What is an Algorithm?

• An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem-solving operations. • According to its formal definition, an algorithm is a finite set of instructions carried out in a specific order to perform a particular task.
• It is not the entire program or code; it is simple logic to a problem represented as an informal description in the form of a flowchart or pseudocode.

Problem: A problem can be defined as a real-world problem or real-world instance problem for which you need to develop a program or set of instructions. An algorithm is a set of instructions.

• **Algorithm:** An algorithm is defined as a step-by-step process that will be designed for a problem.

Input: After designing an algorithm, the algorithm is given the necessary and desired inputs.

Processing unit: The input will be passed to the processing unit, producing the desired output.

Output: The outcome or result of the program is referred to as the output.

Characteristics of an Algorithm: An algorithm has the following characteristics:

Input: An algorithm requires some input values. An algorithm can be given a value other than 0 as input.

Output: At the end of an algorithm, you will have one or more outcomes.

Unambiguity: A perfect algorithm is defined as unambiguous, which means that its instructions should be clear and straightforward.
Finiteness: An algorithm must be finite. Finiteness in this context means that the algorithm should have a limited number of instructions, i.e., the instructions should be countable.

Effectiveness: Because each instruction in an algorithm affects the overall process, it should be adequate.

Language independence: An algorithm must be language-independent, which means that its instructions can be implemented in any language and produce the same results.

Data abstraction:
it is a fundamental concept in computer science that involves focusing on the essential features of data while hiding its underlying implementation details. It's about providing a simplified, external view of data while shielding users from the complexities of how it's stored, structured, or manipulated.

Essential vs. irrelevant details: Data abstraction separates the essential characteristics of data that are relevant to the user from the implementation details that are not.
Simplification: It presents a simplified interface, making it easier to understand and work with data.
Hiding complexity: It conceals the intricate internal workings, making systems more manageable and less prone to errors
Levels of abstraction: It often involves multiple layers, each providing a different level of detail.

Benefits of data abstraction:
Reduces complexity: Makes systems easier to understand, design, and maintain.
Enhances modularity: Promotes code reusability and independent development.
Manages change effectively: Allows modifications to implementation without affecting external interfaces.
Improves security: Can restrict access to sensitive data and implementation details.

Key Differences:
1. **Uniqueness of Elements:**
Sets: Store only unique elements. Each element can appear only once.
Multisets: Allow duplicate elements. An element can appear multiple times.
2. **Underlying Data Structure:**
Sets: Often implemented using self-balancing binary search trees (e.g., red-black trees) for efficient operations.
Multisets: Can be implemented using hash tables or self-balancing binary search trees, depending on the desired performance characteristics.
4. **Common Use Cases:**
Sets:
Removing duplicates from a collection
Implementing mathematical sets
Finding unique elements in a data stream
Checking for membership
Implementing sets of distinct elements (e.g., unique words in a text)
Multisets:
Counting the occurrences of elements in a collection
Implementing histograms
Tracking word frequencies in a document
Representing bag-like structures where element multiplicity matters

Asymptotic notation
It is a mathematical tool used in computer science to describe the rate of growth of functions as their input size approaches infinity. It allows us to categorize algorithms based on their efficiency and compare their performance for large inputs, ignoring constant factors and lower-order terms.

Types of Asymptotic Notation:
1. **Big O Notation (O):**
Represents the upper bound of the growth rate of a function.
Formal definition: $f(n) = O(g(n))$ if there exist positive constants c and n0 such that $f(n) \leq c * g(n)$ for all $n \geq n0$.
Example:
Linear search algorithm has a time complexity of O(n) because its worst-case number of comparisons grows linearly with the size of the input list.
graph showing linear function $f(n) = n$ and O(n) boundary
2. **Big Omega Notation (Ω):**
Represents the lower bound of the growth rate of a function.
Formal definition: $f(n) = \Omega(g(n))$ if there exist positive constants c and n0 such that $f(n) \geq c * g(n)$ for all $n \geq n0$.
Example:
Merge sort algorithm has a best-case time complexity of $\Omega(n \log n)$ because even in the best-case scenario, it needs to perform a logarithmic number of comparisons for each element.

3. **Big Theta Notation (Θ):**
Represents the tight bound of the growth rate of a function.
Formal definition: $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
Example:
Binary search algorithm has a time complexity of $\Theta(\log n)$ because its number of comparisons always grows logarithmically with the size of the input list.

Space and time complexity are two crucial metrics used in computer science to analyze the efficiency of algorithms and data structures. They tell us how much memory (space) and execution time (time) an algorithm or data structure needs to solve a problem as the input size grows.
Time Complexity:
Measures the amount of time an algorithm takes to run as the input size increases.
Typically expressed using asymptotic notation, such as Big O notation, which captures the dominant term affecting the algorithm's execution time for large inputs.
Common time complexities include:
O(1) - Constant time: Runs in the same amount of time regardless of the input size.
O(log n) - Logarithmic time: Execution time grows logarithmically with the input size.
O(n) - Linear time: Execution time grows linearly with the input size.
O(n log n) - Log-linear time: Execution time grows slower than a quadratic but faster than linear.
O(n^2) - Quadratic time: Execution time grows proportionally to the square of the input size.
Space Complexity:
Measures the amount of memory an algorithm or data structure requires to run as the input size increases.
Also expressed using asymptotic notation, particularly Big O notation.
Common space complexities include:
O(1) - Constant space: Uses a constant amount of memory regardless of the input size.
O(n) - Linear space: Memory usage grows linearly with the input size.
O(n^2) - Quadratic space: Memory usage grows proportionally to the square of the input size.

Divide and conquer is a powerful algorithm design paradigm that breaks down a problem into smaller, more manageable subproblems, solves them recursively, and then combines the solutions to solve the original problem. It's often used to design efficient algorithms for complex problems.
Key steps:
Divide:
Partition the problem into two or more subproblems that are smaller instances of the same problem.
The division should be done in a way that makes the subproblems easier to solve.
Conquer:
Recursively solve each subproblem using the same divide and conquer approach.
If a subproblem is small enough, solve it directly (base case).
Combine:
Merge the solutions of the subproblems to obtain the solution to the original problem.
This often involves combining sorted lists, merging trees, or combining partial results.
Example algorithms:
Merge sort: Sorts a list by repeatedly dividing it into halves, sorting each half recursively, and then merging the sorted halves.
Quicksort: Another sorting algorithm that partitions the list around a pivot element and recursively sorts the sublists.
Binary search: Efficiently searches a sorted list by repeatedly dividing the search interval in half.
Closest pair of points: Finds the two closest points in a set of points using a divide and conquer approach.

Greedy Algorithm:
It's a problem-solving approach that involves making locally optimal choices at each step in the hope of finding a global optimum solution.
It prioritizes immediate benefits, selecting the best option available at the current moment without considering the long-term consequences or backtracking to reconsider earlier choices.
Key Characteristics:
Locally Optimal Choices: The algorithm focuses on making the best decision based on the current situation, without considering the implications for future steps.
Construction of a Solution: It builds a solution incrementally, adding one element or decision at a time.
Irreversible Decisions: Once a decision is made, it cannot be reversed, even if it turns out to be suboptimal later.
Not Always Optimal: Greedy algorithms don't guarantee the optimal solution in all cases. They can sometimes get stuck in local optima, missing the globally best solution.
Common Applications:

- Prim's algorithm for minimum spanning trees
- Kruskal's algorithm for minimum spanning trees
- Dijkstra's algorithm for shortest path
- Huffman coding for data compression
- Activity selection problem
- Coin change problem
- Fractional knapsack problem
- Job scheduling problem

Advantages:
Simple and Efficient: Greedy algorithms are often simple to understand and implement, leading to efficient execution.
Faster than Dynamic Programming: They can be faster than dynamic programming, which considers all possible solutions.
Disadvantages:
Not Always Optimal: They may not always find the global optimum solution, especially for problems with complex dependencies between choices.
No Backtracking: They cannot correct suboptimal choices made earlier.

