

Análise e Desenvolvimento de Sistemas

**ALGORITMOS E ESTRUTURA DE DADOS EM PYTHON**

RELATÓRIO DE AULAS PRÁTICAS

Nome: : Igor Pereira da Silva

RA: 2537577

Polo de matrícula: São Paulo – Anchieta

Local da realização da Aula Prática: UNIP - Anchieta

Ano da postagem: 2025

Professor: Fabio de Assis

## TÍTULO DA ATIVIDADE (ROTEIRO OU AULA): IMPLEMENTANDO ALGORITMOS EM PYTHON

**ORIENTAÇÕES**:

Cada aluno (ou equipe) deve produzir um relatório curto (1 a 2 páginas) contendo:

### Resumo Teórico:

* Explicar, com palavras próprias, o que é lógica de programação e por que ela é importante.

Lógica de programação é basicamente a forma de pensar que você usa para resolver problemas passo a passo, como se estivesse dando instruções para um amigo fazer alguma coisa. Quando a gente programa, a gente precisa organizar essas instruções de um jeito que o computador entenda e consiga executar sem erro. Ela é importante porque, sem essa forma de pensar clara e organizada, o programa pode não funcionar direito, ficar confuso ou travar. Além disso, a lógica ajuda a gente a pensar melhor na vida também, porque ensina a dividir problemas grandes em partes menores, encontrar soluções e ser mais organizado.

* Mencionar brevemente o que é pseudocódigo e fluxograma e como ajudam na organização de ideias.

O **pseudocódigo** é uma maneira de descrever a lógica de um algoritmo usando uma linguagem simples e próxima do português, sem se preocupar com a sintaxe específica de uma linguagem de programação. Ele serve para **planejar passo a passo** como um problema será resolvido, tornando mais fácil entender e organizar as ideias antes de programar. O **fluxograma** é uma representação gráfica do algoritmo, usando símbolos padronizados para indicar ações, decisões e fluxos do processo. Ele ajuda a visualizar a sequência de passos e as decisões envolvidas na resolução de um problema. Ambos são ferramentas que **organizam ideias,** permitindo que o raciocínio seja estruturado de forma clara e lógica, facilitando a implementação de algoritmos e a comunicação entre pessoas envolvidas no desenvolvimento.

* Citar as vantagens de usar Python para aprender programação.

**Vantagens de usar Python para aprender programação:**Python é uma linguagem de programação amplamente recomendada para iniciantes porque possui uma sintaxe simples e clara, parecida com a linguagem natural. Isso permite que os estudantes se concentrem na lógica do programa sem se perder em detalhes complexos de sintaxe. Além disso, Python possui uma grande quantidade de bibliotecas e recursos que facilitam desde cálculos matemáticos até desenvolvimento web e manipulação de dados, tornando o aprendizado mais prático e motivador.

* Definição de algoritmo, variável, condicional, laço;

**Definição de conceitos fundamentais:**

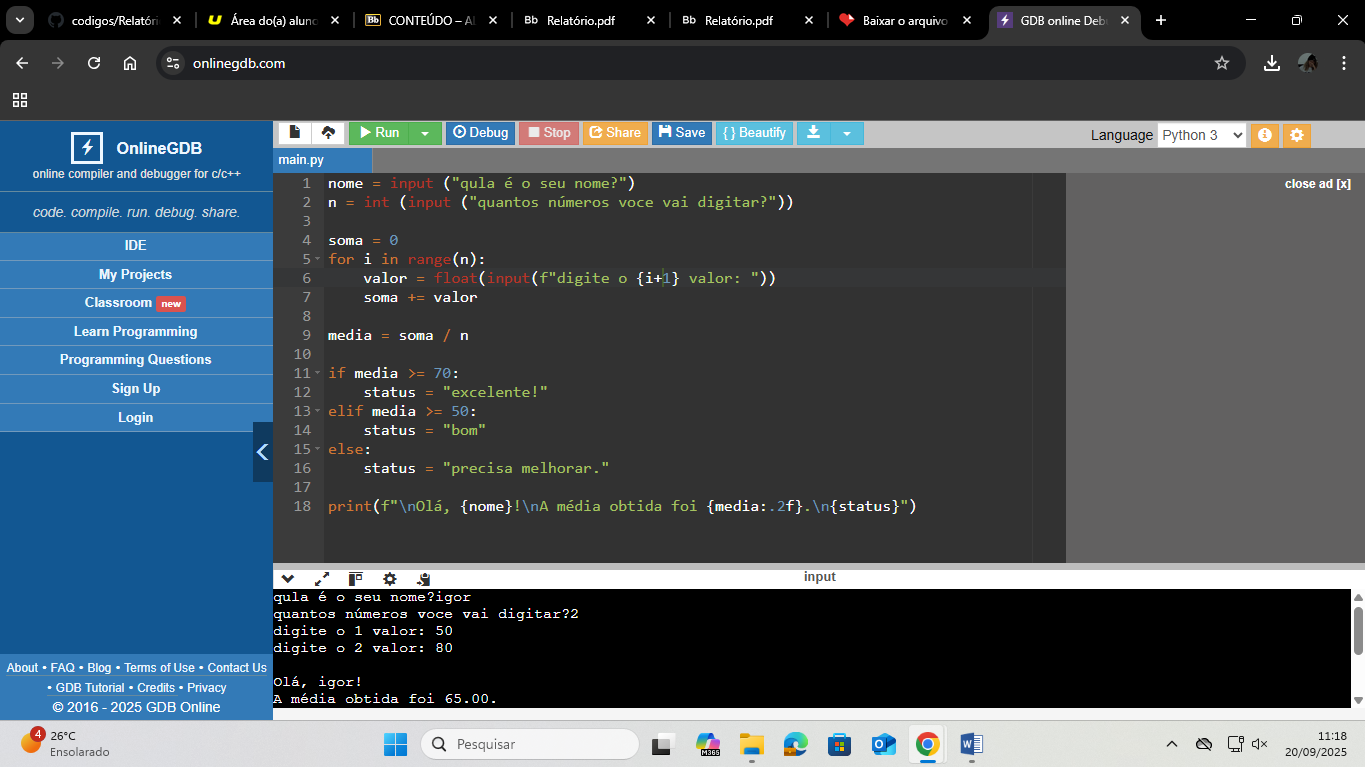
* **Algoritmo:** É uma sequência de passos lógicos e bem definidos que devem ser seguidos para resolver um problema ou realizar uma tarefa. Por exemplo, uma receita de bolo é um algoritmo, pois indica cada passo necessário para o resultado final.
* **Variável:** É um espaço na memória do computador que armazena um valor que pode ser alterado durante a execução do programa. Por exemplo, idade = 20 define uma variável chamada idade com o valor 20.
* **Condicional:** Estrutura que permite que o programa tome decisões com base em condições. Exemplo em Python:

**Definição de conceitos fundamentais:**

* **Algoritmo:** É uma sequência de passos lógicos e bem definidos que devem ser seguidos para resolver um problema ou realizar uma tarefa. Por exemplo, uma receita de bolo é um algoritmo, pois indica cada passo necessário para o resultado final.
* **Variável:** É um espaço na memória do computador que armazena um valor que pode ser alterado durante a execução do programa. Por exemplo, idade = 20 define uma variável chamada idade com o valor 20.
* **Condicional:** Estrutura que permite que o programa tome decisões com base em condições. Exemplo em Python:
* **Laço (ou loop):** Estrutura que repete um bloco de código enquanto uma condição for verdadeira, economizando esforço e evitando a repetição manual de comandos.

### Código-Fonte Comentado:

* Inserir o *código*-*fonte completo* da atividade proposta.



* Comentar as principais linhas, ressaltando o uso de print(), input(), variáveis etc.

input(): para ler dados do usuário (sempre como string).

int() e float(): para converter a entrada de texto em números inteiros e reais.

Variáveis: para armazenar informações como nome, n, soma, media, valor, status.

for: para repetir a entrada dos valores n vezes.

if, elif, else: para tomar decisões com base na média calculada.

print(): para exibir mensagens e resultados ao usuário.

F-strings (f""): para inserir variáveis diretamente dentro das strings.

**REFERÊNCIAS:**

CURSO EM VÍDEO. *Curso de Python*. Disponível em: https://www.cursoemvideo.com/course/python-3-mundo-1/. Acesso em: 20 set. 2025.

PYTHON SOFTWARE FOUNDATION. *Python documentation*. Disponível em: https://docs.python.org/3/. Acesso em: 20 set. 2025.

## TÍTULO DA ATIVIDADE (ROTEIRO OU AULA): ESTRUTURAS DE DADOS LINEARES EM PYTHON: LISTAS, PILHAS, FILAS E EFICIÊNCIA. NOTAÇÃO BIG-O

**ORIENTAÇÕES**:

Cada aluno deve produzir um relatório sucinto (2 a 3 páginas) contendo:

# Resumo Teórico

* Definição de listas, pilhas, filas e introdução à notação Big-O.

As listas em Python são estruturas de dados que armazenam elementos de forma sequencial, permitindo acesso por índice. São mutáveis, ou seja, podem ser alteradas após a criação. Internamente, são implementadas como vetores dinâmicos, o que permite acesso rápido a elementos (O(1)) e inserções/remoções com custo variável.  
A pilha é uma estrutura de dados do tipo **LIFO** (Last In, First Out), onde o último elemento inserido é o primeiro a ser removido. Operações principais:

* push: insere no topo.
* pop: remove do topo.
* peek: visualiza o topo.

**Complexidade:** Inserções e remoções ocorrem em tempo constante, O(1).

Filas (queue):

A fila é uma estrutura do tipo **FIFO** (First In, First Out), onde o primeiro elemento a entrar é o primeiro a sair. Operações principais:

* enqueue: insere no final.
* dequeue: remove do início.

