DESIGN ANALYSIS AND ALGORITHM

CIA 1 TOPICS

Time Complexity Analysis & Recursion	**Time Complexity Analysis – Mathematical Analysis of Recursive and Non Recursive algorithms**
Searching	Linear, Binary search and Bubble sort
Sorting 1	Insertion sort, Selection sort, quick sort
Sorting 2	Divide and Conquer: mergesort , Binary Heap and Heap sort
Greedy Algorithm	Activity selection problem
Greedy Algorithms	Fractional knapsack,Huffman Coding
String algorithms	Naive algorithm, Rabin Karp algorithm,Z algorithm
String Algorithms	KMP algorithm,Manacher’s algorithm

ACTIVITY SELECTION PROBLEM

Problem Statement

Given n activities with their start and finish times, select the maximum number of activities that can be performed by a single person, assuming that a person can work on only one activity at a time.

Algorithm Explanation

The problem is solved using a Greedy Algorithm. The main idea is to select activities based on their finish times, prioritizing those that finish the earliest, as they leave the maximum room for subsequent activities.

ACTIVITY SELECTION PROBLEM

Steps to Solve

1. Sort Activities by Finish Time
    - Arrange all activities in ascending order of their finish times.
2. Initialize Variables
    - Select the first activity since it finishes earliest.
    - Keep track of the finish time of the last selected activity.
3. Iterate Through Activities
    - For each activity, check if its start time is greater than or equal to the finish time of the last selected activity.
    - If true, select the activity and update the last finish time.
4. Repeat Until End
    - Continue the process for all activities.
5. Output
    - The list of selected activities.

ACTIVITY SELECTION PROBLEM

Detailed Example

Input:

Activities:

- Activity 1: Start = 1, Finish = 2
- Activity 2: Start = 3, Finish = 4
- Activity 3: Start = 0, Finish = 6
- Activity 4: Start = 5, Finish = 7
- Activity 5: Start = 8, Finish = 9
- Activity 6: Start = 5, Finish = 9

ACTIVITY SELECTION PROBLEM

Steps:

1. Sort Activities by Finish Time
2. Sorted activities (based on finish time):
    - Activity 1: (1, 2)
    - Activity 2: (3, 4)
    - Activity 3: (0, 6)
    - Activity 4: (5, 7)
    - Activity 5: (8, 9)
    - Activity 6: (5, 9)
3. Select the First Activity
    - Select Activity 1: (1, 2)
    - Update last finish time: 2.
    - Activity 6: Start = 5 < Last Finish = 9 → Skip.

4. Iterate Through Remaining Activities
    - Activity 2: Start = 3 ≥ Last Finish = 2 → Select.
    - Update last finish time: 4.
    - Activity 3: Start = 0 < Last Finish = 4 → Skip.
    - Activity 4: Start = 5 ≥ Last Finish = 4 → Select.
    - Update last finish time: 7.
    - Activity 5: Start = 8 ≥ Last Finish = 7 → Select.
    - Update last finish time: 9.
    - Activity 6: Start = 5 < Last Finish = 9 → Skip.

5. Selected Activities:
    - Activity 1: (1, 2)
    - Activity 2: (3, 4)
    - Activity 4: (5, 7)
    - Activity 5: (8, 9)

Output:

The maximum number of activities is 4, and the selected activities are:

- (1, 2), (3, 4), (5, 7), (8, 9).

ACTIVITY SELECTION PROBLEM

We are given six activities with their start and finish times:

Activity	Start Time	Finish Time
A1	1	2
A2	3	4
A3	0	6
A4	5	7
A5	8	9
A6	5	9

Step-by-Step Visualization

1. Sort the Activities by Finish Time
    - We arrange the activities in ascending order of their finish time:

A1: (1, 2), A2: (3, 4), A3: (0, 6), A4: (5, 7), A5: (8, 9), A6: (5, 9)

2. Start Selection

- Select A1 because it finishes the earliest.
- Highlight A1:

  Selected: A1 (1, 2)

ACTIVITY SELECTION PROBLEM

3. Iterate Through Remaining Activities

- A2: Start = 3 ≥ Last Finish = 2 → Select A2:

Selected: A1 (1, 2), A2 (3, 4)

- A3: Start = 0 < Last Finish = 4 → Skip A3.
- A4: Start = 5 ≥ Last Finish = 4 → Select A4:

Selected: A1 (1, 2), A2 (3, 4), A4 (5, 7)

- A5: Start = 8 ≥ Last Finish = 7 → Select A5:

Selected: A1 (1, 2), A2 (3, 4), A4 (5, 7), A5 (8, 9)

- A6: Start = 5 < Last Finish = 9 → Skip A6.

