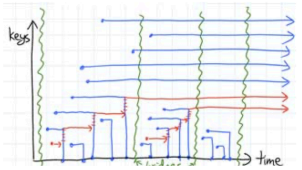
Estrutura de Dados

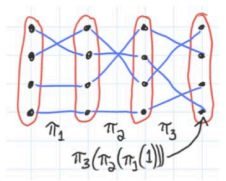
Ordenação e Busca

Curso de Análise e Desenvolvimento de Sistemas

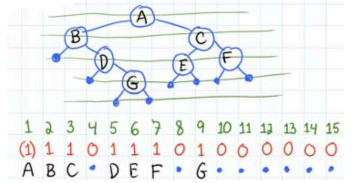
Última revisão: 2024.2



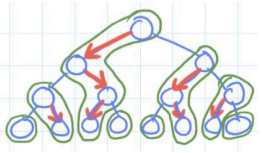
TIME TRAVEL



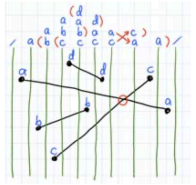
DYNAMIC GRAPHS



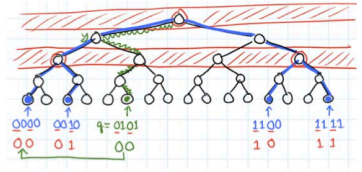
SUCCINCT



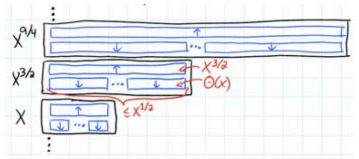
DYNAMIC OPTIMALITY



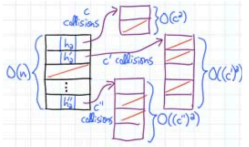
GEOMETRY



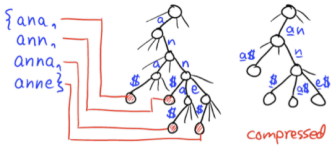
INTEGERS



MEMORY HIERARCHY



HASHING



STRINGS

Ordenação e Busca

Prof. **Saulo Oliveira** | @sauloafoliveira

Isenção de responsabilidade

Isenção de responsabilidade

Os códigos apresentados em Python têm como objetivo principal proporcionar aprendizado e entendimento sobre os conceitos abordados. Embora sejam funcionais e didáticos, não representam necessariamente as soluções mais otimizadas ou recomendadas para aplicações em produção. Sempre considere ajustar e aprimorar os exemplos conforme as necessidades específicas do seu projeto e as boas práticas de programação.

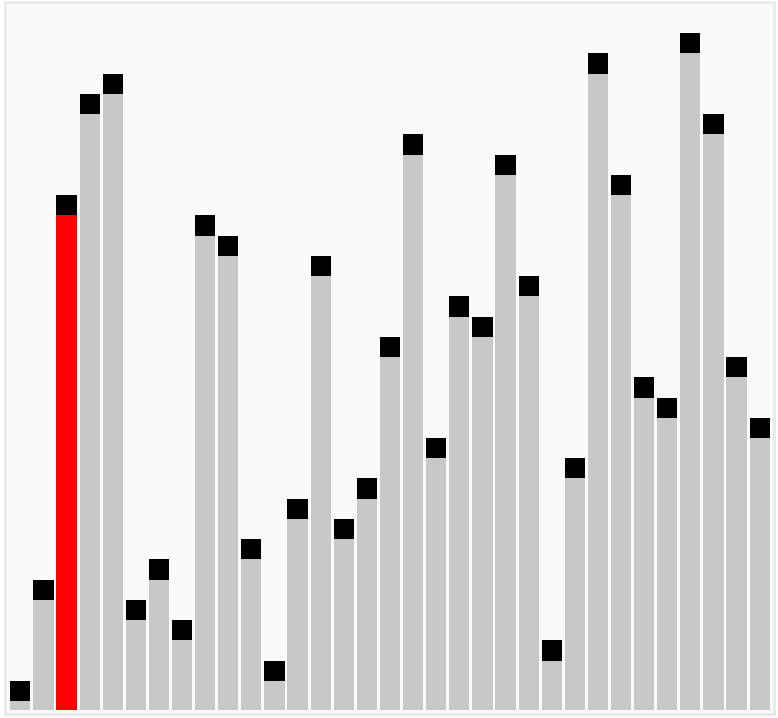
É verdade esse bilhete!

