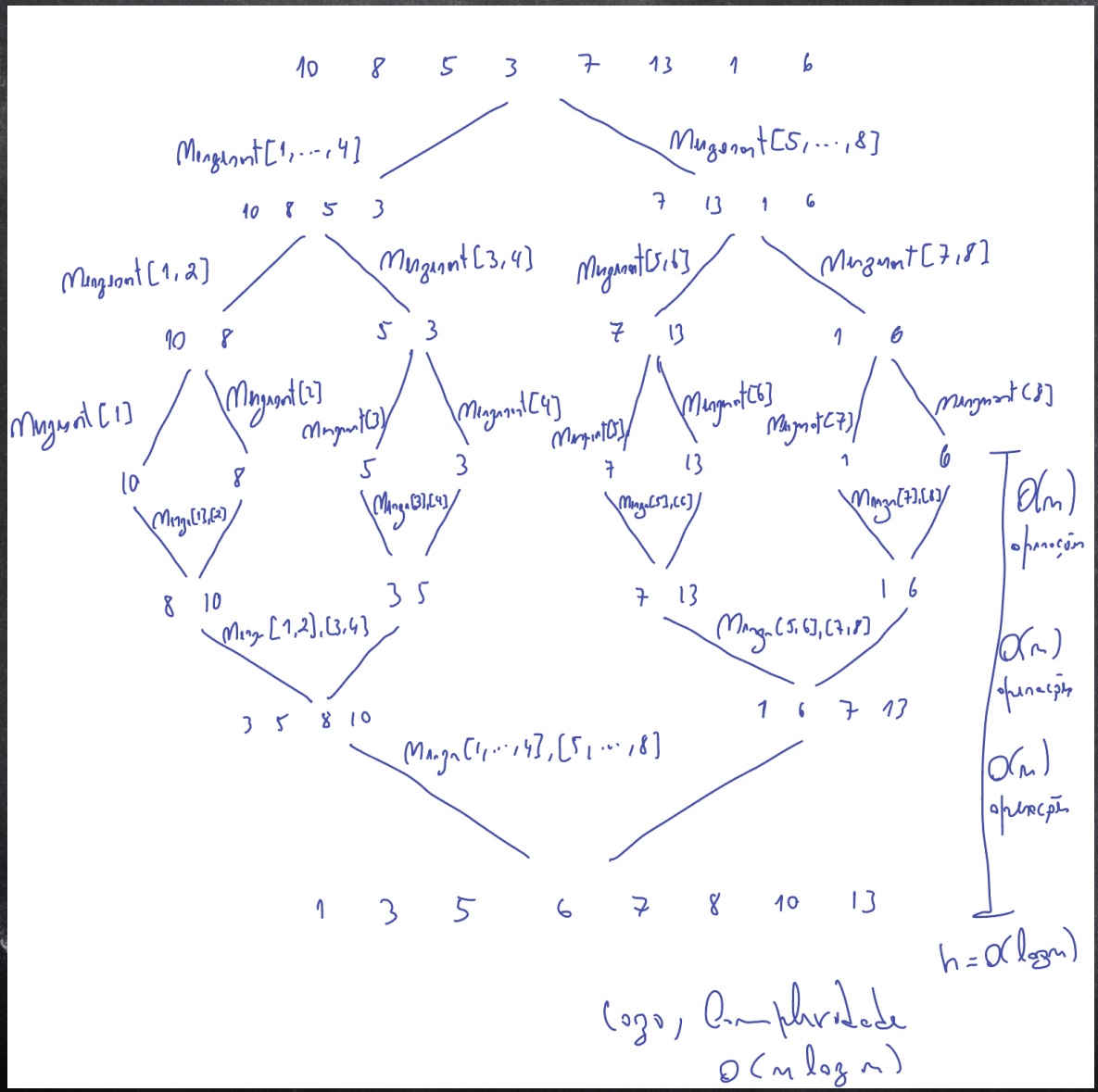
Complexidade vista no exemplo



Luís Felipe

10/11/25

Complexidade vista no exemplo



Luís Felipe

10/11/25

Complexidade do MergeSort

A cada descida na árvore de recursão, cada sublista reduz pela metade. Ou seja:

- No 1o nível, temos uma lista de tamanho n.
- No 2o nível, temos duas listas de tamanho $n/2$
- No 3o nível, temos quatro listas de tamanho $n/4$.
- De modo geral, no k-ésimo nível, temos 2^{k-1} listas de tamanho $\frac{n}{2^{k-1}}$.

A base da recorrência é quando cada lista tem tamanho 1. Ou seja, quando temos n listas de tamanho $\frac{n}{2^{k-1}} = 1$. Isso acontece quando $k = O(\log n)$. Assim, temos $O(\log n)$ níveis.

