Data Structures and Algorithms

Personal Notes – Bhadri Vaidhyanathan

Table of Contents

Terminologies

Defensive Programming: Programming assuming users will make mistakes when they are asked to do something or if program reqests them anything else.

Pseudocode: Write the logic in correct logical sequence but not bothering about syntax. You will mostly use formula for calculations, counter updates, loops (for or While)

Byte: 8 bits. Each bit is a placeholder for binary values 0 or 1. Bytes make up the RAM as in 8GB RAM is 8 Billion Bytes (G-Giga = Billion)

Basics

Byte: 8 bits. Each bit is a placeholder for binary values 0 or 1. Bytes make up the RAM as in 8GB RAM is 8 Billion Bytes (G-Giga = Billion)

Integer are stored typically in 4 BYTES and so 32 bit spots.

ASCII characters like Alphabets take up 1 BYTE or 8 bit spots.

RAM: Random Access Memory

First, Arrays are the simplest Data Structures and it is contiguous in memory.

Since Arrays are contiguous (each address location of the bit is known),accessing the elements of an array is O(1) complexity since the bit address is known and it directly accesses the value.

How to Solve Computational Problems:

And then write an algorithm.

Step-by-Step Approach to Solving Any Computational Problem

1. **Understand the Problem Statement**
   - *Why?* Ensures you know what is being asked before attempting a solution.
   - *How?* Read the problem carefully, identify input/output, constraints, and edge cases.
2. **Break the Problem into Smaller Subproblems**
   - *Why?* Decomposing makes complex problems more manageable and solvable.
   - *How?* Identify independent components (e.g., input parsing, computation, output formatting).
3. **Identify the Most Suitable Algorithm or Approach**
   - *Why?* Helps find an optimal, efficient solution instead of brute force.
   - *How?* Consider known algorithms (sorting, searching, dynamic programming, etc.), and analyze time/space complexity.
4. **Write a Pseudocode or Outline the Solution**
   - *Why?* Provides clarity before implementation, reducing coding mistakes.
   - *How?* Draft step-by-step logic in plain English or pseudocode before coding.
5. **Implement the Solution**
   - *Why?* Converts logic into executable code.
   - *How?* Write code incrementally, using functions/modules for readability.
6. **Test the Solution with Different Inputs**
   - *Why?* Ensures correctness, handles edge cases, and detects bugs early.
   - *How?* Use small test cases, edge cases (e.g., empty input, large values), and expected outputs.
7. **Optimize the Code (if needed)**
   - *Why?* Improves efficiency in terms of time and memory usage.
   - *How?* Use better data structures, remove redundant computations, and analyze Big-O complexity.
8. **Document and Refactor the Code**
   - *Why?* Enhances readability, maintainability, and future usability.
   - *How?* Add comments, modularize code, and follow clean coding principles.
9. **Validate with Real-World Scenarios** (if applicable)
   - *Why?* Ensures the solution works for practical applications.
   - *How?* Run it in production-like conditions, check performance with large-scale data.

Code Efficiency

Space and time: how long or how fast and how much memory/space ? Is there a better data structure?

How long will a computation take to complete?

Lets take a simple problem of calculating number of days (disregard leap years for simplicity) in 1000 years.

Step 1: Define the Problem

We need to compute the number of days between two dates that are 1000 years apart, assuming no leap years. This means each year has exactly **365 days**.

Total days to compute: 1000×365=365,000 days

A "normal computer" today can perform **billions of operations per second**. We will assume a modest processing speed **3 GHz** and walk through the estimation.

Step 2: Define a Reasonable Assumption for Computational Speed

A typical modern CPU has a clock speed of around **3 GHz**, meaning **3 billion cycles per second**. However, not every cycle completes a full computation.
Let's assume:

- A simple subtraction (e.g., `end_date - start_date`) is **1 CPU cycle**.
- A loop iterating 365,000 times (e.g., summing each day) takes **2 cycles per iteration**.

If we use a simple algorithm that loops through each day:

365,000 iterations×2 cycles per iteration=730,000 cycles

Step 3: Compute Time Taken

If the CPU runs at **3 GHz** (i.e., **3×10^9 cycles per second**), the time taken is:

730,000 cycles / 3×10^9 cycles/second = 0.000243 seconds = 243 microseconds

Algorithms

An algorithm is essentially a series of steps for solving a problem. Usually, an algorithm takes some kind of input (such as an unsorted list) and then produces the desired output (such as a sorted list).

Input Size and Code Efficiency

The time taken for a code is directly proportional to the number of times it is run which is the input size.

Big O Notation

Big-O notation describes an algorithm's time complexity in terms of input size n. It helps evaluate Growth rate or how the runtime grows as the input increases. An algorithm searching for a word in a dictionary could luck out and get it in first run but we dont care about luck but consider the worst case situation which is that all words in dictionary will be accessed. The worst case or upper-bound value is used generally as a norm when talking about Big O. It will be specified if its not the case like Avg. Case instead of default worst case.

Asymptotic Behavior

- Big-O notation focuses on **Asymptotic Behavior** – ignore smaller aspects as negligent. Or ignore the early part of the Big O line chart and look when n gets big. With such an approach, small inputs, constants and lower order terms that have little impact on the efficiency are ignored.
- Above focuses on only the growth trend which is not dependent on low or powerful hardware. Powerful or powerless machine, growth trend remains same.

Basic Operations / Instructions

First few operations or tasks are quite simple thus called basic and they each have constant time. Constant Time means even if n increases to millions, the execution time is a constant one so this is defined as O(1).

Eg: Adding 2 number or 10 number takes the same time **Time Taken**(SUM(1,2)) = **Time Taken**(SUM(1,2,3,4,5,6,7,8,9,10)) thus O(1)

- Arithmetic operations
- Logic operations (<, >, ==, etc.)
- Statements like if or return
- Accessing a memory location, such as writing or reading a variable value

Because the number of operations in a basic instruction is constant, it doesn't depend on the amount of data. Since we care about how the execution time grows as the amount of data grows, we can focus on counting the number of basic instructions the algorithm performs.

How to Determine Growth Rate from Code

To determine an algorithm's growth rate, follow these steps:

1 Identify the Dominant Operations

- Look for **loops, recursive calls, complex math calcs and key operations** that contribute to runtime.

2 Express the Number of Executions in Terms of n

- Count how many times the main operations execute.

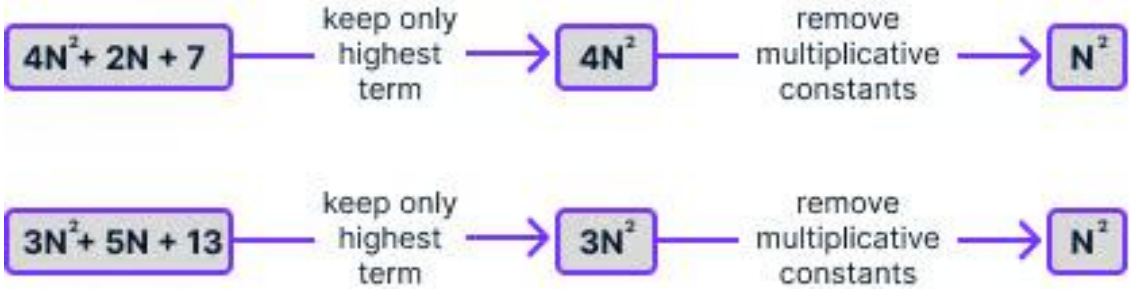
3 Drop Constants and Lower-Order Terms

- Big-O notation describes asymptotic behavior, so we ignore constants and less significant terms.