BUBBLE SORT

Bubble sort

Talvez a estratégia mais simples para se ordenar seja comparar pares de itens consecutivos e permutá-los, caso estejam fora de ordem.

```
def bubble_sort(data: list[Comparable]):
    n = len(data)
    for i in range(n):
        for j in range(n):
            if data[j] > data[i]:
                data[i], data[j] = data[j], data[j]
```



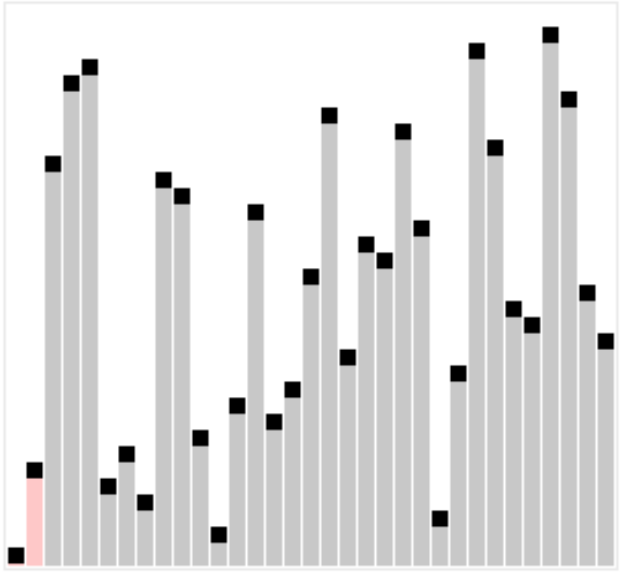
Ordene, a seguir, na mão, usando o BubbleSort: `[92, 80, 71, 63, 55, 41, 39, 27, 14]`.

SELECTION SORT

Selection sort

A estratégia básica desse método é, em cada fase, selecionar um menor item ainda não ordenado e permutá-lo com aquele que ocupa a sua posição na sequência ordenada.

```
def selection_sort(data: list[Comparable]):
    n = len(data)
    for i in range(n):
        k = min(data[i:])
        data[i], data[k] = data[k], data[i]
```



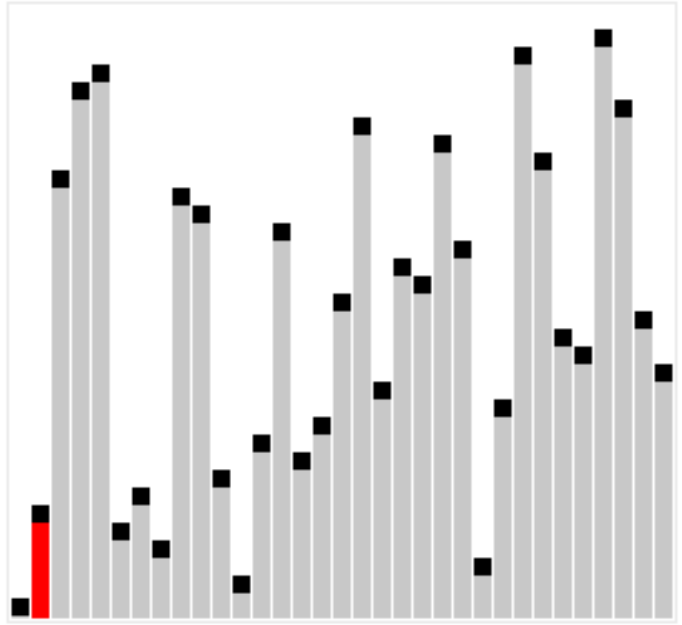
Ordene, a seguir, na mãe, usando o SelectionSort: `[46, 55, 59, 14, 38, 27]`.

Insertion sort

Insertion sort

Ao ordenar uma sequência $L = [L_O, L_D]$, esse método considera uma subsequência ordenada L_O e outra desordenada L_D. Então, em cada fase, um item é removido de L_D e inserido em sua posição correta dentro de L_O.

```
for insertion_sort(data: list[Comparable]):

    for i in range(2, len(data)):
        x = data[i]
        j = i - 1

        while j >= 1 and x < data[j]:
            j = j - 1
            data[ j + 1 ] = data[j]

        data[j + 1] = x
```

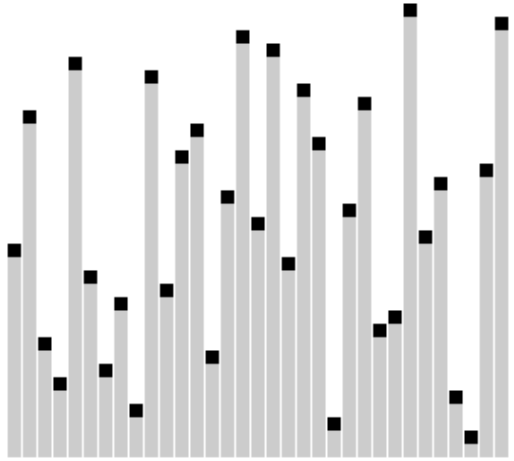


Exercite a ordenação acima na sequência: `[82, 50, 71, 63, 85, 43, 39, 97, 14]`.

Merge sort

Merge sort

O funcionamento do Merge Sort baseia-se em uma rotina fundamental cujo nome é `merge`. Primeiro vamos entender como ele funciona e depois vamos ver como sucessivas execuções de `merge` ordena uma sequência.



A rotina de `merge` é a que combina duas sequências ordenados em um outro também ordenado:

- Uma das duas partes da sequência será consumida em sua totalidade antes da outra;
- Basta então fazer o `append` de todos os elementos faltantes;
- Merge Sort é um algoritmo de divisão-e-conquista;
- Na **divisão**, dividimos recursivamente na metade até que sobre apenas um elemento. Na **conquista**, combinamos duas sequências ordenados em uma também ordenada.

