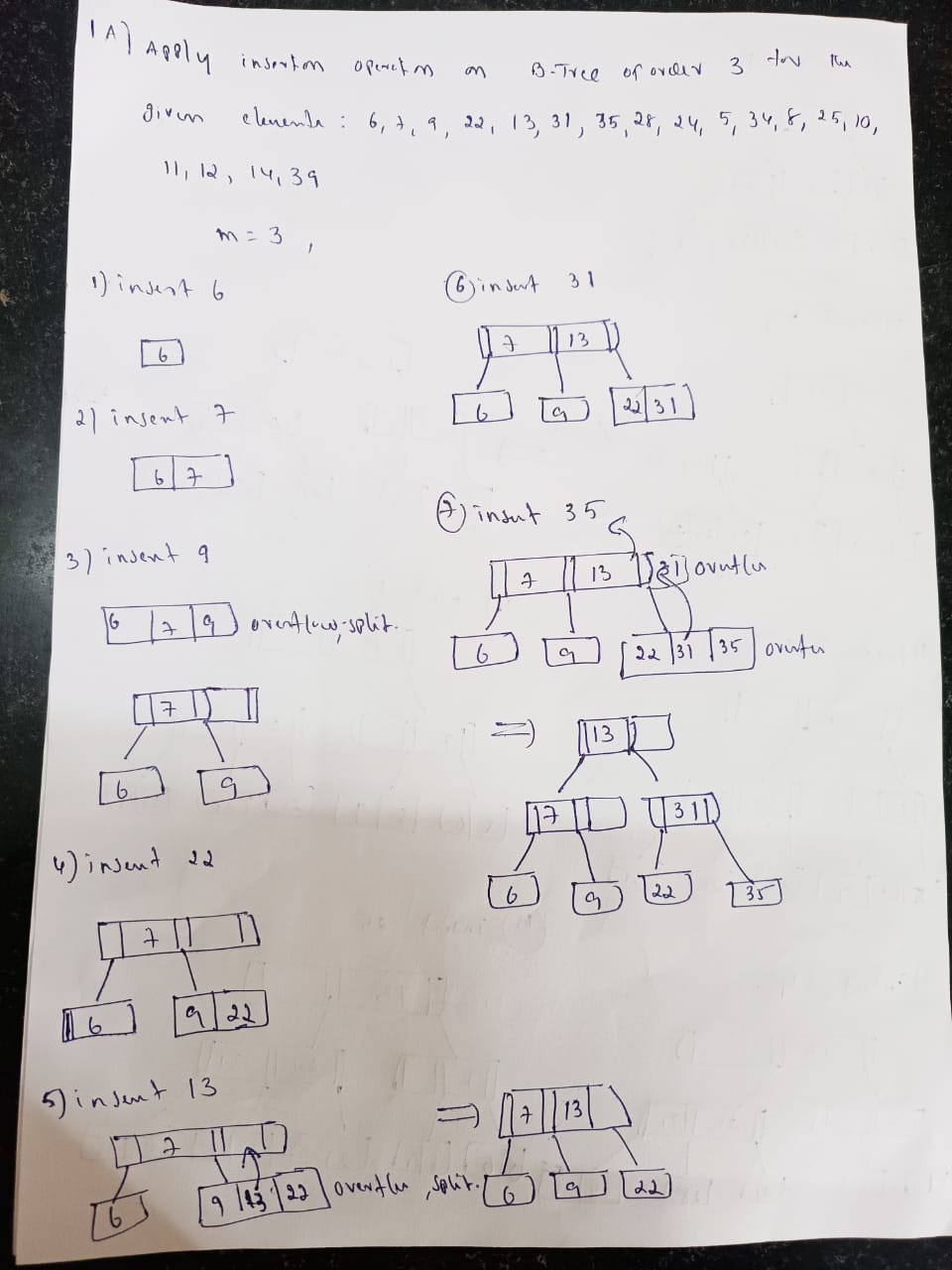
**P.V.P Siddhartha Institute of Technology**