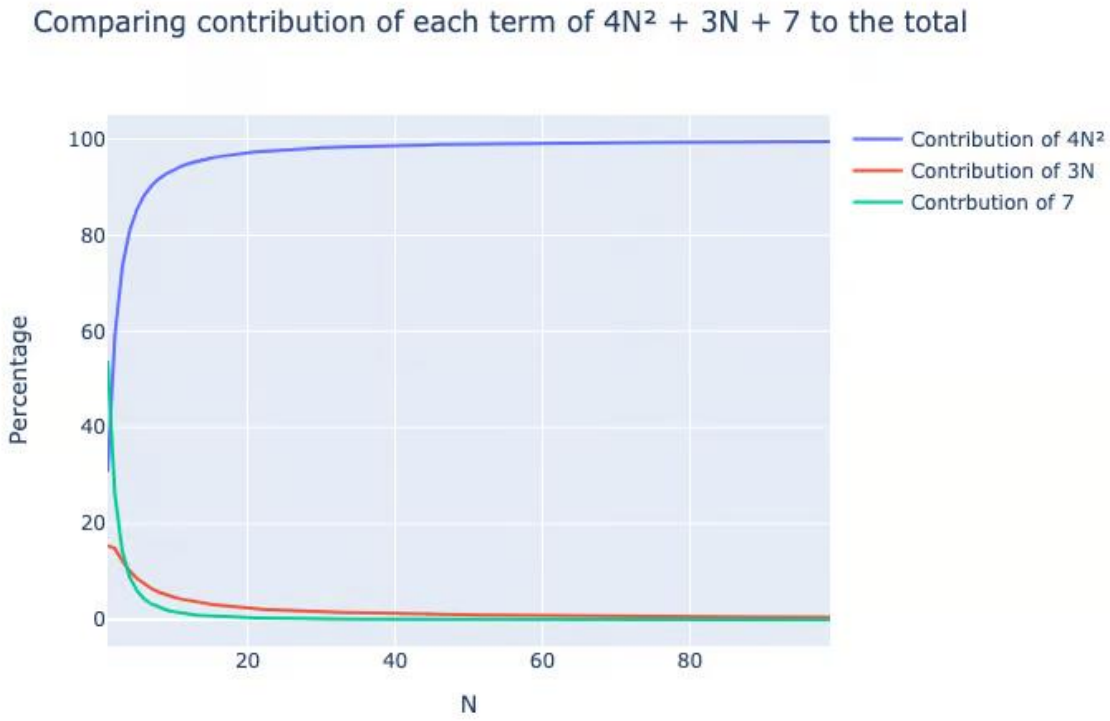
4 Classify into Standard Complexity Classes

- Compare against standard complexities as below

Another example of two Algos:



How:



As above that N^2 is so dominant on the time taken after just little bit increase in N.

Example: Time and Space Complexity

Factors:

Time	**Space**
Operations (assignment, formula)	Variables

Comparison	Data Structures
Loops	Allocations
Pointer References	Function Instantiation (Class call)
Outside Function Calls	

```
example_function(int):
      y = 1
      y = y + 4
      for i in range (10):
            outside_func()
            z = 10
            y += 1
return y
```

	w.r.t Time Complexity	w.r.t Space complexity
`example_function(int):`	Nil	O(1) X 2 for the function and then the int => A + B
`y = 1`	Skipping assignments as negligible	O(1) lets refer to this op as --> C
`y = y + 4`	O(1) ; this is not assignment but operation ; lets refer to this op as --> A	
`for i in range (10):`	O(n) --> B	
`outside_func()`	O(n) --> C	
`z = 10`	Skipping assignments as negligible	O(1); since within the loop z spot is reused every time and so it remains O(1) instead of becoming O(n) => D
`y += 1`	O(n) ; this is not assignment but operation --> D	
`return y`		

Time Complexity:

- A + nB + nC + nD --> A + n (B + C + D) -->
- Lets say A = K1 and B+C+D = K2. So => K1 + nK2
- As n increases to 100 and thousands, K1 constant and K2 a constant become negligible.
- So reduced to n thus it is O(n)

Space Complexity:

- A + B + C + D all constants so it is just O(1)

Logarithms' Role:

First its Log Base 2 since we are in Computer Science Domain.

Logarithms is basically is a scaling function like sigmoid / softmax.

Log_2X --> is asking the question, how many times should the number X be halved (because base is 2) to arrive at 1. Eg: $Log_28 = 3$ (8 > 4> 2 > 1).

$Log_{10}Y$ --> is asking the question, how many times should the number Y be 1/10thed (because base is 10) to arrive at 1. Eg: $Log_{10}1000 = 3$ (1000 > 100> 10 > 1)

Computer science domain uses base 2 widely so in time complexity log n refer to base 2.

Thus Log n refers to Logarithmic time where we are reducing the problem size by half each step.

Types of Complexities:

Complexity	Growth Rate / Name	Example Code
O(1)	Constant	Accessing an array element
O(logn)	Logarithmic	Binary Search
O(n)	Linear	Loop through an array
O(nlogn)	LogLinear	Merge Sort, QuickSort (avg case)
O(n^2)	Quadratic	Nested loops
O(n^3)	**Cubic**	**Matrix Multiplication**
O(n^k)		
O(2^n)	Exponential	fibonacci
O(n!)	Factorial	Recursive permutations

NOTE: $O(N^3)$ is till where hardware changes to superior or GPU helps make things faster but for higher complexities like $O(2^n)$ even superior hardware will be slow and thus the algo needs to be relooked and complexity reduced.

Space Complexity

space complexity : how efficient our algorithm is in terms of memory usage

Python involves additional memory usage for house-keeping overhead activities. In C/C++, an integer type takes up 4 bytes of memory to store the value, but in Python 3 an integer takes 14 bytes of space. Again, this extra space is used for housekeeping functions in the Python language.

Let's ignore this aspect and just assume below:

Type	Storage size
char	1 byte
bool	1 byte
int	4 bytes
float	4 bytes
double	8 bytes

Practical Application

1) If the clean_data() function has complexity O($\mathbf{N^2}$) and analyze_data() has complexity O($\mathbf{N^3}$), then reducing the complexity of clean_data() will not improve the overall complexity of function(). It's better to focus on improving the analyze_data() function.

```
def function(data):
  clean_data(data)
  analyze_data(data)
```

2) Hardware decisions: For example, an O($\mathbf{N^3}$) algorithm, despite its poor scalability, might still be fast enough for tasks with limited data if run on superior hardware or a faster programming language. Conversely, if an algorithm has an exponential time complexity **O($2^{\mathbf{N}}$)**, **no hardware upgrade will suffice**, indicating the necessity for a more efficient algorithm.