Dynamic programming (DP) is a powerful algorithm design technique used to solve complex problems by breaking them down into simpler subproblems and storing the solutions to these subproblems to avoid redundant calculations. Here's a breakdown of its key aspects:
1. **Optimal Substructure:**
DP problems exhibit optimal substructure, meaning that the optimal solution to the overall problem can be constructed from optimal solutions to its subproblems.
2. **Overlapping Subproblems:**
The same subproblems are often solved repeatedly during the solution process. DP capitalizes on this by storing the solutions to these subproblems, eliminating redundant computations.

Steps to Solve a DP Problem:
Identify the subproblems: Break down the problem into smaller, overlapping subproblems.
Define a recurrence relation: Express the solution to the problem in terms of solutions to smaller subproblems, creating a mathematical formula that relates the solutions of different subproblems.
Choose a memoization or tabulation approach:
Memoization: Top-down approach. Solves subproblems as needed and stores solutions in a table for future reference.
Tabulation: Bottom-up approach. Builds up solutions to larger subproblems from solutions to smaller subproblems, typically using a table to store intermediate results.
Build a solution table: Construct a table to store the solutions to subproblems, either iteratively (tabulation) or recursively (memoization).
Construct the optimal solution: Use the stored solutions in the table to reconstruct the optimal solution to the original problem.
Common Applications of DP:
Optimization problems (shortest paths, knapsack problem, sequence alignment)
Counting problems (number of ways to do something)
Graph problems (finding paths, spanning trees)
String problems (edit distance, longest common subsequence)
Combinatorial problems (coin change, matrix chain multiplication)

Advantages of DP:
Can solve complex problems efficiently that are not solvable by other techniques like greedy algorithms or recursion.
Can often find the optimal solution, not just an approximation.
Can be used to solve a wide variety of problems.
Disadvantages of DP:
Can be memory-intensive, especially for large problems.
Can be difficult to come up with the recurrence relation and solution structure for a given problem.

Sorting is a fundamental process in computer science that involves arranging data in a particular order. This order can be ascending (e.g., smallest to largest), descending (largest to smallest), or based on custom criteria. By sorting data, we make it easier to search, analyze, and compare information.
Now, let's delve into internal and external sorting:

Internal Sorting:
Description: Internal sorting algorithms operate on data that can fit entirely within the main memory (RAM) of your computer.
Advantages:
Faster: Since data manipulation happens within RAM, internal sorting is typically much faster than external sorting.
Simpler to implement: Algorithms are generally easier to design and understand for internal sorting.
Disadvantages:
Limited data size: Restricted by the available RAM, internal sorting cannot handle massive datasets that exceed memory capacity.
Common Internal Sorting Algorithms:

- Merge Sort
- Quick Sort
- Bubble Sort
- Insertion Sort
- Heap Sort

External Sorting:
Description: External sorting algorithms are designed to handle data that is too large to fit in RAM at once. This data resides on slower secondary storage devices like hard disks or SSDs.
Advantages:
No size limitations: Can handle massive datasets regardless of RAM capacity.
Disadvantages:
Slower: Accessing data from a disk is significantly slower than RAM, making external sorting less efficient.
More complex algorithms: Designed to minimize disk access and utilize secondary storage effectively, making them more intricate to implement.
Common External Sorting Algorithms:
Merge Sort (modified for external memory)
Polyphase Sort
Sort-Merge algorithm

Bubble Sort:
Simple sorting algorithm that repeatedly steps through a list, comparing adjacent elements and swapping them if they are in the wrong order.
Named for the way larger elements "bubble" to the end of the list.
Algorithm:

1. Iterate through the list n-1 times (where n is the number of elements).
2. For each iteration, compare each pair of adjacent elements.
3. If a pair is in the wrong order, swap them.
4. Repeat steps 1-3 until no more swaps occur, indicating the list is sorted.

Example:

- Unsorted list: [6, 4, 2, 1, 5, 3]
- First pass:
  - [4, 6, 2, 1, 5, 3] (swap 6 and 4)
  - [4, 2, 6, 1, 5, 3] (swap 6 and 2)
  - [4, 2, 1, 6, 5, 3] (swap 6 and 1)
  - [4, 2, 1, 5, 6, 3] (swap 6 and 5)
  - [4, 2, 1, 5, 3, 6] (swap 6 and 3)
- Subsequent passes:
  - [2, 4, 1, 3, 5, 6]
  - [2, 1, 4, 3, 5, 6]
  - [1, 2, 3, 4, 5, 6] (sorted)

Time Complexity:
Worst-case: O(n^2)
Occurs when the list is in reverse order or close to it, requiring the

maximum number of comparisons and swaps.
Average-case: O(n^2)
While some cases might be slightly faster, the average performance is still quadratic.
Best-case: O(n)
Occurs when the list is already sorted, and no swaps are needed. However, this is uncommon in practice.
Space Complexity:
O(1): Bubble sort is an in-place algorithm, meaning it doesn't require additional memory space for sorting beyond a few variables to hold temporary values during swaps.

Insertion Sort:
Simple sorting algorithm that works like arranging a hand of cards: It iterates through the list, inserting each element in its correct position within the already sorted portion of the list.
Algorithm:
1. Start with the second element (index 1).
2. Compare the current element with the elements to its left that are already sorted.
3. Shift elements to the right until a suitable position is found for the current element.
4. Insert the current element into its correct position.
5. Repeat steps 2-4 for the remaining elements.

Example:

- Unsorted list: [6, 4, 2, 1, 5, 3]
- Iteration 1: [4, 6, 2, 1, 5, 3] (insert 4 at index 0)
- Iteration 2: [2, 4, 6, 1, 5, 3] (insert 2 at index 0)
- Iteration 3: [1, 2, 4, 6, 5, 3] (insert 1 at index 0)
- Iteration 4: [1, 2, 4, 5, 6, 3] (insert 5 at index 3)
- Iteration 5: [1, 2, 3, 4, 5, 6] (insert 3 at index 2)
- Sorted list: [1, 2, 3, 4, 5, 6]

Complexities:
Time Complexity:
Worst-case: O(n^2) (when the list is in reverse order)
Average-case: O(n^2)
Best-case: O(n) (when the list is already sorted)
Space Complexity: O(1) (in-place sorting)

Selection Sort:
Simple sorting algorithm that repeatedly finds the minimum (or maximum) element in the unsorted part of the list and places it at the beginning (or end) of the sorted part.
Like repeatedly selecting the smallest card from a hand and placing it in a new, sorted pile.
Algorithm:
Iterate through the list, starting from the beginning.
Find the minimum (or maximum) element in the unsorted part of the list.
Swap the found minimum (or maximum) element with the first element of the unsorted part.
Repeat steps 2-3 for the remaining unsorted part of the list.
Example:

- Unsorted list: [6, 4, 2, 1, 5, 3]
- Iteration 1: [1, 4, 2, 6, 5, 3] (select 1 as the minimum)
- Iteration 2: [1, 2, 4, 6, 5, 3] (select 2 as the minimum)
- Iteration 3: [1, 2, 3, 6, 5, 4] (select 3 as the minimum)
- Iteration 4: [1, 2, 3, 4, 5, 6] (select 4 as the minimum)
- Sorted list: [1, 2, 3, 4, 5, 6]

Complexities:
Time Complexity:
- Worst-case: O(n^2) (in all cases)
- Average-case: O(n^2)
- Best-case: O(n^2)

Space Complexity: O(1) (in-place sorting)

Quick Sort:
Efficient divide-and-conquer sorting algorithm that recursively divides a list into smaller sublists, sorts them independently, and then combines them back into a sorted list.
Known for its average-case efficiency and in-place sorting.
Algorithm:
Choose a pivot element from the list (often the first or last element).
Partition the list around the pivot: elements less than the pivot are placed to its left, and elements greater than the pivot are placed to its right.
Recursively apply quicksort to the sublists to the left and right of the pivot.
Combine the sorted sublists (pivot in the middle) to obtain the final sorted list.
Pseudocode:
```
function quickSort(array, left, right):
  if left < right:
    pivotIndex = partition(array, left, right)
    quickSort(array, left, pivotIndex - 1)
    quickSort(array, pivotIndex + 1, right)

function partition(array, left, right):
  pivotValue = array[right]  // Choose pivot (can be different strategies)
  i = left - 1
  for j = left to right - 1:
    if array[j] <= pivotValue:
      i = i + 1
      swap(array[i], array[j])
  swap(array[i + 1], array[right]) // Place pivot in its correct position
  return i + 1
```
Example:

- Unsorted list: [6, 4, 2, 1, 5, 3]
- Choose pivot: 6 (first element)
- Partition: [4, 2, 1, 5, 3, 6] (elements less than 6 moved to its left)
- Recursively sort sublists:
  - [4, 2, 1, 3] -> [1, 2, 3, 4]
  - [5] -> [5]
- Combine sorted sublists: [1, 2, 3, 4, 5, 6]

Complexities:
Time Complexity:
- Worst-case: O(n^2) (unbalanced partitions, rare)
- Average-case: O(n log n) (efficient for most cases)
- Best-case: O(n log n) (already sorted or nearly sorted)

Space Complexity: O(log n) (due to recursion stack)

Shell Sort:
A generalization of insertion sort that improves its efficiency for larger datasets by breaking the original list into smaller sublists, sorting them partially, and then merging those partially sorted sublists to produce a fully sorted list.
Named after its inventor, Donald Shell.
Algorithm:
Choose a sequence of "gaps" (numbers that determine the intervals for sublists).
For each gap in the sequence:
Perform insertion sort on sublists formed by elements that are gap positions apart.
Repeat step 2 for decreasing gap values until the gap is 1 (effectively performing a final insertion sort on the entire list).
Pseudocode:
```
function shellSort(array):
  gap = length(array) //2  // Initialize gap as half the array size
  while gap > 0:
    for i = gap to length(array) - 1:
      temp = array[i]
      j = i
      while j >= gap and array[j - gap] > temp:
        array[j] = array[j - gap]
        j = j - gap
      array[j] = temp
    gap //= 2  // Decrease gap for the next iteration
```
Example:

- Unsorted list: [6, 4, 2, 1, 5, 3]
- Gap sequence: [3, 1]
- Gap 3: [4, 2, 1, 6, 5, 3] (insertion sort on sublists with elements 3 positions apart)
- Gap 1: [1, 2, 3, 4, 5, 6] (insertion sort on the entire list)

Complexities:
Time Complexity:
Worst-case: Depends on the gap sequence, but typically O(n^2) in practice.
Average-case: Uncertain, but often better than O(n^2), closer to O(n log^2 n).
Best-case: O(n log n) (with specific gap sequences).
Space Complexity: O(1) (in-place sorting).

Merge Sort:
A highly efficient divide-and-conquer sorting algorithm that recursively divides an unsorted list into smaller sublists until each sublist contains only one element, then merges these sublists back together in sorted order.
Algorithm:
Divide:
If the list has more than one element, divide it into two halves.
Recursively apply merge sort to each half.
Conquer:
Merge the two sorted sublists back together into a single sorted list.
Pseudocode:
```
function mergeSort(array):
  if length(array) > 1:
    mid = length(array) // 2
    left = array[:mid]
    right = array[mid:]
    mergeSort(left)
    mergeSort(right)
    merge(left, right, array)

function merge(left, right, array):
  i = j = k = 0
  while i < len(left) and j < len(right):
    if left[i] <= right[j]:
      array[k] = left[i]
      i += 1
    else:
      array[k] = right[j]
      j += 1
    k += 1
  while i < len(left):
    array[k] = left[i]
    i += 1
    k += 1
  while j < len(right):
    array[k] = right[j]
    j += 1
    k += 1
```
Example:

- Unsorted list: [6, 4, 2, 1, 5, 3]
- Divide:
  - [6, 4, 2] and [1, 5, 3]
  - [6] and [4, 2]
  - [1] and [5, 3]
  - [5] and [3]

Conquer (merging):
- [4, 6] and [1, 2] -> [1, 2, 4, 6]
- [3, 5] -> [3, 5]
- [1, 2, 4, 6] and [3, 5] -> [1, 2, 3, 4, 5, 6]

Complexities:
Time Complexity:
Worst-case, Average-case, Best-case: O(n log n)
Space Complexity: O(n) (due to recursion stack and temporary array for merging)
Key Points:

- Highly efficient for large datasets due to its consistent O(n log n) time complexity.
- Stable sort (preserves the order of equal elements).
- Recursive algorithm with a clear divide-and-conquer approach.
- Requires additional memory for merging, making it less suitable for memory-constrained environments.
- Often used in external sorting algorithms for handling large datasets that don't fit in memory.

Heap Sort:
An efficient comparison-based sorting algorithm that leverages a specialized data structure called a heap to repeatedly extract the largest (or smallest) element and place it in its correct position in the sorted array.
Algorithm:
1. Build a max-heap: Arrange the elements of the array into a complete binary tree where each node is greater than or equal to its children.
2. Repeatedly extract the maximum element:
   - Swap the root of the heap (the largest element) with the last element in the unsorted part of the array.
   - Remove the last element from the unsorted part (it's now in its correct position).
   - Heapify the root to restore the max-heap property.
3. Repeat step 2 until the entire array is sorted.

