**Uber Interview**

Introduction

Uber is one of the largest unicorn companies in the world. Uber’s interview is very concise and many of their interview questions are straightforward. Your technical on-site interviews may be done on an actual laptop, or it might be on a whiteboard. You will also have one non-technical discussion with a hiring manager on aspects such as culture fit during the interview.

We organized this list so you can get well-prepared for your Uber interview.

Arrays and Strings

 Two Sum

 Find First and Last Position of Element in Sorted Array

 Group Anagrams

**Text Justification**

 Minimum Window Substring

 Validate IP Address

Linked Lists

 Merge Two Sorted Lists

 Merge k Sorted Lists

Trees and Graphs

**Reconstruct Itinerary**

**Evaluate Division**

 Find Duplicate Subtrees

**Print Binary Tree**

 Serialize and Deserialize N-ary Tree

Heaps, Queues, and, Stacks

 Trapping Rain Water

 Basic Calculator

 Meeting Rooms II

 Task Scheduler

**Exclusive Time of Functions**

**Employee Free Time**

Recursion and Backtracking

 Letter Combinations of a Phone Number

 Subsets

Sorting and Searching

 Merge Intervals

 Word Ladder II

 Number of Islands

 Walls and Gates

 Remove Invalid Parentheses

**24 Game**

**Max Area of Island**

**Bus Routes**

**Shortest Bridge**

Dynamic Programming

 Word Break

 Perfect Squares

**Russian Doll Envelopes**

**Maximum Vacation Days**

**Cherry Pickup**

Design

 LRU Cache

 Serialize and Deserialize Binary Tree

 Moving Average from Data Stream

**Design Snake Game**

 Logger Rate Limiter

**Design Hit Counter**

**Insert Delete GetRandom O(1) - Duplicates allowed**

 Design Search Autocomplete System

Others

 Valid Sudoku

 Candy

 Fraction to Recurring Decimal

**Number of Islands II**

**Random Pick Index**

 Encode and Decode TinyURL

**Solve the Equation**

**Construct Quad Tree**

**Random Pick with Weight**

**Text Justification**

Given an array of words and a width *maxWidth*, format the text such that each line has exactly *maxWidth* characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly *maxWidth* characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no **extra** space is inserted between words.

**Note:**

* A word is defined as a character sequence consisting of non-space characters only.
* Each word's length is guaranteed to be greater than 0 and not exceed *maxWidth*.
* The input array words contains at least one word.

**Example 1:**

**Input:** words = ["This", "is", "an", "example", "of", "text", "justification."], maxWidth = 16

**Output:**

[

   "This    is    an",

   "example  of text",

   "justification.  "

]

**Example 2:**

**Input:** words = ["What","must","be","acknowledgment","shall","be"], maxWidth = 16

**Output:**

[

  "What   must   be",

  "acknowledgment  ",

  "shall be        "

]

**Explanation:** Note that the last line is "shall be " instead of "shall be", because the last line must be left-justified instead of fully-justified.

Note that the second line is also left-justified becase it contains only one word.

**Example 3:**

**Input:** words = ["Science","is","what","we","understand","well","enough","to","explain","to","a","computer.","Art","is","everything","else","we","do"], maxWidth = 20

**Output:**

[

  "Science  is  what we",

"understand      well",

  "enough to explain to",

  "a  computer.  Art is",

  "everything  else  we",

  "do                  "

]

**Constraints:**

* 1 <= words.length <= 300
* 1 <= words[i].length <= 20
* words[i] consists of only English letters and symbols.
* 1 <= maxWidth <= 100
* words[i].length <= maxWidth

**Reconstruct Itinerary**

Given a list of airline tickets represented by pairs of departure and arrival airports [from, to], reconstruct the itinerary in order. All of the tickets belong to a man who departs from JFK. Thus, the itinerary must begin with JFK.

**Note:**

1. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary ["JFK", "LGA"] has a smaller lexical order than ["JFK", "LGB"].
2. All airports are represented by three capital letters (IATA code).
3. You may assume all tickets form at least one valid itinerary.
4. One must use all the tickets once and only once.

