ALGORITHM IMPLEMENTATION IN REAL-WORLD APPLICATIONS

Submitted By:

Gnanada Vanarpathi - SE24UCAM003

Gnapika Chebrolu - SE24UCAM007

Pratheeksha Sumesh - SE24UCAM040

Sruchitha Gandhi - SE24UCAM041

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to our professor, Dr. Sayani Das, for her invaluable guidance, encouragement, and continuous Support throughout this project. We extend our thanks to all faculty members and classmates for their suggestions. Special thanks to our institution, Mahindra University, for giving us the opportunity to work on this project.

Table of Contents

Introduction: 6

Google Maps — Shortest Path Algorithms: 6

__Literature Survey: 7

__Dijkstra's Algorithm (1956): 7

__Applications: 7

__How it works: 7

__A* (A-Star) Algorithm (1968): 8

__Applications: 8

__How it works: 8

__Example: 9

__Code: 11

Pinterest: 12

About Pinterest - 12

__Objective – 12

_Core Algorithms Used – 12

_Data Structures Involved – 13

_Real-World Functionality - 13

_Pinterest's recommendation system – 13

_1] Graphs & Graph Algorithms: 13

_2] Searching and Sorting: 14

_3] Greedy Algorithms: 14

_Code: (Python) 15

Understanding Git Through Data Structures 16

Introduction 16

__Linked List (Commit Chain) 18

__Tree (File Snapshot) 19

__What is a Hash Table? 20

__DAG (Directed Acyclic Graph) 21

__What is a Pointer? 22

__What is a Merge Conflict? 24

__Code (python): 25

Sotify: 30

Introduction 30

Shuffle algorithm (playback order) 30
The challenge of Randomness 30
Algorithm description 31
Time Complexity: 31
Space Complexity: 31
Why It Is Ideal 31
Spotify's perceived randomness adjustments 31
1. Artist Separation Heuristic 31
2. Genre & Mood Distribution 31
3. Device-aware Adaptation 32
Algorithmic Techniques in the Modified Shuffle 32
Spotify recommendation engine 32
Collaborative filtering (CF) 33
User item interaction matrix 33
Item-Based Collaborative Filtering 33
Matrix Factorization 33
Mathematical Model 33
Complexity 34
Content-based filtering 34
Nearest Neighbour Search 34
NLP-Based Recommendation 35
NLP is used to analyse: 35
Three-Stage Recommendation Pipeline (Detailed) 35
Stage 1 — Candidate Generation 35
Stage 2 — Scoring and Ranking 35
Stage 3 — Diversification 36
End-to-End Complexity 36
Prototype code 37
Fisher-yates shuffle 37
Toy collaborative Filtering 37
Real-World Engineering Constraints 38
High Scale 38
Cold Start 38
User Personalization 38
Data structures used in spotify 38
1.Arrays 38
What is an Array? 39

Why Spotify uses it 39
Where used 39
2. Hash Tables 39
What is a Hash Table? 39
Why Spotify uses it 39
3. Sparse Matrices 40
What is a Sparse Matrix? 40
Why Spotify uses it 40
Used for 41
4. Graphs 41
What is a Graph? 41
Why Spotify uses it 41
Real usage 41
5. Vectors (Embeddings) 42
What is a Vector? 42
Why Spotify uses it 42
6. Priority Queues / Heaps 42
What is a Heap? 42
Why Spotify uses it 42
Used for 43
7. KD-Trees / ANN Structures (FAISS, Annoy, HNSW) 43
What they are 43
Why Spotify uses them 43
8. LRU Cache (Least Recently Used Cache) 43

Introduction:

In today's world, many popular applications rely heavily on algorithms and data structures to work efficiently. To understand how these concepts function in real systems, this project focuses on four major applications: Google Maps, Spotify, Git, and Pinterest. Each of these applications uses different algorithms internally such as shortest path finding, randomization, version control graphs, and recommendation techniques.

In this project, we study how these algorithms are used and create simple prototypes that demonstrate their internal logic.

- **Google Maps** — Shortest Path Algorithms
- **Spotify** — Shuffle and Recommendation Algorithms.
- **Git** — Version Control Using Graphs
- **Pinteres**t — Recommendation and Similarity Search

Google Maps — Shortest Path Algorithms:

In modern navigation systems such as Google Maps, finding the shortest or fastest path between two locations is a crucial operation. Every road, intersection, or route can be represented as a graph, where:

- Nodes (vertices) represent locations or intersections.
- Edges represent roads connecting these locations.
- Weights represent distances, travel times, or road costs.

To find the shortest path, two widely used algorithms are:

1. **Dijkstra's Algorithm**
2. **A* (A-Star) Algorithm**

These algorithms play a vital role in Design and Analysis of Algorithms (DAA) by demonstrating how optimization and heuristic search can solve real-world problems efficiently.

Google Maps internally uses modified forms of A* and Dijkstra's Algorithm (often together) to compute routes while considering real-time traffic, distance, and time constraints.

Literature Survey:

Many algorithms have been proposed over time for the shortest path problem in graphs. The following are the most relevant:

Dijkstra's Algorithm (1956):

Proposed by Edsger W. Dijkstra, this algorithm finds the shortest path from a source node to all other nodes in a weighted graph (non-negative weights).
It works by iteratively selecting the node with the smallest tentative distance and updating its neighbours.
Dijkstra's Algorithm is deterministic and guarantees the shortest path.

Applications:

- Basic routing in networks.
- GPS navigation systems (early versions).
- Minimum cost problems in logistics.

How it works:

Initialization

- Set the distance of the start node to 0.
- Set all other node distances to ∞ (infinity).
- Mark all nodes as unvisited.

Select the Node with Minimum Distance

- Pick the unvisited node that has the smallest known distance (initially the start node).

Update Neighbouring Nodes

- For each unvisited neighbour of the current node:

  Calculate the new distance:

- newDistance = currentDistance + edgeWeight

  If this new distance is smaller than the previously known distance, update it.

Mark Node as Visited

- Once you finish checking a node's neighbours, mark it as visited (its shortest path is now fixed).

Repeat

- Repeat steps 2– 4 until all nodes are visited or the destination is reached.

A* (A-Star) Algorithm (1968):

Developed by Peter Hart, Nils Nilsson, and Bertram Raphael, A* is an extension of Dijkstra's Algorithm that adds a heuristic function h(n) to estimate the cost from the current node to the goal.