Pseudocode:
```
function heapSort(array):
  // Build max-heap
  buildMaxHeap(array)

  // Repeatedly extract maximum and heapify
  for i = length(array) - 1 to 1:
    swap(array[0], array[i]) // Move largest element to end
    heapify(array, 0, i) // Heapify the reduced heap

function buildMaxHeap(array):
  for i = length(array) // 2 - 1 to 0:
    heapify(array, i, length(array))

function heapify(array, i, heapSize):
  largest = i
  left = 2 * i + 1
  right = 2 * i + 2
  if left < heapSize and array[left] > array[largest]:
    largest = left
  if right < heapSize and array[right] > array[largest]:
    largest = right
  if largest != i:
    swap(array[i], array[largest])
    heapify(array, largest, heapSize)
```
Example:

- Unsorted list: [6, 4, 2, 1, 5, 3]
- Build max-heap: [6, 4, 5, 1, 3, 2]
- Extract max and heapify:
  - [2, 4, 5, 1, 3, 6]
  - [2, 3, 5, 1, 4, 6]
  - [2, 1, 5, 3, 4, 6]
  - [1, 2, 5, 3, 4, 6]
  - [1, 2, 3, 4, 5, 6] (sorted)

Complexities:
Time Complexity:
Worst-case, Average-case, Best-case: O(n log n)
Space Complexity: O(1) (in-place sorting)

Key Points:

- Efficient for large datasets due to its consistent time complexity.
- In-place sorting, requiring minimal extra memory.
- Not stable (may not preserve the order of equal elements).
- Uses a heap data structure for efficient element selection and sorting.
- Well-suited for scenarios where memory efficiency is crucial.

Count Sort:
A non-comparison-based sorting algorithm that works by counting the occurrences of each unique element in the input array and then using those counts to determine their sorted positions.
Efficient for sorting items with a limited range of possible values.
Algorithm:
1. Find the maximum value (k) in the input array.
2. Create a count array of size k+1, initialized to zeros.
3. Count the occurrences of each element in the input array and store the counts in the count array.
4. Modify the count array to contain the cumulative sum of counts (each element now indicates the position of the corresponding value in the sorted array).
5. Use the count array to place the elements in their sorted positions in a new output array.

Pseudocode:
```
function countSort(array):
  maxValue = findMax(array)  // Find the maximum value
  countArray = [0] * (maxValue + 1)  // Create count array

  // Count occurrences of each element
  for i in array:
    countArray[i] += 1

  // Modify count array to hold cumulative sums
  for i in range(1, maxValue + 1):
    countArray[i] += countArray[i - 1]

  sortedArray = [0] * len(array)  // Create output array

  // Place elements in sorted order
  for i in array:
```

```
    sortedIndex = countArray[i] - 1
    sortedArray[sortedIndex] = i
    countArray[i] -= 1

  return sortedArray
```

Example:

- Unsorted list: [6, 4, 2, 1, 5, 3]
- Count array: [0, 1, 1, 2, 1, 1, 1] (counts of each value)
- Modified count array: [0, 1, 2, 4, 5, 6, 7] (cumulative sums)
- Sorted list: [1, 2, 3, 4, 5, 6]

Complexities:

- **Time Complexity:**
  - Worst-case, Average-case, Best-case: O(n + k), where n is the number of elements and k is the range of values
- Space Complexity: O(k) (due to the count array)

Bucket Sort:

A sorting algorithm that divides elements into a number of "buckets" based on their values and then sorts each bucket individually, often using another sorting algorithm.

Efficient for uniformly distributed data with a known range.

Algorithm:

1. Create empty buckets, typically using an array of lists or arrays.
2. Distribute elements into buckets based on their values (using a hash function or range-based division).
3. Sort each bucket individually using an appropriate sorting algorithm (e.g., insertion sort or merge sort).
4. Concatenate the sorted buckets in order to obtain the final sorted list.

Pseudocode:

```
function bucketSort(array, numBuckets):
  buckets = [[]] * numBuckets  // Create empty buckets

  // Distribute elements into buckets
  for i in array:
    bucketIndex = hashFunction(i) % numBuckets  // Determine bucket using hash function
    buckets[bucketIndex].append(i)

  // Sort each bucket individually
  for i in range(numBuckets):
    insertionSort(buckets[i])  // Example using insertion sort

  // Concatenate sorted buckets
  sortedArray = []
  for bucket in buckets:
    sortedArray.extend(bucket)

  return sortedArray
```

Example:

- Unsorted list: [6, 4, 2, 1, 5, 3], range = [1, 6], numBuckets = 3
- Buckets: [[1, 6], [4, 5], [2, 3]] (after distribution)
- Sorted buckets: [[1, 6], [4, 5], [2, 3]] (after sorting each bucket)
- Sorted list: [1, 2, 3, 4, 5, 6] (after concatenation)

Complexities:

Time Complexity:

Average-case: O(n + k), where n is the number of elements and k is the number of buckets

Worst-case: O(n^2) if elements cluster into a few buckets

Space Complexity: O(n + k) (due to buckets and potential auxiliary space for sorting)

Radix Sort:

A non-comparison-based sorting algorithm that sorts elements by repeatedly distributing them into buckets based on individual digits or characters, starting from the least significant digit (LSD) or most significant digit (MSD).

Efficient for sorting integers or strings with fixed-length keys.

Algorithm (LSD Radix Sort):

1. Determine the maximum number of digits (or characters) in the keys.
2. For each digit position (from least significant to most significant):
   - Create empty buckets for each possible digit (or character) value.
   - Distribute elements into buckets based on the digit at the current position.
   - Concatenate the buckets in order to form the partially sorted list.

Pseudocode:

```
function radixSortLSD(array):
  maxDigits = findMaxDigits(array)  // Find the maximum number of digits

  for d = 0 to maxDigits - 1:  // Iterate through each digit position
    buckets = [[] for _ in range(10)]  // Create empty buckets for 10 digits

    for num in array:
      digit = (num // 10**d) % 10  // Extract the digit at position d
      buckets[digit].append(num)

    array = []
    for bucket in buckets:
      array.extend(bucket)  // Concatenate sorted buckets

  return array
```

Example:

- Unsorted list: [329, 457, 657, 839, 436, 720]
- Iteration 1 (sorting by units digit): [720, 457, 657, 839, 329, 436]
- Iteration 2 (sorting by tens digit): [329, 436, 457, 720, 657, 839]
- Iteration 3 (sorting by hundreds digit): [329, 436, 457, 657, 720, 839] (sorted)

Complexities:

Time Complexity:

Average-case: O(nk), where n is the number of elements and k is the maximum number of digits (or characters)

Worst-case: O(nk)

Space Complexity: O(n + k) (due to buckets and potential auxiliary space)

The choice of sorting technique depends on several factors, including:

- Data size and type: Smaller datasets can tolerate less efficient algorithms, while large datasets need algorithms with better time complexity. For integers or strings with fixed-length keys, radix sort or bucket sort can be very efficient.
- Data distribution: Some algorithms work better for uniformly distributed data (bucket sort), while others are robust to skewed distributions (merge sort).
- Memory constraints: In-place sorting algorithms like quicksort or heap sort require minimal extra memory, while others like count sort or radix sort use additional space for buckets or counting arrays.
- Stability: If preserving the order of equal elements is important, then stable sorting algorithms like merge sort or insertion sort are preferred.

Searching:

- The process of finding a specific item or element within a collection of data.
- Involves examining individual elements to determine if they match a given target value or satisfy a certain condition.
- Common searching algorithms include linear search, binary search, and hash table search.

Key Differences Between Sorting and Searching:

Purpose:

- Sorting: Arranges elements in a specific order (ascending, descending, or based on a custom criterion).
- Searching: Locates a particular element within the collection.

Output:

- Sorting: Produces a new, sorted collection of elements.
- Searching: Returns the position (index) of the target element or indicates its absence.