**Example 1:**

**Input:** [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]

**Output:** ["JFK", "MUC", "LHR", "SFO", "SJC"]

**Example 2:**

**Input:** [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]

**Output:** ["JFK","ATL","JFK","SFO","ATL","SFO"]

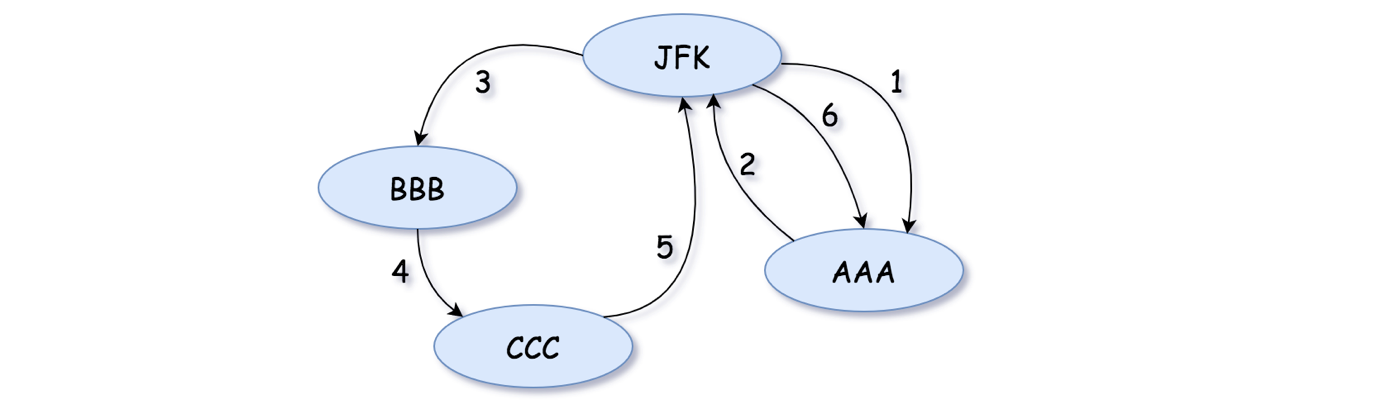
**Explanation:** Another possible reconstruction is ["JFK","SFO","ATL","JFK","ATL","SFO"].

  But it is larger in lexical order.

## Solution

#### **Overview**

Overall, we could consider this problem as a **graph traversal** problem, where an airport can be viewed as a vertex in graph and flight between airports as an edge in graph.



We would like to make a few clarification on the input of the problem, since it is not clear in the description of the problem.

As one might notice in the above example, the input graph is NOT what we call a **DAG** (Directed Acyclic Graph), since we could find at least a cycle in the graph.

In addition, the graph could even have some duplicate edges (i.e. we might have multiple flights with the same origin and destination).

#### **Approach 1: Backtracking + Greedy**

**Intuition**

As common strategies for problems of graph traversal, we often apply the methodologies of **backtracking** or **greedy**. As it turns out, we can apply both of them for this problem.

Typically, [backtracking](https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/) is used to enumerate all possible solutions for a problem, in a trial-fail-and-fallback strategy.

At each airport, one might have several possible destinations to fly to. With backtracking, we enumerate each possible destination. We mark the choice at each iteration (i.e. trial) before we move on to the chosen destination. If the destination does not lead to a solution (i.e. fail), we would then fallback to the previous state and start another iteration of trial-fail-and-fallback cycle.

A [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) is any algorithm that follows the problem-solving heuristic of making locally optimal choice at each step, with the intent of reaching the global optimum at the end.

As suggested by its definition, a greedy algorithm does not necessarily lead to a globally optimal solution, but rather a reasonable approximation in exchange of less computing time.

Nonetheless, sometimes it is the way to produce a global optimum for certain problems. This is the case for this problem as well.

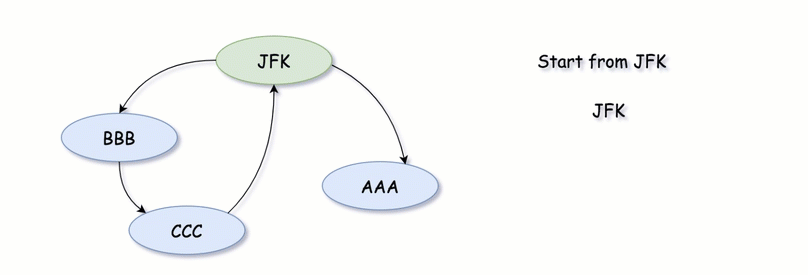
At each airport, given a list of possible destinations, while backtracking, at each step we would pick the destination **greedily** in lexical order, i.e. the one with the smallest lexical order would have its trial first.