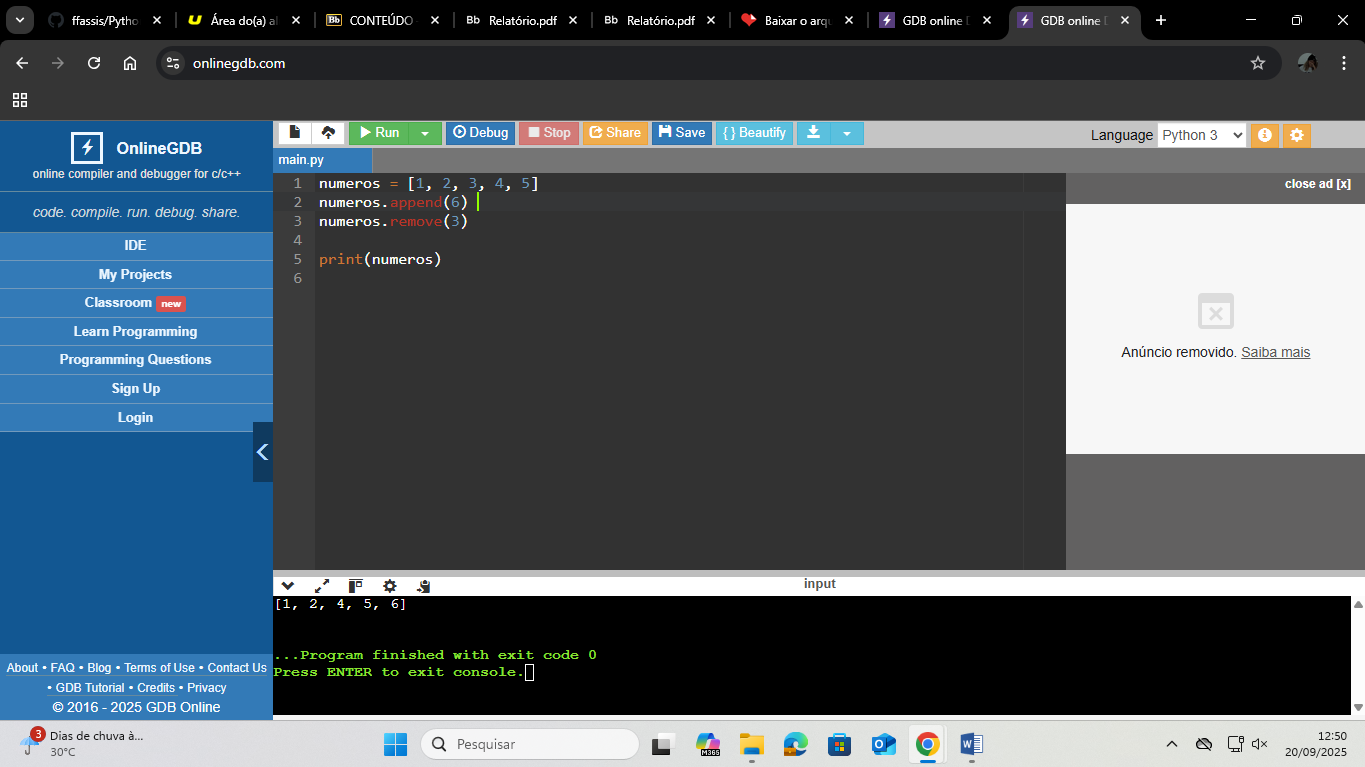
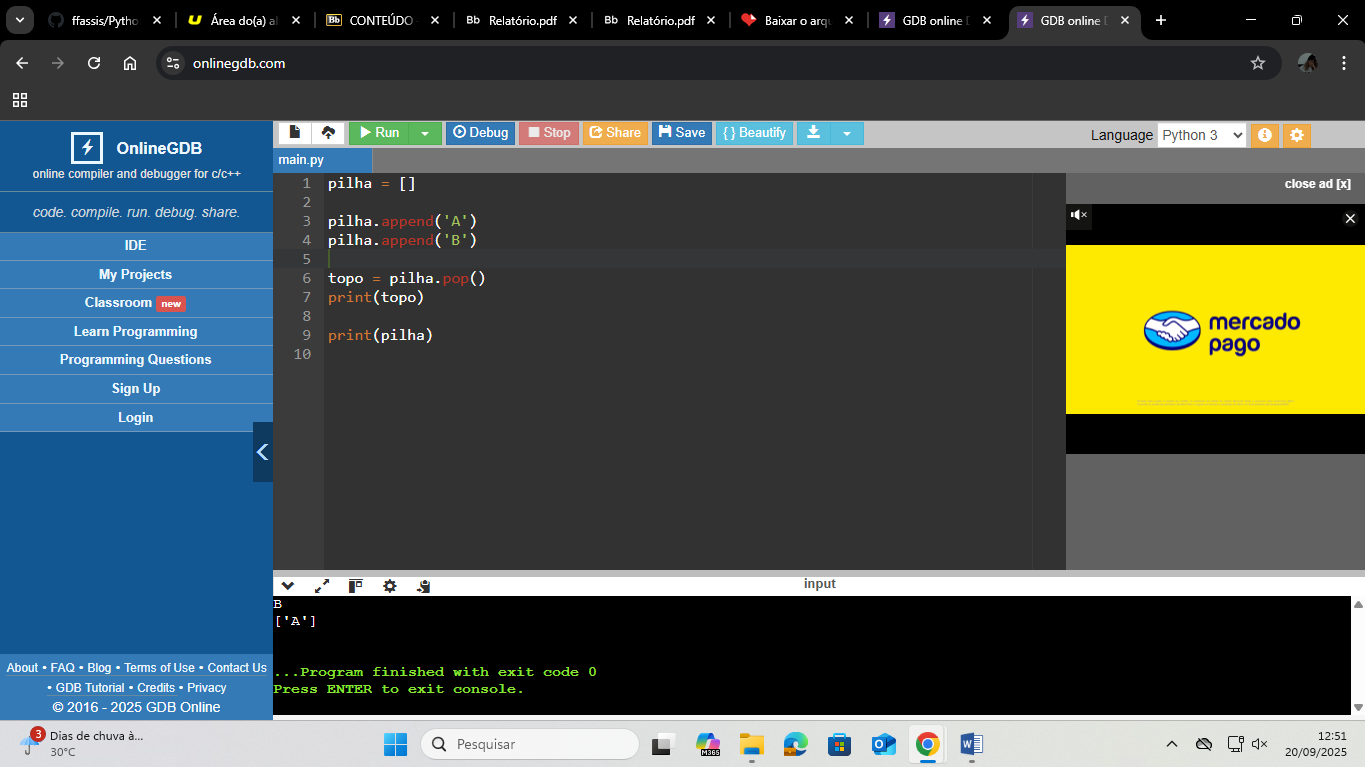
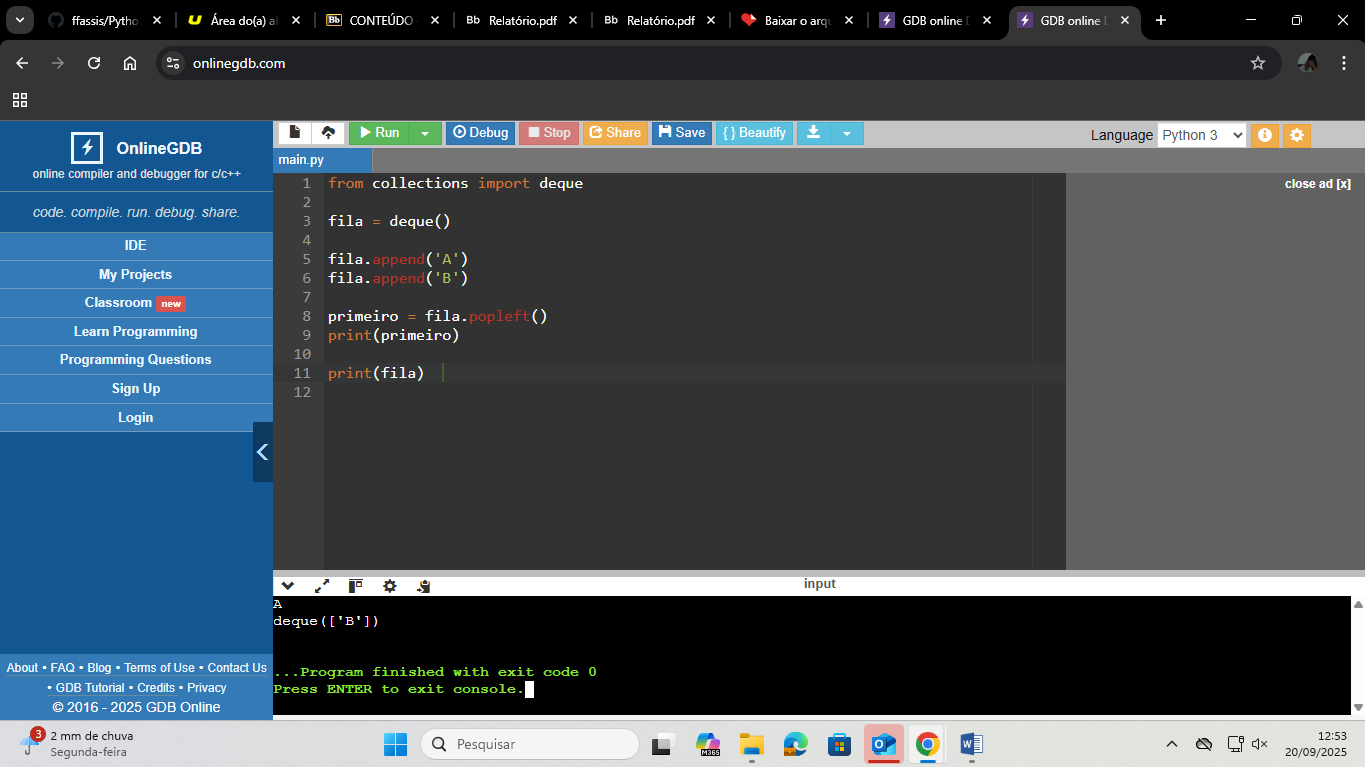
Em Python, usa-se collections.deque para implementar filas com eficiência (O(1) em inserções e remoções nas extremidades).

### Notação Big-O:

A notação Big-O descreve o comportamento assintótico de algoritmos em relação ao tempo ou espaço. Exemplos comuns:

* O(1): constante
* O(n): linear
* O(log n): logarítmica
* O(n^2): quadrática

Ela ajuda a avaliar a eficiência das estruturas e algoritmos.

* Código-fonte comentado das soluções desenvolvidas.
* Reflexão sobre desafios encontrados e critérios para selecionar cada estrutura;

Durante a implementação, um dos principais desafios foi escolher a estrutura de dados ideal para cada problema. Por exemplo, listas são versáteis, mas operações de inserção/remoção no meio são custosas (O(n)). Pilhas são úteis em algoritmos de backtracking, e filas são ideais para problemas de atendimento e simulação.

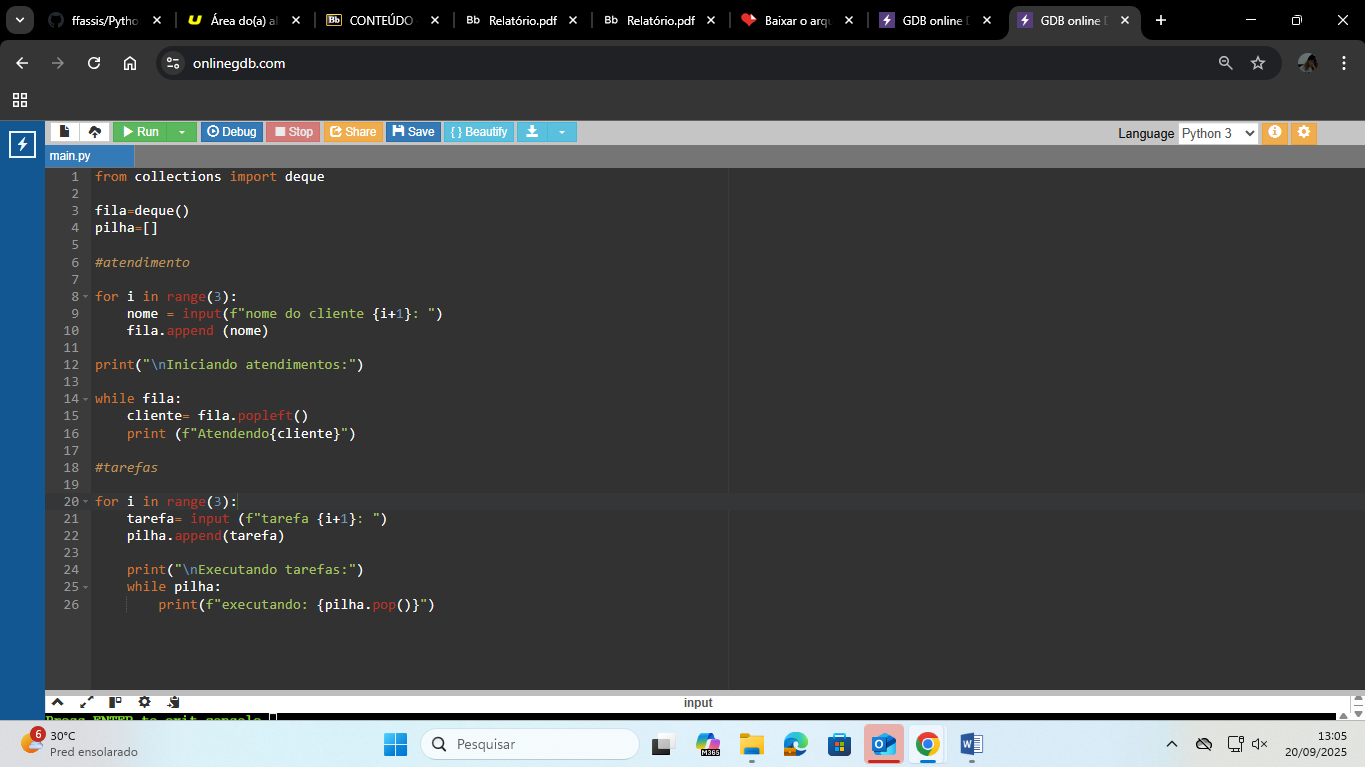
Outro desafio foi compreender a complexidade das operações. O uso da notação Big-O ajudou a prever o comportamento dos algoritmos em grande escala.

A escolha da estrutura foi guiada pelos seguintes critérios:

* **Ordem de acesso:** direto (lista), no topo (pilha), na frente (fila).
* **Complexidade das operações.**
* **Necessidade de ordenação ou reversão.**

# Códigos-Fontes Comentados

* Inserir o *código*-*fonte completo* da atividade proposta.



* Comentar as principais linhas, ressaltando o uso das estruturas de dados.

### Atendimento (Fila - FIFO)

* Utiliza deque para simular uma **fila de clientes.**
* A ordem de atendimento é a mesma ordem de chegada: o primeiro a entrar é o primeiro a ser atendido (**First In, First Out**).
* Métodos usados:

append() para enfileirar.

popleft() para desenfileirar.

### Tarefas (Pilha - LIFO)

* Utiliza uma lista (list) como **pilha de tarefas.**
* A ordem de execução é a **última tarefa adicionada é a primeira a ser executada (Last In, First Out**).
* Métodos usados:

append() para empilhar.

pop() para desempilhar.

**REFERÊNCIAS**:

ZIVIANI, Nivio. Projeto de Algoritmos com Implementações em Pascal e C. 3. ed. São Paulo: Cengage Learning, 2010.

GOODRICH, Michael T.; TAMASSIA, Roberto. Estruturas de Dados e Algoritmos em Python. Rio de Janeiro: LTC, 2016.

BEAZLEY, David. Python Essential Reference. 4. ed. Addison-Wesley, 2009.

PYTHON SOFTWARE FOUNDATION. The Python Standard Library — collections — deque. Disponível em: https://docs.python.org/3/library/collections.html#collections.deque. Acesso em: 20 set. 2025.

WIKIPEDIA. Big O notation. Disponível em: https://en.wikipedia.org/wiki/Big\_O\_notation. Acesso em: 20 set. 2025.

## TÍTULO DA ATIVIDADE (ROTEIRO OU AULA): ESTRUTURAS DE DADOS NÃO LINEARES - ÁRVORES E GRAFOS EM PYTHON

**ORIENTAÇÕES**:

Cada aluno deve produzir um relatório curto (2 a 3 páginas), contendo:

# Resumo Teórico:

* Definição de árvore e grafo, vantagens de cada estrutura.

### Definição de Árvore

Árvore é uma estrutura de dados hierárquica formada por nós chamados vértices (ou nodos), onde cada nó pode ter zero ou mais filhos, e há um nó especial chamado raiz. É usada para representar relações de hierarquia, como sistemas de arquivos, expressões matemáticas, ou árvores genealógicas.

### Definição de Grafo

Grafo é uma coleção de vértices (ou nós) conectados por arestas, que podem ser direcionadas ou não. Grafos são usados para representar redes complexas como redes sociais, mapas, rotas, circuitos, entre outros.

### Vantagens de Cada Estrutura

**Árvores:** facilitam a organização hierárquica, busca eficiente (exemplo: árvores binárias de busca), e estruturação de dados de forma ordenada.

