1. Searching Algorithms

Linear Search

- **Idea:**
  Sequentially checks each element in the array until the target element is found or the array ends.

- **Steps:**
    - Start from the first element of the array.
    - Compare the target element with the current element.
    - If a match is found, return its position.
    - If the array ends without finding the target, return "not found."
- **Example:**
  Array: `[4, 2, 9, 7, 3]`, Target: `7`
    - Start with the first element 4, not equal to 7.
    - Move to 2, not equal to 7.
    - Check 9, not equal to 7.
    - Check 7, found a match.
    - **Output:** Position = 4 (1-based index).
- **Time Complexity:**
    - Best case: O(1).
    - Worst case: O(n).

Binary Search

- **Idea:**
  Search a sorted array by repeatedly dividing it into two halves and eliminating the half where the element cannot exist.

- **Steps:**
    - Start with two pointers: `low` (start) and `high` (end).
    - Find the middle element.

- If the middle element matches the target, return its position.
- If the target is smaller, search the left half; if larger, search the right half.
- Repeat until `low > high`.

- **Example:**

  Array: `[2, 4, 6, 8, 10, 12]`, Target: `8`

  - Initial: `low = 0`, `high = 5`, `mid = 2`.
  - Compare `array[mid] = 6` with `8`. Move to the right half (`low = 3`).
  - New `mid = 4`. Compare `array[mid] = 8` with `8`.
  - **Output:** Position = 4 (0-based index).

- **Time Complexity:**

  - Best case: O(1).
  - Worst case: O(log n).

Bubble Sort

- **Idea:**

  Repeatedly swaps adjacent elements if they are in the wrong order until the array is sorted.

- **Steps:**

  - Start with the first element and compare it to the next.
  - Swap if the first is greater than the second.
  - Repeat this process for the entire array for multiple passes until no swaps occur.

- **Example:**

  Array: `[5, 3, 8, 4]`

  - Pass 1: Compare `5 & 3` (swap), `[3, 5, 8, 4]`; Compare `8 & 4` (swap), `[3, 5, 4, 8]`.
  - Pass 2: Compare `5 & 4` (swap), `[3, 4, 5, 8]`.
  - **Output:** Sorted Array = `[3, 4, 5, 8]`.

- **Time Complexity:**

  - Best case: O(n).
  - Worst case: $O(n^2)$.

---

2. Sorting Algorithms

Insertion Sort

- **Idea:**
  Build a sorted portion of the array one element at a time.

- **Steps:**
  - Start with the second element.
  - Compare it with elements before it to find its correct position.
  - Shift larger elements one position to the right.
  - Insert the current element in its correct position.
  - Repeat for all elements.
- **Example:**
  Array: `[7, 3, 5]`
  - Take `3` and compare it with `7` (shift). Insert `3`. Array becomes `[3, 7, 5]`.
  - Take `5` and compare it with `7` (shift). Insert `5`. Array becomes `[3, 5, 7]`.
  - **Output:** Sorted Array = `[3, 5, 7]`.
- **Time Complexity:**
  - Best case: O(n).
  - Worst case: O(n^2).

Selection Sort

- **Idea:**
  Repeatedly select the smallest element from the unsorted part and move it to the sorted part.

- **Steps:**
  - Start with the first element.
  - Find the smallest element in the unsorted portion.
  - Swap it with the first element of the unsorted portion.
  - Move the boundary of the sorted portion by one.
  - Repeat until the array is sorted.

- **Example:**
  Array: `[8, 3, 1, 7]`
  - Find the smallest (1) and swap with the first (8). `[1, 3, 8, 7]`.
  - Find the smallest in `[3, 8, 7]` (3) and keep it. `[1, 3, 8, 7]`.
  - Repeat for the rest.
  - **Output:** Sorted Array = `[1, 3, 7, 8]`.
- **Time Complexity:**
  - $O(n^2)$.

Quick Sort

- **Idea:**
  Use a pivot element to partition the array into two halves such that elements in one half are smaller than the pivot and elements in the other are larger.

- **Steps:**
  - Pick a pivot element.
  - Partition the array such that smaller elements are to its left and larger to its right.
  - Recursively apply the same logic to the left and right partitions.
- **Example:**
  Array: `[8, 4, 7, 3]`, Pivot: 4
  - Partition: `[3]`, Pivot: 4, `[8, 7]`.
  - Recursively sort: `[3]`, `[7, 8]`.
  - Merge: `[3, 4, 7, 8]`.
  - **Output:** `[3, 4, 7, 8]`.
- **Time Complexity:**
  - Best case: $O(n \log n)$.
  - Worst case: $O(n^2)$.