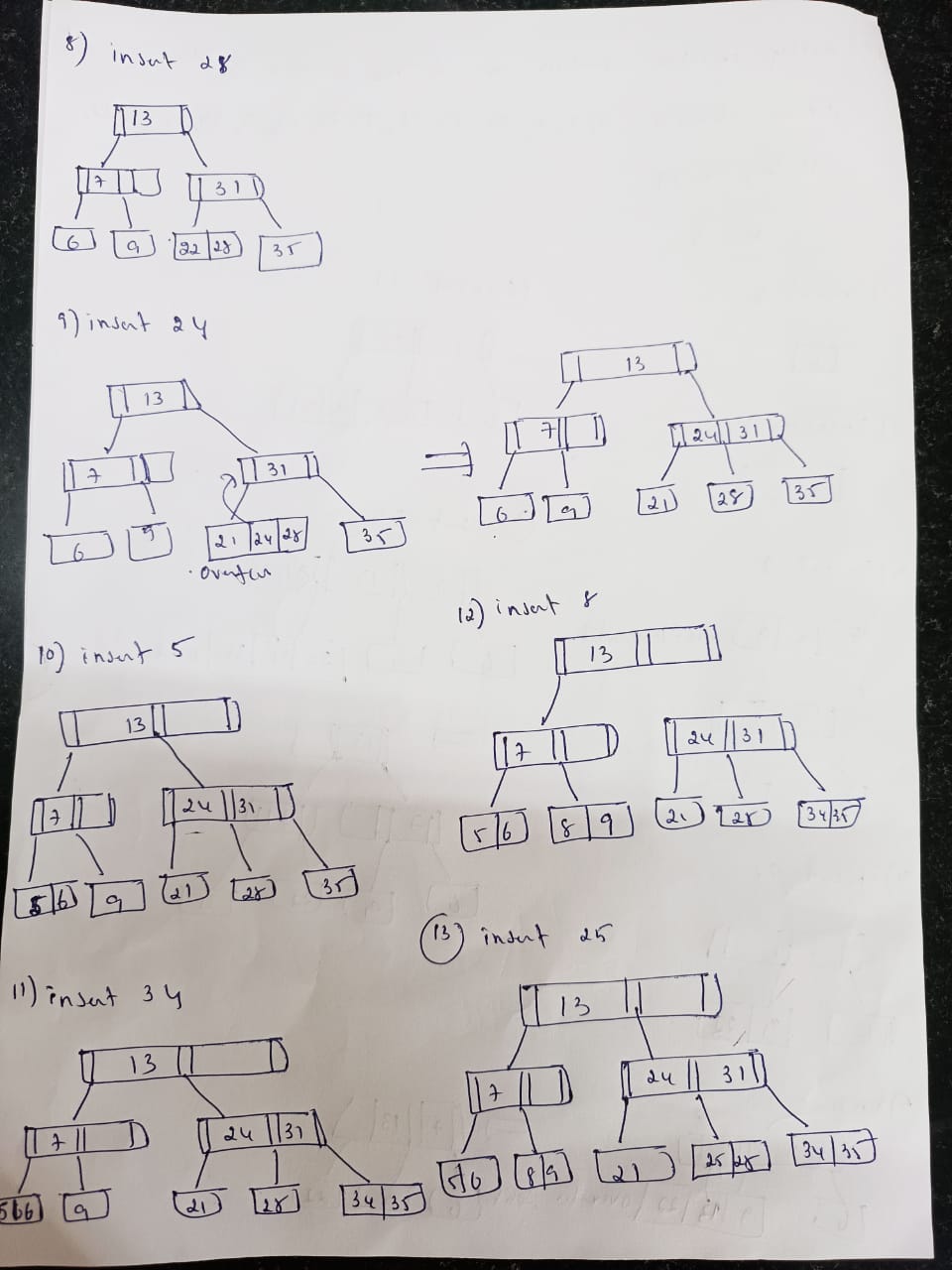
**Department of Computer Science and Engineering**