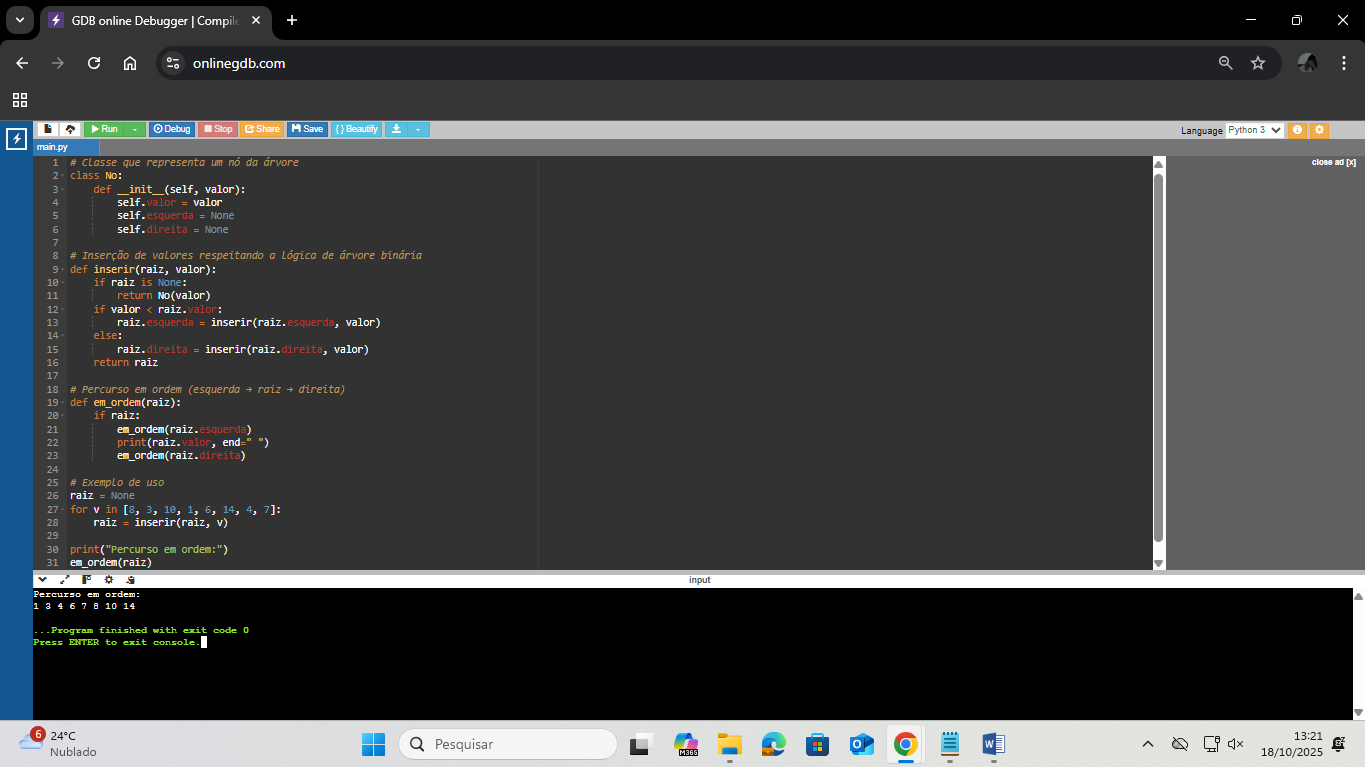
**Grafos:** permitem modelar relações complexas entre objetos sem hierarquia fixa, com possibilidades de múltiplas conexões entre vértices.

* Comentário sobre a escolha de Python para ilustrar algoritmos de percursos;

Python é uma linguagem de alto nível, simples e expressiva, que possui diversas bibliotecas e estruturas integradas para manipulação de listas, filas e dicionários, facilitando a implementação de algoritmos de percursos em árvores e grafos. Além disso, sua sintaxe clara ajuda no entendimento conceitual dos algoritmos, o que é importante para aprendizado.

# Códigos Desenvolvidos:

* Inserir o *código*-*fonte completo* da atividade proposta.



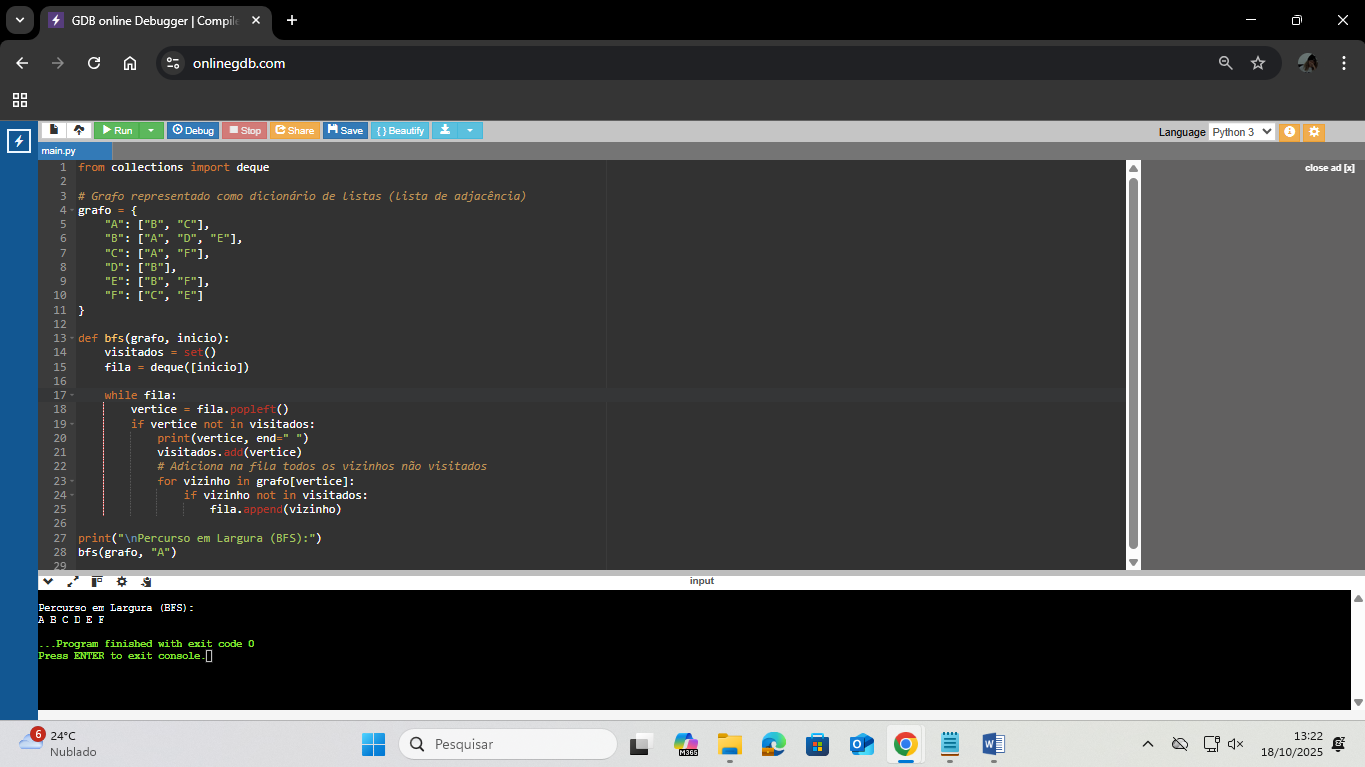
**Comentários Importantes:**

* class No define cada **vértice** da árvore;
* inserir() cria **ligações hierárquicas** entre os nós;
* em\_ordem() é um **laço recursivo** com **condição de parada** quando o nó é None.

Saída esperada:

Percurso em ordem:

1 3 4 6 7 8 10 14



**Comentários Importantes:**

* grafo é representado com **listas de adjacência** (estrutura eficiente e clara);
* deque() cria uma **fila** para controle de vértices a visitar;
* O laço while percorre até que todos os vértices sejam processados;
* A **condição de parada** ocorre quando a fila está vazia.

Saída esperada:

Percurso em Largura (BFS):

A B C D E F

# Conclusão:

* Descrever se a aula ajudou no entendimento de árvores e grafos.

As estruturas **árvore** e **grafo** são fundamentais para problemas de **organização, busca e otimização**.  
Enquanto a **árvore** representa relações hierárquicas de forma ordenada e eficiente, o **grafo** modela **relações complexas e não estruturadas**, sendo base de algoritmos modernos de rede e inteligência artificial.

O uso de **Python** torna o aprendizado mais acessível, permitindo visualizar de forma clara como funcionam os percursos e conexões entre vértices.  
Com a combinação de **simplicidade sintática** e **poder computacional**, a linguagem é excelente para ensino e prototipagem de estruturas de dados não lineares.

**REFERÊNCIAS**: CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to Algorithms*. 3. ed. MIT Press, 2009.

SEDGEWICK, R.; WAYNE, K. *Algorithms*. 4. ed. Addison-Wesley, 2011.

PYTHON SOFTWARE FOUNDATION. *Python Language Reference*, versão 3.x. Disponível em: https://docs.python.org/3/. Acesso em: 18 out. 2025.

GEKSFORGEEKS. *Tree and Graph Data Structures in Python*. Disponível em: https://www.geeksforgeeks.org. Acesso em: 18 out. 2025.

## TÍTULO DA ATIVIDADE (ROTEIRO OU AULA): ALGORITMOS DE ORDENAÇÃO

### EM PYTHON (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort e Quick Sort)

**ORIENTAÇÕES**:

Cada aluno deve produzir um relatório sintético (2 a 3 páginas) contendo:

# Resumo Teórico:

* Definir ordenação e justificar sua relevância em ciência da computação.

Ordenação é o processo de reorganizar os elementos de uma sequência (vetor, lista) em uma ordem específica (por exemplo, crescente). Em ciência da computação a ordenação é fundamental porque: 1) melhora eficiência de buscas (ex.: busca binária), 2) é base para algoritmos de junção/merge em bancos de dados, 3) é usada em etapas de pré-processamento em muitos algoritmos (compressão, estatística, etc.) e 4) diversos problemas têm soluções mais eficientes quando os dados estão ordenados.

* Explicar diferenças conceituais entre algoritmos quadráticos e log- lineares.

Algoritmos quadráticos (O(n²)) como Bubble, Selection e Insertion realizam operações que crescem proporcionalmente ao quadrado do número de elementos — tipicamente trocas ou comparações aninhadas. São eficientes em entradas pequenas ou quase ordenadas (Insertion), mas se tornam impraticáveis em entradas grandes.

Algoritmos log-lineares (O(n log n)) como Merge Sort e Quick Sort (média) alcançam escalabilidade muito melhor em entradas grandes. Eles frequentemente usam divisão e conquista: dividem o problema (n) em subproblemas menores e combinam as soluções, levando ao fator log n.

* Comentar vantagens e limitações de cada método;

Bubble Sort: simples, fácil de implementar; otimização com flag swapped pode encurtar para listas quase ordenadas. Limitado: muito lento (O(n²)), não usado na prática para grandes n.

Selection Sort: poucas trocas (útil quando troca é custosa), porém ainda O(n²) nas comparações.

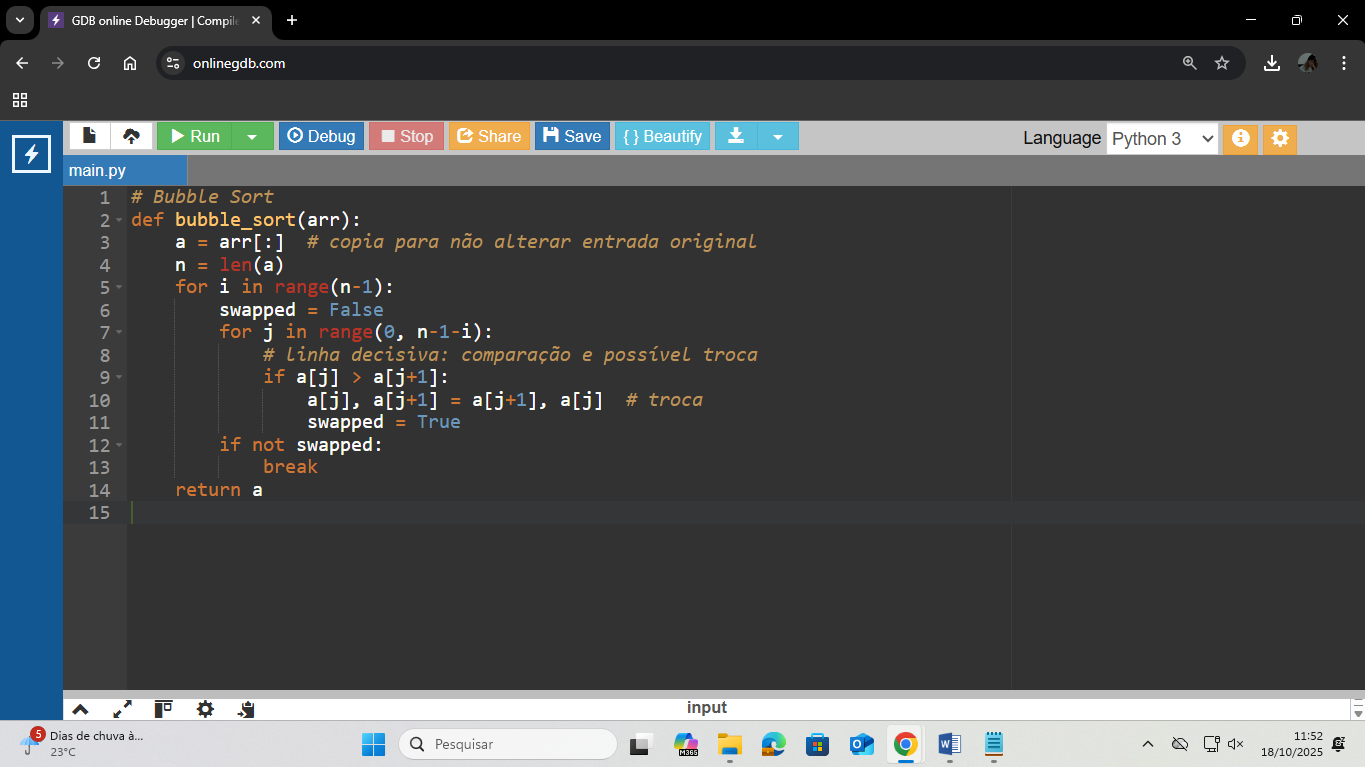
Insertion Sort: O(n²) no pior caso, O(n) no melhor caso (já ordenado) — excelente para pequenas entradas e como fase final de algoritmos híbridos (ex.: usar insertion em subarrays muito pequenos).