ACTIVITY SELECTION PROBLEM

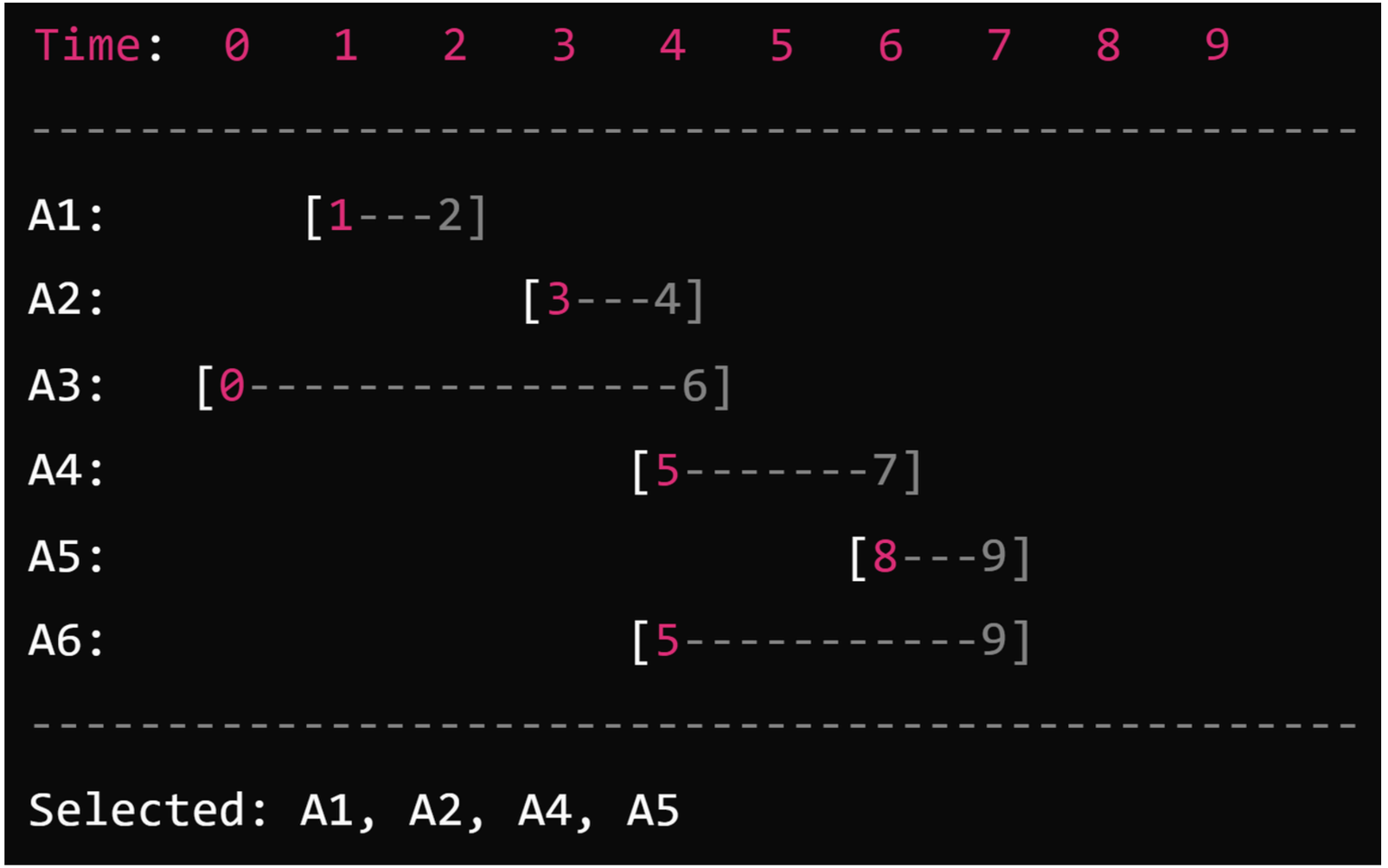
Timeline Visualization

Final Output

The selected activities are:

- A1: (1, 2)
- A2: (3, 4)
- A4: (5, 7)
- A5: (8, 9)

Maximum number of activities: 4.