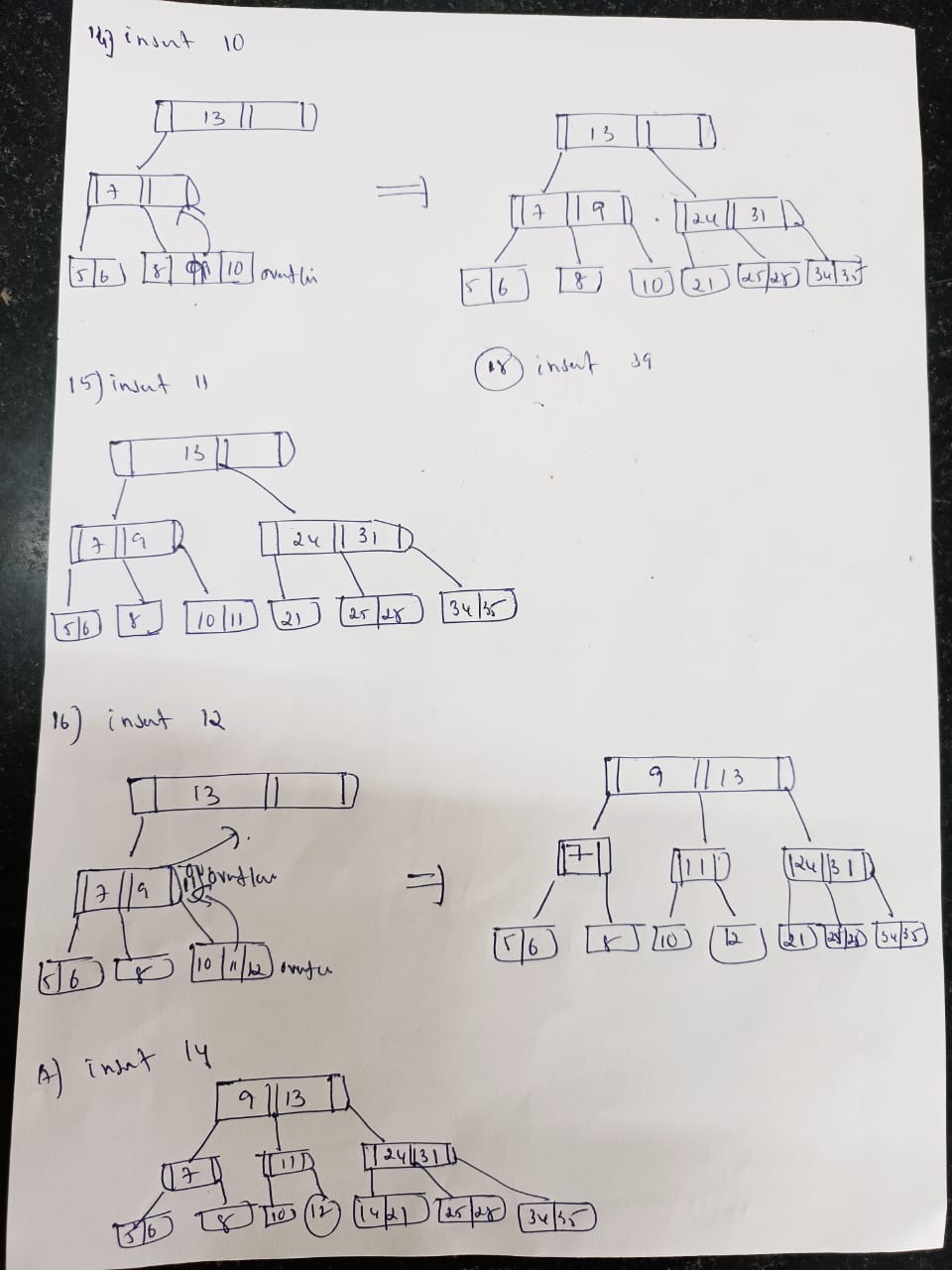
**Subject Code: 20CS3402, Subject Name: Advanced Data Structures, Regulation:PVP20**