Time Complexity:

- Sorting: Typically involves more complex algorithms and has higher time complexities (e.g., O(n log n) for merge sort, O(n^2) for bubble sort).
- Searching: Some algorithms can be very efficient, especially for sorted data (e.g., O(log n) for binary search).

Relationship:

- Sorting can often improve the efficiency of searching, especially for algorithms like binary search that rely on a sorted collection.
- However, sorting isn't always necessary for searching, and some searching algorithms can work efficiently even on unsorted data.

Example:

- Sorting a list of names alphabetically allows for quick binary search to find a specific name.
- Searching for a particular value in an unsorted list might require a linear search, examining each element sequentially.

Linear Search:

Simplest searching algorithm.

Examines each element in the list sequentially, one by one, until the target value is found or the end of the list is reached.

Algorithm:

1. Start at the first element of the list.
2. Compare the current element with the target value.
3. If they match, return the index of the current element.
4. If not, move to the next element and repeat steps 2-3.
5. If the end of the list is reached without finding the target, return "not found."

Time Complexity:

- Worst-case, Average-case, Best-case: O(n) (where n is the number of elements)
- Linearly dependent on the list size.

Binary Search:

Efficient searching algorithm for sorted lists.

Repeatedly divides the search interval in half, comparing the target value with the middle element until the target is found or the interval is empty.

Algorithm:

1. Start with the entire list as the search interval.
2. Find the middle element of the interval.
3. If the middle element matches the target value, return its index.
4. If the target value is smaller than the middle element, search the left half of the interval.
5. If the target value is larger than the middle element, search the right half of the interval.
6. Repeat steps 2-5 until the target is found or the interval is empty.

Time Complexity:

- Worst-case, Average-case, Best-case: O(log n) (where n is the number of elements)
- Logarithmic time, much faster than linear search for large lists.

comparison of linear search and binary search, highlighting their key differences:

Data Requirement:

- Linear Search: Works on both sorted and unsorted data.
- Binary Search: Requires sorted data for efficient operation.

Time Complexity:

- Linear Search: O(n), meaning it takes at most n comparisons to find the target element, where n is the number of elements in the list.
- Binary Search: O(log n), significantly faster than linear search, taking only logarithmic comparisons on average to find the target element.

Efficiency:

- Linear Search: Less efficient for large datasets as the number of comparisons grows linearly with the data size.
- Binary Search: Highly efficient for large sorted datasets due to its rapid elimination of half the search space with each comparison.

Implementation:

- Linear Search: Simpler to implement, requiring basic iteration through the data.
- Binary Search: More complex to implement, requiring maintaining and manipulating indices and sub-lists within the sorted data.

Use Cases:

- Linear Search: Used when data is unsorted or small, or when simplicity is preferred.
- Binary Search: Preferred for efficiently searching large sorted datasets where performance is crucial.

Binary Search Trees (BSTs):

Key Characteristics:

- A type of binary tree where each node has a key greater than all keys in its left subtree and less than all keys in its right subtree.
- This property enables efficient searching, insertion, and deletion.

Searching:

Algorithm:

1. Start at the root node.
2. Compare the target value with the node's key.
3. If they match, return the node.
4. If the target is smaller, search the left subtree.
5. If the target is larger, search the right subtree.
6. Repeat steps 2-5 until the target is found or the search reaches a null node (indicating the target is not present).

Time Complexity:

- Average-case: O(log n)
- Worst-case (skewed tree): O(n)

Insertion:

Algorithm:

1. Perform a search to find the appropriate position for the new node.
2. Create the new node with the given key and value.
3. Insert it as a leaf node at the found position.

Time Complexity:

- Average-case: O(log n)
- Worst-case (skewed tree): O(n)

Deletion:

Algorithm: (more complex than insertion, depends on the node's position and number of children)

1. Find the node to be deleted.
2. If it has no children (leaf node), simply remove it.
3. If it has one child, replace it with its child.
4. If it has two children, find its in-order successor (smallest node in the right subtree) or predecessor (largest node in the left subtree), replace its value with the successor/predecessor's value, and delete the successor/predecessor node.

Time Complexity:

- Average-case: O(log n)
- Worst-case (skewed tree): O(n)

Space Complexity:

- Generally O(n) for a BST with n nodes, considering the space for the nodes and pointers.

Advantages:

- Efficient searching, insertion, and deletion (average-case O(log n))
- Self-balancing variants (e.g., AVL trees, red-black trees) maintain O(log n) performance even in worst-case scenarios.
- Dynamically adaptable to insertions and deletions.

Disadvantages:

- Performance can degrade to O(n) in worst-case scenarios (skewed trees).
- Rebalancing operations can be complex and add overhead.

Applications:

- Sorting algorithms (e.g., merge sort, quicksort): To manage sorted sub-sequences.
- Database systems: To implement indexing and fast retrieval of data.
- In-memory caches: To store and access frequently used data efficiently.
- Network routing: To store routing information for efficient packet forwarding.

- Many other areas where efficient searching, insertion, and deletion are required.

Height Balancing in Balanced Search Trees (BSTs):

Significance:

- Prevents BSTs from becoming skewed (unbalanced), ensuring logarithmic time complexity (O(log n)) for operations like searching, insertion, and deletion even in worst-case scenarios.
- Maintains efficient performance and avoids degradation to linear time (O(n)).

Advantages:

- Guaranteed logarithmic time complexity for basic operations, leading to predictable and consistent performance.
- Improved efficiency for large datasets and frequent operations.
- Adaptability to dynamic changes in data without significant performance impact.

Common Types of Balanced Search Trees:

1. **AVL Trees:**
   - Balance Factor: Difference in height between a node's left and right subtrees is at most 1.
   - Insertion/Deletion: AVL rotations (single or double) to maintain balance.
   - Pros: Strict balance, good for frequent insertions/deletions.
   - Cons: Rotations can be slightly more complex.
2. **Red-Black Trees:**
   - Color Property: Nodes are colored red or black, following specific rules.
   - Insertion/Deletion: Recoloring and rotations (left, right, or color flips) to maintain balance.
   - Pros: Less strict balance, often simpler rotations, good for frequent insertions/deletions.
   - Cons: More complex rules for colors and rotations.
3. **2-3 Trees:**
   - Multi-way Trees: Nodes can have 2 or 3 children.
   - Insertion/Deletion: Splitting and merging nodes to maintain balance.
   - Pros: Very strict balance, good for large datasets.
   - Cons: More complex structure and implementation.

Searching Algorithms:

- Same as in regular BSTs: Traverse the tree based on key comparisons.
- Balanced nature ensures logarithmic time complexity.

Insertion/Deletion Algorithms:

- BST-like insertion/deletion: Find position, insert/remove node.
- Additional rebalancing operations to maintain tree balance after changes.

Hashing:

- A technique for efficiently storing and retrieving data items using a key-value mapping.
- It involves converting keys into indices (called hash values or hash codes) that map to specific locations in a data structure called a hash table.

Hash Tables:

- Data structures that implement hashing.
- Typically consist of an array of buckets, where each bucket can hold multiple key-value pairs.
- Hash functions determine the bucket where a key-value pair is stored.

