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.**

A problem A is *polynomially reducible* to a problem B (A≤ pB) if there exists a polynomial-time algorithm that transforms any instance of A into an instance of B such that the answer to the instance of B corresponds to the answer for the instance of A. This means solving B efficiently (in polynomial time) also solves A.

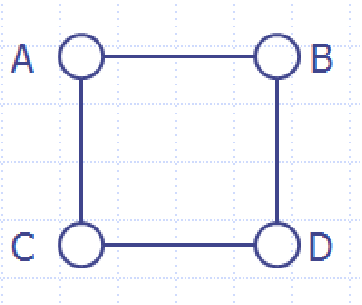
Suppose Prob1≤pProb2 and Prob2 ≤p​Prob3.

Let f1 be the polynomial-time transformation from Prob1 to Prob2, and f2​ be the polynomial-time transformation from Prob2 to Prob3.

Then the composition f(x)=f2(f1(x)) transforms an instance x of Prob1 into an instance of Prob3 in polynomial time because the composition of two polynomial-time functions is still polynomial.

This satisfies the definition of Prob1≤ pProb3, proving the transitivity of polynomial reducibility.

1. **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.**



Using Hamilton Cycle problem (HC)

* Input: A graph G=(V,E).
* Output: Does there exist a cycle visiting every vertex exactly once?

Using the Traveling Salesman Problem

* Input: A weighted graph G′=(V,E′) and a target T.
* Output: Does there exist a tour visiting every vertex exactly once with a total weight ≤T?

Reduction

Convert G from HC into G′ for TSP:

* Create a complete graph G′ where V′ =V and E′ contains all pairs of vertices.
* Assign weights:
  + - w(u,v)= 1 if (u,v) ∈ E.
    - w(u,v) = 2 otherwise.

Set T= ∣V∣ (the number of vertices).

Equivalence

* If G has a Hamiltonian cycle, the corresponding tour in G′ has weight T=∣V∣.
* If G′ has a tour with weight T=∣V∣, this tour only uses edges of weight 1, corresponding to a Hamiltonian cycle in G.

Polynomial Time

The transformation involves constructing G′ with ∣V∣2 edges and assigning weights, which takes O(∣V∣2) time.

1. **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.)**

TSP is in NP

A given tour can be verified in polynomial time:

* Check that all vertices are visited exactly once (O(∣ V ∣).
* Sum the weights and compare to T (O(∣ V ∣)).

Total verification time: O(∣ V ∣)

Reduction from Hamiltonian Cycle (HC)

- HC is NP-complete, and as shown in Question 2, HC reduces to TSP in polynomial time.