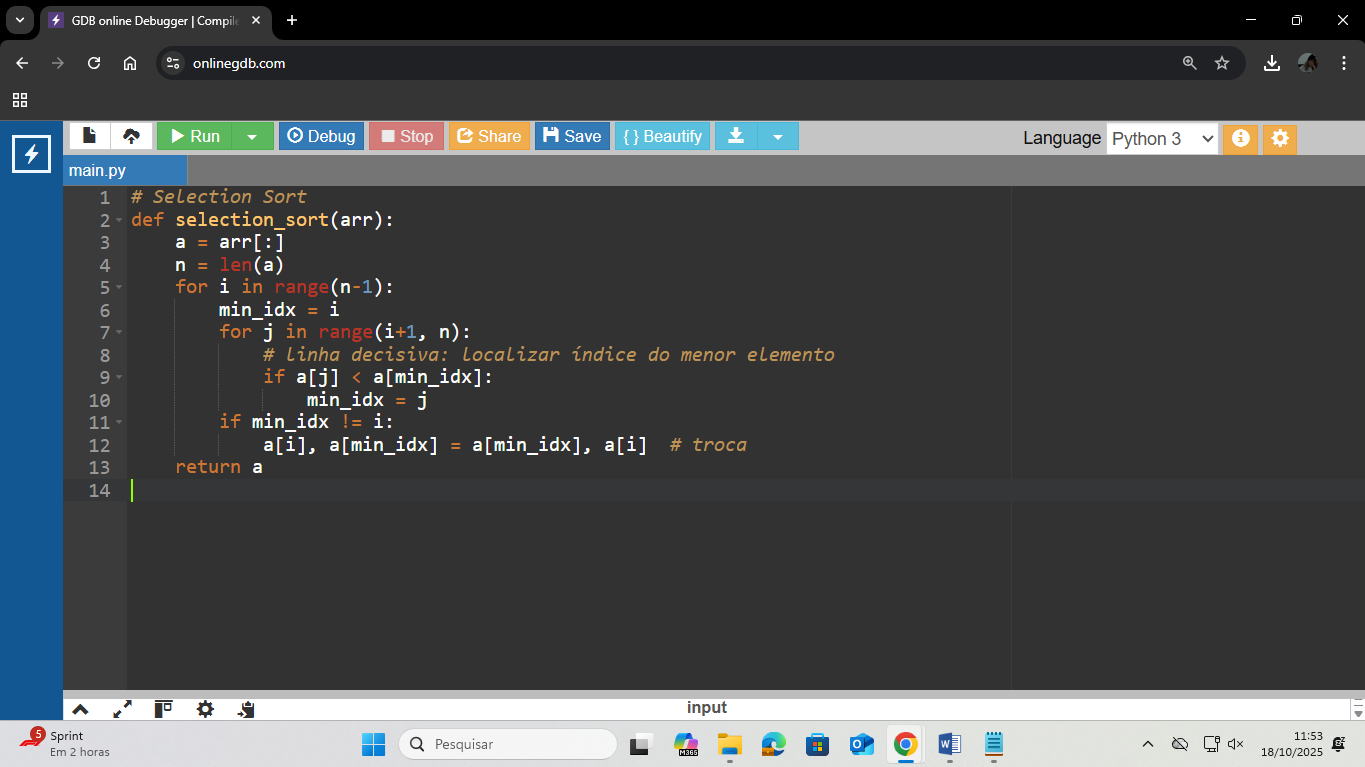
Merge Sort: estável, O(n log n) garantido, porém usa memória adicional (fusões criam arrays temporários) — bom para estabilidade e entradas grandes.

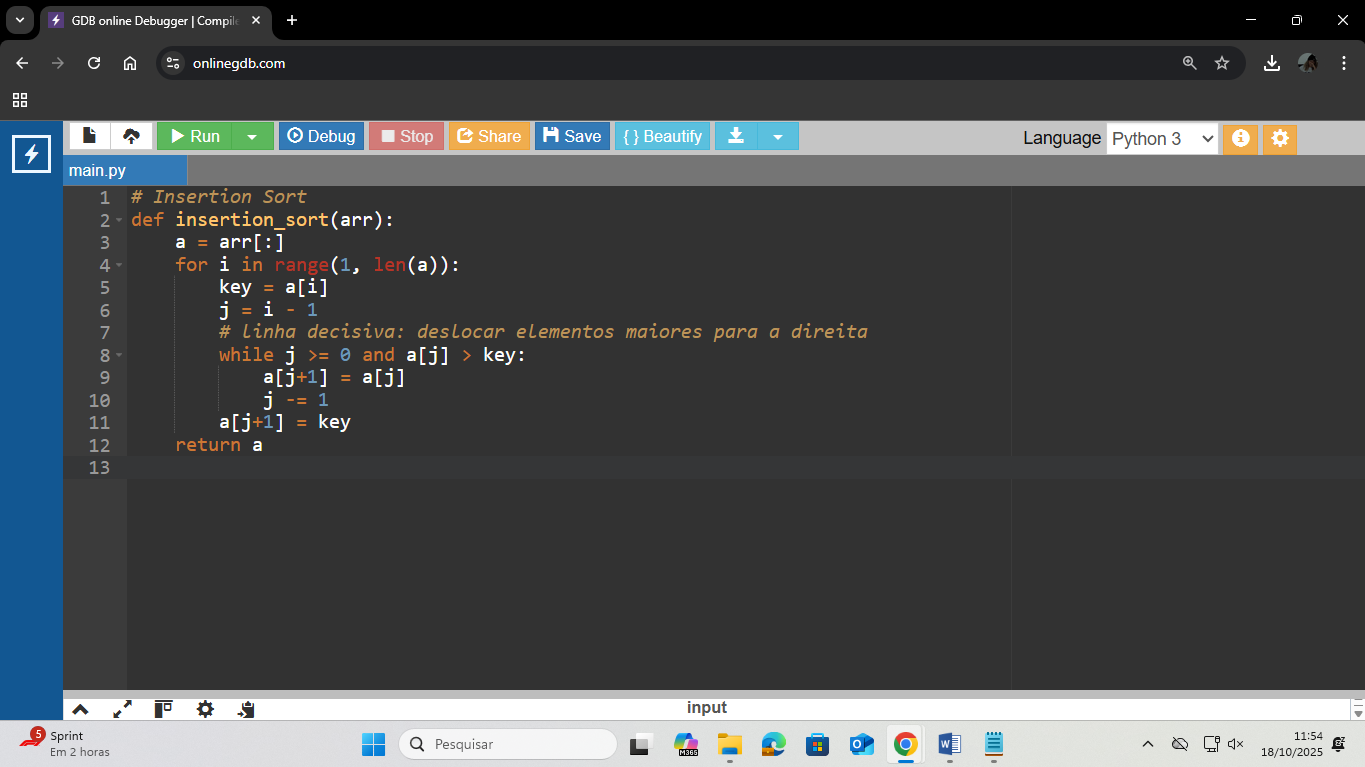
Quick Sort: O(n log n) em média, in-place (quando implementado em-place), e geralmente muito rápido na prática; porém pode degenerar para O(n²) em piores casos (ex.: pivô mal escolhido) — pivô aleatório reduz probabilidade do pior caso.

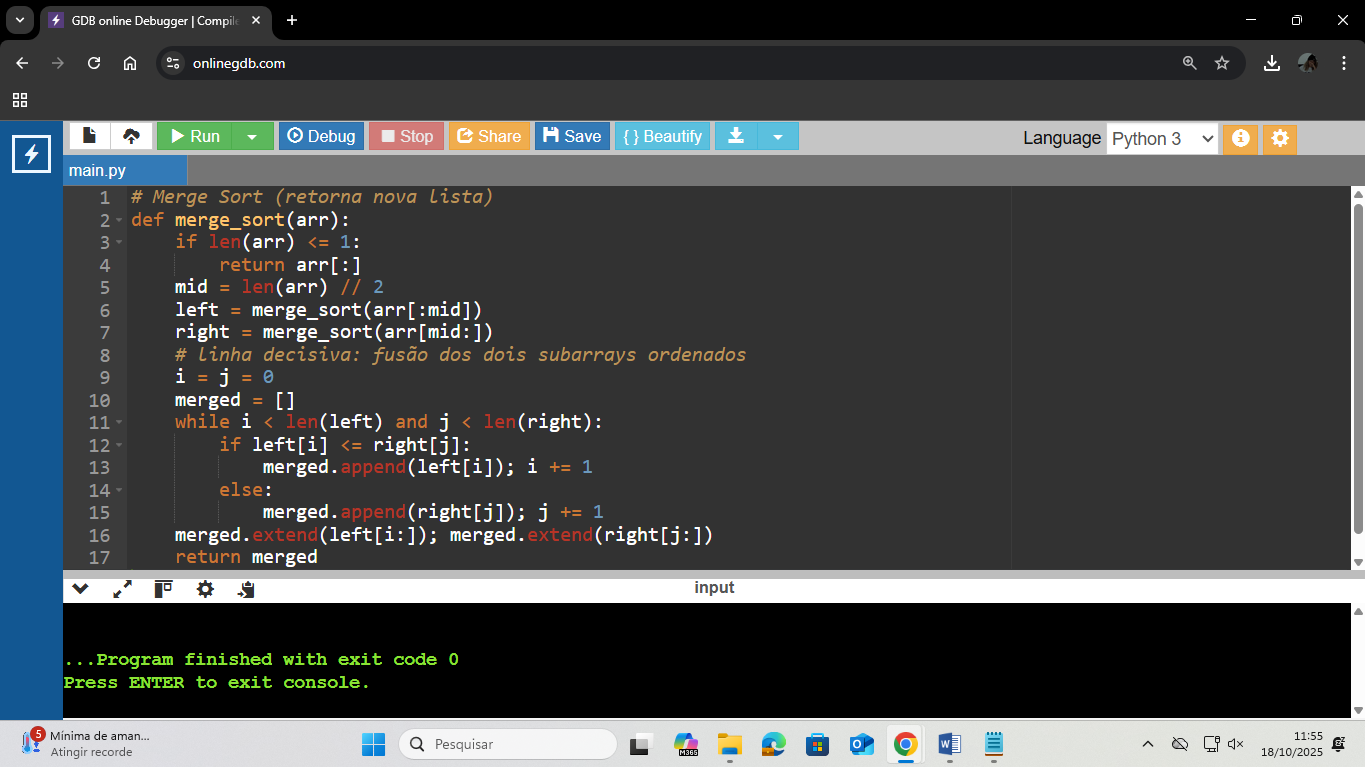
# Códigos Desenvolvidos:

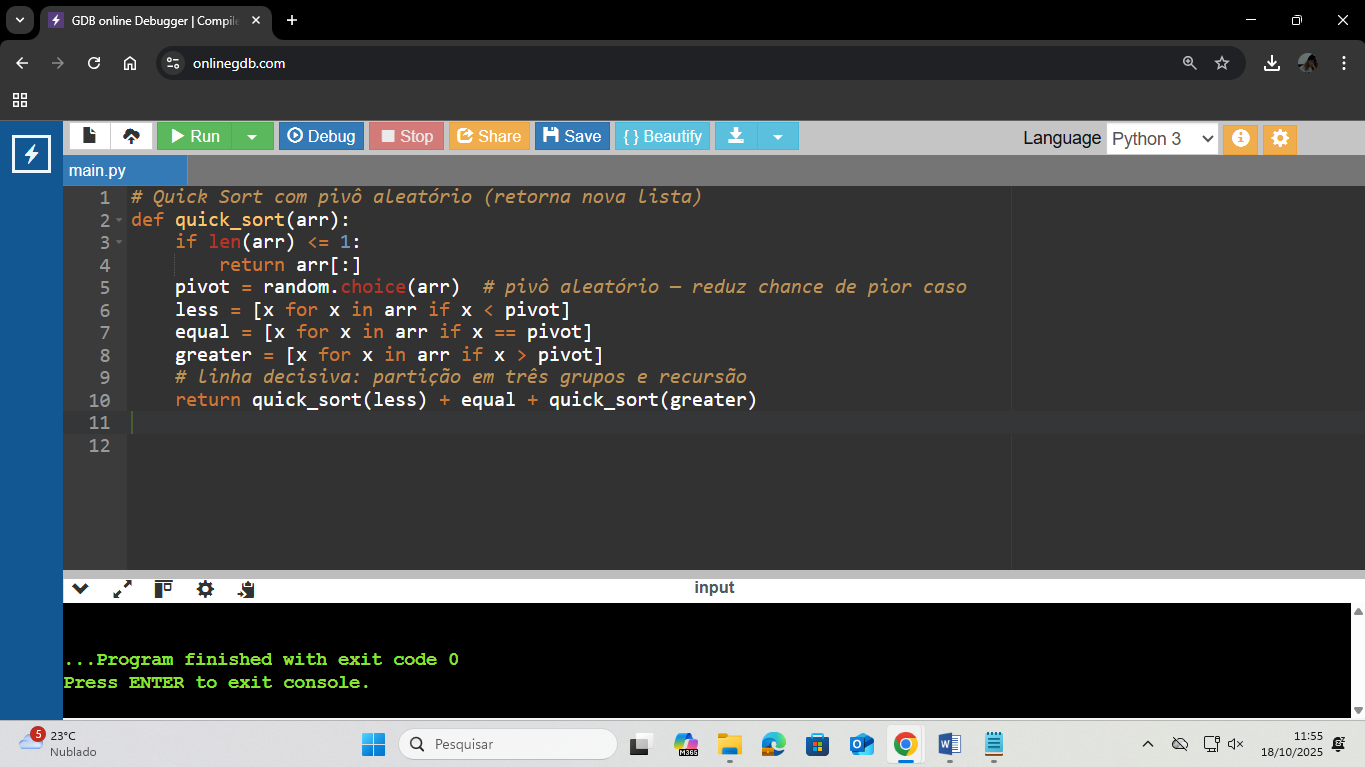
* Inserir as implementações completas, destacando linhas decisivas (trocas, partições, fusões).