3) Time Complexities of popular data science actions:

Time complexity	Type	Examples:
Constant Time: O(1)	**Operations that do not depend on input size**	Accessing a specific element in a NumPy array: arr[i], PD series or df Checking shape: df.shape Assigning a value: arr[i] = 10 Boolean masking on a NumPy array (view-based slicing)
Log Time: O(log n)	**Operations that reduce data size exponentially**	Binary search on a sorted NumPy array: np.searchsorted(arr, value) Some index-based operations in Pandas (e.g., searching in a sorted index)
Linear Time: O(n)	**Operations that iterate over all elements once**	Array or pd.df column stat like min, max, average: np.min(arr), df[col1].avg Filtering a Pandas DataFrame: df[df['column'] > 5] Applying a function row-wise: df.apply(func, axis=1) Appending a new row to a Pandas DataFrame (increases size) Iterating through a DataFrame using .iterrows() (inefficient)
Linearithmic Time: O(n log n)	**Operations involving sorting or divide-and-conquer**	Sorting a Pandas DataFrame: df.sort_values(by='column') Merging two sorted arrays (in general case) Some join operations in Pandas (merge on sorted keys) **Sorting a NumPy array: np.sort(arr)**
Quadratic Time: O(n2)	**Operations that involve nested iterations**	Removing duplicates without hashing: df.drop_duplicates() (worst case) Some DataFrame joins (merge) when keys are not indexed Some Pandas groupby operations with complex aggregations Applying a function across all pairs of rows
Cubic Time and Worse:		Some nested loops in Pandas apply functions Certain brute-force algorithms on data (e.g., computing all possible pairs in a large dataset) Deeply nested joins without indexes

4) ☑ **Fully Vectorized Pandas Operations (Fast)**
These operations **leverage NumPy under the hood**, meaning they are as fast as NumPy:

- **Arithmetic on columns**
  `(df['col'] + 10, df['col1'] * df['col2'])` → **O(n)**
- **Boolean filtering**
  `(df[df['col'] > 5])` → **O(n)**
- **Aggregation**
  `(df.mean(), df.sum(), df.max())` → **O(n)**
- **String operations with `.str`**
  `(df['col'].str.lower())` → **O(n)**
- **Sorting with NumPy backend**

(`df.sort_values()`) → **O(nlogn)**

- **Merging on indexed keys**
  (`df.merge(other_df, on='key')`) → **O(n) average**

- **GroupBy with simple aggregations**
  (`df.groupby('key').sum()`) → **O(n)**

◆ **Why fast?** These are implemented in C via NumPy and optimized for bulk processing.

4B) Partially Vectorized or Slow Pandas Operations

A) Some operations look vectorized but involve loops under the hood, making them slower: `apply()` on rows (`axis=1`):

df['new_col'] = df.apply(lambda row: row['A'] + row['B'], axis=1) **# Slow!**

Why? `apply()` is not fully vectorized for row-wise operations; it loops over rows in Python.
Better alternative: Use column-wise (`axis=0`) operations or NumPy:

- df['new_col'] = df['A'] + df['B'] # Fully vectorized, very fast!

B) Using `.iterrows()` or `.itertuples()`:

- **Why?** `iterrows()` converts rows to Pandas Series, which are slow.
- **Better alternative:** Use vectorized column operations.

C) Merging on non-indexed keys (`df.merge()`):

If merging on a non-indexed column, Pandas has to scan both DataFrames, leading to O(n^2) worst-case complexity.

Fix: Set index first: df1.set_index('key').join(df2.set_index('key')) **# Faster**

Tips and Tricks in Data Science Tasks

A) Use NumPy Instead of Pandas When Possible

Pandas is built on NumPy but adds overhead. When working with large numerical arrays:

- Use **NumPy** instead of Pandas for faster computations.

```
# Slower (Pandas)
df['new_col'] = df['A'] + df['B']  # Vectorized but has overhead

# Faster (NumPy)
df['new_col'] = df['A'].to_numpy() + df['B'].to_numpy()
```

Time Complexity: O(n) in both cases, but NumPy runs faster.

B) Avoid Using `.apply()` and Loops in Pandas

Using, `.apply()` or loops (`.iterrows()`, `.itertuples()`) is much slower than vectorized operations.

- Slow Approach (Loop-based):
  df['new_col'] = df.apply(lambda row: row['A'] * 2, axis=1) # Avoid!
- Faster Approach (Vectorized): df['new_col'] = df['A'] * 2 # 100x faster

C) Use Efficient Data Types in Pandas (Reduce Memory)

Large datasets can consume huge memory if using `float64` or `int64` unnecessarily.

- Use `int32` instead of `int64` when possible.
- Convert object/string columns to category if they have low cardinality.
- Use `float32` instead of `float64` for large numeric columns.

✅ **Space Complexity:** Reduces RAM usage by **2-4x**!

D) Use `.loc[]` Instead of `.iloc[]` for Selecting Rows

- `.loc[]` is faster when selecting by labels/index **if the index is optimized.**
- `.iloc[]` is optimized for selecting by position but may be slower for large DataFrames.

E) Sort Before Searching for Faster Lookups

Binary search is much faster than linear search. If you **sort once**, searches are O(log n) instead of O(n)

```
df = df.sort_values('A')  # Sorting: O(n log n)
df['A'].searchsorted(50)  # Binary search: O(log n)
```

F) Use Multi-Processing or Vectorized Parallelism

Python has GIL (Global Interpreter Lock), so multi-threading isn't always helpful, but:

- NumPy & Pandas are already multi-threaded internally.
- For custom Python functions, use multiprocessing.

🔹 Example: Parallel Pandas Apply Using `swifter`:

```
import swifter
df['new_col'] = df['A'].swifter.apply(lambda x: x * 2)
```

G) Use .merge() Instead of Nested Loops for Joins

If you only need aggregated results for a subset of data, **filter first** instead of aggregating everything and then filtering.

H) Use Sparse Matrices for High-Dimensional Data

If most values in a dataset are **zero**, **don't store them explicitly**.

- **Use Scipy's `sparse` matrix** instead of dense NumPy arrays.

```
from scipy.sparse import csr_matrix
sparse_matrix = csr_matrix(dense_matrix)  # Converts large matrix into compact
form
```

I) Avoid Expensive .groupby() if .pivot_table() is possible

- `Pandas .groupby() is powerful but can be slow on large datasets.`
- Instead of **grouping** + **aggregating**, use `pivot_table()` where applicable.
- Pivot tables are more optimized internally!

```
# Slower:
df.groupby(['col1', 'col2'])['values'].sum()

# Faster:
df.pivot_table(values='values', index='col1', columns='col2', aggfunc='sum')
```

🔥 Bonus: Use Polars for Large-Scale Data

Pandas is not always the best choice for massive datasets.

- **Polars** is a new DataFrame library optimized for performance.
- It is multi-threaded and significantly faster than Pandas for many operations.
- **Polars can be 5-10x faster than Pandas!**

```
import polars as pl
df = pl.read_csv("large_file.csv")
df.groupby("category").agg(pl.col("sales").sum())
```

Data Structures

(Some data structures are just a specific fixed way of using of a basic data struc like array)

Key data structures:

1. **Arrays** – A contiguous memory structure storing elements of the same type. Supports fast indexing but costly insertions and deletions. OG data structure that helped in fast indexing and a contiguous memoery storage.
    - Fixed size or Dynamic (using resizing techniques).
    - Faster access using **indexing**.
    - Efficient for scenarios where memory is pre-allocated.
2. **Linked Lists** – A dynamic collection of nodes, where each node contains data and a reference to the next (or previous) node. Efficient insertions/deletions but slower access than arrays. Overcame arrays' fixed size and expensive insertions/deletions.
    - Flexible size with no predefined limit.
    - Efficient for frequent insertions and deletions.
    - Requires **extra memory for pointers**.
3. **Trees** – Hierarchical data structures composed of nodes, with a root and child nodes. Common types include binary trees, binary search trees (BST), AVL trees, and **heaps**. Used in hierarchical data representation, search optimization, and decision trees.
    - Overcame slow searching in Linked lists and provided sorted storage.
    - Trees are implemented with linked list like data struc in that Tree node could have more than one pointer directing to its children while linkedlist has only one pointer.
4. **Stacks** – A Last-In-First-Out (LIFO) structure. Designed for and mostly used in recursion, backtracking, and expression evaluation.
    - Stacks are usually implemented using Arrays (can be done with LLs as well).
    - A list in Python is a stack thus you have append and pop ops which are usual stack ops.
    - Copy-Paste Item history is LIFO. Most recent item, Last-in, is what gets out, First-Out.
5. **Queues** – A First-In-First-Out (FIFO) structure where elements are added at the rear and removed from the front. Variants include circular queues, priority queues, and deques. Designed for **FIFO** operations like task scheduling.
    - Queues are usually implemented using linked lists or circular arrays.
6. **Hash Map** – Key-value pair structures where hashing technique is used to store and retrieve data efficiently. hashing function maps keys to indices where value is stored. Hash values are not stored but hash function used to determine the index location. Reduces search complexity to near O(1) in ideal cases. Provided constant-time lookup, solving slow search issues in lists/arrays.

