

DAA

Assignment no: 2



Haseeb Irfan 029

Abdullah Nadeem 099

Eshal Atif 023

Q1) Implement the following using a Brute Force strategy:

A) Longest Common Substring

Step 1: Algorithm of Brute Force Longest Common Substring

Algorithm: BRUTE_FORCE_LCS(s1, s2)

```
1. m ← LENGTH(s1)
2. n ← LENGTH(s2)
3. max_length ← 0
4. FOR i ← 0 TO m - 1 DO
5.   FOR j ← 0 TO n - 1 DO
6.     k ← 0
7.     WHILE (i + k < m) AND (j + k < n) AND (s1[i + k] == s2[j + k]) DO
8.       k ← k + 1
9.     END WHILE
10.    IF k > max_length THEN
11.      max_length ← k
12.    END IF
13.  END FOR
14. END FOR
15. RETURN max_length
```

Step 2: Example Execution

Initial Setup:

- s1 = "ABABC" (m = 5)
- s2 = "BABCA" (n = 5)
- max_length = 0

Trace Through Key Iterations:

i = 0 (s1[0] = 'A'):

- j = 0 (s2[0] = 'B'): 'A' ≠ 'B', k = 0
- j = 1 (s2[1] = 'A'): 'A' == 'A', k = 1, then 'B' == 'B', k = 2, then 'A' ≠ 'B'
 - Match = "AB", max_length = 2
- j = 2 (s2[2] = 'B'): 'A' ≠ 'B', k = 0
- j = 3 (s2[3] = 'C'): 'A' ≠ 'C', k = 0
- j = 4 (s2[4] = 'A'): 'A' == 'A', k = 1, then out of bounds, max_length = 2

i = 1 (s1[1] = 'B'):

- j = 0 (s2[0] = 'B'):
 - 'B' == 'B', k = 1

- 'A' == 'A', k = 2
 - 'B' == 'B', k = 3
 - 'C' == 'C', k = 4
 - **Match = "BABC", max_length = 4 ✓**
- j = 1 to 4: shorter or no matches

i = 2 (s1[2] = 'A'):

- All matches will be ≤ 3 characters (remaining length)

i = 3 (s1[3] = 'B'):

- All matches will be ≤ 2 characters

i = 4 (s1[4] = 'C'):

- All matches will be ≤ 1 character

Final Result:

- max_length = 4
- Longest Common Substring = "BABC"

Step 3: Time Complexity

The three loops are nested, so we multiply their worst-case costs:

Time Complexity: $O(m \times n \times \min(m, n))$

Step 4: Time Complexity Analysis

1. **Outer Loop (i):** Runs m times
2. **Middle Loop (j):** For each i, runs n times $\rightarrow m \times n$ iterations so far
3. **Inner While Loop:** In worst case, runs $\min(m, n)$ times
 - Example: s1 = "AAA...A", s2 = "AAA...A" (identical strings)
 - For each (i,j) pair, it compares until end of shorter string
 - Each comparison takes $O(1)$ time

Total Operations: $m \times n \times \min(m, n)$

Worst-case Scenario: Two identical strings of length L

- Time Complexity: **$O(L^3)$**

Step 5: Test Cases

Test ID	String 1 (s1)	String 2 (s2)	Expected Result	Longest Common Substring	Complexity Category	Operations (m×n)
1.1	"ABCD"	"BCDE"	3	"BCD"	Average	16
1.2	"ABCD"	"WXYZ"	0	""	Best Case	16
1.3	"ABCD"	"EFCG"	1	"C"	Average	16
1.4	"HELLO"	"HELLO"	5	"HELLO"	Worst Case	25

B) Closest Pair of Points in 2D Plane

Step 1: Algorithm for Brute Force Closest Pair of Points

Algorithm: BRUTE_FORCE_CLOSEST_PAIR(P[], n)

```
1. min_distance ← ∞
2. point1 ← NULL
3. point2 ← NULL
4. FOR i ← 0 TO n - 2 DO
5.   FOR j ← i + 1 TO n - 1 DO
6.     dx ← P[j].x - P[i].x
7.     dy ← P[j].y - P[i].y
8.     distance ← √(dx² + dy²)
9.     IF distance < min_distance THEN
10.      min_distance ← distance
11.      point1 ← P[i]
12.      point2 ← P[j]
13.   END IF
14. END FOR
15. END FOR
16. RETURN (point1, point2, min_distance)
```