It uses the evaluation function:

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

Where:

- g(n): Actual distance from start to current node.
- h(n): Estimated distance from current node to goal.
- f(n): Total estimated cost of the path through node n.

Because of the heuristic, A* can reach the goal faster sby ignoring unlikely paths. Google Maps and other GPS services rely heavily on A* and its optimized variants such as:

- Bidirectional A*
- ALT (A* + Landmarks + Triangle Inequality)
- Contraction Hierarchies

Applications:

- Google Maps, GPS systems.
- Robotics navigation.
- Game pathfinding (e.g., in AI characters).
- Real-time route optimization.

How it works:

Initialization

- Put the start node in the open list (nodes to be visited).
- Set g(start) = 0 and f(start) = h(start).

Choose the Node with Lowest f(n)

- From the open list, pick the node having the lowest total cost (f).

Goal Test

- If the selected node is the goal, stop you found the path.

Explore Neighbours

- For each neighbour of the current node:
  - Calculate tentative g-score:

$$g_{new} = g(current) + distance(current, neighbor)$$

  - Calculate:

$$f_{new} = g_{new} + h(neighbor)$$

  - If neighbour not in open list or new f is smaller, update it.

Repeat

- Move the current node to the closed list (visited).
- Repeat until goal is reached or open list becomes empty.

Example:

Let us consider a small sample map (graph) of cities or intersections represented by letters (A–E):

From	To	Distance
A	B	6
A	D	1
D	B	2
D	E	1
E	C	5
B	C	5

To find the shortest path from A → C

Using Dijkstra's Algorithm:

1. Start from node A.
2. Initialize distances: A=0, others as ∞.
3. Visit neighbours of A:
    - B = 6
    - D = 1
4. Choose node with smallest distance (D = 1).
   From D → B = 2, E = 1 → update:
    - B = 3 (1+2)
    - E = 2 (1+1)
5. Visit E → C = 7 (2+5)
6. Visit B → C = 8 (3+5)
7. Minimum cost to C = 7, via A → D → E → C

Using A* Algorithm:

In A*, we use a heuristic estimate (h) for each node, representing the approximate distance to goal C.

Node	h(n) (estimated)
A	7
B	6
C	0
D	5
E	3

At each step, A* selects the node with the smallest f(n) = g(n) + h(n) value.

Step	Node	g(n)	h(n)	f(n) = g + h	Path
1	A	0	7	7	A
2	D	1	5	6	A-D
3	E	2	3	5	A-D-E
4	C	7	0	7	A-D-E-C

Shortest Path: A → D → E → C
Total Distance: 7

A* reached the goal faster because it prioritized nodes that seemed closer to the goal (based on h(n)).

Code:

```
        g[start] = 0
        f[start] = heuristic(start, goal)
        path = {}
        while open_set:
            _, current = heapq.heappop(open_set)
            if current == goal:
                break
            for neighbor, weight in graph[current].items():
                temp_g = g[current] + weight
                if temp_g < g[neighbor]:
                    g[neighbor] = temp_g
                    f[neighbor] = g[neighbor] + heuristic(neighbor, goal)
                    path[neighbor] = current
                    heapq.heappush(open_set, (f[neighbor], neighbor))
        return g, path
    # RUN BOTH ALGORITHMS
    print("Dijkstra's Algorithm:")
    distances, path = dijkstra(graph, 'A', 'C')
    print("Shortest distance:", distances['C'])
    print("Path:", path)
    print("\nA* Algorithm:")
    distances, path = astar(graph, 'A', 'C')
    print("Shortest distance:", distances['C'])
    print("Path:", path)
```

Output:

```
Dijkstra's Algorithm:
Shortest distance: 7
Path: {'B': 'D', 'D': 'A', 'E': 'D', 'C': 'E'}

A* Algorithm:
Shortest distance: 7
Path: {'B': 'D', 'D': 'A', 'E': 'D', 'C': 'E'}
```

Both Dijkstra's and A* algorithms effectively find the shortest path in weighted graphs.
However:

- Dijkstra's explores all nodes systematically.
- A* intelligently directs its search using a heuristic.

Therefore, A* is faster and more suitable for real-world navigation applications such as Google Maps, which need to compute millions of routes efficiently while considering traffic and real-time data.

Pinterest:

About Pinterest -

Pinterest is a visual discovery and bookmarking platform where users can find, save, and organize ideas in the form of "pins." Each pin represents an image, video, or link that can be saved to themed collections called "boards."

It helps users explore topics like fashion, food, travel, art, or home décor by recommending visually and contextually similar content. When a user saves or interacts with a pin, Pinterest uses algorithms to find other related pins and show them in the user's feed.

Behind the scenes, it uses advanced **recommendation algorithms** to suggest new pins (images, videos, or products) based on user interests, past activity, and pin similarity. The recommendation engine is built on top of **graph data structures**, **searching and sorting algorithms**, and a **greedy ranking strategy** to select the best content for each user.

Objective –

To explore how Pinterest uses algorithms and data structures to recommend, organize, and search for pins (images and ideas), and to demonstrate a simplified prototype of its recommendation system.

Core Algorithms Used –

- Graph Traversal Algorithms:
  Used to explore relationships between users, pins, and boards.
  (Example: Breadth-First Search (BFS) or Depth-First Search (DFS))
- Similarity Search Algorithms:
  Compare pins based on tags, descriptions, or visual features.
  (Example: cosine similarity or Jaccard index for textual tags)

- Ranking Algorithms:
  Sort recommended pins based on relevance, popularity, and user activity. (Implemented using heaps or priority queues)

Data Structures Involved –

- **Graphs:** Represent relationships between users, boards, and pins. (Nodes = users/pins/boards; Edges = relationships such as "saved", "followed")
- **Hash Tables:** Store pin metadata (title, tags, category) for fast lookups.
- **Priority Queues / Heaps:** Rank and retrieve top recommended pins efficiently.

Real-World Functionality -

Pinterest connects millions of pins using graph-based relationships and machine learning. When a user views or saves a pin, Pinterest uses graph connections and similarity measures to find other pins that are visually or contextually related. This creates a **personalized discovery experience** where the feed continuously improves based on user interaction.

