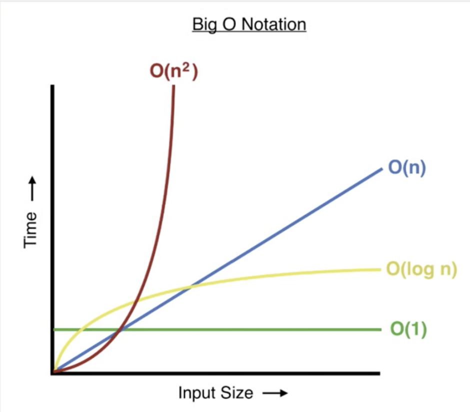
algorithm: a well-defined computational procedure designed to solve a problem. It consists of a sequence of precise steps that take some inputs, process them systematically, and produce corresponding outputs. **data structure**: is a way to organize and manage data, allowing us to write more efficient code in terms of both time and space.

0.1 Big O Notation

Definition: Describes asymptotic behavior of algorithms as input size n grows. **Common Classes**:

- $O(1)$: Constant - Runtime independent of n.
- $O(\log n)$: Logarithmic - Doubling n adds constant time.
- $O(\sqrt{n})$: sub-linear time complexity.
- $O(n)$: Linear - Doubling n doubles time.
- $O(n \log n)$: Linearithmic - Roughly $O(n)$ when n doubles.
- $O(n^2)$: Quadratic - Doubling n multiplies time by 4.
- $O(n^k)$: Polynomial - Time multiplied by 2^k.
- $O(2^n)$: Exponential - Doubling n multiplies time by 2^k.



0.2 Hash Tables

Properties: Keys must be **hashable** (immutable). **Operations**:

- $O(1)$ insert, delete, lookup (average).

Hash Function: Deterministic: same input → same output. **Collisions**: Handled by **chaining** (list per bucket). **Python Dict**:

- Keys must be **hashable**.
- Values can be any type.
- Lists cannot be keys (mutable).

0.3 Graphs

For some graphs, DFS is equivalent to BFS.
Represent a **2D image** having M rows and N columns using a graph
Space Complexity is MN^2 in Adjacency Matrix
Space Complexity is MN in Adjacency List
Definition: $G = (V, E)$ where V=vertices, E=edges. **Types**:

- **Undirected**: edges have no direction (e.g., Twitter followers).
- **Directed**: edges **bidirectional** (e.g., Facebook friends).
- **Weighted**: edges have numerical values.
- **Unweighted**: edges represent presence/absence only.

Adjacency List: Array of lists: $Adj[u]$ is a list of neighbors of u.

- Space: $O(V + E)$.
- Good for **sparse graphs** ($E \ll V^2$).
- Lookup time: $O(V)$ worst case.

Adjacency Matrix: A matrix where $A[i][j] = 1$ if edge exists.

- Space: $O(V^2)$.
- Lookup time: $O(1)$.
- Good for **dense graphs**.

0.4 Breadth-First Search (BFS)

Purpose: Explores neighbors before going deeper. Finds shortest paths in unweighted graphs. **Data Structure**: Uses **Queue** (FIFO). **Complexity**: $O(V + E)$. **Implementation**:

1. Initialize stack with start node.
2. Mark start as **visited**.
3. While stack not empty:
4. - Pop node. #stack.pop(0) in Queue
5. - Process node.
6. - Enqueue unvisited neighbors.
7. - Mark neighbors as **visited**.

```
# --- Option 1: Built-in BFS traversal ---
print("BFS traversal using networkx.bfs_tree:")
bfs_tree = nx.bfs_tree(G, source='A')
print(list(bfs_tree.nodes())) # nodes in BFS order
# OR equivalently:
print("\nUsing networkx.bfs_edges:")
bfs_edges = list(nx.bfs_edges(G, source='A'))
print("BFS edges:", bfs_edges)
# --- Option 2: Custom BFS implementation ---
def bfs_custom(G, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            for neighbor in G.neighbors(node):
                if neighbor not in visited:
                    queue.append(neighbor)
    return visited
```

0.5 Depth-First Search (DFS)

Purpose: Explores as deep as possible before backtracking. **Data Structure**: Uses **Stack** (LIFO). **Complexity**: $O(V + E)$. **Implementation**:

1. Initialize stack with start node.
2. Mark start as **visited**.
3. While stack not empty:
4. - Pop node. #stack.pop() in Queue
5. - Process node.
6. - Push unvisited neighbors.
7. - Mark neighbors as **visited**.

```
# --- Option 2: Custom DFS using recursion ---
def dfs_recursive(G, node, visited=None):
    if visited is None:
        visited = set()
    visited.add(node)
    print(node, end=" ")
    for neighbor in G.neighbors(node):
        if neighbor not in visited:
            dfs_recursive(G, neighbor, visited)
    return visited
print("\nCustom DFS (recursive):")
dfs_recursive(G, 'A')
# --- Option 3: Iterative DFS using a stack ---
def dfs_iterative(G, start):
    visited = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            # Add neighbors to stack
            stack.extend(reversed(list(G.neighbors(node))))
    return visited
```

0.6 Sparse Matrix format (Scipy)

This is "Compressed Sparse Row" format.
It means we store an adjacency list per row.
Sparse matrices come up a lot in practice, beyond just adjacency matrices in graphs. For example:
Word counts: We might represent a document by the words in it, but only a small fraction of all words would appear in a given document.
Ratings: We might represent an Amazon item by the user ratings, but only a small fraction of all users have rated a given item.

0.7 Sparse Matrix Formats

Compressed Sparse Row (CSR) vs. Column (CSC)

Feature	CSR	CSC
Storage Arrays	Values, Column Indices, Row Pointers	Values, Row Indices, Column Pointers
Optimized For	Row operations	Column operations
Access Pattern	Fast row slicing	Fast column slicing
Use Cases	Matrix-vector multiplication, row-wise operations	Column-wise computations, matrix factorization

CSR and CSC are efficient ways to store matrices with mostly zero entries, saving memory and speeding up relevant operations.

```
x = scipy.sparse.random(5, 5, density=0.2, format="csr",
    random_state=321)
```

0.8 Trees

Trees are vital for organizing data **hierarchically** (parent-child relationships). **Modeling Real-World Systems**:

- **File Systems** (folders/subfolders).
- **Organization Charts** (manager-employee).
- **XML/HTML Documents** (nested tags).

Efficiency & Algorithms:

- **Binary Search Trees (BSTs)**: Fast $O(\log n)$ average time for lookup/insertion/deletion.
- **Balanced Trees** (AVL/Red-Black): Consistently fast performance.
- **Heaps**: Used for Priority Queues and heapsort.

Advanced Structures & Applications:

- **Tries**: Autocomplete, spell checking.
- **Syntax Trees**: Compilers/interpreters.
- **B-Trees / B+ Trees**: Database indexing, file systems.
- **Decision Trees**: Machine learning models (random forests).

0.9 Tree Terminology

Node: Fundamental part, stores data. **Root**: The first node in a tree (no parent). **Edge**: A link connecting two nodes. **Parent**: Immediate predecessor (a node has only one parent). **Child**: Immediate successor (a node can have ≥ 0 children). **Leaf Node**: A node with zero children. **Height**: Maximum number of edges from the **root** to a **leaf node**. **Subtree**: A portion of a tree starting from any node and including all its descendants. **Balanced Tree**: A tree where the height of the left and right subtrees of any node are nearly equal.

0.10 NetworkX Code Examples

Create Graphs:

- `G = nx.Graph()` (undirected)
- `G = nx.DiGraph()` (directed)

Add Nodes & Edges:

- `G.add_node("A")`
- `G.add_edge("A", "B")`
- `G.add_edges_from([("A", "B"), ("B", "C"), ("B", "D")])`

0.11 Time Measurement

Code Profiling:

- `timeit`: Python magic for timing small code snippets.
- `line_profiler`: Per-line timing.

Memory Profiling:

- `sys.getsizeof()`: Object size.
- `df.memory_usage()`: Pandas DataFrame memory.
- `np.sum(array.nbytes)`: NumPy arrays memory.

0.12 Tips & Best Practices

Visualization:

- `nx.draw(G, with_labels=True)`

General Tips:

- Use **set** for membership lookup.
- Use **dict** for fast key-value lookups.
- Prioritize coherent **data structure** operations based on needs.
- Consider **space-time tradeoffs**.
- BFS is better for optimizing **shortest paths** in unweighted graphs.
- **Hash table operations** are $O(1)$ avg., not worst-case.
- Prefer **sparse matrices** for graphs with $E \ll V^2$.