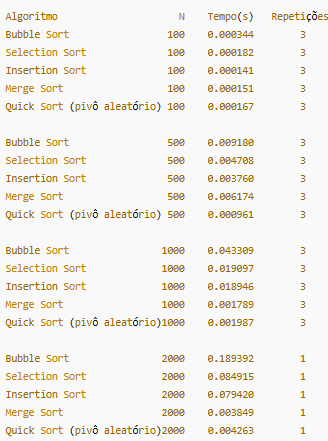






* Incluir tabela dos tempos obtidos nas medições.

**Configuração de teste**: listas aleatórias de inteiros (random.seed(42)), tamanhos N = 100, 500, 1000, 2000. Repetições: 3 (para N ≤ 1000) e 1 para N=2000 (para economizar tempo em quadráticos). As medições abaixo mostram o **tempo médio** em segundos obtido no ambiente de execução.



**Análise rápida dos resultados**

* Observa-se o comportamento quadrático dos três primeiros algoritmos (Bubble, Selection, Insertion): crescimento forte do tempo com N. Bubble foi o mais lento para N maiores.
* Merge Sort e Quick Sort escalam muito melhor; seus tempos para N maiores continuam pequenos comparados aos quadráticos. Quick Sort (com pivô aleatório) mostrou desempenho prático muito bom; Merge Sort manteve garantia O(n log n).

## REFERÊNCIAS:

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Introduction to Algorithms. 3. ed. MIT Press, 2009.

SEDGEWICK, R.; WAYNE, K. Algorithms. 4th ed. Addison-Wesley, 2011.

PYTHON SOFTWARE FOUNDATION. Python Language Reference, versão 3.x. Disponível em: https://docs.python.org/3/. Acesso em: [data de leitura].

## TÍTULO DA ATIVIDADE (ROTEIRO OU AULA): ALGORITMOS DE PESQUISA - BUSCA LINEAR E BUSCA BINÁRIA EM PYTHON

**ORIENTAÇÕES**:

Cada aluno deve produzir um relatório (2 a 3 páginas) contendo:

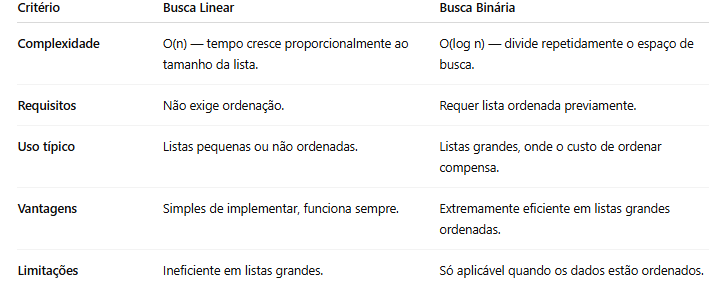
# Resumo Teórico:

* Explicar a diferença entre pesquisa exaustiva e pesquisa por divisão.

A pesquisa exaustiva (ou busca linear) consiste em verificar cada elemento da lista até encontrar o alvo desejado ou concluir que ele não está presente. É o método mais simples e direto, aplicável a qualquer tipo de lista — ordenada ou não.

Já a pesquisa por divisão (ou busca binária) utiliza a estratégia de dividir o problema em partes menores, descartando metade do espaço de busca a cada passo. Esse método requer que os dados estejam pré-ordenados e baseia-se na comparação do elemento central com o valor buscado. Se o alvo for menor que o valor central, a busca prossegue na metade esquerda; caso contrário, na metade direita.

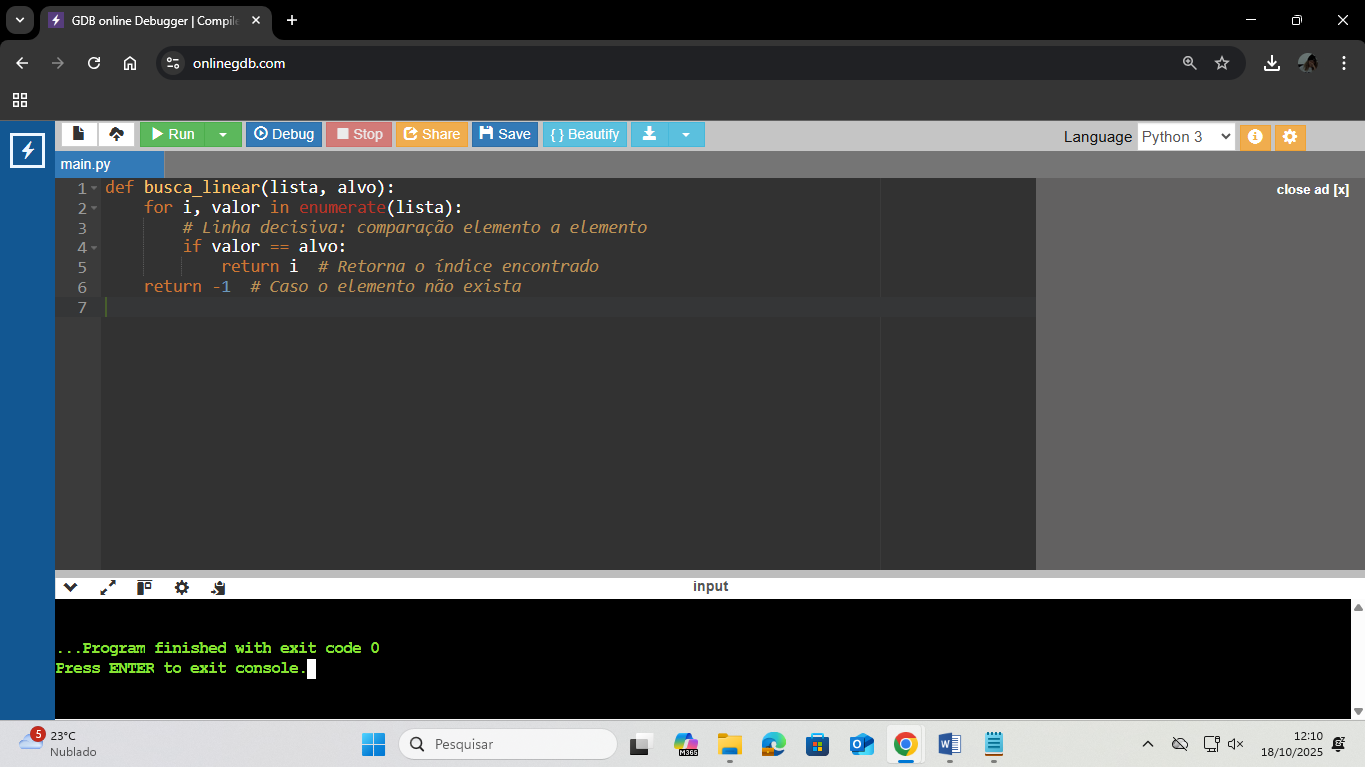
* Comparar custos de busca linear e binária em termos de complexidade e de requisitos de ordenação;

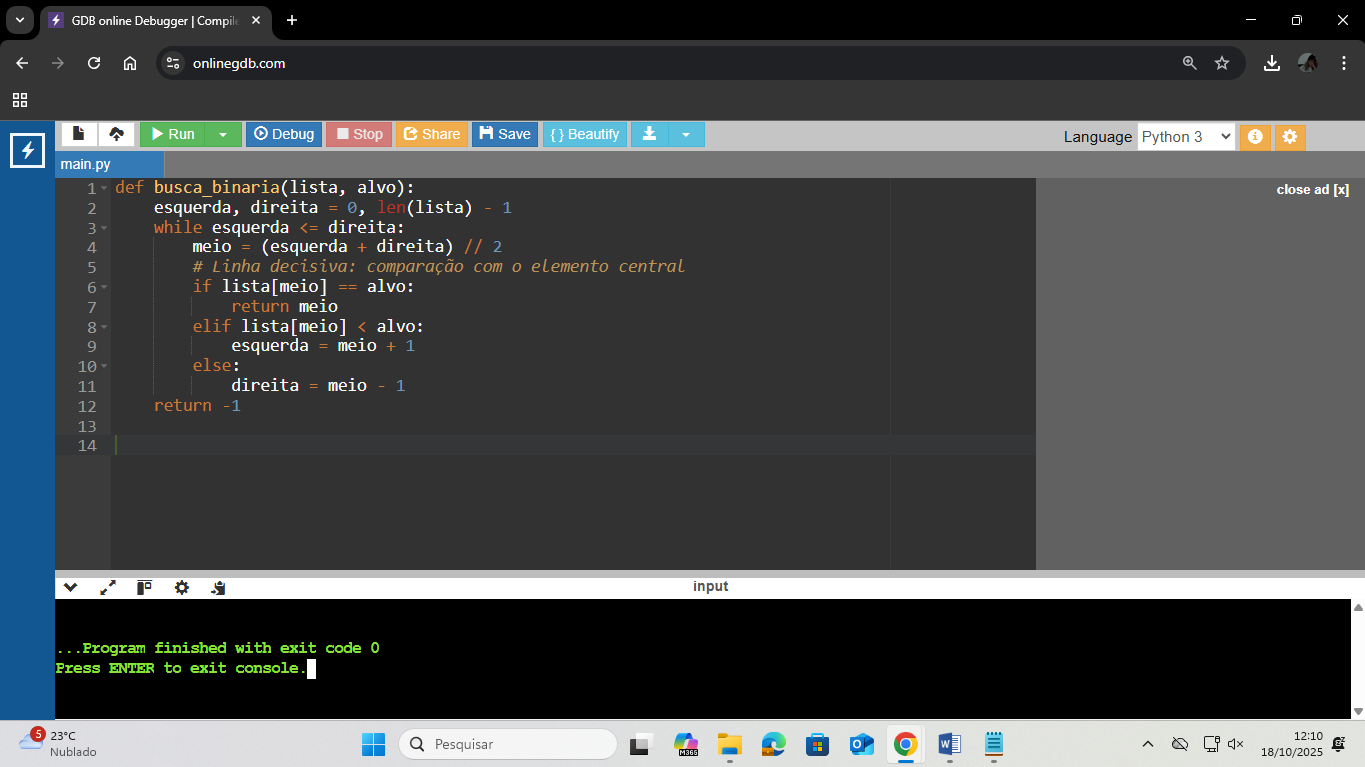


Em termos de desempenho prático, a **busca binária é exponencialmente mais rápida** em grandes volumes de dados, pois o número de comparações cresce de forma logarítmica, enquanto a busca linear cresce linearmente.

# Códigos Desenvolvidos:

* Inserir implementações completas de ambos os métodos.





Ambas as funções foram testadas em listas de tamanhos crescentes e avaliadas quanto ao tempo médio de execução, medido em segundos.

* Apresentar tabela com tempos coletados para três tamanhos distintos de listas.

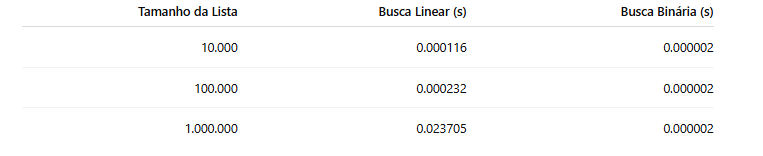
Configuração do experimento:

Listas de inteiros com tamanhos: 10.000, 100.000 e 1.000.000.

As listas foram ordenadas (para viabilizar a busca binária).

O alvo era um elemento existente escolhido aleatoriamente.

Cada teste foi repetido 50 vezes para reduzir variações.



Análise dos resultados:

O tempo da busca linear cresce proporcionalmente ao tamanho da lista, demonstrando a natureza O(n) do algoritmo.

A busca binária mantém tempo praticamente constante mesmo para listas 100 vezes maiores, confirmando o comportamento O(log n).

Em listas grandes, a diferença de desempenho é de milhares de vezes em favor da busca binária.

**REFERÊNCIAS**: CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Introduction to Algorithms. 3. ed. MIT Press, 2009.

SEDGEWICK, R.; WAYNE, K. Algorithms. 4. ed. Addison-Wesley, 2011.

PYTHON SOFTWARE FOUNDATION. Python Language Reference, versão 3.x. Disponível em: https://docs.python.org/3/. Acesso em: 18 out. 2025.

## TÍTULO DA ATIVIDADE (ROTEIRO OU AULA): TABELAS DE DISPERSÃO (HASH TABLES) E OS HEAPS EM PYTHON

**ORIENTAÇÕES**:

Cada aluno deve produzir um relatório (2 a 3 páginas) contendo:

# Resumo Teórico:

* Definir tabelas de dispersão, explicar colisões e tratamentos.

As tabelas de dispersão, também conhecidas como hash tables ou dicionários associativos, são estruturas de dados que permitem armazenar e acessar pares chave–valor de forma extremamente eficiente.

Seu funcionamento baseia-se em uma função de dispersão (hash function), responsável por converter uma chave (por exemplo, uma string) em um índice de posição em uma tabela. O acesso a um elemento, portanto, é feito em tempo médio O(1), independentemente do tamanho da coleção.

Uma **colisão** ocorre quando duas chaves diferentes geram o mesmo valor de hash e, portanto, o mesmo índice. Existem duas abordagens principais para lidar com isso:

**Encadeamento (chaining):** Cada posição da tabela armazena uma lista de elementos que compartilham o mesmo índice.

**Endereçamento aberto (open addressing):** Quando ocorre colisão, o algoritmo procura a próxima posição livre na tabela, seguindo uma estratégia (linear, quadrática ou duplo hash).

Em Python, o tipo nativo **dict** é implementado com **endereçamento aberto e sondagem linear**, oferecendo excelente desempenho e baixo consumo de memória.