FRACTIONAL KNAPSACK PROBLEM

Problem Statement

You are given n items, each with a weight (wi) and value (vi). Your goal is to fill a knapsack with a maximum weight capacity (W) to achieve the maximum total value. You can take fractions of an item if its full weight does not fit into the knapsack.

Algorithm Explanation

This problem is solved using the Greedy Algorithm. The key is to maximize the value at each step by choosing items based on their value-to-weight ratio (vi/wi).

FRACTIONAL KNAPSACK PROBLEM

Steps of the Algorithm

1. Compute Value-to-Weight Ratios:
    - Calculate the ratio vi/wi for each item.
2. Sort Items:
    - Sort the items in descending order based on their value-to-weight ratios.
3. Fill the Knapsack:
    - Iterate through the sorted items:
        - If the item's weight fits completely in the remaining capacity, add it to the knapsack.
        - If it doesn't fit, take a fraction of the item that fills the knapsack.
4. Stop When Full:
    - Stop when the knapsack is full or all items have been considered.

FRACTIONAL KNAPSACK PROBLEM

Example

Given Data:

Item	**Weight ($w\ i$)**	**Value ($v\ i$)**	**Value-to-Weight Ratio ($v\ i$ / $w\ i$)**
1	10	60	6
2	20	100	5
3	30	120	4

FRACTIONAL KNAPSACK PROBLEM

Steps:

1. **Calculate Ratios**:
   - v1/w1=60/10=6v_1/w_1 = 60/10 = 6v1/w1=60/10=6
   - v2/w2=100/20=5v_2/w_2 = 100/20 = 5v2/w2=100/20=5
   - v3/w3=120/30=4v_3/w_3 = 120/30 = 4v3/w3=120/30=4
2. **Sort by Ratios**:
   - Order: Item 1 → Item 2 → Item 3.

FRACTIONAL KNAPSACK PROBLEM

1. **Fill the Knapsack**:
    - Start with capacity $W=50W = 50W=50$.
    - **Select Item 1**:
        - Weight: 10 (fits completely).
        - Value: 60.
        - Remaining capacity: $50−10=4050 - 10 = 4050−10=40$.
    - **Select Item 2**:
        - Weight: 20 (fits completely).
        - Value: 100.
        - Remaining capacity: $40−20=2040 - 20 = 2040−20=20$.
    - **Select a Fraction of Item 3**:
        - Available weight: 20.
        - Fraction: $20/30=2/320/30 = 2/320/30=2/3$.
        - Value: $(2/3)×120=80(2/3)\times 120 = 80(2/3)×120=80$.

FRACTIONAL KNAPSACK PROBLEM

1. Final Output:
    a. Total Value: 60+100+80=24060 + 100 + 80 = 24060+100+80=240.
    b. Items Taken:
        i. Item 1: Full (10/10).
        ii. Item 2: Full (20/20).
        iii. Item 3: 20/3020/3020/30.

FRACTIONAL KNAPSACK PROBLEM

Visualization

Data Representation:

Item	Ratio (vi/wi)	Full Taken	Fraction
Item 1	6	Yes	100%
Item 2	5	Yes	100%
Item 3	4	No	66.67%

Knapsack Timeline:

Initial Capacity: 50

Step 1: Add Item 1 (10/10)

Remaining Capacity: 40

Total Value: 60

Step 2: Add Item 2 (20/20)

Remaining Capacity: 20

Total Value: 160

Step 3: Add 2/3 of Item 3 (20/30)

Remaining Capacity: 0

Total Value: 240

HUFFMAN CODING

Problem Statement

Huffman Coding is an **optimal prefix coding algorithm** used for lossless data compression. The algorithm assigns variable-length binary codes to input characters based on their frequencies, with shorter codes assigned to more frequent characters.

Algorithm Explanation

Huffman Coding uses a **Greedy Algorithm** to build an optimal binary tree (Huffman Tree) from character frequencies. Each character is represented as a leaf in this tree, and binary codes are assigned by traversing the tree.

HUFFMAN CODING

Steps of the Algorithm

1. **Frequency Table**:
    - Count the frequency of each character in the input.
2. **Create Nodes**:
    - Treat each character as a node, with its frequency as the weight.
3. **Build the Huffman Tree**:
    - Place all nodes in a priority queue (min-heap), sorted by frequency.
    - Repeatedly:
        - Remove two nodes with the smallest frequencies.
        - Create a new parent node with a frequency equal to the sum of the two.
        - Add the parent node back to the heap.
    - Continue until one node remains (the root of the Huffman Tree).

