Solution: Compilation Order

Let's solve the Compilation Order problem using the Topological Sort pattern.

We'll cover the following

Statement

There are a total of n classes labeled with the English alphabet (A, B, C, and so on). Some classes are dependent on other classes for compilation. For example, if class B extends class A, then B has a dependency on A. Therefore, A must be compiled before B.

Given a list of the dependency pairs, find the order in which the classes should be compiled.

Constraints:

- Class name should be a character.
- $0 \leq$ `dependencies.length` ≤ 5000
- `dependencies[i].length` $= 2$
- All dependency pairs should be unique.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

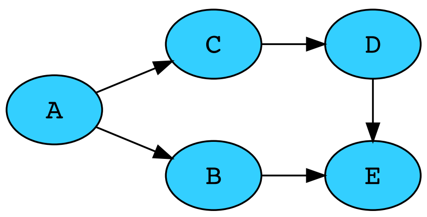
The naive approach is to generate all possible compilation orders for the given classes and then select the ones that satisfy the dependencies.

However, this would be very expensive since there would be an exponential number of possible orders ($n!$, where n is the number of classes) and only a handful of valid ones. The time complexity for this approach is $O(n!)$. The space complexity is $O(1)$.

Optimized approach using topological sort

A more optimized solution to the above problem is topological ordering. Topological sort is used to find a linear ordering of elements that have dependencies on or priority over each other. For example, if A is dependent on B or if B has priority over A, B is listed before A in topological order. Since we're looking for the order of compilation of classes, this problem lends itself naturally to the topological sort pattern.

For this problem, we find the topological order of the given classes using the list of class dependencies. The vertices in the graph represent the classes, and the directed edge represents the dependency relationship.

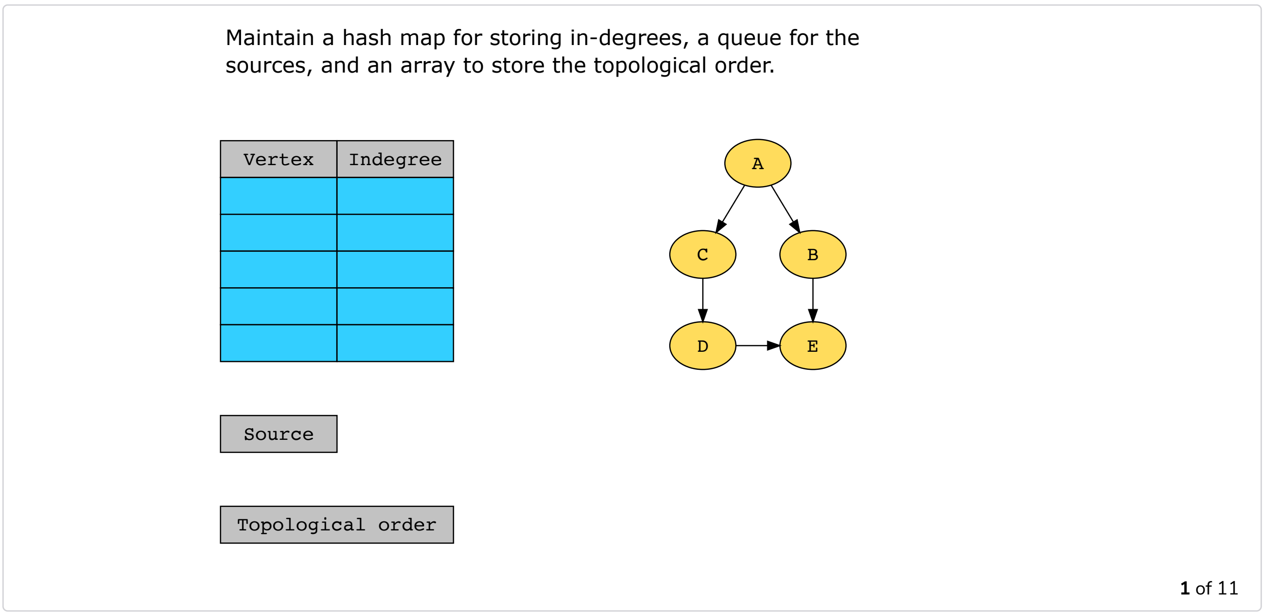


From the example above, we get the order $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$. Two other possible orderings of the classes above can be $A \rightarrow C \rightarrow B \rightarrow D \rightarrow E$ or $A \rightarrow C \rightarrow D \rightarrow B \rightarrow E$. This order of graph vertices is known as a topological sorted order.

We can say that a topological ordering starts with one of the sources and ends at one of the sinks:

- **Source**: Any vertex with no incoming edge and only outgoing edges is called a source.
- **Sink**: Any vertex that has only incoming edges and no outgoing edge is called a sink.