Pinterest's recommendation system –

Let's focus on the topics Searching and Sorting, Graphs & Graph Algorithms and Greedy Algorithms

1] Graphs & Graph Algorithms:

Pinterest's entire platform can be represented as a graph:

- Nodes (Vertices): Users, Boards, and Pins
- Edges: Relationships such as *"user saves pin"* or *"pin belongs to board."*

When a user views a pin, Pinterest explores this graph to find other pins that are closely connected — for example, pins saved to the same board or by similar users.

The simplest way to explore these relationships is by using an algorithm similar to Breadth-First Search (BFS) algorithm.
This helps find all pins “close” to the one being viewed.

2] Searching and Sorting:

Once Pinterest finds all related pins, it needs to rank them — showing the most relevant ones first.

To do this, it:

1. Computes a similarity score between the current pin and each related pin. The similarity is calculated based on several factors, such as shared tags, categories, and even visual similarity detected using image recognition.
2. Once each pin has a similarity score, Pinterest **sorts** all the related pins in decreasing order of their scores. Pins that are more relevant — for example, sharing more common tags or being saved by similar users — move to the top of the list.
3. This searching and sorting step ensures that when a user views a pin, the recommendations they see are not random, but the most contextually and visually related ones.
4. Displays the top-k results.

Sorting algorithms like Merge Sort or Heap Sort are ideal for this.

3] Greedy Algorithms:

After sorting, Pinterest needs to quickly decide **which few pins** to actually show to the user (since it can’t display hundreds of results at once).

For this, Pinterest uses a **greedy approach**: it simply takes the top few pins from the sorted list — those with the highest similarity scores — and recommends them.

This method is efficient because it doesn’t need to check every possible combination of pins; it just picks the best ones immediately based on the ranking order. This approach reflects the **greedy algorithm technique**, where the system

makes the locally optimal choice (highest relevance) at each step to provide immediate and effective recommendations

Code: (Python)

```python
# Pins and their associated tags
pins = {
    'Pin1': {"travel", "beach", "sunset"},
    'Pin2': {"travel", "mountains", "adventure"},
    'Pin3': {"food", "dessert", "cake"},
    'Pin4': {"beach", "vacation", "ocean"},
    'Pin5': {"travel", "beach", "adventure"}
}

def jaccard_similarity(tags1, tags2):
    return len(tags1 & tags2) / len(tags1 | tags2)

def recommend(base_pin, pins):
    base_tags = pins[base_pin]
    scores = []
    for pin, tags in pins.items():
        if pin != base_pin:
            score = jaccard_similarity(base_tags, tags)
            scores.append((pin, score))
    scores.sort(key=lambda x: x[1], reverse=True)
    return scores[:3]

print("Top recommendations for Pin1:", recommend('Pin1', pins))
```

Output:

```
Top recommendations for Pin1: [('Pin5', 0.6), ('Pin4', 0.5), ('Pin2', 0.25)]
```

Explanation:

For a user viewing *Pin1*, the algorithm finds the most similar pins by comparing their tags.

- *Pin5* shares “travel” and “beach,” so it has the highest score.
- *Pin4* also shares “beach,” so it’s second.
- *Pin2* is third, as it shares only one keyword.

This is exactly how Pinterest would rank and recommend visually or thematically similar pins.

Conclusion:

In Pinterest, algorithms like **graph traversal**, **searching and sorting**, and **greedy selection** work together to build a personalized and fast recommendation system.

- Graph algorithms help explore relationships between users, boards, and pins.
- Searching and sorting ensure that only the most relevant results are considered.
- Greedy selection provides quick and effective top suggestions.

Through this, Pinterest demonstrates how theoretical computer science concepts are directly applied in a large-scale, real-world application to enhance user experience.

Understanding Git Through Data Structures

Introduction

What is Version Control?

Main requirements while writing code are:

- Save every version you've ever made
- Go back to any previous version instantly
- Work on multiple experiments simultaneously
- Collaborate with others without overwriting each other's work

Version control is a system that does exactly this for code (and any files). It tracks every change, who made it, and when.

What is Git?

Git is the world's most popular version control system, created by Linus Torvalds (creator of Linux) in 2005. It's used by millions of developers to track changes in their code. Deep down it actually uses data structures like trees, DAG's, et to satisfy its requirements.

This project discusses data structures that Git uses, combined cleverly to create a powerful system.

Git's Core Requirements

Before we look at the data structures, let's understand what Git needs to accomplish:

#	Requirement
R1	Store every version permanently
R2	Identify each version uniquely
R3	Support parallel work (branching)
R4	Merge changes from different people
R5	Be fast and efficient

PART 3: Data Structures in Git

Now that we understand the requirements, we can discuss how data structures are used to make viable solutions:

Data Structure	Git Usage	Requirement Satisfied
Hash Table	Object storage with O(1) lookup time complexity	R2 (Unique IDs), R5 (Speed)
Linked List	Commit chain (each points to parent)	R1 (Store versions)
Tree	File snapshot (directory structure)	R1 (Store versions)
DAG	Branching history (multiple parents)	R3 (Branching)
Pointers	Branches and HEAD	R3 (Branching)
Arrays	Three-way merge comparison	R4 (Merging)

Storing Every version Permanently

Git needs to save a complete snapshot of your project every time you commit, and connect these snapshots in order. It uses linked lists and trees to carry it out.

Linked List (Commit Chain)

What is a Linked List?