- **Generate Codes**:
    - Assign binary codes to each character:
        - Traverse left: Add 0.
        - Traverse right: Add 1.
- **Encode the Input**:
    - Replace each character with its binary code.

HUFFMAN CODING

Example

Input:

- String: "aaabbc"
- Frequency Table:

Character	Frequency
a	3
b	2
c	1

HUFFMAN CODING

Steps:

1. **Create Nodes**:

Initial nodes:

a (3), b (2), c (1)

1. **Build the Huffman Tree**:

Combine nodes c and b:

Node: cb (3)

Combine cb and a:

Root Node: a_cb (6)

Final Tree:

Root (6)

/ \

a (3) cb (3)

/ \

c (1) b (2)

1. **Generate Codes**:

Traverse the tree:

a: 0

c: 10

b: 11

- **Encode the Input**:

Replace each character

with its code:

"aaabbc" → "00011110"

Output

- Encoded String:

  "00011110"

- Character Codes:
  - a: 0
  - b: 11
  - c: 10

HUFFMAN CODING

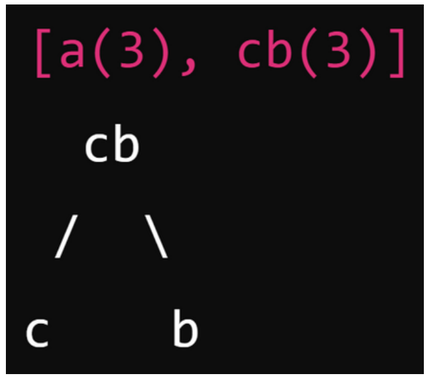
Visualization

Tree Construction

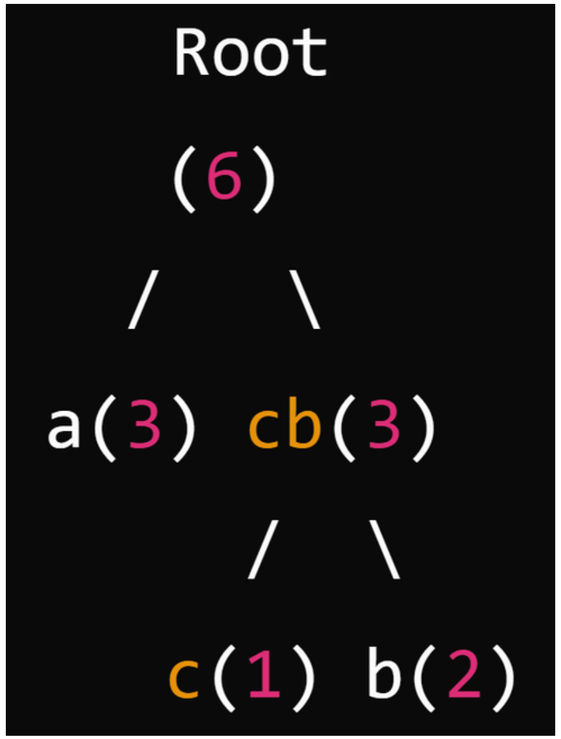
1. Initial:

[a(3), b(2), c(1)]

2. Combine c and b:



3. Combine a and cb:



Code Assignment

- Traverse the tree:
  - Left child: Add 0.
  - Right child: Add 1.

Encoded String Formation

- Replace characters in the string "aaabbc":

a → 0

b → 11

c → 10

Encoded String: 00011110

NAIVE STRING MATCHING ALGORITHM

Problem Statement

The Naive String Matching Algorithm is a straightforward method for finding all occurrences of a pattern P in a text T. It checks for a match by aligning the pattern at every possible position in the text.

Algorithm Explanation

The algorithm works by sliding the pattern P over the text T one character at a time and checking for matches at each alignment. This is why it’s called "naive"—it does not use any optimization

NAIVE STRING MATCHING ALGORITHM

Steps of the Algorithm

1. **Input**:
   - Text T of length n.
   - Pattern P of length m.
2. **Sliding Window**:
   - Start from the first character of T and align P.
   - Compare characters of P with the corresponding characters in T.
3. **Match Check**:
   - If all characters of P match in the current alignment, record the starting index of the match.
4. **Shift**:
   - Shift P one position to the right and repeat the comparison.
5. **End**:
   - Stop when P can no longer fit into T (i.e., when n−m+1n - m + 1 alignments are done).