The illustration below gives a rundown of the solution:



Maintain a hash map for storing in-degrees, a queue for the sources, and an array to store the topological order.

Vertex	Indegree

> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

To find the topological sort of a graph, we use a breadth-first search (BFS) approach for traversing the vertices.

We store the graph in adjacency lists in which each parent vertex has a list containing all of its children. For example, in the above slide deck, the adjacency list for vertex A would be $[C, B]$. To find the sources, we maintain a hash map to count the in-degrees, which is the count of incoming edges of a vertex. Any vertex with a 0 in-degree is a source. We store the topological order in a list called sorted order.

Here is how we implement this solution:

1. We initialize the graph with an empty adjacency list for each vertex. The vertices will have an in-degree of 0. If the length of the graph is 0, that is, there are no vertices, we'll return an empty list.

Java

```java
import java.util.*;

class CompilationOrder {

    public static List<Character> findCompilationOrder(ArrayList<ArrayList<Character>> dependencies){
        List<Character> sortedOrder = new ArrayList<>();
        // a. Initialize the graph and inDegree

        HashMap<Character, List<Character>> graph = new HashMap<>();
        HashMap<Character, Integer> inDegree = new HashMap<>();
        System.out.println("\tInitializing the graph and inDegree");
        for( int x = 0; x < dependencies.size(); x++){
            char parent = dependencies.get(x).get(0);
            char child = dependencies.get(x).get(1);
            graph.put(parent, new ArrayList<>());
            graph.put(child, new ArrayList<>());
            inDegree.put(parent, 0);
            inDegree.put(child, 0);
        }
        System.out.println("\t\tGraph: " + graph);
        System.out.println("\t\tinDegree: " + inDegree + "\n");
        if(graph.size() <= 0){
            System.out.println("\tThere are no vertices");
            System.out.println("\t\tReturning the sorted order: " + sortedOrder);
        }
        return sortedOrder;
    }
    public static void main( String args[] ) {
```

Initializing the graph

2. Next, we build our graph from the input dependencies and populate the in-degrees in the hash map. We traverse over the dependency pairs and identify parent and child classes. Then, we add the child to the parent's adjacency list and increment it's in-degree by 1.

Java

```java
import java.util.*;

class CompilationOrder {
    public static List<Character> findCompilationOrder(ArrayList<ArrayList<Character>> dependencies){
        List<Character> sortedOrder = new ArrayList<>();
        // a. Initialize the graph and inDegree

        HashMap<Character, List<Character>> graph = new HashMap<>();
        HashMap<Character, Integer> inDegree = new HashMap<>();
        System.out.println("\tInitializing the graph and inDegree");
        for( int x = 0; x < dependencies.size(); x++){
            char parent = dependencies.get(x).get(0);
            char child = dependencies.get(x).get(1);
            graph.put(parent, new ArrayList<>());
            graph.put(child, new ArrayList<>());
            inDegree.put(parent, 0);
            inDegree.put(child, 0);
        }
        System.out.println("\t\tGraph: " + graph);
        System.out.println("\t\tinDegree: " + inDegree + "\n");
        if(graph.size() <= 0){
            System.out.println("\tThere are no vertices");
            System.out.println("\t\tReturning the sorted order: " + sortedOrder);
            return sortedOrder;
        }
        System.out.println("\tBuilding the graph and populating inDegree");
        // b. Build the graph
        int k = 0;
```

Building the graph

3. The next step is to find all sources, that is, vertices with in-degree $= 0$. These nodes are the first ones to be removed from our graph and added to the sorted order list.

Java

```java
import java.util.*;

class CompilationOrder {

    public static List<Character> findCompilationOrder(ArrayList<ArrayList<Character>> dependencies){
        List<Character> sortedOrder = new ArrayList<>();
        // a. Initialize the graph and inDegree

        HashMap<Character, List<Character>> graph = new HashMap<>();
        HashMap<Character, Integer> inDegree = new HashMap<>();
        System.out.println("\tInitializing the graph and inDegree");
        for( int x = 0; x < dependencies.size(); x++){
            char parent = dependencies.get(x).get(0);
            char child = dependencies.get(x).get(1);
            graph.put(parent, new ArrayList<>());
            graph.put(child, new ArrayList<>());
            inDegree.put(parent, 0);
            inDegree.put(child, 0);
        }
        System.out.println("\t\tGraph: " + graph);
        System.out.println("\t\tinDegree: " + inDegree + "\n");
        if(graph.size() <= 0){
            System.out.println("\tThere are no vertices");
            System.out.println("\t\tReturning the sorted order: " + sortedOrder);
            return sortedOrder;
        }
        System.out.println("\tBuilding the graph and populating inDegree");
        // b. Build the graph
```

Identify the sources

4. We remove the source from the queue and decrement the in-degree of its children by 1. If a child's in-degree becomes 0, we add it to the source queue. We repeat this step until the source queue is empty and all vertices have been visited.