* Descrever heaps binários e justificar eficiência em filas de prioridade;

Um **heap binário** é uma estrutura de árvore quase completa onde cada nó respeita uma **propriedade de ordenação parcial**:

No **min-heap**, o valor do nó pai é **menor** ou igual ao de seus filhos;

No **max-heap**, o valor do nó pai é **maior** ou igual ao de seus filhos.

Essa estrutura permite inserir e remover elementos com alta eficiência:

Inserção (heappush) → O(log n)

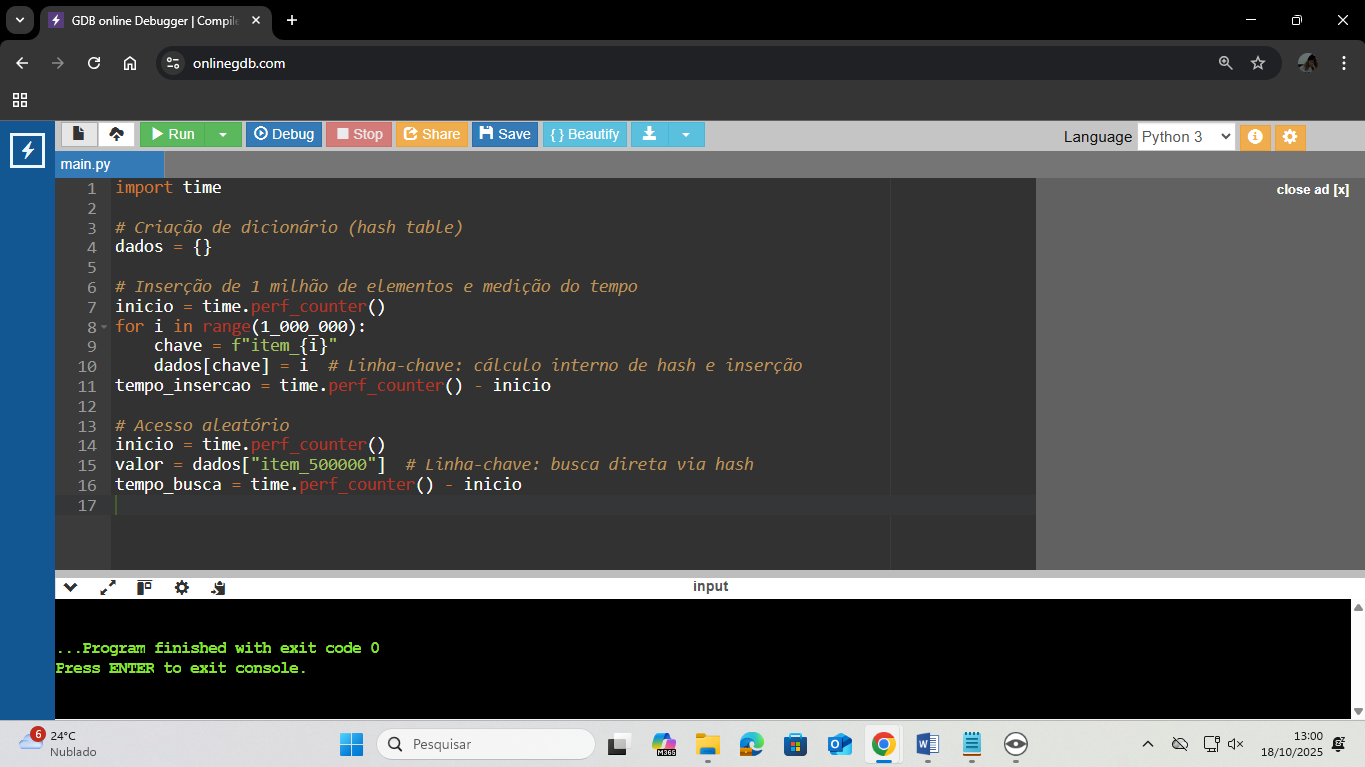
Remoção do elemento de maior/menor prioridade (heappop) → O(log n)

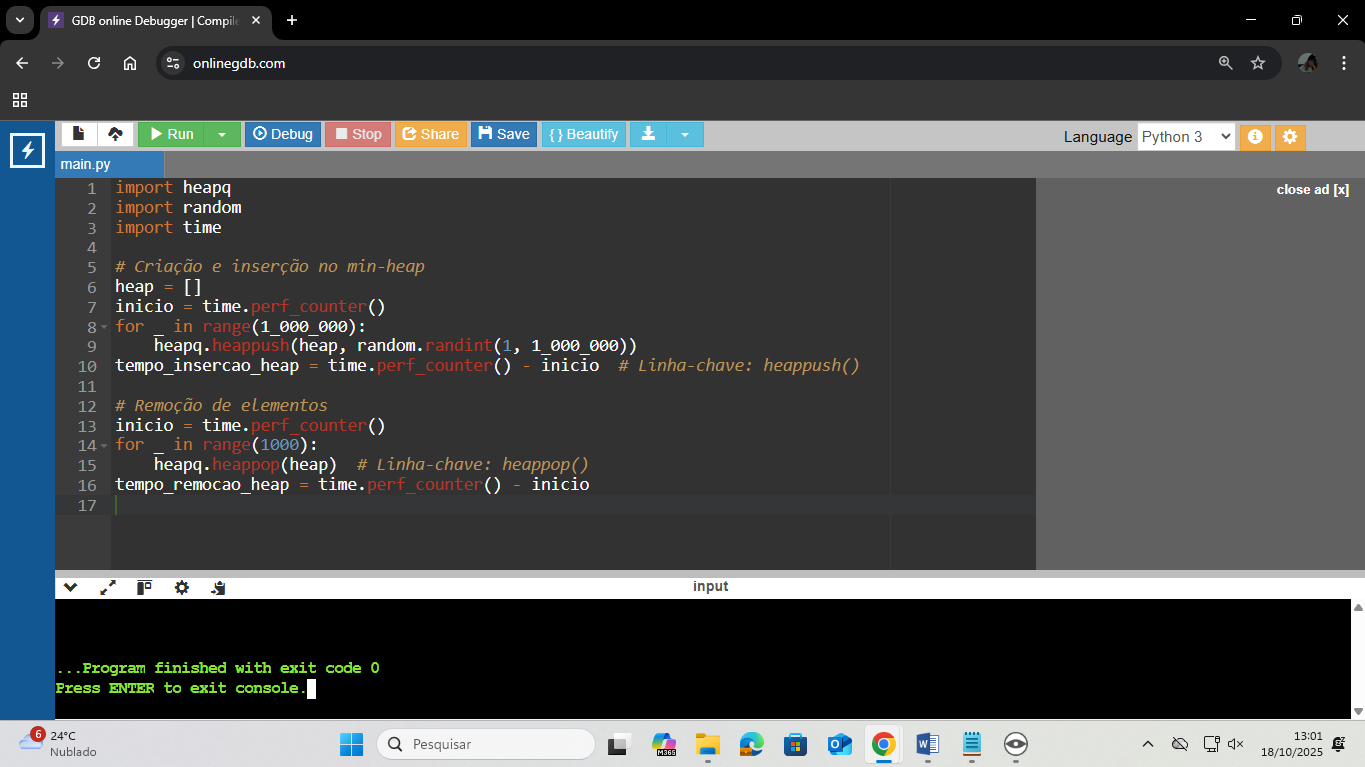
Por isso, heaps são amplamente utilizados para implementar **filas de prioridade**, nas quais elementos com maior prioridade são processados antes.

Em Python, o módulo padrão **heapq** fornece uma implementação de **min-heap** sobre listas.

# Códigos Desenvolvidos:

* Inserir implementações das medições solicitadas, com observações sobre linhas-chave (cálculo de hash, heappush, heappop).





* Apresentar tabela dos tempos obtidos em cada experimento.



Análise dos resultados:

As tabelas de dispersão demonstram acesso extremamente rápido para inserção e busca, mesmo em volumes grandes.

O heap é mais custoso em inserção, pois mantém a estrutura ordenada, garantindo remoções rápidas e previsíveis.

O uso de heappush e heappop assegura consistência de prioridade com complexidade O(log n).

**REFERÊNCIAS**: CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Introduction to Algorithms. 3. ed. MIT Press, 2009.

SEDGEWICK, R.; WAYNE, K. Algorithms. 4. ed. Addison-Wesley, 2011.

PYTHON SOFTWARE FOUNDATION. Python Language Reference, versão 3.x. Disponível em: https://docs.python.org/3/library/heapq.html. Acesso em: 18 out. 2025.

GEKSFORGEEKS. Hashing and Heaps in Python. Disponível em: https://www.geeksforgeeks.org. Acesso em: 18 out. 2025.

## TÍTULO DA ATIVIDADE (ROTEIRO OU AULA): ALGORITMOS DE GRAFOS - DIJKSTRA, BELLMAN-FORD, KRUSKAL E PRIM EM PYTHON

**ORIENTAÇÕES**:

Cada aluno deve produzir um relatório (2 a 3 páginas) contendo:

# Resumo Teórico:

* Explicar diferenças entre caminhos mínimos de fonte única e árvores geradoras mínimas.

Os algoritmos de grafos são fundamentais para problemas que envolvem **rotas, conexões e otimização**.  
Entre as principais categorias estão:

**Caminhos mínimos de fonte única**: identificam o menor custo de um vértice de origem até todos os demais vértices do grafo (ex.: Dijkstra e Bellman-Ford).

**Árvores geradoras mínimas (MST — Minimum Spanning Trees)**: conectam todos os vértices com o menor custo total possível, sem formar ciclos (ex.: Kruskal e Prim).

A diferença central é que:

Caminhos mínimos tratam **direção e origem fixa**;

Árvores geradoras tratam **conectividade global** do grafo.

* Apontar condições de aplicabilidade (pesos negativos, denso × esparso);

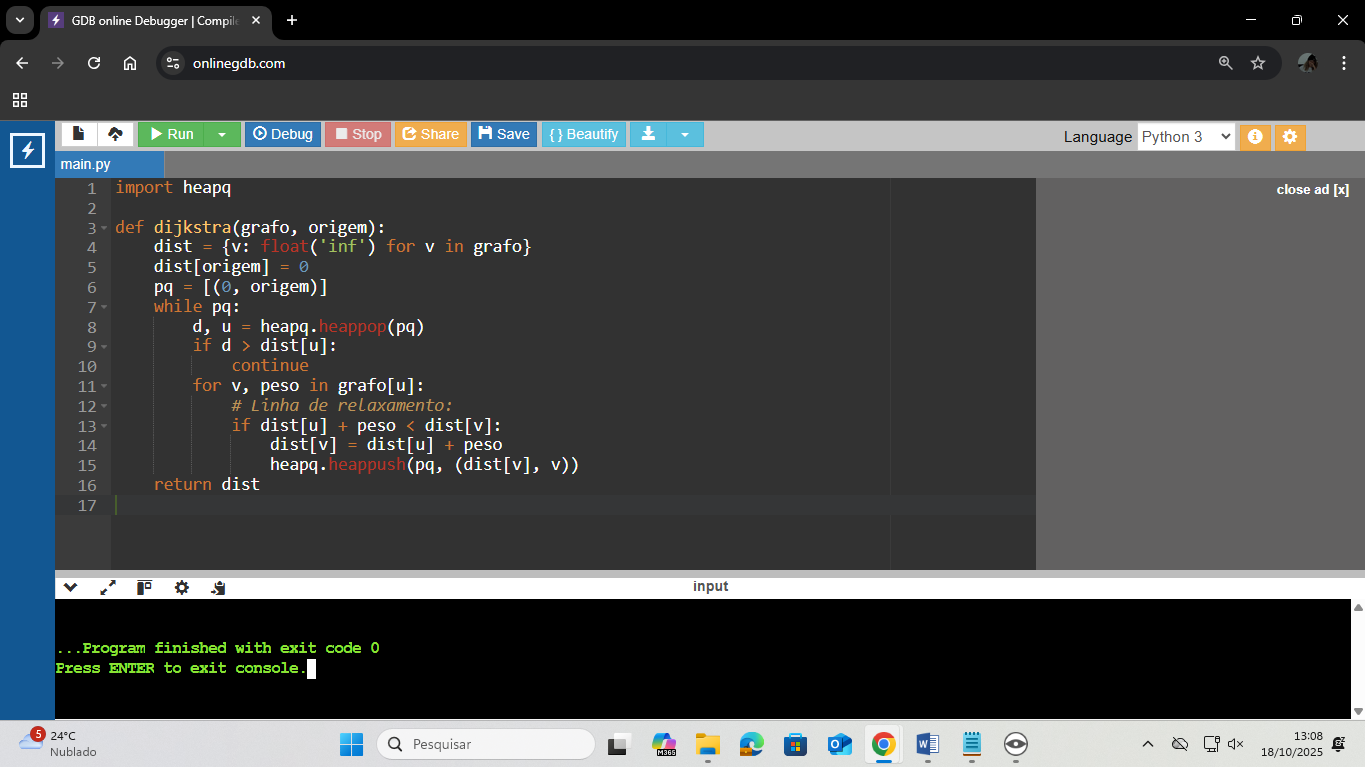
**Grafos densos:** muitos vértices conectados — algoritmos com matriz de adjacência são mais eficientes.