NAIVE STRING MATCHING ALGORITHM

Example

Input:

- Text T="AABAACAADAABAAABAA}
- Pattern P="AABA"

Steps:

Alignment 1: Compare T[0..3] with P:

Text: AABAACAADAABAAABAA

Pattern: AABA

Indices: 01234567890123456 7

Match found at index 0.

Alignment 2: Compare T[1..4] with P:

Text: AABAACAADAABAAABAA

AABA

Indices: 01234567890123456 7

No match.

Alignment 3: Compare T[2..5] with P:

Text: AABAACAADAABAAABAA

AABA

Indices: 01234567890123456 7

No match.

Alignment 4: Compare T[3..6] with P:

Text: AABAACAADAABAAABAA

AABA

Indices: 01234567890123456 7

^^^^

No match.

Alignment 5: Compare T[4..7] with P:

Text: AABAACAADAABAAABAA

AABA

Indices: 01234567890123456 7

^^^^

No match.

NAIVE STRING MATCHING ALGORITHM

Alignment 6: Compare T[9..12] with P:

Text: AABAACAADAABAAABAA

AABA

Indices: 0123456789012345 67

^^^^

Match found at index 9.

Alignment 7: Compare T[13..16] with P:

Text: AABAACAADAABAAABAA

AABA

Indices: 0123456789012345 67

^^^^

Match found at index 13.

Output

The pattern P is found at indices: 0, 9, 13.

NAIVE STRING MATCHING ALGORITHM

Visualization

Text: AABAACAADAABAAABAA

Pattern: AABA

Indices: 01234567890123456 7

Step 1: Match at index 0

Step 2: No match at index 1

Step 3: No match at index 2

Step 4: No match at index 3

Step 5: No match at index 4

Step 6: Match at index 9

Step 7: Match at index 13

RABIN-KARP ALGORITHM

Problem Statement

Given a text T and a pattern P, find all occurrences of P in T using the Rabin-Karp algorithm. This algorithm leverages a hash function to compare the pattern with substrings of the text.

Algorithm

1. Calculate the hash value of the pattern P.
2. Compute the hash value of the first window of text T.
3. Slide the window across the text:
    - If the hash values of the window and the pattern match, compare the actual characters to confirm the match.
    - Compute the hash value for the next window using a rolling hash technique.
4. Continue until the end of the text is reached.

RABIN-KARP ALGORITHM

Steps to Solve

1. Define a hash function for computing hash values of strings.
2. Calculate the hash of the pattern and the initial window in the text.
3. Slide the pattern window:
    - If the hash matches, check characters to verify.
    - Use a rolling hash to efficiently compute the hash for the next window.
4. Return the indices of all matches.

RABIN-KARP ALGORITHM

Example

Text: GEEKS FOR GEEKS

Pattern: GEEK

Steps:

1. Pattern hash: Compute the hash for GEEK.
2. Compute the initial hash of the first window (GEEKS).
3. Slide the pattern window:
    - Compare hash values. If equal, verify characters.
    - Update the hash using a rolling hash for the next window.
4. Matches occur at indices 0 and 10.

Algorithm Explanation

- Hash Function: Typically, a polynomial hash is used.
- Rolling Hash: Update the hash by adding the new character and removing the old character using modular arithmetic.
- Efficiency: Hash comparisons are constant time, making the average-case complexity O(n+m)O(n + m)O(n+m), where nnn is the text length and mmm is the pattern length.

RABIN-KARP ALGORITHM

Step 1: Define a Hash Function

The core idea of Rabin-Karp is to represent both the pattern and substrings of the text as hash values. Typically, a polynomial rolling hash function is used.

Hash Function

For a string S of length m, its hash value can be computed as:

Hash(S)=(S[0]⋅p0+S[1]⋅p1+...+S[m−1]⋅pm−1)mod q\text{Hash}(S) = (S[0] \cdot p^0 + S[1] \cdot p^1 + \dots + S[m-1] \cdot p^{m-1}) \mod qHash(S)=(S[0]⋅p0+S[1]⋅p1+...+S[m−1]⋅pm−1)modq

Where:

- p is a constant (usually a small prime number, e.g., 31).
- q is a large prime number to avoid overflow and ensure hash uniqueness.

The hash function generates a numerical representation for the string, which will allow quick comparisons.

RABIN-KARP ALGORITHM

Step 2: Calculate the Hash of the Pattern P

We start by computing the hash value of the pattern P that we want to search for in the text T.