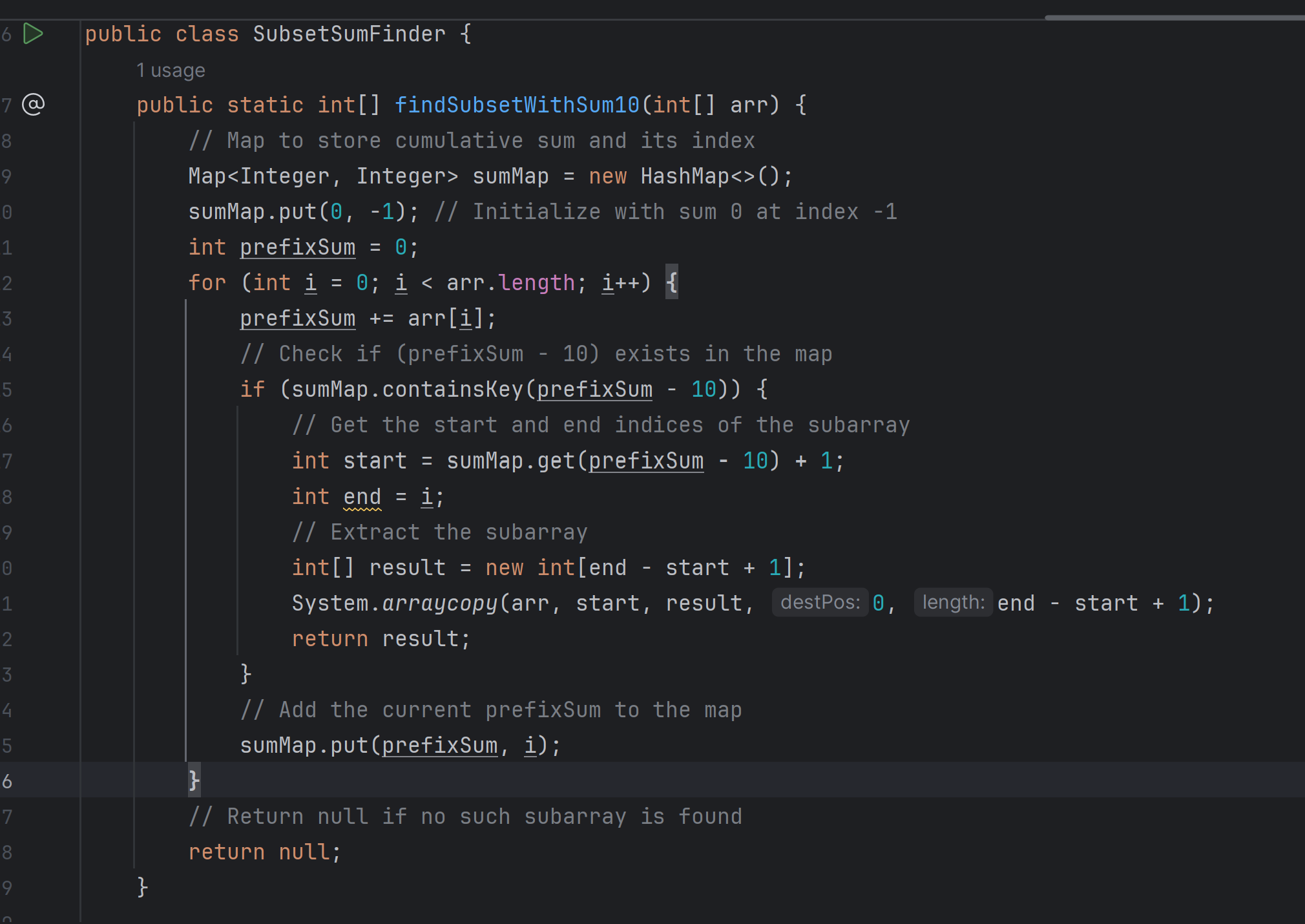
- Thus, any problem in NP can be reduced to HC and then to TSP in polynomial time.