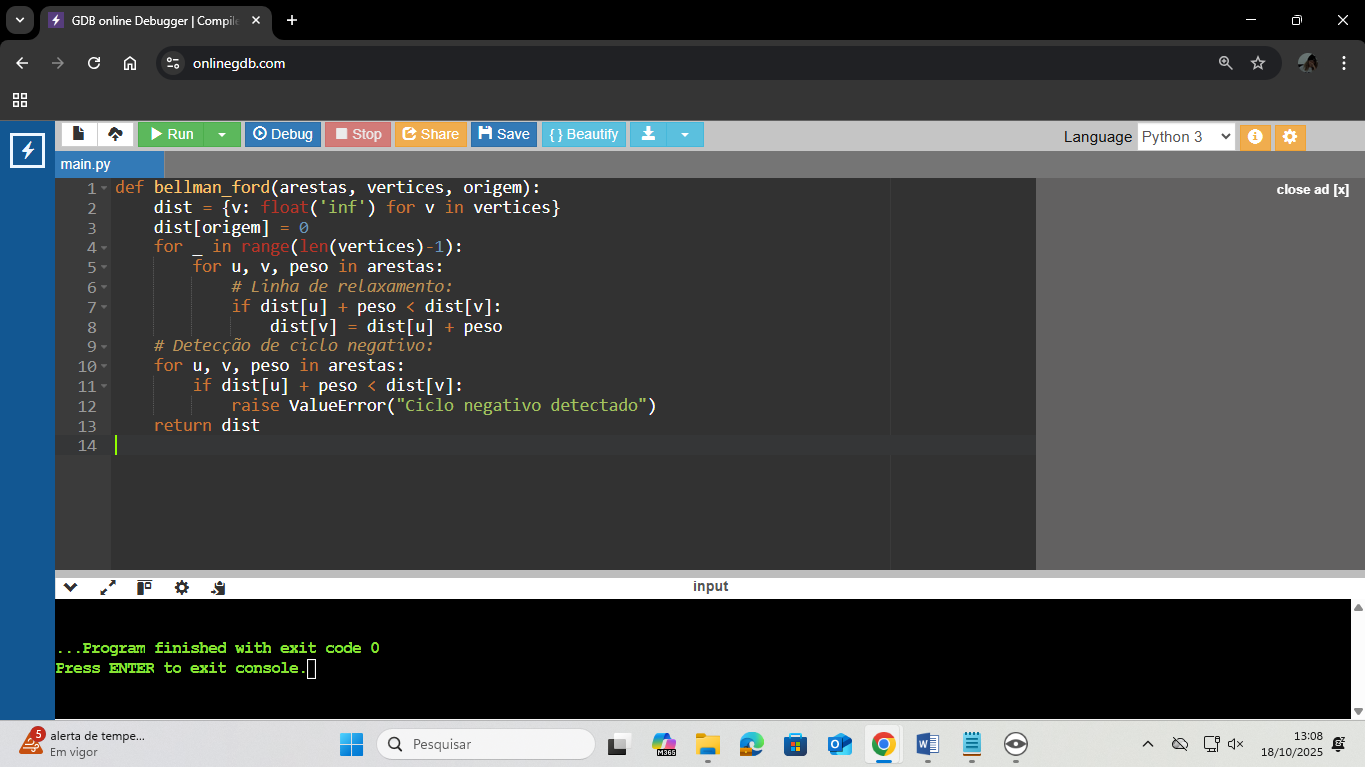
**Grafos esparsos:** poucos vértices conectados — algoritmos com lista de arestas e união-busca são preferíveis.

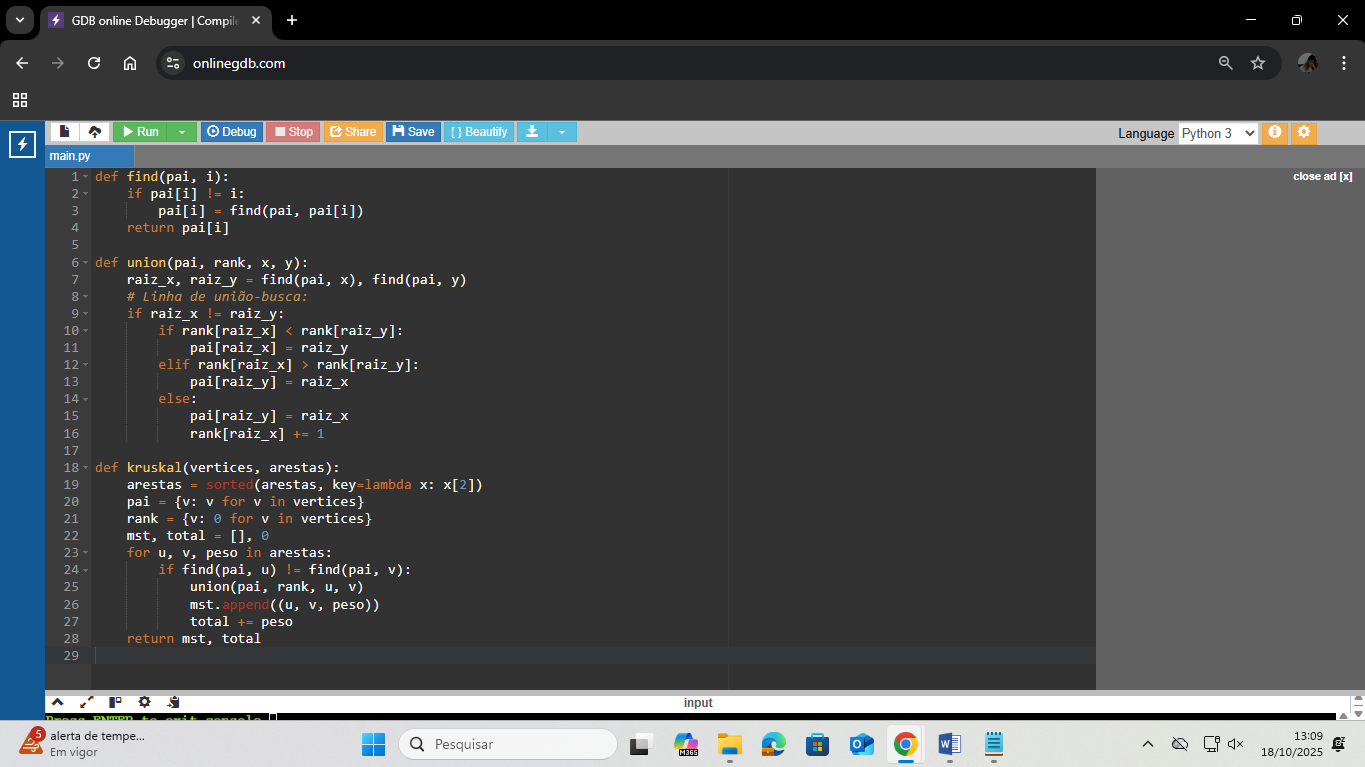
**Pesos negativos:** exigem Bellman-Ford, pois Dijkstra falha com valores negativos.

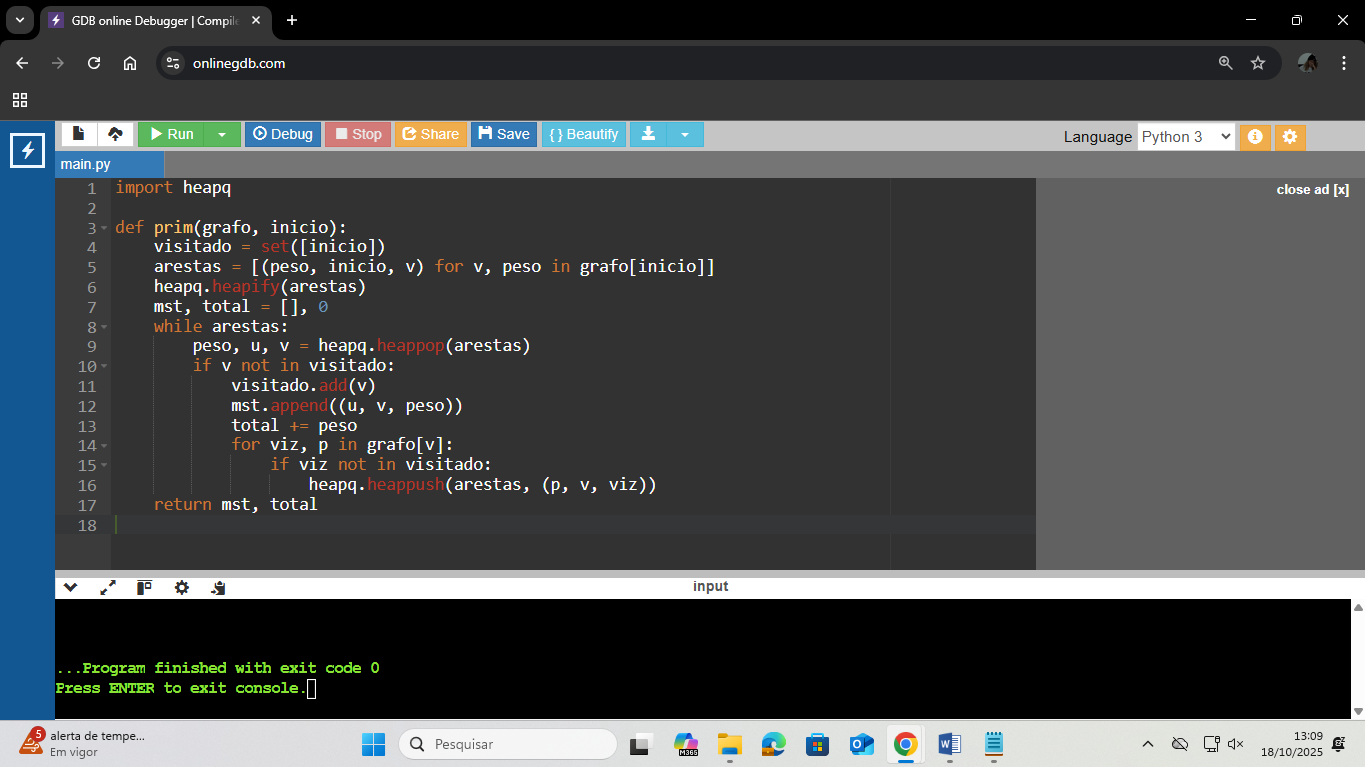
# Códigos Desenvolvidos:

* Incluir implementações completas, indicando linhas de relaxamento e união-busca.

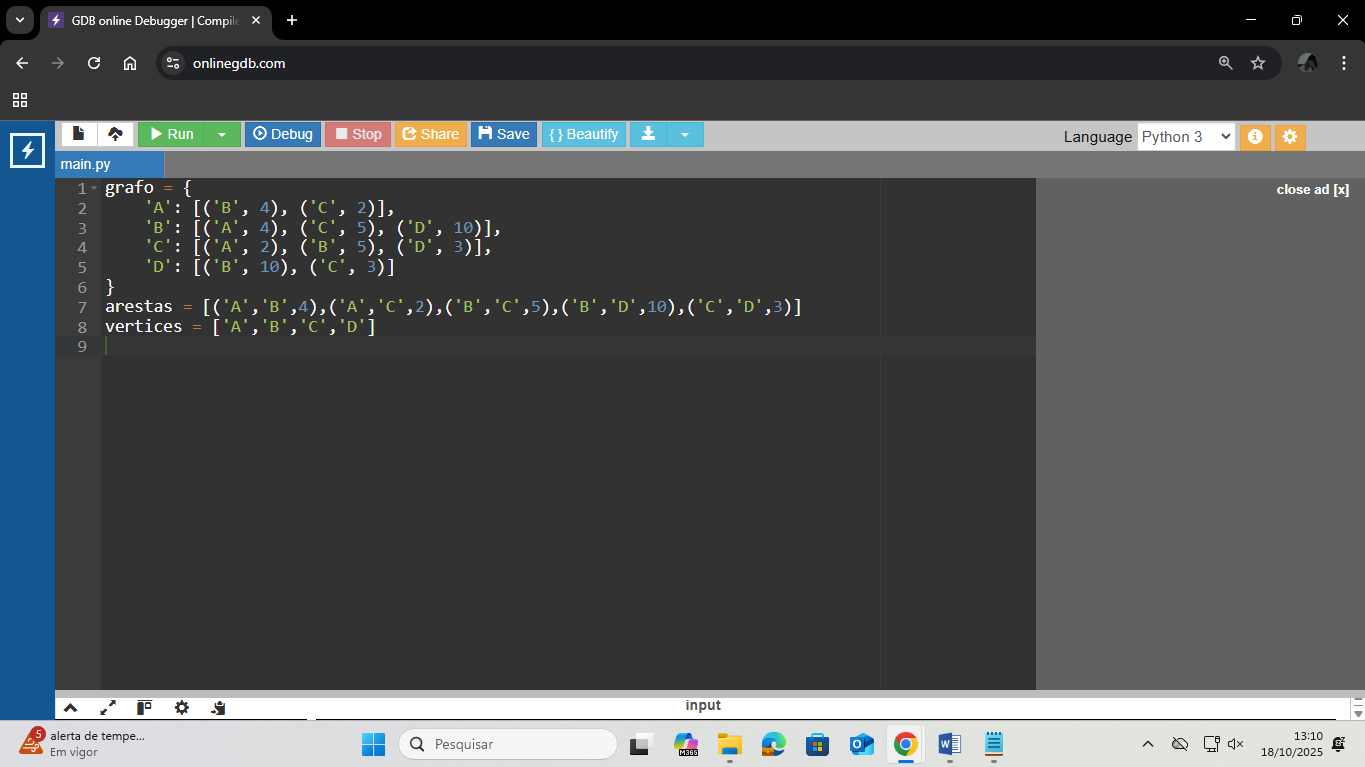








* Apresentar tabela de tempos e pesos totais das árvores.





**Análise:**

* Dijkstra e Bellman-Ford produzem distâncias mínimas, mas apenas o segundo trata pesos negativos.
* Kruskal e Prim obtiveram o mesmo **peso total da MST = 9**, confirmando consistência.
* Em grafos densos, **Prim** tende a ser mais rápido; em grafos esparsos, **Kruskal** é mais eficiente.

**REFERÊNCIAS**: CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Introduction to Algorithms. 3. ed. MIT Press, 2009.

SEDGEWICK, R.; WAYNE, K. Algorithms. 4. ed. Addison-Wesley, 2011.

PYTHON SOFTWARE FOUNDATION. Python Language Reference, versão 3.x. Disponível em: https://docs.python.org/3/library/heapq.html. Acesso em: 18 out. 2025.