Hash Functions:

- Algorithms that take a key as input and produce a hash value (an integer).
- Key properties:
  - Deterministic: Same input key always produces the same hash value.
  - Uniform distribution: Hash values should be evenly distributed across the range of possible indices to minimize collisions.

Example Hash Function (Division Method):

Python

```
def hash_function(key, table_size):
    return key % table_size
```

Collisions:

- Occur when different keys map to the same hash value (bucket).
- Handling techniques:
  - Separate chaining: Each bucket is a linked list to store multiple key-value pairs.
  - Open addressing: Probe for alternative empty buckets using collision resolution strategies (linear probing, quadratic probing).

Advantages of Hashing:

- Average-case O(1) time complexity for insertion, deletion, and searching.
- Efficient for large datasets.
- Adaptable to dynamic changes in data.

Disadvantages of Hashing:

- Worst-case O(n) time complexity if collisions are poorly handled.
- Hash function design is crucial for performance.
- Not suitable for range-based queries or ordered retrieval.

Common Applications:

- Dictionaries and sets in programming languages.
- Caches and in-memory data stores.
- Database indexing.
- Password storage (using secure hashing algorithms).
- File systems (for file name lookup).
- Network routing tables.
- Many other areas involving fast data retrieval.

some common types of hash functions with examples:

1. **Division Method:**
   - Divides the key by a constant (usually the table size) and takes the remainder as the hash value.
   - Simple and efficient, but can lead to uneven distribution if keys cluster around multiples of the divisor.
   - Example:

Python

```
def division_hash(key, table_size):
    return key % table_size
```

2. **Multiplication Method:**
   - Multiplies the key by a constant (usually a fraction between 0 and 1) and takes the fractional part as the hash value.
   - Can provide better distribution than division method, but can be sensitive to the choice of the constant.
   - **Example:**

Python

```
def multiplication_hash(key, table_size):
    A = 0.6180339887  # Golden ratio
    return int(table_size * (key * A % 1))
```

3. **Universal Hashing:**
   - Employs a family of hash functions where any two keys have a high probability of mapping to different hash values, even if the keys are chosen adversarially.
   - Stronger guarantees for collision resistance, but often more computationally expensive.
   - Example:

Python

```
def universal_hash(key, table_size, p=101):
    a = random.randint(1, p - 1)
    b = random.randint(0, p - 1)
    return ((a * key + b) % p) % table_size
```

4. **Folding Method:**
   - Divides the key into segments, adds or multiplies the segments, and takes the resulting value as the hash value.
   - Can be effective for keys with known patterns or structure.
   - **Example:**

Python

```
def folding_hash(key, table_size):
    sum = 0
    for char in key:
        sum += ord(char)  # Sum character codes
    return sum % table_size
```

5. **Mid-Square Method:**
   - Squares the key, extracts a central portion of the result, and uses it as the hash value.
   - Less common due to potential for uneven distribution and overflow issues.
   - **Example:**

Python

```
def mid_square_hash(key, table_size):
    squared = key * key
    center = squared // 2  # Assuming even-length keys
    return center % table_size
```

Hash table vs direct access tables (arrays):

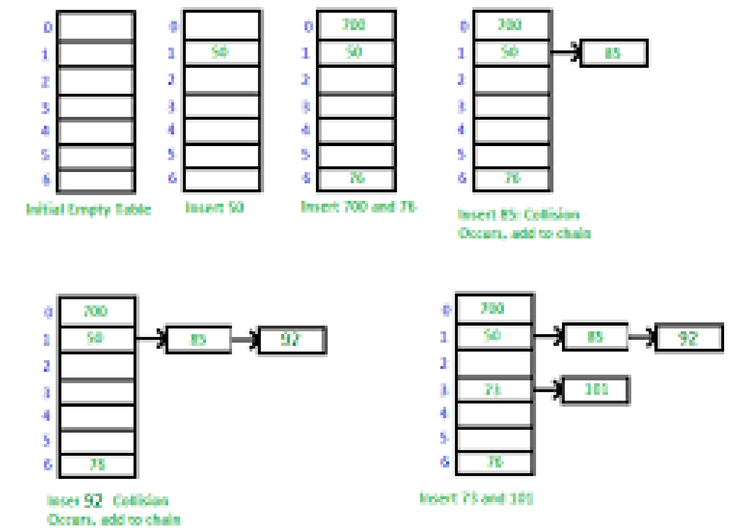
hash tables often outperform direct access tables in terms of:

- Efficiency for large datasets and frequent operations.
- Adaptability to dynamic changes in data.
- Memory efficiency for sparse data.

However, direct access tables can be simpler to implement and are suitable for:

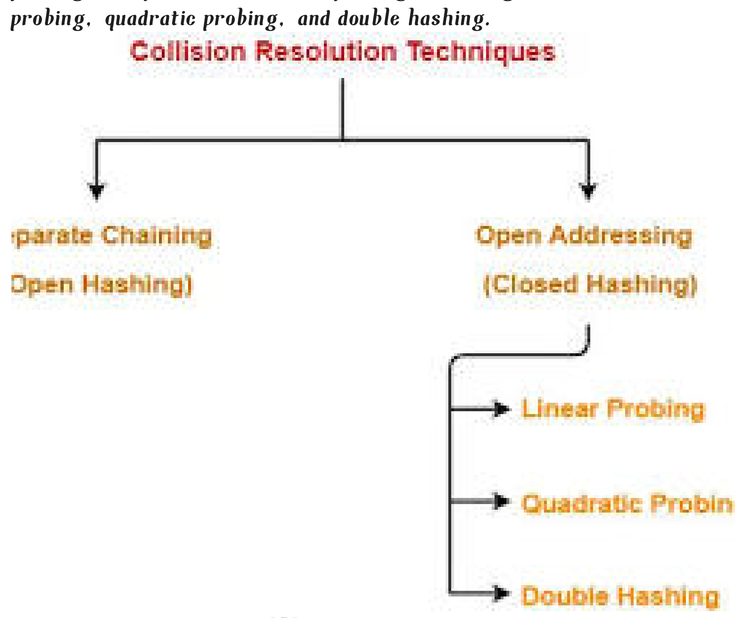
- Basic data storage with known indices.
- Ordered retrieval and range-based queries.

1. **Separate Chaining:**
   - This technique uses linked lists within each bucket to store elements with the same hash value. When a collision occurs, the new element is simply added to the end of the corresponding linked list. Searching involves traversing the linked list to find the desired element.



2. **Open Addressing:**

This technique keeps all elements within the hash table itself, without using any additional data structures. When a collision occurs, the new element is probed for an alternative empty bucket based on a predefined probing sequence. Common probing strategies include linear probing, quadratic probing, and double hashing.



Opens in a new window www.gatevidyalay.com

Open Addressing Collision Resolution Technique in Hashing

3. **Cuckoo Hashing:**

This technique uses two hash tables instead of one. When a collision occurs in the primary table, the new element is inserted into the secondary table using its secondary hash value. If a collision occurs in the secondary table, the element that was displaced is moved back to the primary table using its primary hash value. This process continues until both elements find empty slots.

sample data

REGISTRATION NUMBER	Linear probing (probes)	Quadratic probing (probes)
KUST/SCI/05/356	0	0
KUST/SCI/05/214	4	2
KUST/SCI/05/117	0	0
KUST/SCI/05/714	0	0
KUST/SCI/05/735	1	1
KUST/SCI/05/821	0	0
KUST/SCI/05/434	2	3
KUST/SCI/05/578	1	1

Opens in a new window www.semanticscholar.org

Cuckoo Hashing Collision Resolution Technique in Hashing

Choosing the Right Technique:

The best collision resolution technique for your hash table depends on several factors, including:

- Expected data set size and density: Separate chaining might be better for sparse data, while open addressing might be better for dense data.
- Performance requirements: If fast search times are critical, separate chaining might be preferable. If memory footprint is a concern, open addressing might be better.
- Complexity: Separate chaining is simpler to implement, while cuckoo hashing is more complex.