Como cada nível precisamos aplicar a função merge entre os elementos das listas e dessa forma percorremos todos os n elementos em cada nível.

Ou seja, temos $O(n \log n)$ operações no total.

Luís Felipe

10/11/25

QuickSort

Problema:

- Temos uma lista v de inteiros de tamanho n.
- Objetivo: ordenar v em ordem crescente.

Técnica utilizada: Dividir e Conquistar com recursão.

- Dividir: quebrar o problema P em subproblemas menores.
- Resolver: resolver recursivamente cada subproblema.
- Conquistar: unir as soluções dos subproblemas.

Luís Felipe

10/11/25

QuickSort: Ideia Geral

- Ordena lista entre posições `ini` e `fim`.
- Dividir:
  - Escolhe um elemento como pivô.
  - Particiona a lista em torno do pivô. Elementos menores que o pivô ficam a sua esquerda, elementos maiores a sua direita.
- Recursão: ordenar sublistas à esquerda e à direita do pivô.
- Conquistar: nada a fazer, particionamento garante ordenação local.

Luís Felipe

10/11/25

Um pouco mais a baixo nível

Pergunta: Como fazer a separação de L em S_1 e S_2?

OBS.: Considere todos elementos distintos.

1. Escolha o pivô x;
2. Afaste o pivô de L (ponha x após a última posição de L)
3. Utilize dois ponteiros i e j:
   - i é inicializado apontando para o primeiro elemento de L;
   - j é inicializado apontando para o último elemento de L.
4. Incrementamos a posição de i enquanto os elementos de L são menores que pivô x;
5. Decrementamos a posição de j enquanto os elementos de L são maiores que pivô x.

Luís Felipe

10/11/25

E quando pararmos?

- Ao sair do "enquanto", duas situações podem ocorrer:
  i) Se $i < j$, os elementos de L devem ser trocados e prosseguimos;
  ii) Se $i > j$, a posição está determinada. Troque o elemento com índice i com o pivô x.

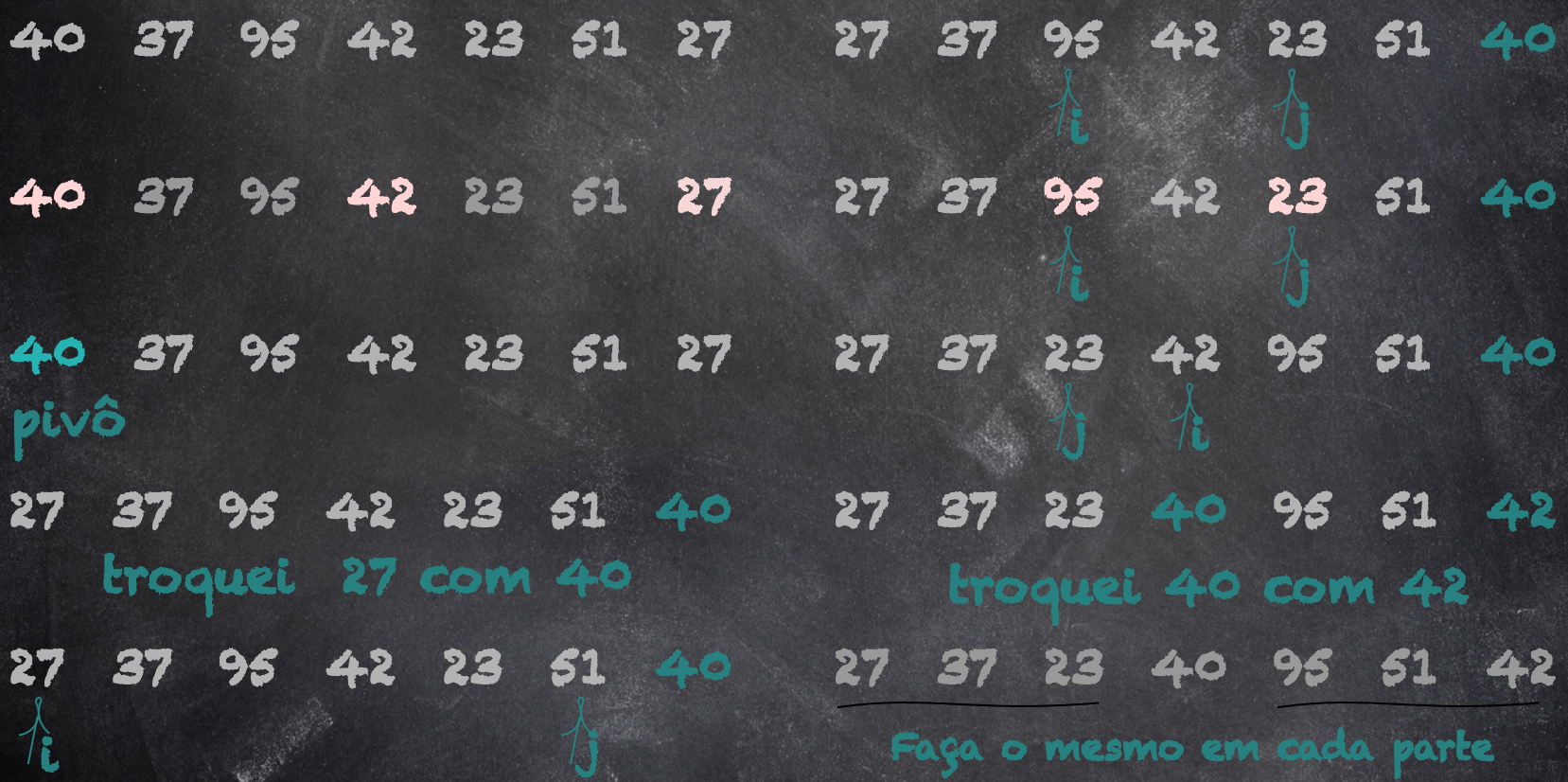
- Consequência: Todos os elementos a esquerda de x serão menores do que x e todos os elementos a direita de x serão maiores do que x.