With this **greedy** strategy, we would ensure that the final solution that we find would have the smallest lexical order, because all other solutions that have smaller lexical order have been trialed and failed during the process of backtracking.

**Algorithm**

Here we explain how we implement a solution for this problem, by combining the strategies of backtracking and greedy.

* As the first step, we build a graph data structure from the given input. This graph should allow us to quickly identify a list of potential destinations, given an origin. Here we adopted the hashmap (or dictionary) data structure, with each entry as <origin, [destinations]>.
* Then due to our greedy strategy, we then should order the destination list for each entry in lexical order. As an alternative solution, one could use PriorityQueue data structure in the first step to keep the list of destinations, which would maintain the order at the moment of constructing the list.
* As the final step, we kick off the backtracking traversal on the above graph, to obtain the final result.
  + At the beginning of the backtracking function, as the bottom case, we check if we have already obtained a valid itinerary.
  + Otherwise, we enumerate the next destinations in order.
  + We mark the status of visit, before and after each backtracking loop.



Note that there is certain code pattern that one can follow in order to implement an algorithm of backtracking. We provide an example in the [Explore card of Recursion II](https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/2793/).

|  |
| --- |
| class Solution {  // origin -> list of destinations  HashMap<String, List<String>> flightMap = new HashMap<>();  HashMap<String, boolean[]> visitBitmap = new HashMap<>();  int flights = 0;  List<String> result = null;  public List<String> findItinerary(List<List<String>> tickets) {  // Step 1). build the graph first  for (List<String> ticket : tickets) {  String origin = ticket.get(0);  String dest = ticket.get(1);  if (this.flightMap.containsKey(origin)) {  List<String> destList = this.flightMap.get(origin);  destList.add(dest);  } else {  List<String> destList = new LinkedList<String>();  destList.add(dest);  this.flightMap.put(origin, destList);  }  }  // Step 2). order the destinations and init the visit bitmap  for (Map.Entry<String, List<String>> entry : this.flightMap.entrySet()) {  Collections.sort(entry.getValue());  this.visitBitmap.put(entry.getKey(), new boolean[entry.getValue().size()]);  }  this.flights = tickets.size();  LinkedList<String> route = new LinkedList<String>();  route.add("JFK");  // Step 3). backtracking  this.backtracking("JFK", route);  return this.result;  }  protected boolean backtracking(String origin, LinkedList<String> route) {  if (route.size() == this.flights + 1) {  this.result = (List<String>) route.clone();  return true;  }  if (!this.flightMap.containsKey(origin))  return false;  int i = 0;  boolean[] bitmap = this.visitBitmap.get(origin);  for (String dest : this.flightMap.get(origin)) {  if (!bitmap[i]) {  bitmap[i] = true;  route.add(dest);  boolean ret = this.backtracking(dest, route);  route.pollLast();  bitmap[i] = false;  if (ret)  return true;  }  ++i;  }  return false;  }  } |

**Complexity**

* Time Complexity: \mathcal{O}(|E|^d)O(∣*E*∣*d*) where |E|∣*E*∣ is the number of total flights and d*d* is the maximum number of flights from an airport.
  + It is tricky to estimate the time complexity of the backtracking algorithm, since the algorithm often has an early stopping depending on the input.
  + To calculate a loose upper bound for the time complexity, let us consider it as a combination problem where the goal is to construct a sequence of a specific order, i.e. |V\_1V\_2...V\_n|∣*V*1​*V*2​...*Vn*​∣. For each position V\_i*Vi*​, we could have d*d* choices, i.e. at each airport one could have at most d*d* possible destinations. Since the length of the sequence is |E|∣*E*∣, the total number of combination would be |E|^d∣*E*∣*d*.
  + In the worst case, our backtracking algorithm would have to enumerate all possible combinations.
* Space Complexity: \mathcal{O}(|V| + |E|)O(∣*V*∣+∣*E*∣) where |V|∣*V*∣ is the number of airports and |E|∣*E*∣ is the number of flights.
  + In the algorithm, we use the graph as well as the visit bitmap, which would require the space of |V| + |E|∣*V*∣+∣*E*∣.
  + Since we applied recursion in the algorithm, which would incur additional memory consumption in the function call stack. The maximum depth of the recursion would be exactly the number of flights in the input, i.e. |E|∣*E*∣.
  + As a result, the total space complexity of the algorithm would be \mathcal{O}(|V| + 2\cdot|E|) = \mathcal{O}(|V| + |E|)O(∣*V*∣+2⋅∣*E*∣)=O(∣*V*∣+∣*E*∣).