Properties:

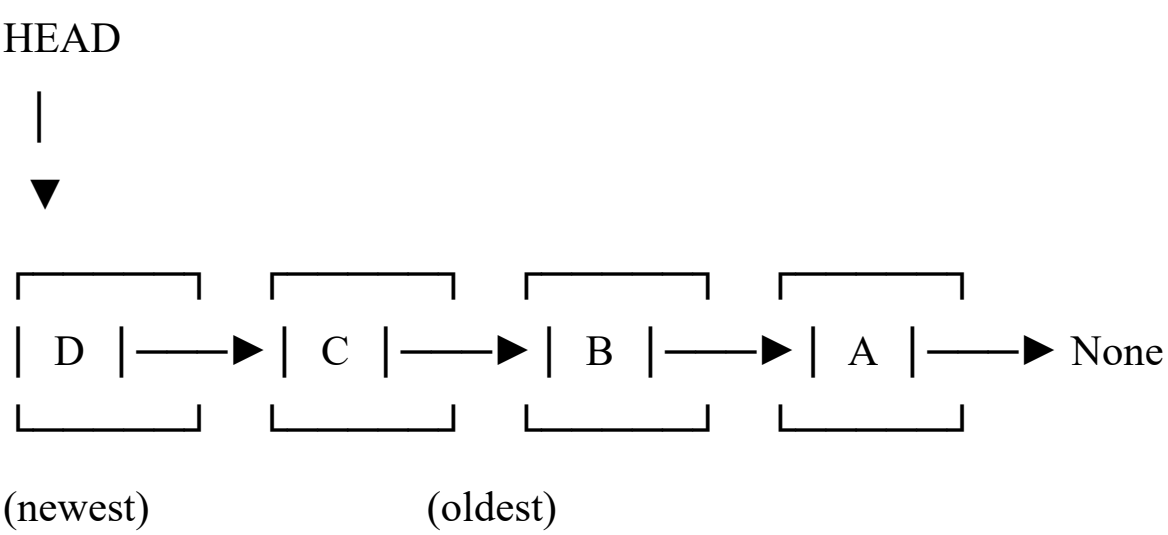
- Sequence of nodes connected by pointers
- Data + Pointer to next node
- Follow pointers from one node to next
- Easy insertion, dynamic size

Used to:

- Each commit points to its parent commit
- Forms a chain from newest to oldest
- git log traverses this chain

- Each node commit stores a message, parent pointer, snapshot of files

Visualization



Tree (File Snapshot)

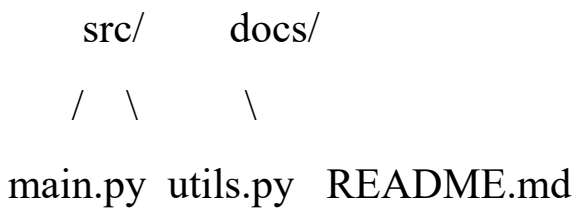
What is a Tree?

- Hierarchical (parent-child relationships)
- Single topmost node(parent)
- Nodes with no children are possible
- Represents nested/hierarchical data naturally

How Git Uses It

- Each commit points to a tree (snapshot of all files)
- Trees store: {filename → content_hash}, needs lesser time to access with a hash index.
- Represents directory structure at that moment

Visualization



Therefore every commit stores a complete snapshot (Tree) and links to previous versions (Linked List).

Unique identification of each version and efficiency

The Challenge

- Git needs to give every version a unique ID, find any object instantly (even with millions of files) and avoid storing duplicate content

Solution:

Hash Table

What is a Hash Table?

- Key-value pairs
- Computed from content using hash function
- O(1) - constant time (instant!)
- Fastest possible lookup
- Input can be any data type while output is a fixed size and unique.

How Git Uses It

- Every file's content gets a unique hash (SHA-256)
- Same content = same hash = stored only once (no duplication)
- Finding any object is O(1)

Visualization

Content: "Hello World" ──hash──► Key: "a1b2c3"

Support Parallel Work (Branching)

Multiple developers need to work on different features simultaneously their changes should not affect each other until ready and they should be fast to create.

DAG (Directed Acyclic Graph)

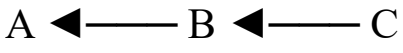
- Edges have direction (arrows)
- No cycles (can't loop back to same node)
- Nodes can have MULTIPLE parents
- Represents branching and merging naturally

How Git Uses It

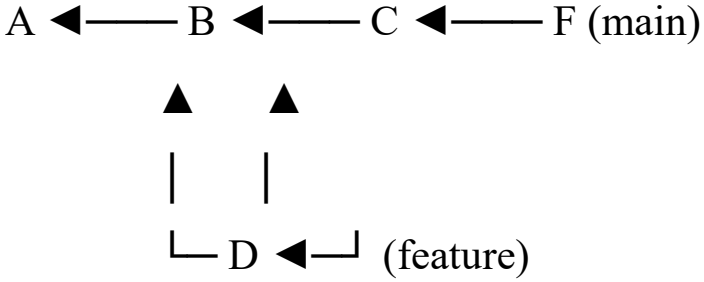
- Normal commit = 1 parent (like linked list)
- Merge commit = 2 parents (this is what makes it a DAG!)
- Allows branches to split and rejoin

Visualization

Simple Linked List (1 parent each):



DAG with Branch + Merge (F has 2 parents):



F has two parents: C and D

Pointers (Branches and HEAD)

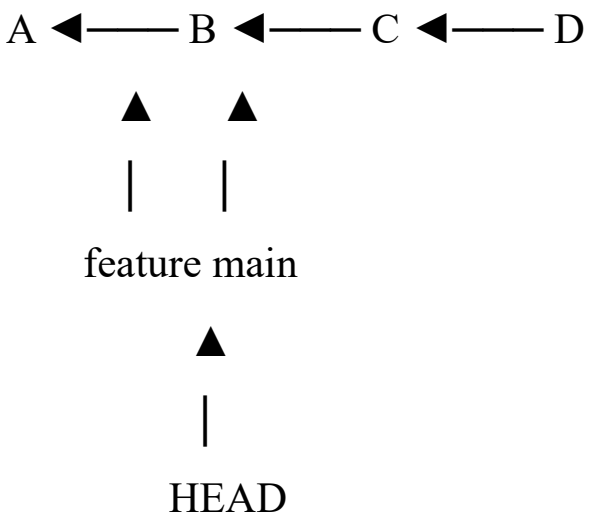
What is a Pointer?

- A reference to another location/object
- Address/reference, NOT the actual data
- Lightweight, changing pointer doesn't copy data

How Git Uses It

- **Branches are just pointers to commits.**
- Creating a branch = creating a new pointer (
- HEAD = special pointer to current branch
- No data is copied when branching

Visualization



(No files copied, it is an instant operation)

Therefore, branches split and merge (DAG) and are lightweight (Pointers).

Merging Changes

The Challenge

Two people changed the same file differently so Git needs to combine their changes automatically and must detect when it cannot merge automatically (conflicts)

Arrays (Three-Way Merge)

What is an Array?