Graph:

- A non-linear data structure that represents a set of objects (called vertices or nodes) and their relationships (called edges or arcs).
- Visualized as a collection of points connected by lines or curves.
- Used to model a wide range of real-world scenarios involving interconnected entities.

Types of Graphs:

1. **Directed Graph (Digraph):**
   - Edges have a specific direction, indicating a one-way relationship.
   - Example: A social network where following someone doesn't imply being followed back.
   -
2. **Undirected Graph:**

- Edges have no direction, representing a two-way relationship.
- Example: A road network where travel between cities is possible in both directions.
-

3. **Weighted Graph:**
   - Edges have associated weights (numerical values), representing costs, distances, or other measures.
   - Example: A map where distances between cities are marked.
   -
4. **Cyclic Graph:**
   - Contains at least one cycle, a closed path that starts and ends at the same node.
   - Example: A flowchart where a decision can lead back to an earlier step.
   -
5. **Acyclic Graph:**
   - Contains no cycles.
   - Example: A family tree where no person can be their own ancestor.
   -
6. **Connected Graph:**
   - Every pair of nodes is connected by a path.
   - Example: A group of friends where everyone knows each other directly or indirectly.
7. **Disconnected Graph:**
   - Not all pairs of nodes are connected by a path.
   - Example: A social network where different groups of friends don't interact.
8. **Complete Graph:**
   - Every pair of nodes is connected directly by an edge.
   - Example: A round-robin tournament where every team plays every other team.
9. **Bipartite Graph:**
   - Nodes can be divided into two sets such that no edges exist within a set.
   - Example: A matching problem where people are matched to jobs.
10. **Tree:**
    - A special type of acyclic graph with a single root node and unique paths to all other nodes.
    - Example: A file system hierarchy or a decision tree.

Paths:

- A path in a graph is a sequence of vertices connected by edges, where each vertex is visited only once.
- It represents a way to travel from one vertex to another within the graph.
- Example: In a map graph, a path could represent a route between two cities.
- Types of Paths:
  - Simple Path: A path that doesn't repeat any vertices.
  - Shortest Path: The path with the minimum total weight (in weighted graphs).

Cycles:

- A cycle is a closed path that starts and ends at the same vertex, visiting other vertices exactly once in between.
- It represents a loop within the graph.
- Example: In a social network graph, a cycle could represent a group of friends who are all connected to each other.

Types of Cycles:

- Simple Cycle: A cycle that doesn't repeat any vertices or edges except for the starting/ending vertex.
- Hamiltonian Cycle: A cycle that visits every vertex in the graph exactly once (not all graphs have Hamiltonian cycles).

Spanning Trees:

- A spanning tree of a graph is a subgraph that includes all of the graph's vertices and a subset of its edges, forming a tree structure.
- It connects all vertices without any cycles.
- Example: In a computer network graph, a spanning tree could represent the minimum set of connections needed for all devices to communicate.
- Types of Spanning Trees:
  - Minimum Spanning Tree (MST): The spanning tree with the minimum total edge weight (in weighted graphs).
  - Shortest Path Tree: A spanning tree rooted at a specific vertex, where the paths from the root to all other vertices are the shortest paths in the graph.

Topological Sorting Algorithm:

What it does:

- Arranges the vertices of a directed acyclic graph (DAG) in a linear order such that for every directed edge (u, v), vertex u comes before vertex v in the ordering.
- Ensures that dependencies between vertices are respected.

Algorithm:

1. **Initialize:**
   - Create an empty list to store the sorted order.
   - Indegree: Number of incoming edges for each vertex.
2. **Find vertices with indegree 0:**
   - These have no incoming dependencies, so they can be processed first.
3. **Process vertices with indegree 0:**
   - For each vertex with indegree 0:
     - Add it to the sorted list.
     - Decrement indegree of its outgoing neighbors.
4. **Repeat until all vertices are processed:**
   - Find new vertices with indegree 0 (after previous processing).
   - Process them as in step 3.
5. **Detect cycles:**
   - If any vertices remain with non-zero indegree after all possible processing, the graph has a cycle and topological sorting is not possible.

minimum spanning tree (MST)
is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles, and with the minimum possible total edge weight. In simpler terms, it's the most cost-effective way to connect all the nodes in the graph while avoiding loops.
Key Properties:

- Connects all vertices: Every vertex in the graph is reachable from any other vertex through the edges in the MST.
- No cycles: There are no closed paths formed by the edges in the MST.
- Minimum total weight: The sum of the weights of all edges in the MST is the smallest possible among all spanning trees of the graph.

Prim's Algorithm:

- Purpose: Finds the minimum spanning tree (MST) of a connected, weighted, undirected graph.
- Approach: Grows the MST from a starting vertex by iteratively adding the cheapest edge that connects the existing tree to new vertices.

Algorithm Steps:

1. **Initialize:**
   - Choose any starting vertex.
   - Create a set of vertices in the MST (initially just the starting vertex).
   - Create a set of edges in the MST (initially empty).
2. Repeat until all vertices are in the MST:
   - Find the cheapest edge that connects a vertex in the MST to a vertex not yet in the MST.
   - Add this edge to the MST.
   - Add the new vertex to the set of vertices in the MST.

Example 1: Simple Graph
Consider a graph with vertices A, B, C, D, E and edges with weights:

- AB = 2
- AC = 3
- BC = 5
- BD = 7
- CD = 4
- DE = 1

Steps:

1. Start at A (arbitrary choice).
2. Add edge AB (cheapest edge) and vertex B to MST.
3. Add edge DE (next cheapest) and vertex E to MST.
4. Add edge AC (next cheapest) and vertex C to MST.
5. Add edge CD (next cheapest) to MST (completing MST).

MST: A-B-E-C-D with total weight 6.
Kruskal's Algorithm:
Purpose: Finds the minimum spanning tree (MST) of a connected, weighted, undirected graph. Approach: Sorts all edges by weight and iteratively adds them to the MST if they don't create a cycle, starting with the cheapest edges.
Algorithm Steps:

1. **Sort edges:** Sort all edges in the graph in non-decreasing order of their weights.
2. **Initialize MST:** Create an empty set to store the edges of the MST.
3. **Iterate through edges:**

For each edge (u, v) in the sorted order:
If adding (u, v) to the MST would not create a cycle:
Add (u, v) to the MST.

4. Stop when MST is complete: Continue until the MST contains V-1 edges (where V is the number of vertices).

Example 1: Simple Graph
Consider the same graph with vertices A, B, C, D, E and edge weights as in the Prim's Algorithm example.
Steps:

1. Sort edges: DE (1), AB (2), AC (3), CD (4), BC (5), BD (7).
2. Add DE to MST.
3. Add AB to MST.
4. Add AC to MST.
5. Skip BC (creates cycle).
6. Add CD to MST (completing MST).