Java

```java
import java.util.*;

class CompilationOrder {
    public static List<Character> findCompilationOrder(ArrayList<ArrayList<Character>> dependencies){
        List<Character> sortedOrder = new ArrayList<>();
        // a. Initialize the graph and inDegree

        HashMap<Character, List<Character>> graph = new HashMap<>();
        HashMap<Character, Integer> inDegree = new HashMap<>();
        System.out.println("\tInitializing the graph and inDegree");
        for( int x = 0; x < dependencies.size(); x++){
            char parent = dependencies.get(x).get(0);
            char child = dependencies.get(x).get(1);
            graph.put(parent, new ArrayList<>());
            graph.put(child, new ArrayList<>());
            inDegree.put(parent, 0);
            inDegree.put(child, 0);
        }
        System.out.println("\t\tGraph: " + graph);
        System.out.println("\t\tinDegree: " + inDegree + "\n");
        if(graph.size() <= 0){
            System.out.println("\tThere are no vertices");
            System.out.println("\t\tReturning the sorted order: " + sortedOrder);
            return sortedOrder;
        }
        System.out.println("\tBuilding the graph and populating inDegree");
        // b. Build the graph
```

Updating the in-degrees and removing the sources from the graph

Our algorithm is now complete and returns a topological ordering for the classes. However, it throws an error if there's a cycle in our dependencies. Therefore, we need to add a condition that caters to this case.

If the length of the sorted order list is not equal to the length of the graph, there's a cycle and we return an empty list.

Java

```
import java.util.*;

class CompilationOrder {

    public static List<Character> findCompilationOrder(ArrayList<ArrayList<Character>> dependencies){
        List<Character> sortedOrder = new ArrayList<>();
        // a. Initialize the graph and inDegree

        HashMap<Character, List<Character>> graph = new HashMap<>();
        HashMap<Character, Integer> inDegree = new HashMap<>();
        System.out.println("\tInitializing the graph and inDegree");
        for( int x = 0; x < dependencies.size(); x++){
            char parent = dependencies.get(x).get(0);
            char child = dependencies.get(x).get(1);
            graph.put(parent, new ArrayList<>());
            graph.put(child, new ArrayList<>());
            inDegree.put(parent, 0);
            inDegree.put(child, 0);
        }
        System.out.println("\t\tGraph: " + graph);
        System.out.println("\t\tinDegree: " + inDegree + "\n");
        if(graph.size() <= 0){
            System.out.println("\tThere are no vertices");
            System.out.println("\t\tReturning the sorted order: " + sortedOrder);
            return sortedOrder;
        }
        System.out.println("\tBuilding the graph and populating inDegree");
```

Compilation Order

Just the code

Here's the complete solution to this problem:

Java

```
import java.util.*;
```

```java

class CompilationOrder {
    public static String repeat(String str, int pValue){
        String out = "";
        for(int i = 0; i < pValue; i++)
        {
            out += str;
        }
        return out;
    }
    public static List<Character> findCompilationOrder(ArrayList<ArrayList<Character>> dependencies){
        List<Character> sortedOrder = new ArrayList<>();

        HashMap<Character, List<Character>> graph = new HashMap<>();
        HashMap<Character, Integer> inDegree = new HashMap<>();
        for( int x = 0; x < dependencies.size(); x++){
            char parent = dependencies.get(x).get(0);
            char child = dependencies.get(x).get(1);
            graph.put(parent, new ArrayList<>());
            graph.put(child, new ArrayList<>());
            inDegree.put(parent, 0);
            inDegree.put(child, 0);
        }
        if(graph.size() <= 0){
            return sortedOrder;
        }
```

Solution summary

To recap, the solution to this problem can be divided into the following parts:

1. Build a graph from the dependencies list and keep track of the in-degrees of vertices in a hash map.
2. Populate the sources deque with vertices with in-degrees $= 0$.
3. Add the sources to the sorted list.
4. Remove the sources from the graph and update the in-degrees of their children. If the in-degree of a child becomes 0, it becomes a source in the next iteration.
5. Repeat until all vertices are visited.

Time complexity

The time complexity of the above algorithm is $O(V + E)$, where V is the total number of vertices and E is the total number of edges in the graph.

Space complexity

The space complexity is $O(V)$ since we are creating a deque data structure that will have $O(V)$ elements in the worst case. We are also maintaining a hash table with the in-degree of the vertices. Its size is $O(V)$ as well.

← Back
Compilation Order

Next →
Alien Dictionary

Mark as Completed