- Maps / Dictionaries are implemented with hash maps – Key-value pair structures for fast lookups, insertions, and deletions. Implemented using hash tables or balanced trees.

7. **Heap**

   A **Heap** is a specialized **binary tree** that satisfies the **heap property** - a rule between parent and child nodes. It is commonly used in algorithms that require efficient extraction of the smallest or largest elements, like **priority queues**.

8. **Graph** – A set of nodes (vertices) connected by edges. Can be **directed** or **undirected**, **weighted** or **unweighted**. Used in network modeling, pathfinding algorithms, and social network analysis. Created to model complex relationships like networks and social connections.

Top Applications:

Arrays	Omnipresent in Data Science
Hash Maps	Dictionaries in Python , Word embeddings (e.g., `word2vec` stores word-vector mappings in a hash table)
Trees	Binary Search Tree are Used in **database indexing (SQL databases)**
Graphs	Used in network analysis, recommendation systems, and graph-based ML models. Social network analysis (e.g., LinkedIn's People You May Know).
Linked Lists	Used in OS's memory allocation, MS Word Undo/Redo uses it, Browser back /forwrd uses it. Music/Video apps use a doubly-linked list to manage playlists.

Static Array vs Dynamic Array vs Linked List

Aspect	Static Array	Dynamic Array	Linked List
Size	Fixed at creation	Grows or shrinks as needed	No fixed size, grows dynamically
Memory Allocation	Continuous block	Continuous block, resizes by copying	Nodes scattered, linked using pointers
Access Time	O(1) (Direct access via index)	O(1) (Direct access via index)	O(n) (Sequential traversal)
Insertion/Deletion	O(n) (Shifting elements)	O(n) (Shifting elements)	O(1) at head/tail, O(n) elsewhere
Memory Efficiency	More efficient, no overhead	Wastes memory during resizing	Extra memory for pointers
Use Case	Best for fixed-size data	Best for variable-size data	Best for frequent insertions/deletions

- **Static Array:** Best when the **size is known and quick access is essential**.
- **Dynamic Array:** Useful when **size changes are unpredictable**, like in dynamic data storage.

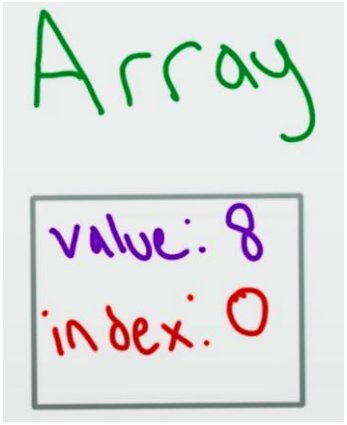
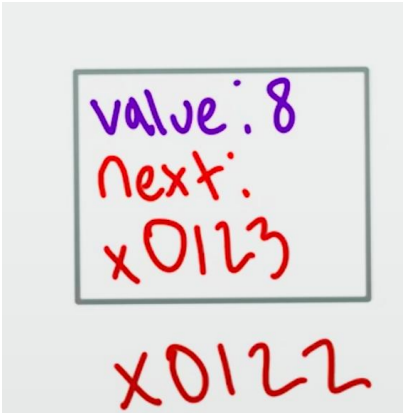
- **Linked List:** Ideal when memory fragmentation is not a concern and **frequent insertions/deletions** are required. slower access than arrays.

Linked Lists

A dynamic collection of nodes, where each node contains data and a reference to the next (or previous) node. Efficient insertions/deletions but slower access than arrays. Why not just use arrays? Because in big arrays/lists, it is a pain and very inefficient inserting or deleting an item in the array/list while very smooth in Linked Lists.

Other Types:

- Doubly Linked – Has two pointers, one points at previous item and another pointing at the next item. Helps in going forward and backward in list when needed.
- Circle Linked – Last item points at the first item. Help in traversing around the list

	
Array holds the value and the index num,ber for the value. The Array is held in contiguous memory	LL holds the value and the memory location of the next value. If null then it is the last value. Thus insertion takes Constant Time instead of Linear Time.

- Note: Care must be taken when deleting an item in LL such that the “next” pointer is retained and not lost when deleting the item

Stacks and Queues

Stacks and Queues utilize arrays and linked lists as building blocks to create slightly more complex data structures with different uses and efficiencies.

Stacks/Queues Just "Specialized Arrays / LinkedLs"?

Yes, in Python.

but **not in lower-level languages** where managing memory and efficiency is more manual. The core takeaway is that **understanding their distinct behaviors helps apply them effectively in algorithms and system design**.

Note: Stacks (using normal list) and Queues(using collections.deque – doubly-linked package) in Python are actually implemented with just array data struc type or linked list.

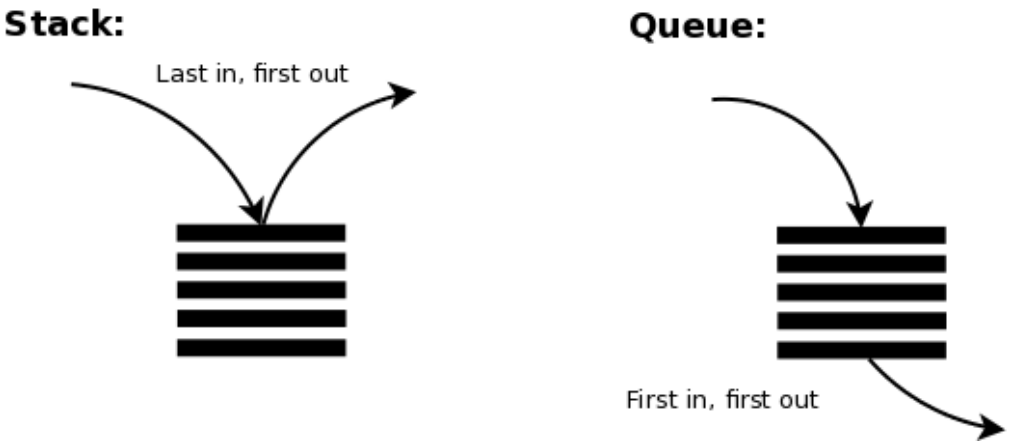
Why then a different Data Struc and why separate and teach them?

Many problems naturally require stacks/queues:

- Stacks: Used in DFS, expression evaluation, backtracking (e.g., undo feature in apps)
- Queues: Used in BFS, scheduling, caching, and message queues

Memory and Performance Considerations →

- Stacks help in function call management (call stack)
- Queues ensure efficient order processing in simulations, web servers, etc.



Stacks

Focus on the more recent items added to the collection than what went in much before. Stacks can be implemented with Arrays and LL as needed.

- A Python list is a stack. Even has pop() which are OG Stack ops. pop() removes&returns the last item and thus achieves LIFO. Accessing any other item makes it only as good as a list and not a stack. So to create one, just create a list [] in python

Popular usages:

1. Function Call Stack (Recursion & Execution)
Used in programming languages to manage function calls. When a function is called, its state (local variables, return address) is pushed onto the stack. When the function returns, it is popped from the stack.

2. Undo/Redo in Text Editors

Used to track changes in applications like Microsoft Word, Google Docs, Photoshop.
Each operation is pushed onto the stack. Undo pops the last action, and redo pushes it back.