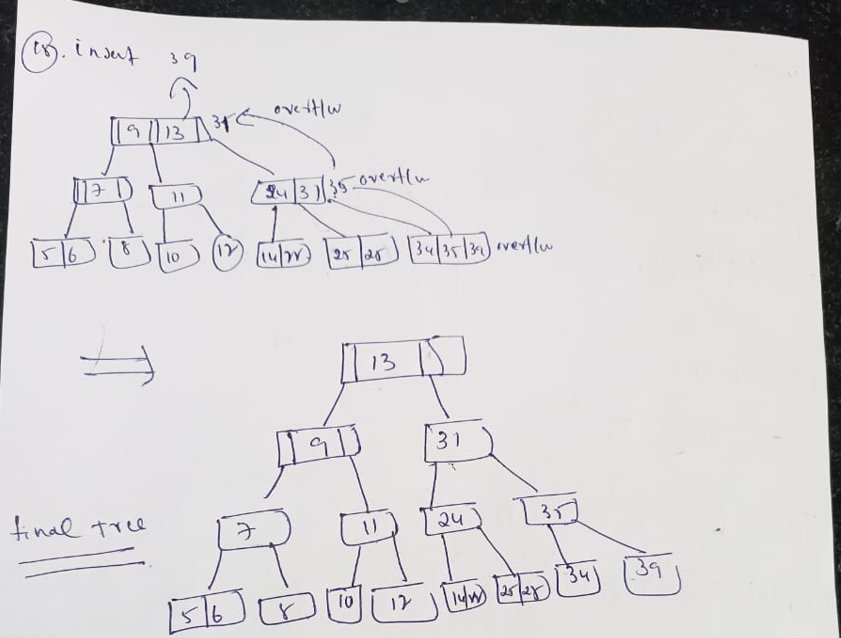
**Descriptive Examination -2**

**1 a)** Apply insertion operation on B- Tree of order 3 for the given elements 6, 7, 9, 22, 13, 31, 35, 28, 24, 5, 34, 8, 25, 10, 11, 12, 14, 39, CO3 L3 4M







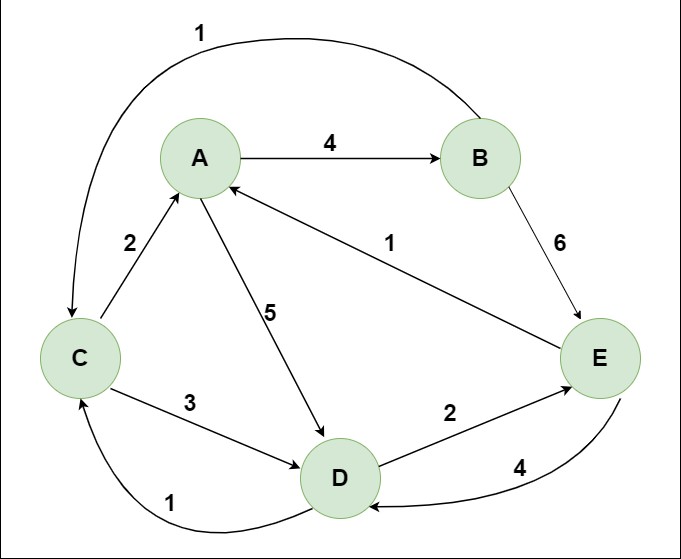


B) Explain pattern matching and give its applications CO2, L2 1M

Pattern Matching is the process of finding a specific sequence of characters (called a pattern) within a larger text or dataset.