- Ordered collection of elements
- By index: arr[0], arr[1], arr[2]...
- Easy comparison by index

The Three Arrays in Merge

Array	Contains
BASE[]	Original file (common ancestor)
OURS[]	Our version of the file
THEIRS[]	Their version of the file

Comparison Works (Index by Index)

For each index i:

Compare base[i], ours[i], theirs[i]

Condition	Result
ours[i] == theirs[i]	✓ No conflict, use either
ours[i] == base[i]	✓ Only they changed, use theirs
theirs[i] == base[i]	✓ Only we changed, use ours
All three different	✗ user must decide

What is a Merge Conflict?

A conflict occurs when both versions changed the **same line differently**. Git cannot automatically decide which version to keep.

Merge Conflict Example

BASE: "Hello World"

OURS: "Hello India" (We changed it)

THEIRS: "Hello America" (They changed it)

This creates a conflict.

Human must choose which to keep (or keep both).

Visualization: Array Index-by-Index Comparison

	BASE[]	OURS[]	THEIRS[]	RESULT
[0]	"Hello"	"Hello"	"Hello"	Same in all
[1]	"World"	"India"	"World"	Use ours (only we changed)
[2]	"Goodbye"	"Goodbye"	"Bye"	Use theirs (only they changed)
[3]	"End"	"Finish"	"Done"	Conflict

Therefore changes merged automatically when possible, conflicts detected when not.

Mini-Git Implementation

The Python code below demonstrates all data structures working together.

Code (python):

```python
# MINI-GIT: Understanding Git Through Data Structures

import hashlib                      # Library for SHA-256 hashing
from datetime import datetime       # Library for timestamps



# PART 1: HASH TABLE

objects = {}                        # Empty dictionary = our hash table

def hash_content(content):
    encoded = content.encode()      # Convert string to bytes
    hash_obj = hashlib.sha256(encoded)  # Create SHA-256 hash object
    full_hash = hash_obj.hexdigest()    # Get hash as hex string (64 chars)
    return full_hash[:7]            # Return first 7 characters only

def store_object(content):
    key = hash_content(content)     # Generate hash key from content
    objects[key] = content          # Store in hash table: key -> content
    return key                      # Return the key

def get_object(hash_id):
    return objects.get(hash_id, None)   # Lookup by key, return None if not found



# PART 2: TREE (File Snapshot)
class Tree:
    def __init__(self):
        self.files = {}             # Empty dict to store {filename: hash}

    def add_file(self, filename, content):
        content_hash = store_object(content)    # Store content, get hash
        self.files[filename] = content_hash     # Map filename to hash

    def get_hash(self):
        items = sorted(self.files.items())  # Sort files alphabetically
        tree_str = str(items)               # Convert to string
        return hash_content(tree_str)       # Hash the whole tree
# PART 3: COMMIT (Linked List / DAG Node)
class Commit:
    def __init__(self, message, tree_hash, parent=None, parent2=None):
        self.message = message      # Commit message
        self.tree_hash = tree_hash  # Hash of file snapshot
        self.parent = parent        # Pointer to parent commit (linked list)
        self.parent2 = parent2      # Second parent (for merge = DAG)
        self.timestamp = datetime.now().strftime("%H:%M:%S")  # Current time

        # Create unique hash for this commit
        commit_data = f"{message}{tree_hash}{parent}{self.timestamp}"
        self.hash = hash_content(commit_data)
# PART 4: REPOSITORY
```

```python
# PART 4: REPOSITORY
class MiniGit:
    def __init__(self):
        self.commits = {}               # Hash table to store all commits
        self.branches = {"main": None}  # Branch pointers {name: commit_hash}
        self.head = "main"              # Current branch name
        self.staging = Tree()           # Staging area for files
        print("Initialized MiniGit repository\n")

    def add(self, filename, content):
        self.staging.add_file(filename, content)  # Add file to staging
        print(f"Staged: {filename}")

    def commit(self, message):
        parent = self.branches[self.head]   # Get current commit as parent
        tree_hash = self.staging.get_hash() # Get hash of staged files

        new_commit = Commit(message, tree_hash, parent)  # Create commit
        self.commits[new_commit.hash] = new_commit       # Store in hash table
        self.branches[self.head] = new_commit.hash       # Update branch pointer

        print(f"Committed: {new_commit.hash} - {message}")
        return new_commit.hash

    def log(self):
        print("\n--- Commit Log (Linked List Traversal) ---")
        current = self.branches[self.head]  # Start at current branch

        while current:                      # Loop until None
            c = self.commits[current]       # Get commit object
            print(f"  {c.hash} | {c.message} | Parent: {c.parent}")
            current = c.parent              # Move to parent (traversal)
        print()

    def branch(self, name):
        current_commit = self.branches[self.head]   # Get current commit
        self.branches[name] = current_commit        # New branch points to same commit
        print(f"Branch '{name}' created -> {current_commit}")

    def checkout(self, name):
        self.head = name                    # Change current branch
        print(f"Switched to '{name}'")

    def merge(self, branch_name):
        our_hash = self.branches[self.head]     # Our current commit
        their_hash = self.branches[branch_name] # Their branch commit

        # Create merge commit with TWO parents (this makes it a DAG!)
        merge_commit = Commit(
            f"Merge {branch_name} into {self.head}",  # Message
            self.staging.get_hash(),                  # Tree hash
            our_hash,                                 # Parent 1
            their_hash                                # Parent 2
        )

        self.commits[merge_commit.hash] = merge_commit  # Store commit
        self.branches[self.head] = merge_commit.hash    # Update branch
        print(f"Merged! Commit {merge_commit.hash} has 2 parents")

    def visualize(self):
        print("\n--- Branch Pointers ---")
        for name, hash in self.branches.items():    # Loop through branches
            marker = " <-- HEAD" if name == self.head else ""
            print(f"  {name}: {hash}{marker}")
```

```
PART 5: THREE-WAY MERGE (Arrays)
f three_way_merge():
  print("\n--- Three-Way Merge Demo (Arrays) ---\n")

  # Three versions of same file as arrays
  base   = ["Hello World",   "Line 2", "Goodbye"]     # Original
  ours   = ["Hello India",   "Line 2", "Goodbye"]     # Our changes
  theirs = ["Hello America", "Line 2", "See you"]     # Their changes

  print("BASE:  ", base)
  print("OURS:  ", ours)
  print("THEIRS:", theirs)
  print()

  # Compare each index
  for i in range(len(base)):
      if ours[i] == theirs[i]:                # Both same
          print(f"[{i}] Same in both -> '{ours[i]}'")
      elif ours[i] == base[i]:                # Only they changed
          print(f"[{i}] Only they changed -> '{theirs[i]}'")
      elif theirs[i] == base[i]:              # Only we changed
          print(f"[{i}] Only we changed -> '{ours[i]}'")
      else:                                   # Both changed differently
          print(f"[{i}] CONFLICT! ours='{ours[i]}' vs theirs='{theirs[i]}'")

PART 6: RUN DEMO
 __name__ == "__main__":

  # --- DEMO 1: Hash Table ---
  print("=" * 50)
  print("DEMO 1: HASH TABLE")
  print("=" * 50)
  h1 = store_object("Hello World")        # Store content
  h2 = store_object("Hello World")        # Same content = same hash
  h3 = store_object("Different")          # Different content = different hash
  print(f"'Hello World' -> {h1}")
  print(f"'Hello World' -> {h2}  (same hash!)")
  print(f"'Different'   -> {h3}")
  print()

  # --- DEMO 2: Commits (Linked List) ---
  print("=" * 50)
  print("DEMO 2: COMMITS (Linked List)")
  print("=" * 50)
  repo = MiniGit()                        # Create repository
  repo.add("file.txt", "Hello")           # Stage file
  repo.commit("First commit")             # Create first commit
  repo.add("file.txt", "Hello World")     # Stage updated file
  repo.commit("Second commit")            # Create second commit
  repo.log()                              # Show history

  # --- DEMO 3: Branching (Pointers) ---
  print("=" * 50)
  print("DEMO 3: BRANCHING (Pointers)")
  print("=" * 50)
  repo.branch("feature")                  # Create new branch
  repo.checkout("feature")                # Switch to it
  repo.add("new.txt", "Feature work")     # Add new file
  repo.commit("Feature commit")           # Commit on feature branch
  repo.visualize()                        # Show branch pointers
  print()
```

```
# --- DEMO 4: Merge (DAG) ---
print("=" * 50)
print("DEMO 4: MERGE (DAG - Two Parents)")
print("=" * 50)
repo.checkout("main")                    # Switch back to main
repo.merge("feature")                    # Merge feature into main
repo.visualize()                         # Show updated pointers

# --- DEMO 5: Three-Way Merge (Arrays) ---
print()
print("=" * 50)
print("DEMO 5: THREE-WAY MERGE (Arrays)")
print("=" * 50)
three_way_merge()                        # Run merge demo

# --- Summary ---
print()
print("=" * 50)
print("SUMMARY")
print("=" * 50)
print("""
Hash Table   -> Store objects, O(1) lookup
Linked List  -> Commits point to parent
Tree         -> Snapshot of files
DAG          -> Merge commits have 2 parents
Pointers     -> Branches point to commits
Arrays       -> Compare versions index by index
""")
```