4. Browser Back/Forward Navigation
Web browsers (Chrome, Firefox) use two stacks for back and forward navigation.

5. Parentheses & Syntax Validation
Stacks are used to check balanced parentheses in expressions like ({[ ]}). Used in compilers & interpreters for parsing code structures (if-else, {}, () matching).

6. Memory Management in Operating Systems
OS uses a stack for managing memory allocation (e.g., local variable storage, return addresses).
Example: Stack Overflow occurs when recursive function calls exceed memory limits.

Queue

A **queue (FIFO: First In, First Out)** is used when **order of processing matters**, ensuring elements are processed in the sequence they arrive.

In Python, you can use collections.deque which is a doubly-linked list that will make FIFO fast and efficient.

Popular usages:

1. Task Scheduling (CPU & I/O Scheduling)
Operating systems (OS) and Processor use queues for process scheduling.
Example: CPU Scheduling → Ready Queue in Round Robin scheduling.

2. Print Queue in Printers
Print jobs are queued up and processed in order.

3. Handling Requests in Web Servers
Web servers process incoming client requests in a queue.
Example:HTTP request handling in web servers like Apache, Nginx.
Load balancers distribute requests across multiple servers.

4. Data Buffers in Streaming Services (Netflix, YouTube)
Media streaming platforms queue up video/audio chunks for smooth playback.

Trees

A **tree** is a hierarchical / **sequential** data structure that consists of **nodes connected by edges**. Unlike arrays, linked lists, stacks, or queues, trees allow **efficient hierarchical representation** of data.

Trees like in Linked Lists are connected in that they store the locations of the child nodes. In Trees, one node can have more than 1 child nodes.

- No Cycling in Trees, you must not be able to come across same node twice whatever directions u take

What Kind of Data is Stored in Trees?
Trees can store structured, hierarchical, or ordered data, such as:

- File system structures (folders & files).
- Database indexes (B-Trees in databases).
- Searchable data (binary search trees, decision trees).

Popular Usage:

Tree Type	Usage
Binary Search Tree (BST)	Fast searching & sorting (Dictionaries, Autocomplete).
B-Trees / B+ Trees	Database indexing (MySQL, PostgreSQL).
Huffman Trees	Data compression (JPEG, MP3, ZIP).
Merkle Trees	Blockchain security (Bitcoin, Ethereum).

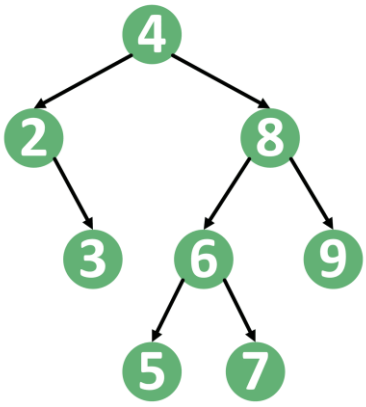
Binary Search Tree (BST)

– A Specialized Tree for Fast Searching

A **Binary Search Tree (BST)** is a special type of **binary tree** where each node follows a strict ordering rule:

- **Left subtree** contains values **smaller** than the node's value.
- **Right subtree** contains values **greater** than the node's value.

This ordering makes **searching, insertion, and deletion** highly efficient.



- Note how the number of the nodes is lowest on left and rises slowly as we move right. No node is exactly on top of another but slightly to left or right of nodes along a vertical axes
- Each Parent node has only two child nodes thus the name "BINARY"

Data Storage or Guide/Template for Search? Both

✓ BST stores data in applications like databases, dictionaries, and file systems.

✓ BST traversal approach is used in searching algorithms (like Binary Search) even when data is in arrays.

Hash Maps / Tables

Hash Maps (Maps / Dictionaries) – Key-Value Data Structure
A Hash Map (also called Map, Dictionary, or Associative Array) is a key-value data structure that allows efficient insertion, deletion, and lookup based on a unique key.

Hash Maps were created to solve the problem of **fast lookups** without needing to search through a list or tree.

- Searching in **arrays/lists** takes **O(n)** time.
- Searching in **BSTs** takes **O(log n)** time.
- **Hash Maps provide O(1) average-time complexity for lookups**, making them much faster than lists or trees.

How Does a Map Work?

- Uses a **hash function** to compute an index from the key.

Hashing is really a custom on-demand index that is created on the spot when new value comes to be stored. Same way the hash was created, it is redone, when a word is searched: instead of going one-by-one or even using fancy search algos, we simply re-calculate the hash from the search value and directly jump to hash index location.

- Stores values in a **hash table** at the computed index.
- Resolves collisions using techniques like **chaining (linked lists) or open addressing**.

How Hashing Works in a Hash Table

1. **Compute Hash:** Convert a key into an index using a **hash function**.
    - Example: `"JohnDoe"` → Hash Function → `index 5`
2. **Store the Value:** Store data at that index in the hash table.
3. **Lookup:** Given `"JohnDoe"`, compute the same hash to get back `index 5` and retrieve the value instantly.

Why Not Just Search in an Array Instead?

- Array search (linear or binary search) takes O(n) or O(log n).
- Hash table lookup takes O(1) (on average).

Hash tables **skip searching** by directly jumping to the computed index.

Why Hashing Wins

Imagine a **student database** storing student records using their roll number or email as a key:

Without Hashing (Direct Indexing or using rollnumber for indexing)

- Roll numbers: `1001, 2034, 5060, 9023`
- You'd need an array of **size 9023**, even though only 4 students exist! **Waste of space.**

With Hashing

- Hash function maps $1001 \rightarrow 2, 2034 \rightarrow 8, 5060 \rightarrow 4, 9023 \rightarrow 1$.
- Uses a **small hash table of size ~10 instead of 9023**.

Collisions

When two different values end up with same hash id. The data to be stored is not concatenated but a Linked List or better a BST is created in which both the values are stored. BST is better since it has better Time complexity than Linked List. Chaining is the name of this technique.

Chaining (Linked List, BST, or Buckets)

Instead of finding a new slot, store multiple values at the same index using a linked list, BST, or dynamic array (bucket).

Algorithms

Important CS Algorithms (Non core ML algos) for Data Scientists:

Sorting & Searching (for Data Processing & Feature Engineering)

- **Sorting Algorithms (QuickSort, MergeSort, HeapSort, Bucket Sort, Radix Sort)**
  - Used in large-scale data preprocessing (e.g., sorting datasets for indexing).
  - **Decision point**: Choosing the most efficient sorting algorithm for large datasets.
- **Binary Search**
  - Used in search operations within large sorted datasets.
  - **Decision point**: Whether to pre-sort data for efficient searching.

Hashing & Feature Encoding Algorithms

- **MinHash (for Duplicate Detection)**

- Used in near-duplicate detection and locality-sensitive hashing.
- **Decision point**: When working with text similarity or deduplication.

🔹 **Bloom Filters (for Large-Scale Lookups)**

- Probabilistic data structure used in massive dataset lookups.
- **Decision point**: When to use an approximate lookup vs. Exact.

Sampling & Statistical Algorithms

🔹 **Bootstrapping & Resampling**

- Used in statistical modeling and A/B testing.
- **Decision point**: Choosing sampling techniques for unbiased estimation.

🔹 **Monte Carlo Methods**

- Used in probabilistic modeling and simulations.
- **Decision point**: When modeling uncertain scenarios.

Binary search

Binary search is a search algorithm where we find the position of a target value by comparing the middle value with this target value.