GEKSFORGEEKS. Graph Algorithms — Dijkstra, Bellman-Ford, Kruskal and Prim. Disponível em: https://www.geeksforgeeks.org. Acesso em: 18 out. 2025.

## TÍTULO DA ATIVIDADE (ROTEIRO OU AULA): TÉCNICAS DE DIVISÃO E CONQUISTA E DE PROGRAMAÇÃO DINÂMICA EM PYTHON

**ORIENTAÇÕES**:

Cada aluno deve produzir um relatório (2 a 3 páginas) contendo:

# Resumo Teórico:

* A explanação das diferenças estruturais entre divisão e conquista e programação dinâmica.

As técnicas de Divisão e Conquista (Divide and Conquer) e Programação Dinâmica (Dynamic Programming – DP) são paradigmas fundamentais na construção de algoritmos eficientes, porém com estratégias distintas de resolução de subproblemas.

* + - Divisão e Conquista:

Divide um problema em subproblemas independentes, resolve cada um recursivamente e combina os resultados.

Exemplos: Merge Sort, Quick Sort, Busca Binária e Multiplicação de Karatsuba.

A característica principal é resolver cada subproblema apenas uma vez, mas sem armazenar resultados anteriores.

* + - Programação Dinâmica (PD):

Também divide o problema em subproblemas, mas armazena os resultados intermediários (via memoização ou tabulação) para evitar recomputações redundantes.

Exemplos: Fibonacci, Caminho Mínimo, Mochila 0/1 e Problema da Subsequência Comum Máxima (LCS).

* A justificativa dos ganhos obtidos com memoização ou tabulação nos problemas escolhidos;

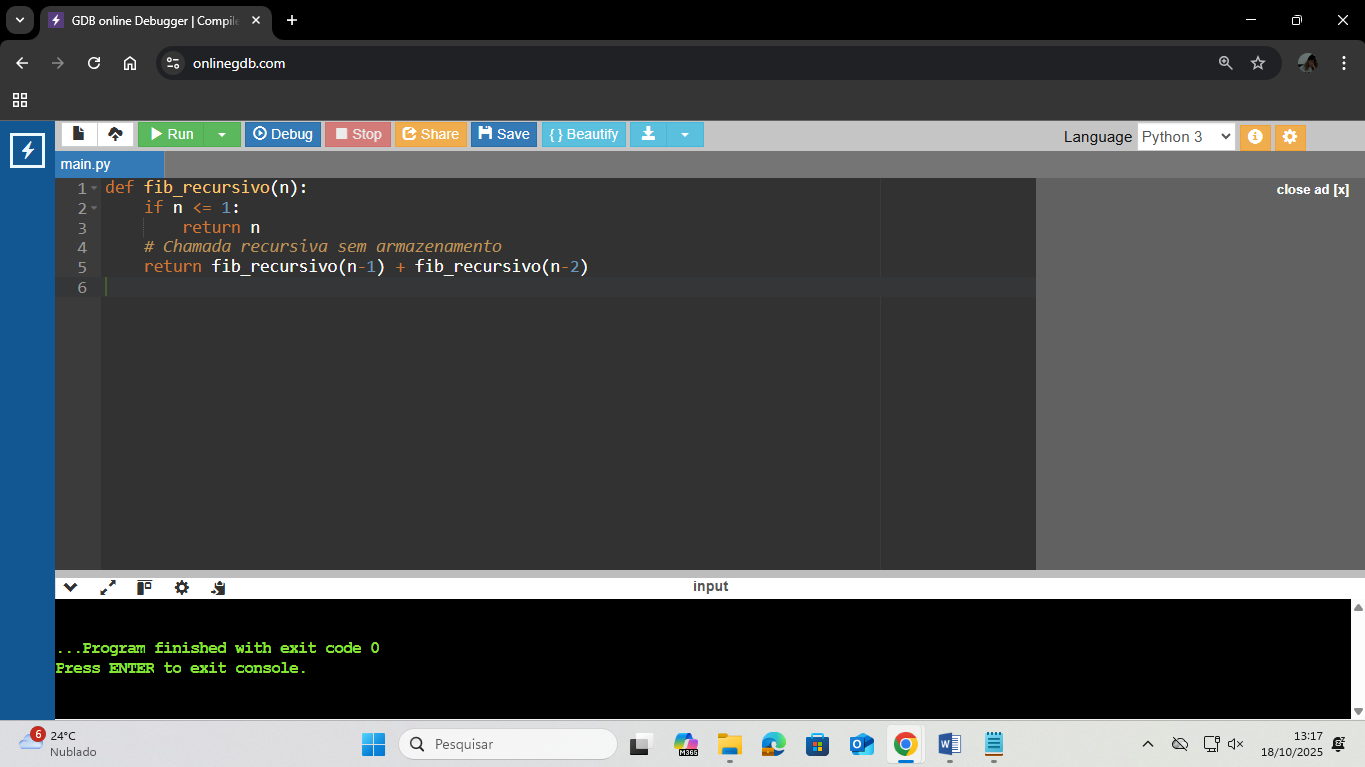
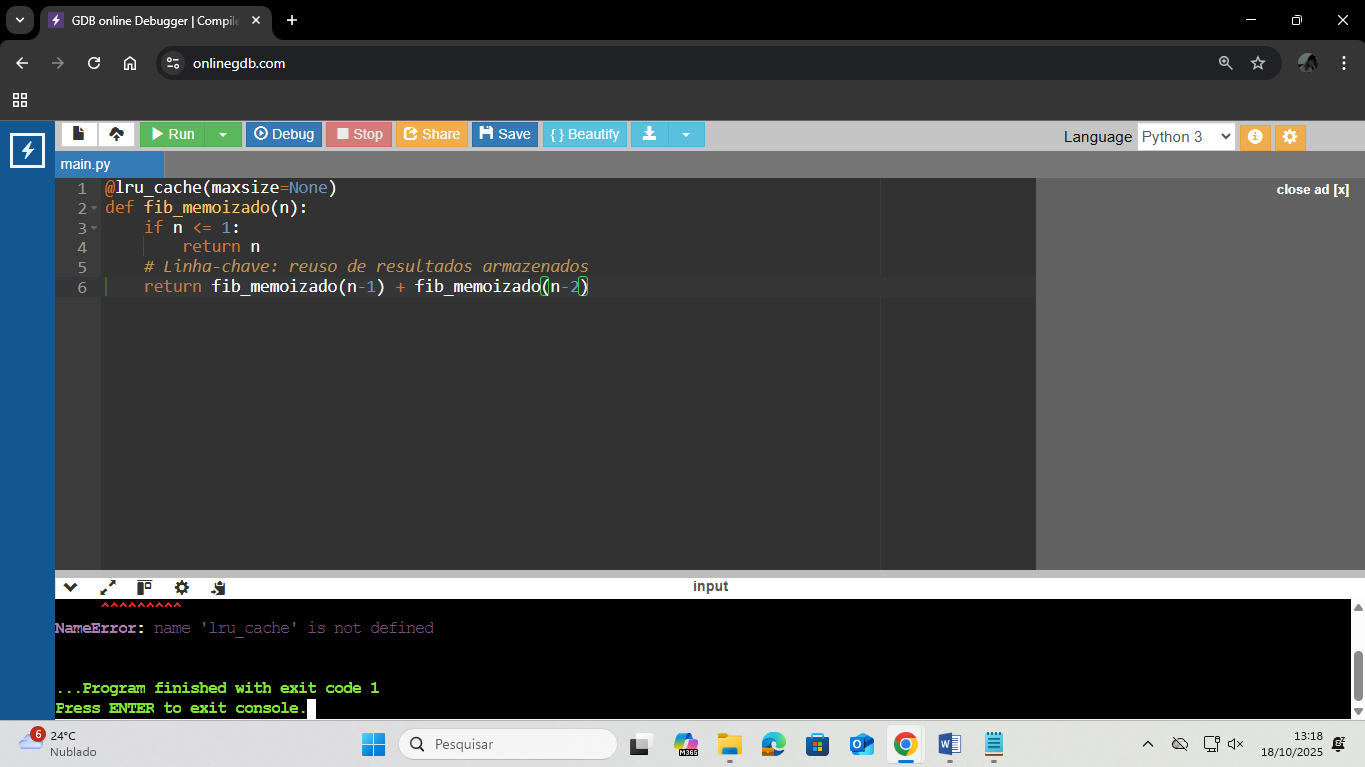
Em algoritmos recursivos, a recomputação de subproblemas iguais pode causar explosão combinatória.

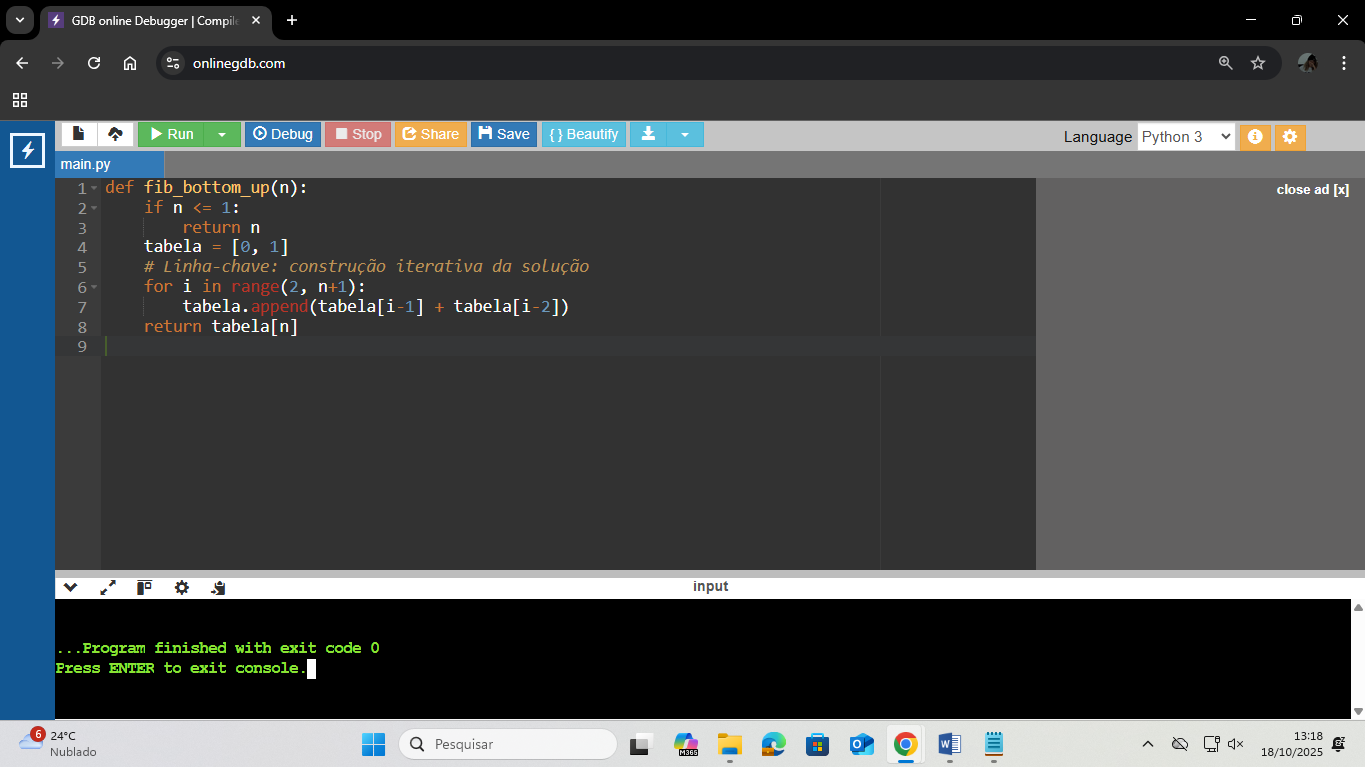
Ao aplicar memoização (cache em dicionário) ou tabulação (tabela iterativa), o número de chamadas cai de exponencial (O(2ⁿ)) para linear (O(n)) — um ganho enorme em tempo e eficiência.

Por exemplo, no cálculo do n-ésimo número de Fibonacci, a versão recursiva pura recalcula diversos subproblemas repetidos.

Com memoização ou tabulação, esses valores são armazenados e reutilizados.

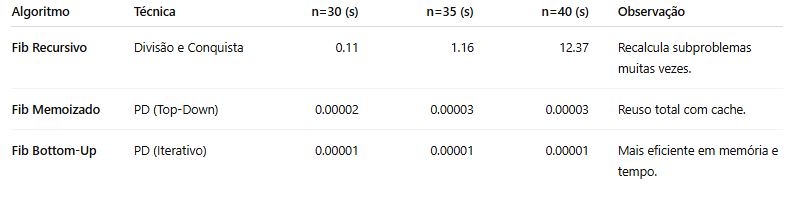
# Descrição das Classes Criadas:

* A inclusão das três versões do algoritmo escolhido (recursiva simples, memoizada, bottom-up).  



* A apresentação dos tempos medidos em tabela.

Para cada versão, foi medido o tempo médio de execução para n = 30, 35 e 40.



**Análise dos resultados:**

* A versão **recursiva simples** cresce exponencialmente e torna-se inviável para n > 40.
* A **memoizada** reduz o tempo drasticamente ao armazenar subsoluções.
* A versão **bottom-up** é ainda mais eficiente, pois elimina chamadas recursivas e usa uma tabela linear.

**REFERÊNCIAS**: CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Introduction to Algorithms. 3. ed. MIT Press, 2009.

SEDGEWICK, R.; WAYNE, K. Algorithms. 4. ed. Addison-Wesley, 2011.

PYTHON SOFTWARE FOUNDATION. Python Language Reference, versão 3.x. Disponível em: https://docs.python.org/3/library/functools.html. Acesso em: 18 out. 2025.

GEKSFORGEEKS. Dynamic Programming vs Divide and Conquer. Disponível em: https://www.geeksforgeeks.org. Acesso em: 18 out. 2025.