Step 2: Example Execution

Input Points:

P = [(2,3), (12,30), (40,50), (5,1), (12,10), (3,4)]

Step-by-step Brute-Force Comparison:

1. Compare (2,3) and (12,30):

- dx = 10, dy = 27
- distance = $\sqrt{(10^2 + 27^2)} = \sqrt{(100 + 729)} = \sqrt{829} \approx 28.79$
- min_distance = 28.79
- closest pair: (2,3), (12,30)

2. **Compare (2,3) and (40,50):**
 - $dx = 38, dy = 47$
 - $distance = \sqrt{38^2 + 47^2} = \sqrt{1444 + 2209} = \sqrt{3653} \approx 60.44$
 - $min_distance = 28.79$ (no change)
3. **Compare (2,3) and (5,1):**
 - $dx = 3, dy = -2$
 - $distance = \sqrt{9 + 4} = \sqrt{13} \approx 3.61$
 - $min_distance = 3.61$
 - closest pair: (2,3), (5,1)
4. **Compare (2,3) and (12,10):**
 - $dx = 10, dy = 7$
 - $distance = \sqrt{100 + 49} = \sqrt{149} \approx 12.21$
 - $min_distance = 3.61$ (no change)
5. **Compare (2,3) and (3,4):**
 - $dx = 1, dy = 1$
 - $distance = \sqrt{1 + 1} = \sqrt{2} \approx 1.41$
 - $min_distance = 1.41$
 - **closest pair: (2,3), (3,4) ✓**
6. **Compare (12,30) and (40,50):**
 - $dx = 28, dy = 20$
 - $distance = \sqrt{784 + 400} = \sqrt{1184} \approx 34.41$
 - $min_distance = 1.41$ (no change)
7. **Compare (12,30) and (5,1):**
 - $dx = -7, dy = -29$
 - $distance = \sqrt{49 + 841} = \sqrt{890} \approx 29.83$
 - $min_distance = 1.41$ (no change)
8. **Compare (12,30) and (12,10):**
 - $dx = 0, dy = -20$
 - $distance = 20$
 - $min_distance = 1.41$ (no change)
9. **Compare (12,30) and (3,4):**
 - $dx = -9, dy = -26$
 - $distance = \sqrt{81 + 676} = \sqrt{757} \approx 27.51$
 - $min_distance = 1.41$ (no change)
10. **Compare (40,50) and (5,1):**
 - $dx = -35, dy = -49$
 - $distance = \sqrt{1225 + 2401} = \sqrt{3626} \approx 60.22$
 - $min_distance = 1.41$ (no change)
11. **Compare (40,50) and (12,10):**
 - $dx = -28, dy = -40$
 - $distance = \sqrt{784 + 1600} = \sqrt{2384} \approx 48.83$
 - $min_distance = 1.41$ (no change)
12. **Compare (40,50) and (3,4):**
 - $dx = -37, dy = -46$
 - $distance = \sqrt{1369 + 2116} = \sqrt{3485} \approx 59.04$
 - $min_distance = 1.41$ (no change)
13. **Compare (5,1) and (12,10):**

- $dx = 7, dy = 9$
- $distance = \sqrt{49 + 81} = \sqrt{130} \approx 11.40$
- $min_distance = 1.41$ (no change)

14. Compare (5,1) and (3,4):

- $dx = -2, dy = 3$
- $distance = \sqrt{4 + 9} = \sqrt{13} \approx 3.61$
- $min_distance = 1.41$ (no change)

15. Compare (12,10) and (3,4):

- $dx = -9, dy = -6$
- $distance = \sqrt{81 + 36} = \sqrt{117} \approx 10.82$
- $min_distance = 1.41$ (no change)