Output:

```
==================================================
DEMO 1: HASH TABLE
==================================================
'Hello World' -> a591a6d
'Hello World' -> a591a6d  (same hash!)
'Different'   -> e9cdbd4

==================================================
DEMO 2: COMMITS (Linked List)
==================================================
Initialized MiniGit repository

Staged: file.txt
Committed: 290532c - First commit
Staged: file.txt
Committed: 10a00a8 - Second commit

--- Commit Log (Linked List Traversal) ---
  10a00a8 | Second commit | Parent: 290532c
  290532c | First commit | Parent: None

==================================================
DEMO 3: BRANCHING (Pointers)
==================================================
Branch 'feature' created -> 10a00a8
Switched to 'feature'
Staged: new.txt
Committed: b4bd6ed - Feature commit

--- Branch Pointers ---
  main: 10a00a8
  feature: b4bd6ed <-- HEAD

==================================================
DEMO 4: MERGE (DAG - Two Parents)
==================================================
Switched to 'main'
Merged! Commit b7da0ea has 2 parents

--- Branch Pointers ---
  main: b7da0ea <-- HEAD
  feature: b4bd6ed

==================================================
DEMO 5: THREE-WAY MERGE (Arrays)
==================================================

--- Three-Way Merge Demo (Arrays) ---

BASE:   ['Hello World', 'Line 2', 'Goodbye']
OURS:   ['Hello India', 'Line 2', 'Goodbye']
THEIRS: ['Hello America', 'Line 2', 'See you']

[0] CONFLICT! ours='Hello India' vs theirs='Hello Am
[1] Same in both -> 'Line 2'
[2] Only they changed -> 'See you'

==================================================
SUMMARY
==================================================

    Hash Table   -> Store objects, O(1) lookup
    Linked List  -> Commits point to parent
    Tree         -> Snapshot of files
    DAG          -> Merge commits have 2 parents
    Pointers     -> Branches point to commits
    Arrays       -> Compare versions index by index
```

Spotify:

Introduction

Spotify is the world's largest music-streaming platform, serving personalized recommendations, curated playlists, and adaptive playback to hundreds of millions of users. Behind this experience lies a set of sophisticated algorithms designed to handle large-scale data, model user behaviour, and deliver instantaneous results with high accuracy.
Here we try to analyse the following:

1. **Shuffle Playback** – generating an unbiased yet "human-pleasing" random play order.
2. **Recommendation Engine** – predicting what a user may enjoy, using collaborative filtering, content analysis, and machine-learning ranking systems.

Shuffle algorithm (playback order)

The challenge of Randomness

Although randomness seems simple, humans perceive randomness differently from actual mathematical randomness.
A purely random shuffle may produce:

- songs from the same artist consecutively,
- genre clusters,
- repeated patterns.

Users mistake these *expected artifacts* of randomness for "bad shuffle".
Therefore, Spotify needed an algorithm that is:

- **statistically fair**,
- **visually balanced**,
- **computationally efficient**,
- **scalable to large playlists**.

Fisher-yates shuffle

Algorithm description

Given an array A[0…n−1], the Fisher–Yates shuffle generates a uniform random permutation:

1. For **i = n−1** down to **1**:
2. Generate a random **index j between 0 and i**
3. **Swap A[i] and A[j]**

Time Complexity:

- **O(n)** — exactly one swap per element.

Space Complexity:

- **O(1)** — in-place algorithm.

