Yogendra Pokhrel

617585

1. Suppose Prob1, Prob2, and Prob3 are decision problems and Prob1 is polynomial reducible to Prob2, and Prob2 is polynomial reducible to Prob3. Explain why Prob1 must be polynomial reducible to Prob3.

Answer:

In computational complexity theory, polynomial-time reducibility ($\leq p$) is transitive. That is, if a problem A can be reduced to B in polynomial time, and B can be reduced to C in polynomial time, then A can also be reduced to C in polynomial time. Applying this concept:

Given:

- Prob1 $\leq p$ Prob2: There exists a polynomial-time computable function f_1 such that for every input x,

  $x \in$ Prob1 $\Leftrightarrow f_1(x) \in$ Prob2.

- Prob2 $\leq p$ Prob3: There exists a polynomial-time computable function f_2 such that for every input y,

  $y \in$ Prob2 $\Leftrightarrow f_2(y) \in$ Prob3.

To reduce Prob1 to Prob3, we can compose the reductions:

- Define $f(x) = f_2(f_1(x))$.

Because both f_1 and f_2 are polynomial-time computable, and the composition of two polynomial-time functions is still polynomial-time, f is also a polynomial-time computable function.

Thus:

- $x \in$ Prob1 $\Leftrightarrow f_1(x) \in$ Prob2 $\Leftrightarrow f_2(f_1(x)) \in$ Prob3
- Therefore, Prob1 $\leq p$ Prob3

Conclusion:

This demonstrates that polynomial-time reducibility is closed under composition, which is a fundamental property used in proving the NP-completeness of problems. As a result, Prob1 is polynomial-time reducible to Prob3.

2. Illustrate the proof that the HamiltonianCycle problem is polynomial reducible to TSP by considering the following Hamiltonian graph—an instance of HamiltonianCycle—and transforming it to a TSP instance in polynomial time so that a solution to the HC problem yields a solution to the TSP problem, and conversely

Perfect, Yogendra. Since you're experienced in algorithms, let's approach this with a formal yet clear explanation and illustration of how the **Hamiltonian Cycle (HC)** problem reduces to the **Travelling Salesman Problem (TSP)** in **polynomial time** — using the provided undirected graph as a concrete example.

Answer:

Reduction Strategy (HC ≤p TSP)

We want to reduce the Hamiltonian Cycle (HC) problem to an instance of the Travelling Salesman Problem (TSP) such that solving the TSP yields a solution to HC and vice versa.

Objective:

- For HC: Determine if there exists a simple cycle that visits every vertex exactly once and returns to the starting vertex.
- For TSP (decision version): Given a weighted complete graph and a cost limit K, is there a tour (visiting each vertex once and returning to the start) with total cost ≤ K?

Step 1: Represent the HC Graph

Let's represent your graph **G(V, E)**:

- **Vertices (V)**: {A, B, C, D}
- **Edges (E)**: {(A-B), (B-D), (D-C), (C-A)}

This forms a cycle: **A → B → D → C → A**, which is indeed a Hamiltonian cycle.

Step 2: Construct a Complete Graph G′ for TSP

To reduce HC to TSP, we transform the original graph G into a complete graph G′ with artificial weights:

- For each pair of vertices (u, v):
    - If (u, v) ∈ E (in original HC graph), assign weight = 1.
    - If (u, v) ∉ E, assign weight = ∞ or a sufficiently large constant (say, 1000), to discourage traversal.

This ensures that:

- A tour of cost = number of vertices (i.e., 4) exists iff there is a Hamiltonian cycle in the original graph.

- **Step 3: TSP Graph (G′) Construction**

From/To	A	B	C	D
A	0	1	1	∞
B	1	0	∞	1
C	1	∞	0	1
D	∞	1	1	0

Here:

- Valid HC edges are assigned weight = 1.
- Invalid (non-HC) edges are assigned weight = ∞.

Step 4: Define Cost Limit K for TSP

- Set K = number of vertices = 4.

Step 5: Equivalence of Solutions

- If the TSP instance has a tour of total weight ≤ K = 4, then it must only use the edges of weight 1, which means a Hamiltonian cycle exists in the original graph.
- If no such tour exists under this constraint, then no Hamiltonian cycle exists in the original graph.

Conclusion:

This reduction transforms any instance of the Hamiltonian Cycle problem to a TSP decision problem in polynomial time:

- Constructing the complete graph takes $O(n^2)$
- Edge weighting is deterministic and local
- A TSP solver’s output directly maps back to an HC solution

Thus, HC ≤p TSP.

3. Show that TSP is NP-complete. (Hint: use the relationship between TSP and HamiltonianCycle discussed in the slides. You may assume that the HamiltonianCycle problem is NP-complete.)

Answer:

To prove that the TSP (decision version) is NP-complete, we need to establish two things:

1. TSP ∈ NP

Given a tour (a sequence of cities), we can:

- Check if it visits each vertex exactly once, and
- Compute the total cost of the tour and verify if it is ≤ K

  All of this can be done in polynomial time.

So, TSP is in NP.

2. TSP is NP-hard

We will prove this by a polynomial-time reduction from Hamiltonian Cycle (HC) to TSP.