Final Output:

- **Closest pair:** (2,3) and (3,4)
- **Minimum distance:** $\sqrt{2} \approx 1.41$ km

Step 3: Time Complexity Analysis

The number of comparisons made:

$$\begin{aligned} T(n) &= (n-1) + (n-2) + (n-3) + \dots + 2 + 1 \\ &= \sum_{k=1}^{n-1} k \\ &= (n-1) \times n / 2 \\ &= n^2/2 - n/2 \end{aligned}$$

Asymptotic Analysis:

- Big O (Upper Bound):**
 - $T(n) = n^2/2 - n/2 \leq n^2$ for all $n \geq 1$
 - Therefore, $T(n) = O(n^2)$
- Big Ω (Lower Bound):**
 - $T(n) = n^2/2 - n/2 \geq n^2/4$ for all $n \geq 2$
 - Therefore, $T(n) = \Omega(n^2)$
- Big Θ (Tight Bound):**
 - Since $T(n) = O(n^2)$ and $T(n) = \Omega(n^2)$
 - Therefore, $T(n) = \Theta(n^2)$

Conclusion: This brute-force approach has quadratic time complexity **$O(n^2)$**

Step 4: Test Cases

Test ID	Input Points	Expected Closest Pair	Expected Distance	Description
1.1	[(0,0), (1,1), (2,2)]	(0,0) & (1,1)	$\sqrt{2} \approx 1.414$	Simple case
1.2	[(0,0), (3,4)]	(0,0) & (3,4)	5.0	Only two points

Test ID	Input Points	Expected Closest Pair	Expected Distance	Description
1.3	[(1,1), (1,1), (2,2)]	(1,1) & (1,1)	0.0	Duplicate points
1.4	[(0,0), (1,0), (0,1)]	(0,0) & (1,0) or (0,0) & (0,1)	1.0	Multiple equidistant

Q2) Implement the following using a Decrease-and-Conquer strategy:

A) Binary Search

Step 1: Algorithm

Algorithm: BINARY_SEARCH(array, target, left, right)

1. IF left > right THEN
2. RETURN -1 // Base case: not found
3. END IF
- 4.
5. mid \leftarrow (left + right) / 2 // Find middle
- 6.
7. IF array[mid] == target THEN
8. RETURN mid // Base case: found
9. ELSE IF array[mid] < target THEN
10. RETURN BINARY_SEARCH(array, target, mid + 1, right)
11. ELSE
12. RETURN BINARY_SEARCH(array, target, left, mid - 1)
13. END IF

Step 2: How Decrease-and-Conquer is Applied

Step 1: Look at the Middle

- Find the middle element of your current search range
- Compare it with your target number

Step 2: Make a Smart Decision

- If the middle element equals your target: **You found it! Done.**
- If the middle element is smaller than your target: **Your target must be in the right half** (discard left half)
- If the middle element is larger than your target: **Your target must be in the left half** (discard right half)

Step 3: Repeat on the Smaller Problem

Space Complexity:

- Recursive: $O(\log n)$ due to call stack
- Iterative: $O(1)$

Step 5: Test Cases

Test ID	Input Array	Target	Expected Result	Description
1.1	[1, 3, 5, 7, 9, 11]	5	2 (index)	Target found in middle
1.2	[1, 3, 5, 7, 9, 11]	1	0	Target found at beginning
1.3	[1, 3, 5, 7, 9, 11]	11	5	Target found at end
1.4	[1, 3, 5, 7, 9, 11]	7	3	Target found
1.5	[1, 3, 5, 7, 9, 11]	9	4	Target found
1.6	[1, 3, 5, 7, 9, 11]	4	-1	Target not found

Q3) Implement a problem that requires using both Brute Force and Decrease-and-Conquer strategies:

Traveling Salesman Problem (TSP)

Approach 1: Brute Force Algorithm

Algorithm: BRUTE_FORCE_TSP(distances, cities)