---

3. Divide and Conquer Algorithms

Merge Sort

- **Idea:**
  Divide the array into halves, recursively sort them, and merge the sorted halves.

- **Steps:**
  - Divide the array into two halves.
  - Recursively sort each half.
  - Merge the sorted halves into a single sorted array.
- **Example:**
  Array: `[5, 3, 8, 6]`
  - Divide: `[5, 3]` and `[8, 6]`.
  - Sort: `[3, 5]` and `[6, 8]`.
  - Merge: `[3, 5, 6, 8]`.
  - **Output:** `[3, 5, 6, 8]`.
- **Time Complexity:**
  - O(n log n).

Binary Heap

- **Idea:**
  A binary heap is a complete binary tree used to implement priority queues, where the parent node either has a higher priority (max-heap) or lower priority (min-heap) than its children.

- **Steps to Build a Heap:**
  - Insert elements level-wise in a binary tree structure.
  - Ensure the heap property (either max-heap or min-heap) by comparing each node with its children and swapping as needed (heapify).
  - For sorting, extract the root (highest/lowest priority) repeatedly and heapify the remaining tree.
- **Example (Max-Heap):**
  Array: `[4, 10, 3, 5, 1]`

Step 1: Build heap:

```
    10
   / \
  5   3
 / \
4   1
```

-

Step 2: Extract `10` and rebuild heap:

```
    5
   / \
  4   3
 /
1
```

-
- Repeat until sorted array: `[1, 3, 4, 5, 10]`.

- **Time Complexity:**
  - Build heap: O(n).
  - Heapify: O(log n).
  - Sorting: O(n log n).

Heap Sort

- **Idea:**
  A sorting algorithm that uses a binary heap to repeatedly extract the largest/smallest element and sorts the array.

- **Steps:**
  - Build a max-heap from the array.
  - Swap the root with the last element of the heap.
  - Reduce the heap size and heapify the root.
  - Repeat until the heap is empty.

- **Example:**
  Array: `[3, 8, 5, 4]`
  - Step 1: Build max-heap: `[8, 4, 5, 3]`.
  - Step 2: Swap root `8` with last element `3`: `[3, 4, 5, 8]`.
  - Step 3: Rebuild heap for remaining elements: `[5, 4, 3]`.
  - Repeat until sorted: `[3, 4, 5, 8]`.
- **Time Complexity:**
  - O(n log n).

---

4. Greedy Algorithms

Activity Selection Problem

- **Idea:**
  Select the maximum number of non-overlapping activities that can be completed.
- **Steps:**
  - Sort activities by their finishing times.
  - Select the first activity and add it to the solution.
  - For each subsequent activity, check if its start time is greater than or equal to the finish time of the last selected activity.
  - If yes, select the activity.
- **Example:**
  Activities: `{(1, 2), (3, 4), (0, 6), (5, 7), (8, 9)}` (start, finish)
  - Sort by finish time: `{(1, 2), (3, 4), (5, 7), (8, 9), (0, 6)}`.
  - Select `(1, 2)`, then `(3, 4)`, then `(5, 7)`, and finally `(8, 9)`.
  - **Output:** `{(1, 2), (3, 4), (5, 7), (8, 9)}`.
- **Time Complexity:**
  - O(n log n) (due to sorting).

Fractional Knapsack

- **Idea:**
  Maximize the total value of items that can fit into a knapsack of given capacity, allowing fractional items.

- **Steps:**
  - Calculate the value-to-weight ratio for each item.
  - Sort items in decreasing order of this ratio.
  - Add items to the knapsack starting from the highest ratio, taking fractions if necessary.
- **Example:**
  Items: `{(value, weight)} = {(60, 10), (100, 20), (120, 30)}`, Capacity = 50
  - Ratio: `6, 5, 4`.
  - Take full `(60, 10)` and `(100, 20)` and `2/3` of `(120, 30)`.
  - **Output:** Total value = `60 + 100 + 80 = 240`.
- **Time Complexity:**
  - O(n log n) (due to sorting).

Huffman Coding

- **Idea:**
  Build an optimal binary prefix code for characters based on their frequencies.