**Applications of Pattern Matching: Any two applications**

1. **Text Search Engines (e.g., Google Search):**
2. **DNA/Protein Sequence Analysis (Bioinformatics):**
3. **Spam Filtering and Intrusion Detection:**
4. **Plagiarism Detection:**
5. **Compiler Design (Lexical Analysis):**

2 A) Apply all pairs shortest path algorithm on the given graph CO3, L3 4M

**Initial Setup**

**A = 0, B = 1, C = 2, D = 3, E = 4**

We initialize the distance matrix D[5][5] using the weights from the graph. If there's no direct edge, we use ∞ (infinity). If i = j, the distance is 0.

**Initial Distance Matrix (D₀):**

| **From\To** | **A(0)** | **B(1)** | **C(2)** | **D(3)** | **E(4)** |
| --- | --- | --- | --- | --- | --- |
| **A** | **0** | **4** | **2** | **5** | **1** |
| **B** | **∞** | **0** | **∞** | **∞** | **6** |
| **C** | **1** | **∞** | **0** | **3** | **∞** |
| **D** | **∞** | **∞** | **1** | **0** | **2** |
| **E** | **∞** | **∞** | **∞** | **4** | **0** |

Iteration with Intermediate Node A (k = 0):

Try improving paths via node A.

Update rule: D[i][j] = min(D[i][j], D[i][0] + D[0][j])

**Updated matrix D₁:**

| **From\To** | **A** | **B** | **C** | **D** | **E** |
| --- | --- | --- | --- | --- | --- |
| **A** | **0** | **4** | **2** | **5** | **1** |
| **B** | **∞** | **0** | **∞** | **∞** | **6** |
| **C** | **1** | **5** | **0** | **3** | **2** |
| **D** | **∞** | **∞** | **1** | **0** | **2** |
| **E** | **∞** | **∞** | **∞** | **4** | **0** |

C → A → B: 1 + 4 = 5  
C → A → E: 1 + 1 = 2

**Iteration with Intermediate Node B (k = 1):**

Try improving paths via node **B**.

**D[i][j] = min(D[i][j], D[i][1] + D[1][j])**

Only updates from nodes that can reach B (A and C), but B has no outgoing paths except to E.

Updated matrix D₂:

No change in most rows, except:

| **From\To** | **A** | **B** | **C** | **D** | **E** |
| --- | --- | --- | --- | --- | --- |
| **B** | ∞ | 0 | ∞ | ∞ | 6 |
| **C** | 1 | 5 | 0 | 3 | 2 |
| **A** | 0 | 4 | 2 | 5 | 1 |

No further improvements using B as intermediate.

Iteration with Intermediate Node C (k = 2):

Now use C to check if paths improve. D[i][j] = min(D[i][j], D[i][2] + D[2][j])

| **From\To** | **A** | **B** | **C** | **D** | **E** |
| --- | --- | --- | --- | --- | --- |
| **A** | 0 | 4 | 2 | 5 | 1 |
| **B** | ∞ | 0 | ∞ | ∞ | 6 |
| **C** | 1 | 5 | 0 | 3 | 2 |
| **D** | 2 | 6 | 1 | 0 | 2 |
| **E** | ∞ | ∞ | ∞ | 4 | 0 |

**Updated matrix D₃:**

D → C → A: 1 + 1 = 2  
D → C → B: 1 + 5 = 6

Iteration with Intermediate Node D (k = 3):

Use D as intermediate node.

D[i][j] = min(D[i][j], D[i][3] + D[3][j])

**Updated matrix D₄:**

| **From\To** | **A** | **B** | **C** | **D** | **E** |
| --- | --- | --- | --- | --- | --- |
| **A** | **0** | **4** | **2** | **5** | **1** |
| **B** | **∞** | **0** | **∞** | **∞** | **6** |
| **C** | **1** | **5** | **0** | **3** | **2** |
| **D** | **2** | **6** | **1** | **0** | **2** |
| **E** | **6** | **10** | **7** | **4** | **0** |

E → D → A: 4 + 2 = 6  
 E → D → B: 4 + 6 = 10  
 E → D → C: 4 + 1 = 5 (but no direct path yet — updated next)

Iteration with Intermediate Node E (k = 4):

Use E as intermediate node.