Merge sort

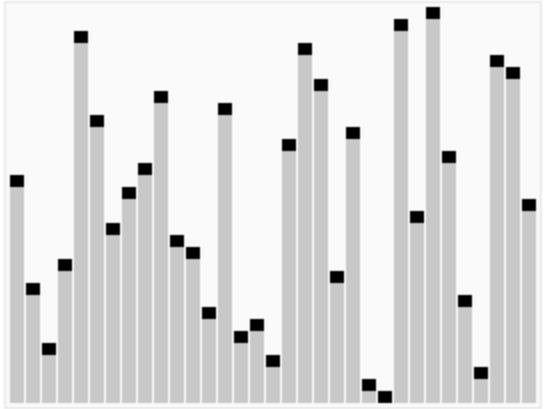
```python
def merge(a: list[int], b: list[int]):

    c = []

    while len(a) > 0 and len(b) > 0:

        if a[0] > b[0]:
            c.append( b[0] )
            b.pop(0)
        else:
            c.append( a[0] )
            a.pop(0)

    c.extend(a)
    c.extend(b)

    return c
```

```python
def merge_sort(data: list[int]):

    if len(data) < 2:
        return data

    # divisão
    meio = len(data) // 2

    esquerda = merge_sort(data[:meio])
    direita  = merge_sort(data[meio:])

    # conquista
    return merge(esquerda, direita)
```

QUICK SORT

Quick sort

O Quick Sort baseia-se em uma rotina fundamental cujo nome é `particionamento`. Particionar significa escolher um valor qualquer presente, chamado de **pivot**, e colocá-lo em uma posição tal que todos os elementos à esquerda são menores ou iguais e todos os elementos à direita são maiores.



```
def quick_sort(data: list[Comparable]):

    if len(data) <= 1:
        return data

    pivot = data[0]

    m, M = partition(data[1:])

    return quicksort(m) + [pivot] + quicksort(M)
```

```
def partition(data: list, pivot: Any) -> tuple:

    m = [x for x in data if x <= pivot]
    M = [x for x in data if x > pivot]

    return m, M
```

Análise de complexidade

Segue uma tabela de comparação com as notações dos algoritmos de ordenação.

Algoritmo/Tempo	Melhor	Médio	Pior
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insert sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Busca

Busca linear

A busca é o processo em que se determina se um particular elemento x é membro de uma determinada sequência $\mathcal{S}$. Dizemos que a busca tem sucesso se $x \in \mathcal{S}$ e que fracassa em cc.

```python
def linear_search(data: list, x: Any) -> bool:

    for y in data:

        if x == y:
            return True

    return False
```

No pior caso, quando o item procurado não consta na sequência, a busca linear precisa verificar todos os elementos.

A vantagem da busca linear é que ela sempre funciona, independentemente da seuquência estar ou não ordenada. A desvantagem é que ela para encontrar um determinado item `x`, precisa examinar todos os itens que precedem `x`.

Busca binária

Se não sabemos nada a respeito da ordem em que os itens aparecem, o melhor que podemos fazer é uma busca linear. Entretanto, se os itens aparecem ordenados, podemos usar um método de busca muito mais eficiente.

Analogia com um Dicionário do mundo real

Esse método é semelhante àquele que usamos quando procuramos uma palavra num dicionário: primeiro abrimos o dicionário numa página aproximadamente no meio; se tivermos sorte de encontrar a palavra nessa página, ótimo; senão, verificamos se a palavra procurada ocorre antes ou depois da página em que abrimos e então continuamos, mais ou menos do mesmo jeito, procurando a palavra na primeira ou na segunda metade do dicionário...

Como a cada comparação realizada o espaço de busca reduz-se aproximadamente à metade, esse método é denominado busca binária.

Busca binária

```python
def binary_search(data: list[Sorted], x: Any) -> bool:

    # Caso base: intervalo vazio
    if len(data) == 0:
        return False

    pos_meio = len(data) // 2
    meio = data[pos_meio]

    if meio == x:
        return True

    if x < meio:
        return binary_search(data[:meio], x)

    return binary_search(data[meio:], x)
```

O número de comparações feitas pelo algoritmo de busca binária tende a ser muito menor que aquele feito pela busca linear.

Exemplo. Para encontrar um item num dentro 5000 itens, o número de comparações chega se reduzir a 13.

Tamanho reduz na ordem de $\log_2 n$:

$$5000 \Rightarrow 2500 \Rightarrow 1250 \Rightarrow 625 \Rightarrow 312 \Rightarrow 156 \Rightarrow 78 \Rightarrow 39 \Rightarrow 19 \Rightarrow 9 \Rightarrow 4 \Rightarrow 2 \Rightarrow 1 \Rightarrow 0$$

Referências

- João Arthur Brunet. **Ordenação por Comparação: Merge Sort.** Disponível em: https://joaoarthurbm.github.io/eda/posts/merge-sort/.
- Silvio Lago. **Ordenação e Busca**. Disponível em: https://www.ime.usp.br/~slago/slago-ordena-busca.pdf.