1. $n \leftarrow$ number of cities
2. $\text{min_distance} \leftarrow \infty$
3. $\text{min_path} \leftarrow$ empty list
- 4.
5. FOR each permutation p of cities $[1, 2, \dots, n-1]$:
6. $\text{current_path} \leftarrow [0] + p + [0]$ // Start and end at city 0
7. $\text{current_distance} \leftarrow 0$
- 8.
9. FOR $i \leftarrow 0$ TO $\text{length}(\text{current_path}) - 2$:
10. $\text{current_distance} += \text{distances}[\text{current_path}[i]][\text{current_path}[i+1]]$
11. END FOR
- 12.
13. IF $\text{current_distance} < \text{min_distance}$:
14. $\text{min_distance} \leftarrow \text{current_distance}$
15. $\text{min_path} \leftarrow \text{current_path}$
16. END IF
17. END FOR
- 18.
19. RETURN (min_path , min_distance)

Time Complexity Analysis (Brute Force)

Overall Time Complexity: $O(n!)$

Line 5: Generating Permutations

- We need to generate all permutations of $(n-1)$ cities (excluding the starting city 0)
- Number of permutations = $(n-1)!$
- Generating each permutation: $O(n)$ per permutation
- Total for generation: $O(n \cdot (n-1)!) = O(n!)$

Lines 6-11: Processing Each Permutation

- Constructing current_path (line 6): $O(n)$
- The FOR loop (lines 9-10) iterates n times (visiting $n+1$ cities means n transitions)
- Each distance lookup (line 10): $O(1)$
- Total per permutation: $O(n)$

Lines 13-15: Comparison and Update

- Comparison (line 13): $O(1)$
- Path assignment (line 15): $O(n)$
- These operations don't dominate, happening only when a better path is found

Final Calculation:

- Outer loop iterations: $(n-1)!$ permutations
- Work per iteration: $O(n)$
- **Total complexity: $O((n-1)! \cdot n) = O(n!)$**

Practical Implications:

Cities (n)	Permutations (n-1)!	Approximate Operations
5	24	~120
10	362,880	~3.6 million
15	87 billion	~1.3 trillion
20	121 quintillion	Infeasible

Approach 2: Decrease-and-Conquer (Nearest Neighbor)

Algorithm: NEAREST_NEIGHBOR_TSP(distances)

1. $n \leftarrow$ number of cities
2. visited \leftarrow array of size n , all false

```

3. path ← [0]           // Start from first city
4. visited[0] ← true
5. total_distance ← 0
6.
7. FOR i ← 1 TO n-1:
8.   current_city ← path[last element]
9.   nearest_city ← -1
10.  min_dist ← ∞
11.
12.  FOR each unvisited city j:
13.    IF distances[current_city][j] < min_dist:
14.      min_dist ← distances[current_city][j]
15.      nearest_city ← j
16.    END IF
17.  END FOR
18.
19.  path.append(nearest_city)
20.  visited[nearest_city] ← true
21.  total_distance += min_dist
22. END FOR
23.
24. // Return to starting city
25. total_distance += distances[path[last]][0]
26. path.append(0)
27.
28. RETURN (path, total_distance)

```

Time Complexity Analysis (Decrease-and-Conquer)

Overall Time Complexity: $O(n^2)$

Lines 1-5: Initialization

- Creating visited array: $O(n)$
- Creating path list: $O(1)$
- Setting visited[0]: $O(1)$
- Total: $O(n)$

Lines 7-22: Main Loop (Building the Tour)

Outer loop (line 7): Iterates $(n-1)$ times

For each iteration:

- Line 8: Get current city from path: $O(1)$
- Lines 9-10: Initialize variables: $O(1)$
- Lines 12-17: Inner loop - Find nearest unvisited city
 - Iteration 1: Check $(n-1)$ unvisited cities
 - Iteration 2: Check $(n-2)$ unvisited cities
 - Iteration 3: Check $(n-3)$ unvisited cities

- ...
- Iteration (n-1): Check 1 unvisited city
- Lines 19-21: Update path and distance: O(1) each