Why It Is Ideal

- Produces each permutation with equal probability (1/n!).
- No statistical bias.
- Works efficiently even for very large playlists.

Spotify's perceived randomness adjustments

Spotify found that users frequently complained that shuffle "was not random", even when Fisher–Yates was used.
Thus, they introduced "perceived randomness", which includes:

1. Artist Separation Heuristic

Ensure that the same artist does not appear too close together.

Algorithmically:

- After Fisher–Yates, scan through the playlist.
- If the same artist appears within a fixed window (e.g., distance < k), swap it with another random position.

2. Genre & Mood Distribution

Spotify groups tracks by audio features:

- energy,
- valence (happiness),
- tempo,

- loudness,
- danceability.

After shuffle, Spotify ensures:

- mood transitions are smooth,
- no harsh jumps (e.g., calm → heavy metal abruptly).

3. Device-aware Adaptation

Shuffle behaviour may vary depending on:

- user listening history,
- playlist length,
- device type (mobile/desktop),
- whether smart shuffle is enabled.

Algorithmic Techniques in the Modified Shuffle

Spotify's final shuffle process combines:

- Fisher–Yates (base random permutation),
- local constraint optimization (artist & genre spread),
- random re-sampling for violations,
- greedy swaps or simulated annealing-like adjustments for balanced spacing.

This transforms shuffle into a **constraint satisfaction problem (CSP)** solved via:

- local search,
- greedy corrections,
- lightweight heuristics.

Spotify recommendation engine

Spotify's recommendation system is one of the most advanced in the world. It integrates **collaborative filtering**, **audio content analysis**, **NLP**, and **machine-learning ranking models**.

It follows a **three-stage architecture**:

1. Candidate Generation

2. Scoring / Ranking
3. Diversification & Playlist Construction

Collaborative filtering (CF)

Collaborative Filtering predicts user interests by analysing behavioural patterns.

User item interaction matrix

Spotify stores listening data in a massive sparse matrix:

Row = users

Columns = songs

Values = implicit feedback (play count, likes, skips, add to playlist)

This matrix is extremely sparse (less than 1% filled).

Item-Based Collaborative Filtering

Similarity between two tracks is computed using:

- **Cosine similarity**
- **Jaccard similarity**
- **Count-based co-occurrence**

A simplified similarity formula:

$$\mathbf{sim(i, j)} = \frac{|\mathbf{U_i} \cap \mathbf{U_j}|}{\sqrt{|\mathbf{U_i}| \cdot |\mathbf{U_j}|}}$$

where U_i is the set of users who played song i.

Matrix Factorization

To handle millions of users and tracks, Spotify factorizes the user–item matrix into low-dimensional embeddings.

Mathematical Model

$$R \approx P \cdot Q^T$$

- P = user latent matrix
- Q = song latent matrix

Optimized via:

- Alternating Least Squares (ALS)
- Stochastic Gradient Descent (SGD)

Complexity

- ALS: $O(n \cdot k^2 + m \cdot k^2)$
- SGD: O(#interactions)

These reduced embeddings power many Spotify features:

- Discover Weekly
- Daily Mix
- Artist Radio

Content-based filtering

Spotify analyses the audio files directly using convolutional neural networks (CNNs) in their pipeline called **"Audio Analysis"**.

Extracted features include:

- MFCCs (Mel-Frequency Cepstral Coefficients)
- Beat detection
- Chroma features
- Spectral contrast
- Instrumentation

These features create a **128–2048 dimensional embedding vector** for each track.

Nearest Neighbour Search

To find similar songs quickly, Spotify uses:

- KD-Trees (small datasets)
- Approximate Nearest Neighbour (ANN) solutions like:
    - FAISS (Facebook AI)
    - Annoy
    - HNSW graphs

NLP-Based Recommendation

Spotify treats playlists, descriptions, and lyrics as natural-language data.

NLP is used to analyse:

- Playlist names (“lofi chill mix”, “study beats”, etc.)
- Lyrics using token vectors
- Blog posts about music
- Artist biographies
- User-generated playlist titles

Models used:

- Word2Vec
- BERT-style embeddings
- TF-IDF for fast textual similarity

This helps Spotify cluster songs with similar themes or moods even if listening patterns are sparse.

Three-Stage Recommendation Pipeline (Detailed)

Stage 1 — Candidate Generation

Generate thousands of candidate tracks using:

- User-based CF
- Item-based CF
- Audio-similarity search
- NLP similarity
- Graph-based traversal (playlist graph)
- Popular trending songs
- Followed artists’ new releases

Stage 2 — Scoring and Ranking

Each candidate song receives multiple scores:

- *Collaborative filtering score*

- *Audio similarity score*
- *User-history alignment score*
- *Freshness / novelty score*
- *Skip penalty score*

Spotify uses a **learning-to-rank model**, typically:

- Gradient Boosted Decision Trees (GBDT)
- Neural Ranking Models
- Logistic Regression
- Multi-task learning (predicting like, skip, playlist add)

Stage 3 — Diversification

Even a high-ranked list might be:

- too repetitive,
- too mainstream,
- too niche,
- too homogenous.

Therefore, Spotify applies constraints:

- No repeating artists within a fixed window
- Genre diversity
- Tempo variety
- Mood transitions

Algorithms used:

- Maximal Marginal Relevance (MMR)
- Determinantal Point Processes (DPPs)
- Greedy diversification based on dissimilarity

End-to-End Complexity

- Candidate generation: O(log n) to O(n) depending on ANN structure

- Ranking with ML model: O(k log k)
- Diversification: $O(k^2)$ worst case
- Overall: optimized to run in **milliseconds**

Prototype code

Fisher-yates shuffle

```
1. import random
2. def fisher_yates(arr):
3.     for i in range(len(arr)-1, 0, -1):
4.         j = random.randint(0, i)
5.         arr[i], arr[j] = arr[j], arr[i]
6.     return arr
7.
```

Toy collaborative Filtering

```
 1. from math import sqrt
 2. plays = {
 3.     'u1': {'s1','s2','s3'},
 4.     'u2': {'s2','s3','s4'},
 5.     'u3': {'s1','s4','s5'}
 6. }
 7.
 8. def cosine(a, b):
 9.     inter = len(a & b)
10.     return inter / (sqrt(len(a)) * sqrt(len(b)))
11.
12. def recommend(user):
13.     seen = plays[user]
14.     scores = {}
15.     all_songs = {s for ss in plays.values() for s in ss}
16.
17.     for song in all_songs:
18.         if song in seen: continue
19.         score = 0
20.         for u in plays:
21.             if song in plays[u]:
22.                 score += cosine(plays[user], plays[u])
23.         scores[song] = score
24.
```

```
25.     return sorted(scores.items(), key=lambda x: x[1], reverse=True)
26.
```

Real-World Engineering Constraints

High Scale

- 500M+ users
- 100M+ tracks
- billions of interactions per day

To handle this, Spotify uses:

- distributed computing
- vector databases
- GPU-accelerated audio analysis
- offline precomputations
- multi-level caching

Cold Start

Handled by:

- content-based features (audio)
- metadata
- “similar artists” graph

User Personalization

Models adapt in real-time:

- session-based behaviour,
- skip patterns,
- time-of-day listening,
- device type.