For example:

- Pattern: P = "GEEK"
- We calculate its hash value using the above hash formula.

Step 3: Compute the Hash of the First Window in the Text T

Now, we need to compare the pattern P with substrings of the text T. We start by computing the hash value of the first window (substring) of T that is the same length as the pattern.

For example:

- Text: T = "GEEKS FOR GEEKS"
- Initial window: T[0..m-1] = "GEEK"

We calculate the hash for this first window, using the same hash function as we did for the pattern.

RABIN-KARP ALGORITHM

Step 4: Slide the Pattern Window Over the Text

Now, we slide the pattern window one character at a time across the text.

For each new window:

1. We compare the hash values of the current window and the pattern.
2. If the hash values match, we then compare the actual characters to check for a real match (this step handles hash collisions).

RABIN-KARP ALGORITHM

Step 4: Slide the Pattern Window Over the Text

Now, we slide the pattern window one character at a time across the text. For each new window:

1. We compare the hash values of the current window and the pattern.
2. If the hash values match, we then compare the actual characters to check for a real match (this step handles hash collisions).

Z ALGORITHM EXPLANATION

Z Algorithm Explanation

The Z Algorithm is an efficient string-matching algorithm that computes the Z-array for a given string. The Z-array is used to find occurrences of a pattern within a text, and the algorithm runs in O(n) time, where n is the length of the string. The Z-array is a helpful structure because it provides information about the longest common prefix between the pattern and substrings of the text.

Problem Statement

Given a text T and a pattern P, find all occurrences of P in T using the Z algorithm. We concatenate the pattern P and the text T with a special delimiter and then compute the Z-array for the concatenated string. The Z-array for a string S is an array of integers where Z[i] is the length of the longest substring starting from S[i] that is also a prefix of S.

Z ALGORITHM EXPLANATION

Steps to Solve the Problem Using Z Algorithm

Step 1: Concatenate the Pattern and Text

We concatenate the pattern P, a delimiter (which is a character that does not appear in either P or T), and the text T to form a new string S. For example:

$$S=P+\text{delimiter}+T$$

This ensures that we can compute the Z-array in one pass without having to compute the Z-values separately for P and T.

Z ALGORITHM EXPLANATION

Step 2: Compute the Z-array for the Concatenated String

We now compute the Z-array for the concatenated string S. To compute the Z-array efficiently, we use the following approach:

1. Initialize two variables:
    - l (left boundary of the current Z-box)
    - r (right boundary of the current Z-box) These boundaries define the segment of the string where the Z-values are already computed.
2. Iterate through the string:
    - For each position i, check if it lies outside the current Z-box (i.e., i > r). If so, compute the Z-value from scratch by comparing S[i] with S[i + j] and keep increasing j until we find a mismatch or reach the end of the string.

Z ALGORITHM EXPLANATION

- If i lies within the current Z-box, use the previously computed Z-values to speed up the computation. Specifically, if i <= r, you can start comparing from Z[i-l] and check the substring starting from S[r+1] to avoid redundant comparisons.
- Update the Z-box boundaries: If the new Z-value extends beyond r, update l and r.

Z ALGORITHM EXPLANATION

Step 3: Find Matches in the Text

Once we have the Z-array, we can find occurrences of the pattern P in the text T. For each index i in the Z-array, if Z[i] equals the length of the pattern P, then the pattern occurs at the position i - (m + 1) in the text T, where m is the length of the pattern.

Example Walkthrough

Let’s go through an example to demonstrate the Z Algorithm step by step.

Example

Text: T = "GEEKS FOR GEEKS"

Pattern: P = "GEEK"

Step 1: Concatenate the Pattern and Text

We concatenate the pattern P, a delimiter "$", and the text T to get:

S = "GEEK$GEEKS FOR GEEKS”

Z ALGORITHM EXPLANATION

Step 2: Compute the Z-array for S

Now, we compute the Z-array for the string S = "GEEK$GEEKS FOR GEEKS". Here's how we do it:

1. Initialize Z[0] = 0 (the first character of the string always has a Z-value of 0).
2. Start with l = 0 and r = 0.

Now, process each character:

- i = 1: Compare S[1] with S[0], which gives Z[1] = 0 because they don't match.
- i = 2: Compare S[2] with S[0] again, which gives Z[2] = 1 because S[2] = S[0] = 'G'.
- i = 3: Compare S[3] with S[1] and S[2]. We find that Z[3] = 2 because the substring "GE" matches the prefix "GE".
- i = 4: At S[4], we reach the delimiter "$". The Z-value will be Z[4] = 0 because the delimiter is not a match.