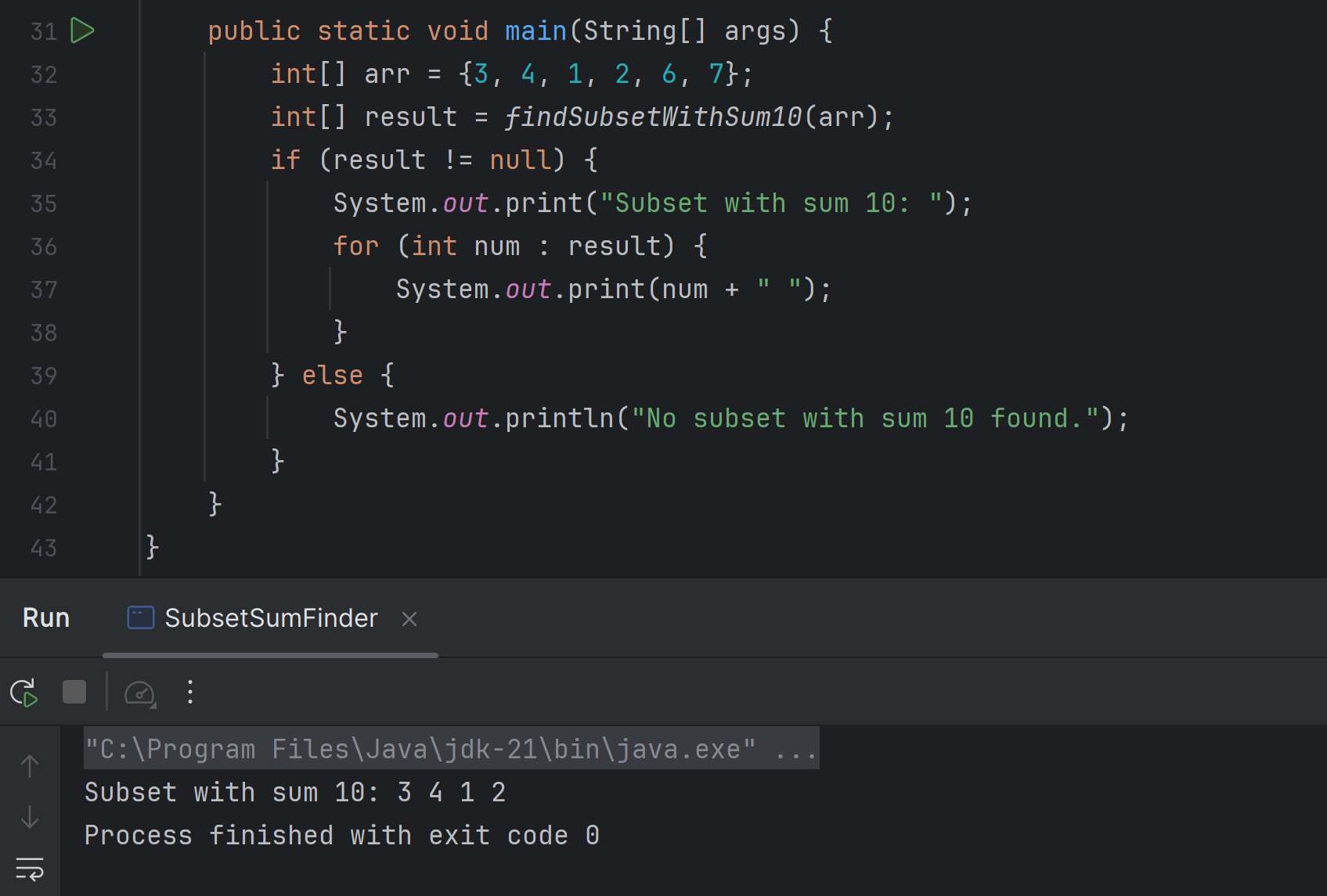
Since TSP is in NP and NP-hard (by reduction), TSP is NP-complete.

1. **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).**

Use a sliding window with a hash map to track the cumulative sum.

* Maintain a variable *prefixSum* to store the running sum of elements as you iterate through the array.
* Use a hash *map* where keys are cumulative sums and values are their indices.

**

**

* The sumMap stores cumulative sums as keys and their corresponding indices as values.This helps quickly check if a subarray with the required sum exists.
* As we iterate through the array, maintain the running sum (prefixSum), we check if prefixSum - 10 exists in the map:
* If it does, retrieve the starting index and form the subarray.
* Add the current prefixSum to the map.

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

**Detailed Steps**:

DP Table:

Define dp*[i][j]:* Whether a subset of the first i elements has sum j.

Initialization:

* + dp*[0][0]*= True (sum 0 is always achievable with no elements).
  + dp*[i][0]*= True (sum 0 is achievable by not selecting any elements).
  + dp*[0][j]*= False for j>0 (non-zero sum not achievable with 0 elements).

Transition:

* + If S[i−1] ≤ j:
    - dp*[i][j]*=dp[i−1][j] (exclude S[i−1] OR dp[i−1][j−S[i−1]] (include S[i−1]).
  + Else:
    - d*p*[i][j] = d*p*[i−1][j].

Table Filling:

* + dp[4][4] will determine if a subset exists.

An example:

S={3,2,1,5}, k=4. Result: Subset {3, 1} satisfies the condition.

