TASK-1

With Understanding Identify The Scope, Advantages And Disadvantages Of The Given Searching Algorithm .

Name: Kaushal Dahal

Reg No. : 2023006226

Searching Technique	Scope	Constraints	Application
Linear Search	• Works on **unsorted or sorted** data. • Suitable for **small datasets**. • Useful when data structure does not support random access (e.g., linked lists).	• **Time complexity is O(n)** → slow for large datasets. • Cannot exploit sorted structure efficiently. • Not suitable for performance-critical applications.	• Searching small unsorted lists. • Lookup in **linked lists**. • When the dataset is very small or rarely searched.
Binary Search	• Works only on **sorted arrays/lists**. • Efficient on static datasets with random access.	• Requires **sorted input**. • **Not suitable** for linked lists unless modified. • Updates (insert/delete) are costly because sorting must be maintained.	• Search in **databases**, dictionaries. • Finding elements in sorted arrays. • Used in libraries like `bisect()` in Python.
Depth-First Search (DFS)	DFS goes **deep along one path** before backtracking. Used on: • **Graphs and Trees** • Both directed and undirected graphs	• Can get stuck in cycles without "visited" tracking • Recursive DFS may cause **stack overflow** for deep graphs • Takes longer to find shortest paths	• Pathfinding in puzzles (mazes) • Detecting cycles in graphs • Topological sorting • Solving tree-based problems

Breadth-First Search (BFS)	• Explores nodes **level by level**, making it perfect for finding the **shortest path**. • Useful in real-world graphs like social networks and maps. • Works for finite graphs and trees	• Requires **large memory** because all neighbors must be stored in a queue. • Slower when the branching factor is very high. • Not ideal for very deep graph	• Finding **shortest path in unweighted graphs** • GPS navigation • Social networks (friend recommendations) • Web crawling
Hash-Based Search	• Allows **very fast searches** (O(1) average time). • Used when you need instant lookup, such as dictionaries or symbol tables. • Works best with **unique keys** and well-distributed hash functions.	• Hash collisions can reduce speed. • Requires more memory (extra buckets and overhead). • Performance depends heavily on quality of hash function.	• Databases indexing • Language compilers (symbol tables) • Password hashing and validation • Implementing dictionaries/maps
Jump Search	• A middle-ground between linear and binary search for **sorted arrays**. • Uses fixed jumps to reduce search time in large datasets. • Helpful in systems where jumping is faster than random access.	• Only works for **sorted** arrays. • Slower compared to binary search in most cases. • Choosing optimal jump size is required for best performance.	• Searching in large sorted lists • Used where binary search overhead is expensive

Interpolation Search	• Estimates the position of the target based on **value distribution**. • Very fast when data is **evenly spread** (e.g., student roll numbers). • Works only on sorted data with numeric or uniform keys.	• Performs poorly when values are **clustered** or unevenly distributed. • Cannot be used with strings or unsorted data. • Worst-case complexity becomes O(n) if distribution is bad.	• Searching in uniform numeric datasets • Databases with evenly distributed keys • Statistical lookup tables
Exponential Search	• Useful when the array size is **unknown** or very large. • Quickly finds a range using exponential jumps before applying binary search. • Works for sorted lists, especially streaming or unbounded data sources.	• Needs random access (array). • Inefficient for small datasets. • Only works for **sorted** data.	• Searching in **unbounded** or **infinite lists** • Used in online data streams • Useful in exponentially growing data structures
Fibonacci Search	• Similar to binary search but uses Fibonacci sequence to divide search range. • Reduces comparisons in systems where memory access is costly. • Best for sorted arrays stored in slow memory (e.g., old tapes).	• More complex to implement than binary search. • Slightly slower in practice due to Fibonacci calculations. • Only works on sorted lists.	• Searching when memory access is expensive • Embedded systems • Systems using sequential memory blocks (tapes)

Ternary Search	• Splits data into three parts instead of two. • Primarily used to find peak/minimum in **unimodal** functions. • Works well in optimization problems.	• Not useful for regular array searching. • Slower than binary search for discrete datasets. • Requires the function to have only **one peak or valley** (unimodal).	• Optimization problems • Finding minimum/maximum of mathematical functions • AI algorithms requiring optimization
Sublist Search (KMP Algorithm)	• Efficient for **pattern matching** in text (string search). • Uses preprocessing to avoid re-checking characters. • Works on very large texts.	• Needs preprocessing of pattern (computing lps table). • Only applicable to sequential data like strings or lists. • Higher initial setup cost for short searches.	• Text editors (Find/Replace) • DNA/protein sequence analysis • Search engines • Plagiarism detection
A* Search Algorithm	• Combines cost so far (**g**) + heuristic estimate (**h**) to find best path. • Works on weighted graphs and maps. • Finds **shortest optimal paths** if heuristic is correct.	• Requires a **good heuristic**; bad heuristic slows it down. • Can use large amounts of memory (open/closed lists). • Not ideal for extremely large graphs.	• Google Maps, GPS navigation • Video game AI for pathfinding • Robotics movement and planning • Network routing protocols
Beam Search	• Heuristic search that keeps only a limited number of best candidates. • Useful in huge search problems where full exploration is impossible. • Trades accuracy for speed.	• May miss the optimal solution due to limited beam width. • Depends heavily on heuristics. • Not guaranteed to find the best or even correct result.	• Speech recognition • Machine Translation (NLP) • Optimization problems • Large decision-tree searches

Grover's Search Algorithm	• A quantum search algorithm that finds a target in **√n time**. • Works on unsorted databases better than classical methods. • Useful in fields needing massive search optimization.	• Requires quantum computing hardware (not widely available). • Only gives quadratic speedup, not exponential. • Sensitive to quantum noise and decoherence.	• Cryptography (breaking symmetric keys faster) • Searching quantum databases • Quantum AI and optimization