Z ALGORITHM EXPLANATION

- i = 5: Compare S[5] with S[0], and it matches, so Z[5] = 1.
- i = 6: Continue comparing, and find Z[6] = 2 because the substring "GE" matches the pattern prefix.
- i = 7: Compare S[7] with S[0], and the substring "G" matches, so Z[7] = 1.
- i = 8: Finally, for the last part of the string, we find Z[8] = 4, indicating that the substring "GEEK" matches the prefix of the string.

At the end of this process, we get the Z-array for S:

Z=[0,0,1,2,0,1,2,1,4,0,1,2,3,4,0,1]

Z ALGORITHM EXPLANATION

Step 3: Find Matches

We want to find all occurrences of the pattern "GEEK" in the text. The length of the pattern m = 4, so we check the Z-array starting from index m + 1 = 5 onward.

We look for any index i where Z[i] = 4 (the length of the pattern). In this case, Z[8] = 4 and Z[12] = 4.

Thus, the pattern "GEEK" occurs at indices i - (m + 1) in the text:

- For Z[8] = 4, the match is at index 8 - (4 + 1) = 3.
- For Z[12] = 4, the match is at index 12 - (4 + 1) = 7.

Visualization

Let's visualize the Z-array computation for the string S = "GEEK$GEEKS FOR GEEKS":

String: G E E K $ G E E K S F O R G E E K S F O R G E E K S

Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Z-Array: [0, 0, 1, 2, 0, 1, 2, 1, 4, 0, 1, 2, 3, 4, 0, 1]

Z ALGORITHM EXPLANATION

Step 3: Find Matches

We want to find all occurrences of the pattern "GEEK" in the text. The length of the pattern m = 4, so we check the Z-array starting from index m + 1 = 5 onward.

We look for any index i where Z[i] = 4 (the length of the pattern). In this case, Z[8] = 4 and Z[12] = 4.

Thus, the pattern "GEEK" occurs at indices i - (m + 1) in the text:

- For Z[8] = 4, the match is at index 8 - (4 + 1) = 3.
- For Z[12] = 4, the match is at index 12 - (4 + 1) = 7.

Visualization

Let's visualize the Z-array computation for the string S = "GEEK$GEEKS FOR GEEKS":

String: G E E K $ G E E K S F O R G E E K S F O R G E E K S

Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Z-Array: [0, 0, 1, 2, 0, 1, 2, 1, 4, 0, 1, 2, 3, 4, 0, 1]

KMP ALGORITHM (KNUTH-MORRIS-PRATT)

The KMP Algorithm is an efficient string-matching algorithm that avoids unnecessary comparisons by using partial matching information. It preprocesses the pattern to create a prefix table (also called the failure function) which is then used to skip over parts of the text that have already been matched.

Problem Statement for KMP

Given a text T and a pattern P, find all occurrences of P in T using the KMP algorithm. The KMP algorithm preprocesses the pattern to avoid re-evaluating characters that have already been compared, leading to faster matching.

KMP ALGORITHM (KNUTH-MORRIS-PRATT)

Steps to Solve the Problem Using KMP Algorithm

Step 1: Preprocess the Pattern to Create the Prefix Table

The prefix table is an array LPS (Longest Prefix Suffix) where LPS[i] gives the length of the longest proper prefix of the substring P[0..i] that is also a suffix of P[0..i]. The value of LPS[i] indicates how much of the pattern we can skip when a mismatch occurs.

1. Initialize LPS[0] = 0 because a single character has no proper prefix.
2. For each subsequent character in the pattern, check the longest prefix-suffix match and fill in the LPS array.

KMP ALGORITHM (KNUTH-MORRIS-PRATT)

Step 2: Use the LPS Array to Match the Text

1. Start comparing characters of the text T with the pattern P from the beginning.
2. If a mismatch occurs and the prefix table suggests that part of the pattern has already been matched, shift the pattern accordingly without rechecking the previously matched part.
3. If a match is found, move to the next character of both the text and the pattern.

Step 3: Find Matches

Once a complete match of the pattern with a substring of the text is found, record the index of the match and continue the comparison using the LPS table to skip already checked portions.