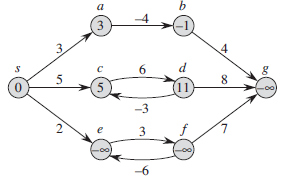
D[i][j] = min(D[i][j], D[i][4] + D[4][j])

**Final Distance Matrix D₅:**

| **From\To** | **A** | **B** | **C** | **D** | **E** |
| --- | --- | --- | --- | --- | --- |
| **A** | **0** | **4** | **2** | **5** | **1** |
| **B** | **7** | **0** | **9** | **12** | **6** |
| **C** | **1** | **5** | **0** | **3** | **2** |
| **D** | **2** | **6** | **1** | **0** | **2** |
| **E** | **6** | **10** | **7** | **4** | **0** |

B → E → D: 6 + 4 = 10  
 B → E → C: 6 + 7 = 13 (but B→C was ∞, now 9 via other paths)

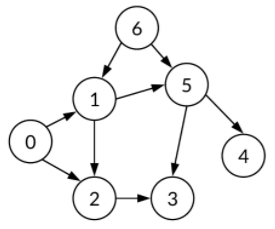
B) Which algorithm will be suitable on the graph below to find shortest path from a single source to all other vertices CO1, L2 1M



The **graph contains negative edge weights** (e.g., edge 3 → 1 has weight **-4**, edge 5 → 4 has weight **-3**, etc.).  
Therefore, the best algorithm to find the **shortest path from a single source** to **all other vertices** in such a graph is: Bellman-Ford Algorithm

3 A) Organize the ordering of vertices produced by topological sort when it is run on the below graph

CO4, L4 1.5M



To produce a **topological sort** of the given **Directed Acyclic Graph (DAG)**, we must arrange the vertices in **linear order** such that for every directed edge u → v, **u comes before v** in the ordering.

Step 1: Calculate In-degrees of All Vertices

| **Vertex** | **In-Degree** |
| --- | --- |
| 0 | 0 |
| 1 | 1 (from 0) |
| 2 | 2 (from 0, 1) |
| 3 | 1 (from 2) |
| 4 | 1 (from 5) |
| 5 | 2 (from 1, 6) |
| 6 | 0 |

**Start with nodes of in-degree 0 → [0, 6]**

Let’s process them in this order:

**Pop 0** → result = [0]

* 0 → 1 → decrease in-degree of 1 → now 0
* 0 → 2 → decrease in-degree of 2 → now 1
* Queue: [6, 1]

**Pop 6** → result = [0, 6]

* 6 → 5 → in-degree of 5 becomes 1
* Queue: [1]

**Pop 1** → result = [0, 6, 1]

* 1 → 2 → in-degree of 2 becomes 0
* 1 → 5 → in-degree of 5 becomes 0
* Queue: [2, 5]

**Pop 2** → result = [0, 6, 1, 2]

* 2 → 3 → in-degree of 3 becomes 0
* Queue: [5, 3]

**Pop 5** → result = [0, 6, 1, 2, 5]

* 5 → 4 → in-degree of 4 becomes 0
* Queue: [3, 4]

**Pop 3** → result = [0, 6, 1, 2, 5, 3]

* No outgoing edge
* Queue: [4]

**Pop 4** → result = [0, 6, 1, 2, 5, 3, 4]

* Done

0 → 6 → 1 → 2 → 5 → 3 → 4

Note: Topological sort is **not unique**. Other valid orders are possible as long as the rule “parents come before children” is satisfied.

B) Analyze the KMP algorithm that will effectively search for a given text – abcabcabdabc for a given pattern- abcabdabc CO4, L4 1.5M

**Text (T)** = abcabcabdabc

**Pattern (P)** = abcabdabc

Goal: Use **KMP** to search the pattern in the text efficiently.