Reduction: HC ≤p TSP

Given:

An undirected graph **G = (V, E)** for the **Hamiltonian Cycle** problem. We want to determine whether there exists a cycle that visits every vertex exactly once.

Construct:

A complete weighted graph G′ = (V, E′) for the TSP decision problem:

1. Vertices: Same as in G

2. Edges and Weights:

    - If an edge (u, v) exists in the original HC graph G, assign weight = 1.

    - If (u, v) is not in G, assign weight = ∞ (or a very large number).

3. Cost limit K = |V|

Claim:

There exists a Hamiltonian Cycle in G ⇔ There exists a TSP tour of total cost ≤ K in G′

- If G has a Hamiltonian Cycle: Then there exists a tour using only edges of cost 1 in G′ → Total cost = |V| → TSP answer is "Yes".

- If TSP has a tour of cost ≤ K: Then it must use only edges of weight 1 → those edges exist in G → which forms a Hamiltonian Cycle.

Therefore, solving TSP allows us to solve HC — meaning TSP is at least as hard as HC, which is NP-complete.

Conclusion:

- TSP is in NP.
- HC ≤p TSP in polynomial time.
- Hence, TSP is NP-complete.

4. Find an O(n) algorithm that does the following: Given a size n input array of integers, output the first numbers in the array (from left to right) whose sum is exactly 10 (or indicate that no such numbers can be found).

Problem Statement:

Design an **O(n)** algorithm that:

- Takes an array arr of size n as input.

- Finds the **first subarray (from left)** whose **sum equals exactly 10**.

- Return that subarray (or indicate no such subarray exists).

Optimal Approach – Sliding Window (O(n))

This is a **classic prefix-sum sliding window** problem, assuming **only non-negative integers**.

If the array can contain **negative integers**, a sliding window won't work reliably — we'd then use a hash map (still O(n) for expected time).

Algorithm (for non-negative integers):

```
public static List<Integer> findSubarraySum10(int[] arr) {
    int start = 0, sum = 0;

    for (int end = 0; end < arr.length; end++) {
        sum += arr[end];

        // Shrink the window if sum > 10
        while (sum > 10 && start <= end) {
            sum -= arr[start++];
        }

        // Check for exact match
        if (sum == 10) {
            List<Integer> result = new ArrayList<>();
            for (int i = start; i <= end; i++) {
                result.add(arr[i]);
            }
            return result;
        }
    }

    // No subarray found
    return new ArrayList<>();
}
```

Example Usage:

```
int[] arr = {1, 2, 3, 4, 5, 1};
System.out.println(findSubarraySum10(arr)); // Output: [2, 3, 4, 1]
```

Time & Space Complexity:

- **Time:** O(n) — Each element is processed at most twice (once added, once removed).
- **Space:** O(k) — At most, a list of k elements with sum 10 is returned.

5. Work through the steps of the Dynamic Programming solution to SubsetSum in the case in which S = {3,2,1,5} and k = 4.

Step-by-Step: Dynamic Programming Table Construction

We'll use a DP table dp[i][j], where:

- i = number of elements considered (from 0 to n)
- j = target sum (from 0 to k)
- dp[i][j] = true if a subset of the first i elements has a sum of j

Let's initialize:

- n = 4 (number of elements)
- k = 4 (target sum)
- S = [3, 2, 1, 5]

Initialize DP Table (Size: (n+1) x (k+1) → 5 x 5):

i \ j	**0**	**1**	**2**	**3**	**4**
0	T	F	F	F	F
1					
2					
3					
4					

- **dp[0][0] = true** → 0 sum is always possible with 0 items.
- **dp[0][j] = false** for j > 0

Fill the table row by row:

Step 1: i = 1 (element = 3)

We check whether sum j is possible using just {3}.

i \ j	**0**	**1**	**2**	**3**	**4**
1	T	F	F	**T**	F

- dp[1][3] = true → sum 3 is possible using just element 3.

Step 2: i = 2 (elements = {3, 2})

i \ j	**0**	**1**	**2**	**3**	**4**
2	T	F	**T**	**T**	F

- dp[2][2] = true → using {2}
- dp[2][3] = true → already true from above
- dp[2][0] always stays true

Step 3: i = 3 (elements = {3, 2, 1})

i \ j	**0**	**1**	**2**	**3**	**4**
3	T	**T**	**T**	**T**	**T**

- dp[3][1] = true → using {1}
- dp[3][4] = true → 3 + 1 or 2 + 1 + 1

Now we have a subset that sums to 4!

Step 4: i = 4 (element = 5, full set {3, 2, 1, 5})

i \ j	**0**	**1**	**2**	**3**	**4**
4	T	T	T	T	T

- Nothing changes for k=4, as adding 5 would overshoot it.

Final DP Table:

i \ j	**0**	**1**	**2**	**3**	**4**
0	T	F	F	F	F
1	T	F	F	T	F
2	T	F	T	T	F
3	T	T	T	T	**T**
4	T	T	T	T	**T**

Answer: Yes, a subset that sums to 4 exists (e.g., {3, 1} or {2, 1, 1} depending on repetitions).