MST: A-B-E-C-D with total weight 6 (same as Prim's).

Shortest Path Algorithms are a family of algorithms designed to find the shortest path between two vertices (or nodes) in a graph. They have extensive applications in various fields, including:
- Routing in networks: Finding the most efficient routes for data packets in computer networks or for vehicles in navigation systems.
Logistics and supply chain management: Optimizing delivery routes to minimize travel time and costs.
Travel planning: Finding the quickest or cheapest routes between multiple destinations.
Gaming: Pathfinding for characters in video games.
Social network analysis: Measuring distance and influence between individuals in social networks.

Dijkstra's Algorithm:
Purpose:
Finds the shortest paths from a single source vertex to all other vertices in a weighted, directed or undirected graph, where the weights are non-negative.
Key Steps:
Initialization:
Create two sets:
unvisited: All vertices in the graph.
visited: Initially empty.
Assign a tentative distance value to each vertex:
0 for the source vertex.
Infinity for all other vertices.
Repeat until all vertices are visited:

- Find the unvisited vertex with the smallest tentative distance.
- Mark it as visited and move it from unvisited to visited.
- For each of its unvisited neighbors:

Calculate the tentative distance to the neighbor through the current vertex.
If this new tentative distance is smaller than the neighbor's current tentative distance, update it.
Algorithm Visualized:

graph with vertices and edge weights, showing the steps of Dijkstra's algorithm
Time Complexity:
O(V^2) for dense graphs using an adjacency matrix.
O(E log V) for sparse graphs using a priority queue to efficiently select the vertex with the smallest tentative distance in each iteration.
Example:
Consider a graph with vertices A, B, C, D, E and edge weights:

- AB = 2
- AC = 5
- BC = 4
- BD = 1
- CD = 8
- DE = 3

Applying Dijkstra's algorithm from source A:
1. Visit A, distances: A(0), B(2), C(5), D(∞), E(∞)
2. Visit B, distances: A(0), B(2), C(4), D(1), E(∞)
3. Visit D, distances: A(0), B(2), C(4), D(1), E(3)
4. Visit E, distances: A(0), B(2), C(4), D(1), E(3)
5. Visit C, distances: A(0), B(2), C(4), D(1), E(3)

Shortest paths from A: A-B-D (2), A-B-D-E (4), A-C (4)
The Bellman-Ford algorithm is another powerful tool for finding shortest paths in graphs, but it offers some unique capabilities compared to Dijkstra's algorithm. Here's a breakdown of its key characteristics:
Purpose:
Finds the shortest paths from a single source vertex to all other vertices in a weighted, directed graph, even if the graph contains negative edge weights.
Key Differences from Dijkstra's Algorithm:
Handles Negative Weights: Unlike Dijkstra's, Bellman-Ford can handle graphs with negative edge weights, making it suitable for situations where costs or distances can decrease as you progress.
Cycle Detection: It can also detect the presence of negative weight cycles, which can create infinitely decreasing paths and render finding a true shortest path impossible
.
Algorithm Steps:
Initialization:
Similar to Dijkstra's, create sets of unvisited and visited vertices and assign tentative distances.
Set all distances to infinity initially.
Relaxation Iterations:
Repeat for a specific number of iterations (typically the number of vertices) or until no distances change:
For each unvisited vertex:
Relax all its incoming edges: check if the tentative distance through any incoming edge is smaller than the current tentative distance, and update if necessary.
Negative Cycle Detection:
After the iterations, if any distances decrease further, the graph contains a negative weight cycle.
Time Complexity:
O(VE), where V is the number of vertices and E is the number of edges. This can be slower than Dijkstra's O(E log V) for non-negative weights.
Example:
Consider a graph with vertices A, B, C, D, E and edge weights:

- AB = 2
- AC = -1
- BC = 4
- BD = -5
- CD = 8
- DE = 3

Applying Bellman-Ford from A:

- Iteration 1: A(0), B(2), C(1), D(3), E(∞)
- Iteration 2: A(0), B(1), C(0), D(-2), E(3)
- Iteration 3: A(0), B(1), C(0), D(-7), E(3) (detects negative weight cycle)

the Floyd-Warshall Algorithm, designed to find all-pairs shortest paths in a weighted graph:

Purpose:

Computes the shortest paths between all pairs of vertices in a weighted, directed graph (can also be applied to undirected graphs).

Handles both positive and negative edge weights (but not negative weight cycles).

Key Steps:

Initialization:

Create a distance matrix D where D[i][j] represents the current shortest known distance from vertex i to vertex j.

Initialize D with the direct edge weights (if an edge exists) or infinity (if no direct edge).

Iteratively consider intermediate vertices:

For each intermediate vertex k (from 1 to V):

For each pair of vertices i and j:

Check if the path i -> k -> j is shorter than the current shortest path i -> j:

If so, update D[i][j] with the new shorter distance.

Algorithm Visualized:

graph with vertices and edge weights, showing the steps of FloydWarshall algorithm

Time Complexity:

O(V^3), where V is the number of vertices.

Example:

Consider a graph with vertices A, B, C, D and edge weights:

- AB = 4
- AC = 2
- BC = 5
- BD = 1
- CD = 3

Applying Floyd-Warshall:

Initial D:

0 A B C D A | 0 4 2 ∞ B | ∞ 0 5 1 C | ∞ ∞ 0 3 D | ∞ ∞ ∞ 0

After considering k = A:

0 A B C D A | 0 4 2 4 B | ∞ 0 5 1 C | ∞ ∞ 0 3 D | 8 ∞ 7 4

... (after considering k = B, C, D)

Final D contains shortest paths between all pairs.

Rabin-Karp Algorithm, a string searching algorithm known for its efficiency in multiple pattern searching:

Purpose:

Efficiently finds occurrences of a pattern string within a larger text string, especially when searching for multiple patterns.

Key Features:

Rolling hash function: Uses a hash function to quickly compare portions of the text with the pattern, avoiding character-by-character comparisons in most cases.

Multiple pattern searching: Excels at finding multiple patterns simultaneously, making it useful for tasks like plagiarism detection or virus scanning.

Algorithm Steps:

Preprocessing:

Calculate the hash value of the pattern string using a rolling hash function (e.g., Rabin fingerprint).

Text scanning:

Calculate the hash value of the first substring of the text with the same length as the pattern.

If the hash values match, perform a character-by-character comparison to confirm a true match.

If not, slide the window one character to the right and recalculate the hash value, repeating until the end of the text.

Multiple patterns:

For multiple patterns, store their hash values in a hash table for efficient lookups.

Time Complexity:

Average case: O(n + m), where n is the text length and m is the pattern length.

Worst case (rare): O(nm), when hash collisions occur frequently.

Advantages:

Fast average-case performance, especially for multiple patterns.

Handles multiple patterns efficiently.

Disadvantages:

- Can have a worst-case performance of O(nm) in rare cases with hash collisions.
- Requires careful choice of hash function to minimize collisions.

Key Points:

- Well-suited for multiple pattern searching and large text files.
- Efficiency depends on the quality of the hash function.
- Can be adapted for approximate string matching.

Common Applications:

- Plagiarism detection
- Virus scanning
- DNA sequence analysis
- Text compression
- Intrusion detection systems