Inner Loop Total Work:

$$(n-1) + (n-2) + (n-3) + \dots + 2 + 1 = (n-1) \cdot n / 2 = O(n^2)$$

Lines 24-26: Closing the Tour

- Adding distance back to start: O(1)
- Appending to path: O(1)
- Total: O(1)

Final Calculation:

- Outer loop: (n-1) iterations
- Inner loop per outer iteration: O(n) on average
- **Total complexity: $O(n-1) \times O(n) = O(n^2)$**

Practical Implications:

Cities (n)	Operations (n ²)	Approximate Time
10	100	Instant
100	10,000	Instant
1,000	1,000,000	< 1 second
10,000	100,000,000	Few seconds

Example: European Cities TSP

Distance Matrix (in km):

	London (L)	Paris (P)	Berlin (B)	Rome (R)
London (L)	0	344	931	1432
Paris (P)	344	0	878	1106
Berlin (B)	931	878	0	1184
Rome (R)	1432	1106	1184	0

Solution 1: Brute Force Approach

All possible routes starting and ending at London (L):

1. **L→P→B→R→L**
 - $344 + 878 + 1184 + 1432 = 3838$ km
2. **L→P→R→B→L**
 - $344 + 1106 + 1184 + 931 = 3565$ km ✓
3. **L→B→P→R→L**
 - $931 + 878 + 1106 + 1432 = 4347$ km
4. **L→B→R→P→L**
 - $931 + 1184 + 1106 + 344 = 3565$ km ✓
5. **L→R→P→B→L**
 - $1432 + 1106 + 878 + 931 = 4347$ km
6. **L→R→B→P→L**
 - $1432 + 1184 + 878 + 344 = 3838$ km

Optimal Solution: L→P→R→B→L or L→B→R→P→L with distance **3565 km**

Solution 2: Nearest Neighbor (Decrease-and-Conquer)

Starting from London (L):

Step 1: From London, find nearest city

- P: 344 km (nearest)
- B: 931 km
- R: 1432 km
- **Choose: L→P (344 km)**

Step 2: From Paris, find nearest unvisited city

- B: 878 km
- R: 1106 km
- **Choose: P→B (878 km)**

Step 3: From Berlin, find nearest unvisited city

- R: 1184 km (only option)
- **Choose: B→R (1184 km)**

Step 4: Return to London

- R→L: 1432 km
- **Choose: R→L (1432 km)**

Nearest Neighbor Path: $L \rightarrow P \rightarrow B \rightarrow R \rightarrow L$ **Total Distance:** $344 + 878 + 1184 + 1432 = 3838$ km

Comparison:

- Brute Force (Optimal): **3565 km**
 - Nearest Neighbor: **3838 km**
 - Difference: 273 km (7.7% suboptimal)
-

Comparative Analysis: Brute Force vs. Decrease-and-Conquer

Brute Force Approach

Definition: Exhaustively check all possible solutions until the correct one is found.

Advantages:

Simplicity: Easy to implement and understand

Guaranteed Solution: Will always find the optimal solution if one exists

No Special Cases: Works uniformly across all inputs

Minimal Preprocessing: Often requires little to no setup

Proven Correctness: Easy to verify correctness

Disadvantages:

Inefficient: Time complexity is often exponential or polynomial

Resource Intensive: Consumes significant memory and processing power

Not Scalable: Performance degrades rapidly with input size

Wasteful: Explores many obviously wrong solutions

Decrease-and-Conquer Approach

Definition: Reduce problem instance to a smaller instance of the same problem, solve the smaller instance, and extend the solution.

Advantages:

Efficiency: Often achieves logarithmic or linear time complexity

Optimal Solutions: Frequently produces optimal or near-optimal results

Scalable: Handles large inputs efficiently

Elegant Design: Clean recursive or iterative structure

Mathematical Foundation: Well-defined recurrence relations

Disadvantages:

Implementation Complexity: Can be harder to implement correctly

Problem Constraints: Requires problems to have optimal substructure

Overhead: Recursive versions have stack overhead

Preconditions: Often requires sorted data or specific input structure