#### **Approach 2: Hierholzer's Algorithm**

**Eulerian Cycle**

In graph theory, an Eulerian trail (or **Eulerian path**) is a trail in a finite graph that visits every edge exactly once (allowing for revisiting vertices).

In our problem, we are asked to construct an itinerary that uses all the flights (edges), starting from the airport of "JFK". As one can see, the problem is actually a variant of [Eulerian path](https://en.wikipedia.org/wiki/Eulerian_path), with a fixed starting point.

Similarly, an Eulerian circuit or **Eulerian cycle** is an Eulerian trail that starts and ends on the same vertex.

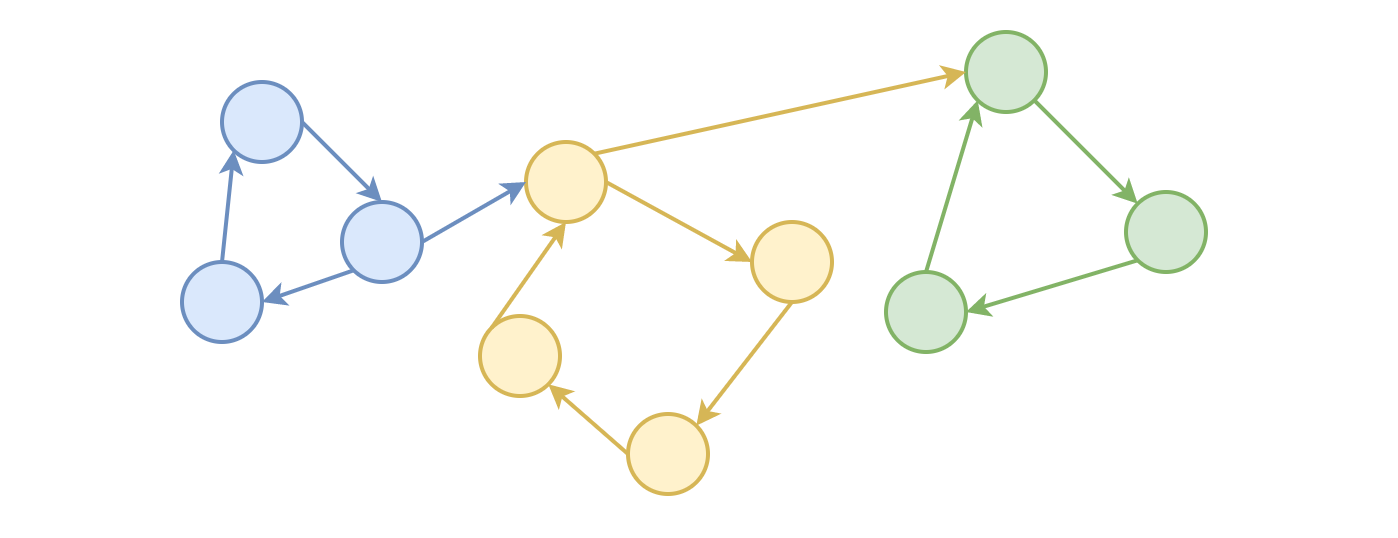
The Eulerian cycle problem has been discussed by [Leonhard Euler](https://en.wikipedia.org/wiki/Leonhard_Euler) back in 1736. Ever since, there have been several algorithms proposed to solve the problem.

In 1873, Hierholzer proposed an efficient algorithm to find the Eulerian cycle in linear time (\mathcal{O}(|E|)O(∣*E*∣)). One could find more details about the Hierholzer's algorithm in this [course](https://www-m9.ma.tum.de/graph-algorithms/hierholzer/index_en.html).

The basic idea of Hierholzer's algorithm is the stepwise construction of the Eulerian cycle by connecting disjunctive circles.

To be more specific, the algorithm consists of two steps:

* It starts with a random node and then follows an arbitrary unvisited edge to a neighbor. This step is repeated until one returns to the starting node. This yields a first circle in the graph.
* If this circle covers all nodes it is an Eulerian cycle and the algorithm is finished. Otherwise, one chooses another node among the cycles' nodes with unvisited edges and constructs another circle, called subtour.



By connecting all the circles in the above process, we build the Eulerian cycle at the end.