Binary Search Algo

- Lets search for a number 1 in ordered list of numbers. [1,2,3,4,5,6,7,8].
- Binary search finds the middle number and decides if it is the number and if not, is it bigger or smaller. It drops the right / bigger half if smaller and left / smaller half if bigger. Thus since we are looking for 1, the middle number is 4 which is bigger than 1 so left / bigger half is dropped
- We are left is only [1,2,3,4]. we repeat above step and will be left with [1,2]
- We will finally find the number 1 in the 3^{rd} step. Thus we took 3 steps.
- $Log_2 8 = 3$ or n=8 here and log 8 = 3 thus log n time complexity.

How It Works

1. Start with the middle element.
2. If the target value is **smaller**, search the left half.
3. If the target value is **larger**, search the right half.
4. Repeat until the target is found or the search space is empty.

✅ **Time Complexity: O(log n)** (Much faster than linear search **O(n)**)
✅ **Space Complexity: O(1)** (Iterative) or **O(log n)** (Recursive due to call stack)

When Binary Search Is NOT Ideal

🚫 **If the data is unsorted** → Linear search might be the only option.

🚫When searching in **high-dimensional data** → k-d trees or hashing work better.
🚫 **If random access is expensive** → Linked lists don't support binary search efficiently.
🚫 **If distribution is uniform** → Interpolation search can be much faster.

Related Algos: As seen been below, there are specific algos for specific use cases

Algorithm	Time Complexity	When to Use?
Linear Search	O(n)	When the dataset is **unsorted** or very small.
Binary Search	O(log n)	When the dataset is **sorted** and random access is possible.
Ternary Search	O(log n)	When searching for the **minimum/maximum** of a unimodal function.
Interpolation Search	O(log log n) (Best)	When data is **sorted and uniformly distributed**.
Exponential Search	O(log n)	When working with **unbounded or infinite-sized arrays**.
Jump Search	O(√n)	When data is **sorted but too large for binary search**.
Hashing (e.g., Hash Table Search)	O(1) (Best Case), O(n) (Worst Case)	When fast lookup is needed with **unordered data**.

Popular Usage:

1. **Routine Datascience Tasks**

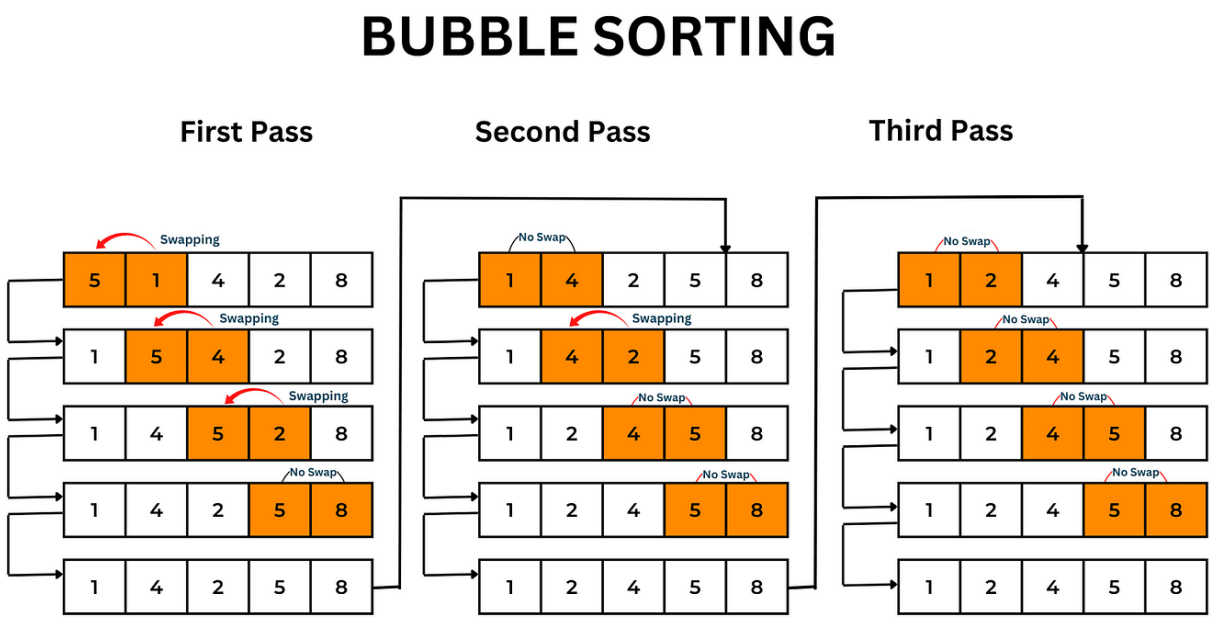
Python pandas uses it when using index searches

2. **Machine Learning & Data Science Applications**

- Hyperparameter Optimization (Binary Search on Learning Rate)
    - In deep learning, binary search is used to **find the best learning rate** (e.g., in fastai's `lr_find()`).
- Finding Decision Boundaries in Models
    - Binary search can be used to optimize model thresholds (e.g., finding the best decision boundary in classification tasks).
- Efficient Data Sampling
    - Used in stratified sampling and quantile computation.
- Anomaly Detection

- Used in threshold-based anomaly detection where you find an optimal boundary for flagging outliers.

Sorting Algorithms



1 Bubble Sort

🔹 **Idea**:

- Repeatedly swaps adjacent elements if they are in the wrong order.
- Large elements "bubble up" to the correct position in each pass.

🔹 **Steps:**

1. Compare adjacent elements.
2. Swap if they are out of order.
3. Repeat until the list is sorted.

🔹 **Time Complexity:**

Best Case	Average Case	Worst Case	Space Complexity
O(n)	O(n^2)	O(n^2)	O(1)

🔹 **Characteristics:**

✅ Simple and easy to implement.

❌ Very slow for large datasets (quadratic complexity).

📌 **Best suited for nearly sorted or very small datasets.**

---

2 Merge Sort (Divide & Conquer)

🔹 **Idea**:

- Recursively splits the array into halves until single elements remain.
- Merges sorted halves back together.

🔹 **Steps:**

1. Divide the array into two halves.
2. Recursively sort both halves.
3. Merge them back into a sorted sequence.

🔹 **Time Complexity:**

Best Case	Average Case	Worst Case	Space Complexity
O(n log n)	O(n log n)	O(n log n)	O(n)

🔹 **Characteristics:**

✅ **Stable** (preserves order of equal elements).
✅ **Guaranteed O(n log n) complexity** in all cases.
❌ Requires **O(n) extra space** for merging.
📌 **Best for linked lists, external sorting (huge datasets that don't fit in memory).**

---

3 Quick Sort (Divide & Conquer, In-Place)

🔹 **Idea**:

- Selects a **pivot** element and partitions the array around it.
- Recursively sorts left and right partitions.

🔹 **Steps:**

1. Pick a pivot (usually last, first, or median element).
2. Partition elements into two groups (smaller and larger than pivot).
3. Recursively apply QuickSort on both partitions.

🔹 **Time Complexity:**

Best Case	Average Case	Worst Case	Space Complexity
O(n log n)	O(n log n)	$O(n^2)$ (when pivot is poorly chosen)	O(log n) (in-place)

🔹 **Characteristics:**

✅ **Faster than Merge Sort in practice due to cache efficiency**.
✅ **In-place sorting (O(log n) space)**.

❌ **Worst case ($O(n^2)$) occurs if pivot selection is bad (e.g., already sorted list)**.
📌 **Best for general-purpose sorting when optimized with median-of-three pivoting.**

Comparison Table

Algorithm	Best Case	Average Case	Worst Case	Space Complexity	Stability	Best Used For
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	✅ Stable	Small/Nearly Sorted Data
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	✅ Stable	Large Data, Linked Lists
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	❌ Not Stable	General Sorting (Optimized Pivot)

Summary

📌 If stability is needed → Merge Sort
📌 If space efficiency is important → Quick Sort (in-place) or Heap Sort
📌 If worst-case performance is a concern → Merge Sort (guaranteed O(n log n))
📌 If sorting large datasets with efficient memory usage → Heap Sort
📌 If dataset is nearly sorted → Bubble Sort can be efficient (O(n))

Inversion Pairs in Sorting

An **inversion pair** in an array is a situation where **two elements are out of order** with respect to their correct sorted positions.

j which is bigger than i seems to be before i in the order, Thus this pair is one inversion pair.