Luís Felipe

10/11/25

Vamos passar uma vez?

Exemplo:



Luís Felipe

10/11/25

Quicksort

```python
def quicksort(L, ini, fim):
    # Caso base: sublista de tamanho 0 ou 1
    if fim - ini < 2:
        if fim - ini == 1 and L[ini] > L[fim]:
            L[ini], L[fim] = L[fim], L[ini]
        return

    # Escolha do pivô (pode ser o meio)
    mediana = (ini + fim) // 2
    L[mediana], L[fim] = L[fim], L[mediana]

    i = ini
    j = fim - 1
    key = L[fim]

    # Particionamento
    while j >= i:
        while i <= j and L[i] < key:
            i += 1
        while j >= i and L[j] > key:
            j -= 1
        if j >= i:
            L[i], L[j] = L[j], L[i]
            i += 1
            j -= 1

    # Coloca o pivô na posição correta
    L[i], L[fim] = L[fim], L[i]

    # Chama recursivamente para as duas metades
    quicksort(L, ini, i - 1)
    quicksort(L, i + 1, fim)

print("Lista ordenada:", ordenados)
```

```python
def aplicar_quicksort(L):
    # Função que chama o quicksort
    if len(L) > 1:
        quicksort(L, 0, len(L) - 1)
    return L

valores = [38, 27, 43, 3, 9, 82, 10]
ordenados = aplicar_quicksort(valores)
```

Luís Felipe

10/11/25

Complexidade do QuickSort

- Caso médio: $O(n \log n)$.
- Pior caso: $O(n^2)$ (partições muito desbalanceadas).
- Pior caso ocorre, por exemplo, em listas já ordenadas se pivô fixo for usado.

Obs.: Conteúdo da disciplina de Análise e Projeto de Algoritmos.

Luís Felipe

10/11/25

Tratando o Pior Caso: Random QuickSort

- O pior caso do QuickSort ocorre quando o pivô divide mal o vetor, por exemplo, em listas já ordenadas.
- Para reduzir essa chance, podemos escolher o pivô de forma aleatória.
- Isso faz com que, estatisticamente, o algoritmo tenda ao comportamento médio, com tempo esperado de $O(n \log n)$.
- Podemos usar a função `random.randint(a,b)` da biblioteca `random`, que retorna um número inteiro aleatório entre a e b.

Ideia principal: Escolher o pivô aleatoriamente a cada chamada recursiva, trocando-o com o último elemento antes do particionamento.

Luís Felipe

10/11/25

Implementação: Random QuickSort

```python
import random

def random_quicksort(L, ini, fim):
    if fim - ini < 2:
        if fim - ini == 1 and L[ini] > L[fim]:
            L[ini], L[fim] = L[fim], L[ini]
        return

    pivo = random.randint(ini, fim)             # Escolhe pivô aleatoriamente
    L[pivo], L[fim] = L[fim], L[pivo]
    key = L[fim]

    i = ini
    j = fim - 1

    while j >= i:
        while i <= j and L[i] < key:
            i += 1
        while j >= i and L[j] > key:
            j -= 1
        if j >= i:
            L[i], L[j] = L[j], L[i]
            i += 1
            j -= 1

    L[i], L[fim] = L[fim], L[i]

    random_quicksort(L, ini, i - 1)
    random_quicksort(L, i + 1, fim)

def aplicar_random_quicksort(L):
    if len(L) > 1:
        random_quicksort(L, 0, len(L) - 1)
    return L
```