KMP ALGORITHM (KNUTH-MORRIS-PRATT)

Example Walkthrough for KMP

Text: T = "ABABDABACDABABCABAB"

Pattern: P = "ABABCABAB"

Step 1: Compute the Prefix Table

We compute the LPS (Longest Prefix Suffix) array for the pattern:

1. P[0] = "A": No prefix or suffix, so LPS[0] = 0.
2. P[1] = "B": No match, so LPS[1] = 0.
3. P[2] = "A": Match prefix "A" with suffix, so LPS[2] = 1.
4. P[3] = "B": Match "AB", so LPS[3] = 2.
5. P[4] = "C": No match, so LPS[4] = 0.
6. P[5] = "A": Match "A" again, so LPS[5] = 1.
7. P[6] = "B": Match "AB", so LPS[6] = 2.
8. P[7] = "A": Match "ABA", so LPS[7] = 3.
9. P[8] = "B": Match "ABAB", so LPS[8] = 4.

KMP ALGORITHM (KNUTH-MORRIS-PRATT)

So, the LPS array is:

LPS = [0, 0, 1, 2, 0, 1, 2, 3, 4]

Step 2: Match the Pattern with the Text

1. Start comparing the first character of the pattern with the text.
2. If a mismatch occurs, use the LPS array to shift the pattern and skip redundant comparisons.
3. Continue until the pattern matches or the end of the text is reached.

Step 3: Find Matches

We find that the pattern "ABABCABAB" occurs in the text at index 10.

KMP ALGORITHM (KNUTH-MORRIS-PRATT)

KMP Algorithm Visualization

Let’s visualize the KMP matching process for T = "ABABDABACDABABCABAB" and P = "ABABCABAB":

Text: A B A B D A B A C D A B A B C A B A B

Pattern: A B A B C A B A B

LPS Array: 0 0 1 2 0 1 2 3 4

The algorithm compares characters efficiently, skipping unnecessary comparisons using the LPS array.

MANACHER’S ALGORITHM

Manacher's Algorithm is a linear-time algorithm used to find the longest palindromic substring in a given string. lt works by expanding palindromes around each center while avoiding redundant calculations using a clever trick called mirror expansion.

Problem Statement for Manacher’s Algorithm

Given a string S, find the longest palindromic substring in linear time using Manacher’s Algorithm.

MANACHER'S ALGORITHM

Steps to Solve the Problem Using Manacher's Algorithm

Step 1: Transform the String

To handle even-length palindromes and simplify the process, we transform the string by inserting a special character (e.g., #) between every character of the string and at the ends. This ensures that we treat both even and odd-length palindromes uniformly.

For example, transform S = "abac" into T = "#a#b#a#c#".

Step 2: Initialize Arrays

1. P array: The P[i] array stores the length of the palindrome centered at index i in the transformed string.
2. C: The center of the rightmost palindrome found so far.
3. R: The right edge of the rightmost palindrome found so far.

MANACHER'S ALGORITHM

Step 3: Expand Around Centers

Iterate through the transformed string. For each position i, attempt to expand the palindrome centered at i by comparing characters symmetrically around i. If you reach a mismatch or the bounds of the string, stop.

If a palindrome at i extends past the right boundary R, update C and R.

Step 4: Use Mirror Expansion

For each position i, we check the mirror of i relative to C (i.e., mirror = 2*C - i). If the palindrome at mirror is within the bounds of the current palindrome, we can use it to avoid redundant checks.

Step 5: Find the Longest Palindrome

The maximum value in the P array gives the length of the longest palindromic substring.

MANACHER'S ALGORITHM

Example Walkthrough for Manacher's Algorithm

Input String: S = "babad"

Step 1: Transform the String

We transform the string S = "babad" into:

T = "#b#a#b#a#d#"

Step 2: Initialize Arrays

- P = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
- C = 0
- R = 0

Step 3: Expand Around Centers

- Start with i = 1 (center a):
  - Compare T[0] and T[2], they are equal, so P[1] = 1.
- Move to i = 2 (center b):
  - Compare T[1] and T[3], they are equal, so P[2] = 1.
- Continue expanding and updating P.

Step 4: Find Longest Palindrome

At the end of the expansion, the largest value in P will correspond to the longest palindrome. In this case, the longest palindrome is "bab" (length 3), centered at index 2.

MANACHER'S ALGORITHM

Manacher's Algorithm Visualization

Input: b a b a d

Transformed: # b # a # b # a # d #

P array: [0, 1, 1, 3, 1, 1, 3, 1, 1, 1, 0]

Here, the palindrome of maximum length is "bab" with a center at position 2.