Count of number of inversion pairs in a dataset is a measure of how unsorted the dataset is.

Importance of Inversion Pairs

- If an array has **zero inversions**, it is **already sorted**.
- The **maximum number of inversions** in an array of **n** elements is **n(n-1)/2** (for a fully reversed array).
- **Merge Sort** can count inversions efficiently in **O(n log n)** time.

Divide and Conquer paradigm

Divide and Conquer approach is suitable to solve a big (scale) problem by breaking it into smaller sub-problems, where each sub-problem looks exactly similar to the original problem. In general, there are three phases:

Divide - Break the given problem into smaller sub-problems

Conquer - Solve each sub-problem using recursion. The smallest sub-problem (base case) would have a simple straightforward solution.

Combine - This phase will automatically execute as a part of the recursion call stack, which combines the solution of smaller sub-problems to generate the final solution.

Quicksort and **Mergesort** are Divide and Conquer. There are a few points to note while deciding if one should go for faster Divide and Conquer approach:

- The problem should be on a bigger scale.
- The sub-problem must look precisely similar to the original problem in hand.
- Use recursion to solve the problem. It means that the solution will be built for the smallest sub-problem (base case) first.
- There is a trade-off between memory usage and speed of execution. Recursion comes with a price of extra memory usage for executing the call stack. But, if you use multi-threading, you can compute the solution even much faster.
- In case if many sub-problems look precisely the same, then we don't want to re-execute the same again and again. In such cases, you can consider storing the results of the execution, and thus reuse them whenever required. This strategy is called Memoization (in Dynamic Programming approach).

Greedy Algorithms

Concept

A **Greedy Algorithm** makes the **locally optimal choice** at each step with the hope that these choices will lead to a **globally optimal solution**. Unlike **Divide and Conquer**, it does **not** break the problem into subproblems but rather makes decisions step-by-step.

locally optimal choice : choose the next option as the **closest / easiest** to current instead of doing a thorough global search of all approaches and then deciding. The idea is that it will help later on.

Examples of Greedy Algorithms

Algorithm	Use Case	Time Complexity
Huffman Coding	Data Compression (e.g., ZIP files)	O(n log n)
Dijkstra's Algorithm	Shortest Path (non-negative weights)	O((V+E) log V)
Prim's Algorithm	Minimum Spanning Tree (MST)	O(E log V)
Kruskal's Algorithm	Minimum Spanning Tree (MST)	O(E log E)
Fractional Knapsack	Resource Allocation	O(n log n)
Activity Selection	Scheduling	O(n log n)

Advantages of Greedy Algorithms

✓ **Fast and Efficient** → Usually **O(n log n)** or **O(n)**.
✓ **Simple to Implement** → Often requires sorting followed by a single pass.
✓ **Works for Many Optimization Problems** → Particularly useful in graph and scheduling problems.

Limitations of Greedy Algorithms

❌ **Doesn't always guarantee the best solution** → Can get stuck in local optima.
❌ **May require a proof of correctness** → Not every problem can be solved greedily.
❌ **Fails if the problem doesn't have optimal substructure**.

Dijkstra's Algorithm

(~Diks-stra)

https://www.youtube.com/watch?v=bZkzH5x0SKU

Dijkstra's Algorithm is used in Google maps to find the shortest path between two locations.

- One location A is set as base and then for this base the shortest distance to all relevant points is estimated.
- The Algo start with A and then looks at all the unvisited nodes from A and marks the shortest distance.
- It then moves to next un-visited node and updates the shortest distance to base A and so on.'
- Algo keeps track of two things
  - List of visited and Unvisited nodes – visited nodes are never visited again
  - The previous node that has the shortest distance to A
- Eventually, you simply follow the shortest distance path from destination to A

Real-Life Applications of Dijkstra's Algorithm

🔹 **Google Maps & GPS Navigation** → Finding the fastest route between two locations.
🔹 **Network Routing (Internet Protocols)** → Used in **OSPF (Open Shortest Path First)** for routing data efficiently.
🔹 **AI Pathfinding (Game Development)** → Used in *A Algorithm** for pathfinding in video games.

✅ Greedy works well when the problem has an optimal substructure (like shortest path problems).

Optimal Substructure → A problem has optimal substructure if its **optimal solution can be constructed from optimal solutions of its subproblems**.

✅ Dijkstra's Algorithm is fast and efficient for graphs with non-negative weights.

Bellman-Ford Algorithm works even when **negative weights**

✅ It's widely used in real-world applications like maps, routing, and AI.

X Greedy does fail when local optimal choices don't lead to the global optimal solution.

Greedy Algorithm vs. Dynamic Programming

Both **Greedy algorithms** and **Dynamic Programming (DP)** are optimization techniques, but **Greedy fails** in certain cases where DP succeeds.

Where Does Greedy Fail?

- **Greedy makes the best local choice at each step** without considering the global picture.
- **DP optimally solves subproblems and builds up the global solution**, ensuring an optimal result.
- **Greedy fails when local optimal choices don't lead to the global optimal solution.**

Example Where Greedy Fails: The Coin Change Problem

Problem Statement

Given coins of denominations `{1, 3, 4}` and a total amount `6`, find the minimum number of coins needed to make `6`.

1. Greedy Approach (Fails)

- Pick the **largest coin first** at each step.
- Steps:
  - Take `4` → Remaining = `6 - 4 = 2`
  - Take `1` → Remaining = `2 - 1 = 1`
  - Take `1` → Remaining = `1 - 1 = 0`
- Total coins used = **3** (`4, 1, 1`)

🔴 **Wrong Answer!**

2. Dynamic Programming Approach (Correct)

DP builds solutions by solving smaller subproblems:

- `f(6) = min(f(6-1), f(6-3), f(6-4)) + 1`
- Using DP, the optimal way is:
  - Take `3` → Remaining = `6 - 3 = 3`
  - Take `3` → Remaining = `3 - 3 = 0`
- **Correct answer: Only 2 coins (`3, 3`)** ✅

🔹 **DP considers all possibilities, ensuring the best solution. Greedy fails because it picks the largest coin first and gets stuck.**

General Rule: When to Use Greedy vs. DP

Feature	Greedy	Dynamic Programming
Decision Making	Local best choice	Solves all subproblems first
Optimality	Sometimes fails	Always optimal
Overlapping Subproblems?	No	Yes
Used When?	Problems with the **Greedy Choice Property** & **Optimal Substructure**	Problems requiring **global optimization**

Dynamic Programming (DP)

Dynamic Programming (DP) is a Optimization paradigm, not a single algorithm. It is a method for solving optimization problems by **breaking them into overlapping subproblems**, solving each subproblem once, and storing the results to avoid redundant computations.

"Programming" in DP ≠ Computer Programming

- In the 1950s, **Richard Bellman**, who coined DP, used "programming" to mean **"planning" or "tabular computation"**, which was common in **optimization** at that time.
- Dynamic is a later addition to differentiate DP with Computer programming
- DP is a subfield of Optimization or one way of doing optimizations.
    - **Optimization** involve finding the **best possible solution** from a set of possible choices.
    - DP is a type of **optimization** where by **storing intermediate results** to prevent recomputation and optimization.