- **Steps:**
  - Create a priority queue with nodes for each character and its frequency.
  - While the queue has more than one node:
    - Remove two nodes with the smallest frequencies.
    - Merge them into a new node with a combined frequency.
    - Add the new node back to the queue.
  - The final tree represents the Huffman encoding.
- **Example:**
  Characters: `{A: 5, B: 9, C: 12, D: 13, E: 16, F: 45}`

Build tree:

```
     100
    /  \
  45    55
       / \
     25   30
    /\   /\
   A  B C  D
```

-
- Encoding: `A: 1100, B: 1101, ....`
- **Time Complexity:**
   - O(n log n).

5. String Algorithms

Naive String Matching Algorithm

- **Idea:** The simplest method to find all occurrences of a pattern in a text. It checks for the pattern starting from every position in the text.
- **Steps:**
   - Iterate through each character in the text up to the point where the remaining characters are less than the pattern's length.
   - For each starting position, compare the pattern with the substring of the text.
   - If all characters match, record the position as a match.
- **Example:**
   - Text: `"AABAACAADAABAABA"`
   - Pattern: `"AABA"`
   - Process:
      - Compare starting from index `0`: Match.
      - Index `1`: No match.
      - Index `9`: Match.
      - **Output:** Pattern found at indices `0`, `9`, `12`.

- **Time Complexity:**
    - Worst case: O(nm), where `n` is the text length and `m` is the pattern length.

Rabin-Karp Algorithm

- **Idea:** Uses hashing to find patterns. Instead of checking all characters, it compares hash values of the pattern and substrings.
- **Steps:**
    - Compute the hash value of the pattern and the first substring of the text with the same length.
    - Slide the window one character at a time, updating the hash.
    - If the hash values match, check the characters to confirm.
    - Continue until the end of the text.
- **Example:**
    - Text: `"GEEKS FOR GEEKS"`
    - Pattern: `"GEEK"`
    - Process:
        - Compute hash for `"GEEK"` and the first 4 characters of the text.
        - Slide and compare.
        - **Output:** Pattern found at indices `0`, `10`.
- **Time Complexity:**
    - Average case: O(n + m).
    - Worst case: O(nm) due to hash collisions.

Z Algorithm

- **Idea:** Computes the Z-array, where each element `Z[i]` stores the length of the substring starting at `i` that matches the prefix.
- **Steps:**

- Build the Z-array for the combined string `Pattern + $ + Text`.
- For each index in the Z-array corresponding to the text, check if the value equals the pattern length.
- If true, it's a match.

- **Example:**
  - Text: `"ABABAB"`
  - Pattern: `"AB"`
  - Combined string: `"AB$ABABAB"`.
  - Z-array: `[0, 0, 0, 2, 0, 2, 0, 2]`.
  - **Output:** Pattern found at indices `0, 2, 4`.
- **Time Complexity:**
  - O(n + m).

KMP Algorithm (Knuth-Morris-Pratt)

- **Idea:** Avoids redundant comparisons by precomputing a prefix-suffix table (LPS array).
- **Steps:**
  - Build the LPS (Longest Prefix Suffix) array for the pattern.
  - Traverse the text, using the LPS array to skip unnecessary comparisons.
  - If a match occurs, report it and continue from the LPS value.
- **Example:**
  - Text: `"ABABDABACDABABCABAB"`
  - Pattern: `"ABABCABAB"`
  - LPS array: `[0, 0, 1, 2, 0, 1, 2, 3, 4]`.
  - **Output:** Pattern found at index `10`.
- **Time Complexity:**
  - O(n + m).

Manacher's Algorithm (Longest Palindromic Substring)

- **Idea:** Efficiently finds the longest palindromic substring by using the symmetry of palindromes.
- **Steps:**
  - Preprocess the string by inserting delimiters (e.g., `#`) to handle even-length palindromes.
  - Use a center and right boundary to expand around potential centers and compute palindromic lengths.
  - Track the maximum length and position.
- **Example:**
  - Text: `"abacdfgdcaba"`
  - Preprocessed: `"#a#b#a#c#d#f#g#d#c#a#b#a#"`
  - Palindromic lengths: `[0, 1, 0, 1, 0, ...]`.
  - **Output:** Longest palindrome: `"aba"` or `"aca"`.
- **Time Complexity:**
  - O(n).