Data structures used in spotify

1.Arrays

Used in: **Shuffle Algorithm (Fisher–Yates)**

What is an Array?

- A continuous block of memory.
- Each element is accessed using an index.
- Example: playlist songs stored as A[0], A[1], A[2], …

Why Spotify uses it

The shuffle algorithm must:

- Pick random indices
- Swap values quickly
- Traverse the playlist only once

Arrays allow:

- O(1) access
- Very fast swapping
- Simple, predictable memory use

Where used

- Fisher–Yates shuffle
- Reordering playlist songs
- Applying artist-spacing adjustments

2. Hash Tables

Used in:

- **Artist metadata lookup**
- **Song features lookup**
- **Fast retrieval of lyrics, tags, genres**

What is a Hash Table?

A key–value structure where:

- Keys (song IDs) map to values (metadata)
- Lookup takes **O(1)** time

Why Spotify uses it

Spotify stores massive metadata:

- song → features
- artist → profile
- track → audio vectors
- playlist → attributes

Hash tables make this ultra-fast:

- retrieving metadata
- checking artist repetition
- fetching audio features for recommendation

3. Sparse Matrices

Used in: **Collaborative Filtering (CF)**

What is a Sparse Matrix?

A matrix with MOST entries = 0

Example:

Users × Songs matrix:

- 1 = user played song

- 0 = no interaction

Since each user plays very few songs compared to 100M tracks:

- Most cells are empty → sparse.

Why Spotify uses it

For CF, Spotify needs to store:

- play counts
- likes
- skips
- add-to-playlist actions

A dense matrix would waste massive memory.
A sparse matrix stores **only non-zero entries**, saving space and speeding up algorithms.

Used for

- Matrix Factorization
- User–user similarity
- Item–item similarity

4. Graphs

Used in:

- Playlist relationships
- User similarity graphs
- Track co-occurrence networks
- Artist–genre networks

What is a Graph?

Nodes + Edges

Examples:

- node = track
- edge = “these tracks appear together in many playlists”

Why Spotify uses it

Graph structures allow:

- finding related tracks
- learning communities/clusters
- performing random walks (like Pinterest)
- discovering new music via connections

Real usage

- Playlist-to-playlist similarity
- Artist recommendation
- “Fans also like” feature

5. Vectors (Embeddings)

Used in: **Content-based & Audio-based recommendations**

What is a Vector?

A list of numbers representing features of a song.

For example, a song might be:

[tempo, energy, danceability, valence, loudness, …]

Deep learning models convert audio into **128–2048 dimensional vectors**.

Why Spotify uses it

Vectors help compare songs using distance metrics:

- Cosine similarity
- Euclidean distance

This allows Spotify to:

- Match songs with similar beats
- Find songs with similar mood
- Generate "Discover Weekly" embeddings

6. Priority Queues / Heaps

Used in: **Top-k Recommendation Ranking**

What is a Heap?

A tree-like structure that always gives:

- The smallest element (min-heap) or
- The largest element (max-heap)

Why Spotify uses it

When ranking thousands of candidate songs:

- Spotify doesn't need ALL scores
- It only needs the **Top k (like top 50)**

Priority Queues give:

- O(log n) insertion
- O(1) extraction of top item

Used for

- Selecting top recommendations
- Re-ranking during diversity checks
- Sorting shuffle constraints

7. KD-Trees / ANN Structures (FAISS, Annoy, HNSW)

Used in: **Finding similar songs quickly**

What they are

Advanced data structures for nearest-neighbor search in high-dimensional spaces.

Why Spotify uses them

When a playlist is created, Spotify needs to search:

- "Which 100 songs are closest to this one in vector space?"

ANN search reduces retrieval time from **seconds to milliseconds** — essential for real-time music recommendations.

8. LRU Cache (Least Recently Used Cache)

Used in: **Speeding up frequently-used results**

Spotify caches:

- user's top artists
- last played tracks
- recommendations based on daily habits
- local playlist metadata

LRU ensures that:

- most frequently used data stays in memory
- least used data is removed

This keeps the app fast.

Conclusion:

Spotify's shuffle and recommendation systems combine classical algorithm design with modern machine learning and large-scale engineering. Fisher–Yates provides unbiased randomization, improved through heuristic constraints to match user expectations. The recommendation engine uses a hybrid of collaborative filtering, content-based modelling, NLP, and advanced ranking strategies.
Together, these components form a robust, scalable system that delivers highly personalized music experiences.

Resources:

- **Graph Convolutional Neural Networks for Web-Scale Recommender Systems** (KDD 2018 / PinSage paper) (kdd.org)
- **Related Pins at Pinterest: The Evolution of a Real-World Recommender System** (WWW 2017 companion) (archives.iw3c2.org)
- **Pixie: A System for Recommending 3+ Billion Items to 200+ Million Users in Real-Time** (arXiv) (arXiv)
- **Applying Deep Learning to Related Pins** (Pinterest Engineering Blog via Medium) (Medium)
- **Introducing Pixie, an Advanced Graph-Based Recommendation System** (Pinterest Engineering Blog via Medium) (Medium)

Conclusion:

This project successfully demonstrates how core algorithms and data structures used in popular real-world applications can be understood by building simplified prototypes. By studying systems like Google Maps, Spotify, Git, and Pinterest, we explored how graphs, trees, arrays, hash maps, and probability-based algorithms form the backbone of many modern technologies.