Key Concepts of Dynamic Programming

1. **Optimal Substructure** → A problem has optimal substructure if its **optimal solution can be constructed from optimal solutions of its subproblems**.
2. **Overlapping Subproblems** → The same subproblems are solved multiple times, making **memoization** or **tabulation** useful.

Two Approaches to DP

- **Top-Down (Memoization)** → Solve the problem recursively and store results to avoid redundant calculations.
    - Biggest problems first and saves the lessons to reuse.
- **Bottom-Up (Tabulation)** → Solve small subproblems first and build up to the final solution using an iterative table.
    - Smallest or easiest tasks first and save calculations so it is not repeated when doing higher level tasks

---

Is DP an Algorithm?

🚀 **No, DP is not a specific algorithm**, but a technique used in designing efficient algorithms for optimization problems.

👉 **Example tasks DP Algorithms used for:**

1. **Fibonacci Sequence** (Avoid redundant calculations)
2. **Knapsack Problem** (Optimizing item selection)
3. **Longest Common Subsequence** (Used in text comparison, DNA sequencing)
4. **Edit Distance (Levenshtein Distance)** (Used in spell check and NLP)

Popular Applications:

Knapsack Problem

The **Knapsack Problem** is a classic **optimization problem** where you must maximize the total value of items placed in a knapsack **without exceeding its weight capacity**.

DP is the best algo for this problem.

Graph Algorithms

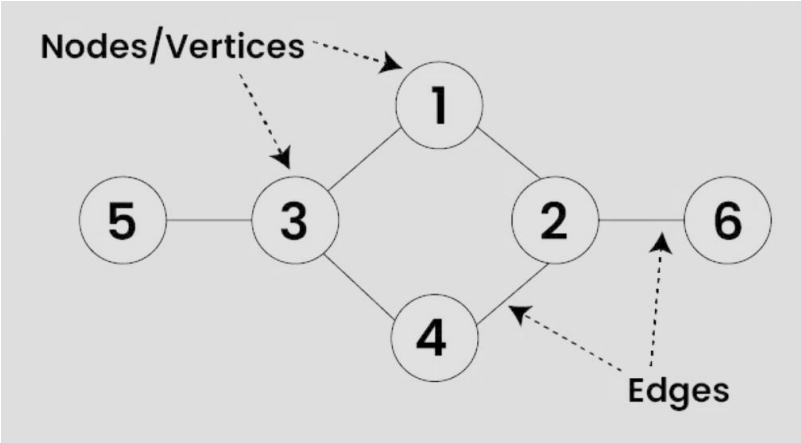
Graph Data Structure – aka Network Data Structure

Graphs are data structures, while **Graph Algorithms** are algos used specifically for Graphs. Graph algorithms operate on graph structures to solve problems like shortest paths, connectivity, traversal, and network flow.

Note: Tree is a type of Graph 🚀

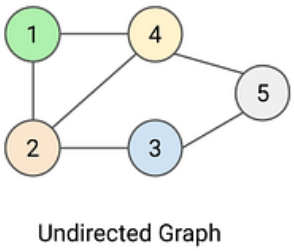
A **graph** consists of:

- **Nodes (Vertices, V)**: Represent entities (e.g., cities, web pages).
- **Edges (E)**: Represent relationships or connections between nodes.
    - **Directed (one-way)** or **Undirected (two-way)**
    - **Weighted (cost assigned)** or **Unweighted**

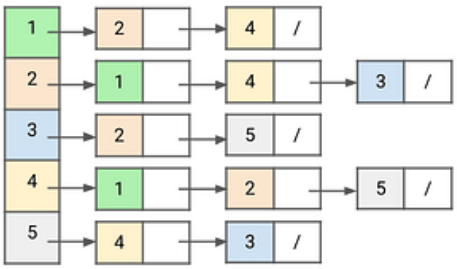


Graphs can be represented as:

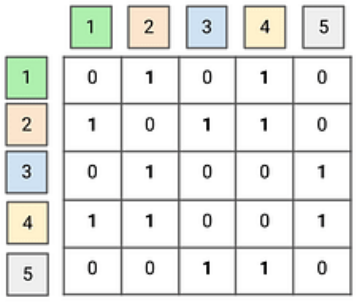
- **Adjacency List** (efficient for sparse graphs)
- **Adjacency Matrix** (efficient for dense graphs)



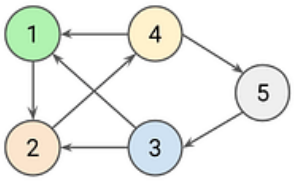
Undirected Graph



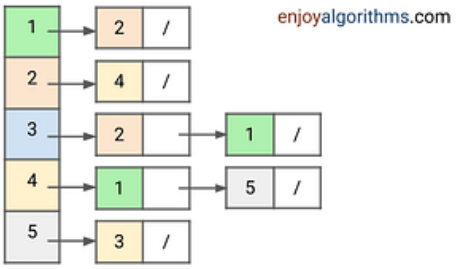
Adjacency List Representation

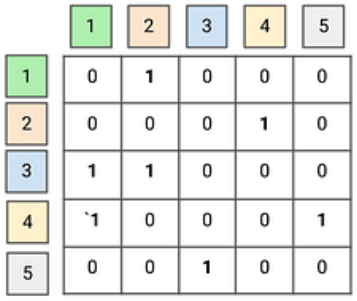


Adjacency Matrix Representation



Directed Graph





Graph Algorithms

(Algorithms on Graphs)

Graph algorithms help **search**, **traverse**, **find shortest paths**, **detect cycles**, etc.

Important Graph Algorithms

Algorithm Type	Algorithms Names	Purpose
Traversal	BFS, DFS	Explore or search graphs
Shortest Path	**Dijkstra's**, Bellman-Ford, Floyd-Warshall	Find the shortest path between nodes
Minimum Spanning Tree (MST)	Kruskal's, Prim's	Find the least-cost connection of all nodes
Cycle Detection	DFS-based cycle check, Union-Find	Detect cycles in directed/undirected graphs
Graph Partitioning	Karger's Algorithm	Divide a graph into smaller components
Flow Algorithms	Ford-Fulkerson	Maximize flow in a network

Popular Graph Algorithms in use:

Yes! Graph algorithms have applications in:

- **Social Networks** (Finding influencers, friend recommendations)
- **Search Engines** (Google's PageRank is a graph algorithm)
- **Recommendation Systems** (Graph-based collaborative filtering)
- **Natural Language Processing** (Dependency parsing in text)

*A (A-Star) Algorithm**

🔹 *A (A-Star) is a graph traversal and pathfinding algorithm** used in AI, robotics, and computer games. It **finds the shortest path** from a start node to a goal node by combining the strengths of **Dijkstra's Algorithm (guarantees shortest path)** and **Greedy Best-First Search (fast heuristic search).**

How A Works

A* uses a **cost function** to decide which node to explore next:

$f(n) = g(n) + h(n)$

Where:

- **g(n)** → Cost from the start node to node nnn (**actual cost**)
- **h(n)** → Estimated cost from node nnn to the goal (**heuristic**)
- **f(n)** → Total estimated cost of the path through node nnn

*A prioritizes nodes with the lowest f(n)f(n)f(n) value.**

Why A is Powerful

✅ **Guaranteed Shortest Path** (if h(n)h(n)h(n) is admissible → never overestimates)
✅ **Efficient** (explores fewer nodes than Dijkstra)
✅ **Customizable** (heuristics can be adjusted for different applications)

Real-World Applications

📌 **Pathfinding in Maps & Games** → Google Maps, GPS, AI in video games
📌 **Robotics** → Robot navigation
📌 **AI & Machine Learning** → Planning and optimization