**Step 1: Build the LPS Array (Longest Prefix Suffix)**

The **LPS array** helps avoid unnecessary comparisons by storing the length of the longest proper prefix which is also a suffix for the pattern.

**Pattern: a b c a b d a b c**

| **Index (i)** | **P[i]** | **LPS[i]** |
| --- | --- | --- |
| 0 | a | 0 |
| 1 | b | 0 |
| 2 | c | 0 |
| 3 | a | 1 |
| 4 | b | 2 |
| 5 | d | 0 |
| 6 | a | 1 |
| 7 | b | 2 |
| 8 | c | 3 |

Final **LPS array**: [0, 0, 0, 1, 2, 0, 1, 2, 3]

**Step 2: Apply KMP Matching**

Let:

* i = 0 → index for text T
* j = 0 → index for pattern P

We compare characters at T[i] and P[j], and do the following:

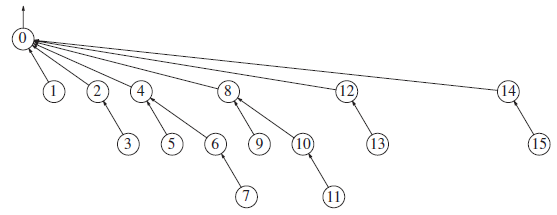
1. If T[i] == P[j], increment both i and j.
2. If mismatch and j > 0, then j = LPS[j-1]
3. If mismatch and j == 0, just increment i.

**Matching Process:**

| **Text Index (i)** | **Text Char** | **Pattern Index (j)** | **Pattern Char** | **Action** |
| --- | --- | --- | --- | --- |
| **0** | **a** | **0** | **a** | **match → i=1, j=1** |
| **1** | **b** | **1** | **b** | **match → i=2, j=2** |
| **2** | **c** | **2** | **c** | **match → i=3, j=3** |
| **3** | **a** | **3** | **a** | **match → i=4, j=4** |
| **4** | **b** | **4** | **b** | **match → i=5, j=5** |
| **5** | **c** | **5** | **d** | **mismatch → j=LPS[4]=2** |
| **5** | **c** | **2** | **c** | **match → i=6, j=3** |
| **6** | **a** | **3** | **a** | **match → i=7, j=4** |
| **7** | **b** | **4** | **b** | **match → i=8, j=5** |
| **8** | **d** | **5** | **d** | **match → i=9, j=6** |
| **9** | **a** | **6** | **a** | **match → i=10, j=7** |
| **10** | **b** | **7** | **b** | **match → i=11, j=8** |
| **11** | **c** | **8** | **c** | **match → i=12, j=9 (pattern matched fully) ✅** |

**Pattern found at index 3** in the text.

C) Explain in detail how to find the 11 in the below data structure using smart find algorithm

CO2, L2 2M 

The **Smart Find** algorithm is used in **Disjoint Set Union (DSU)** or **Union-Find** data structures to efficiently determine the **representative/root of the set** a node belongs to. It **compresses the path** from the node to the root so that **all nodes on that path directly point to the root**. This improves future query speed to nearly constant time.

Follow the parent pointers for 11:

11 → 10 → 8 → 0

11’s parent = 10

10’s parent = 9

9’s parent = 6

6’s parent = 8

8’s parent = 0 (root)

So, the root (representative) of 11 is **0**.

**Path Compression**

We now **update parent pointers** of all nodes in the path (11, 10, 9, 6, 8) to point directly to the root 0.

After Path Compression

11 → 0

10 → 0

9 → 0

6 → 0

8 → 0

This **flattens** the tree structure for this path.

**Final Updated Tree (Simplified View)**

After compression, the tree will look like this:

0

/ /|\ \ \ \

1 2 4 6 8 12 14

|

9 10 11

All now directly or indirectly point to 0 as their parent, resulting in a **flatter tree**.