**Eulerian Path**

To find the Eulerian path, inspired from the original Hierzolher's algorithm, we simply change one condition of loop, rather than stopping at the starting point, we stop at the vertex where we do not have any unvisited edges.

To summarize, the main idea to find the Eulerian path consists of two steps:

* Step 1). Starting from any vertex, we keep following the unused edges until we get **stuck** at certain vertex where we have no more unvisited outgoing edges.
* Step 2). We then backtrack to the nearest neighbor vertex in the current path that has unused edges and we **repeat** the process until all the edges have been used.

The first vertex that we got stuck at would be the **end point** of our ***Eulerian path***. So if we follow all the stuck points backwards, we could reconstruct the Eulerian path at the end.

**Algorithm**

Now let us get back to our itinerary reconstruction problem. As we know now, it is a problem of Eulerian path, except that we have a fixed starting point.

More importantly, as stated in the problem, the given input is guaranteed to have a solution. So we have one less issue to consider.

As a result, our final algorithm is a bit simpler than the above Eulerian path algorithm, without the backtracking step.

The essential step is that starting from the fixed starting vertex (airport 'JFK'), we keep following the ordered and unused edges (flights) until we get **stuck** at certain vertex where we have no more unvisited outgoing edges.

The point that we got stuck would be the last airport that we visit. And then we follow the visited vertex (airport) **backwards**, we would obtain the final itinerary.

Here are some sample implementations which are inspired from a [thread of discussion](https://leetcode.com/problems/reconstruct-itinerary/discuss/78768/Short-Ruby-Python-Java-C%2B%2B) in the forum.

|  |
| --- |
| class Solution {  // origin -> list of destinations  HashMap<String, LinkedList<String>> flightMap = new HashMap<>();  LinkedList<String> result = null;  public List<String> findItinerary(List<List<String>> tickets) {  // Step 1). build the graph first  for(List<String> ticket : tickets) {  String origin = ticket.get(0);  String dest = ticket.get(1);  if (this.flightMap.containsKey(origin)) {  LinkedList<String> destList = this.flightMap.get(origin);  destList.add(dest);  } else {  LinkedList<String> destList = new LinkedList<String>();  destList.add(dest);  this.flightMap.put(origin, destList);  }  }  // Step 2). order the destinations  this.flightMap.forEach((key, value) -> Collections.sort(value));  this.result = new LinkedList<String>();  // Step 3). post-order DFS  this.DFS("JFK");  return this.result;  }  protected void DFS(String origin) {  // Visit all the outgoing edges first.  if (this.flightMap.containsKey(origin)) {  LinkedList<String> destList = this.flightMap.get(origin);  while (!destList.isEmpty()) {  // while we visit the edge, we trim it off from graph.  String dest = destList.pollFirst();  DFS(dest);  }  }  // add the airport to the head of the itinerary  this.result.offerFirst(origin);  }  } |

**Discussion**

To better understand the above algorithm, we could look at it from another perspective.

Actually, we could consider the algorithm as the **postorder DFS** (Depth-First Search) in a directed graph, from a fixed starting point.

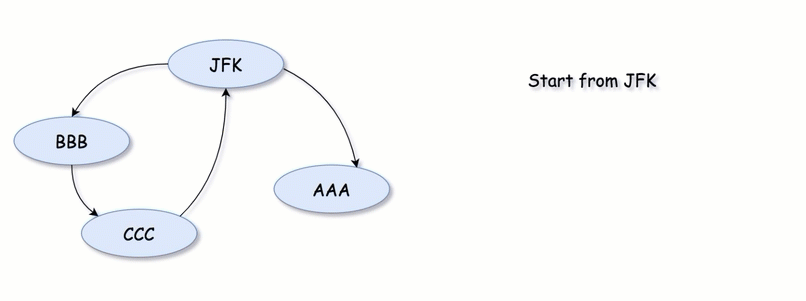
As we know that, each input is guaranteed to have a solution. Therefore, the task of the problem can be interpreted as that given a list of flights (*i.e.* edges in graph), we should find an order to use each flight *once and only once*.

In the resulted path, before we visit the last airport (denoted as V), we can say that we have already used all the rest flights, *i.e.* if there is any flight starting from V, then we must have already taken that before.

Or to put it another way, before adding the last airport (vertex) in the final path, we have visited all its **outgoing** vertex.

Actually, the above statement applies to each airport in the final itinerary. *Before adding an airport into the final itinerary, we must first visit all its outgoing neighbor vertex.*

If we consider the outgoing vertex in a directed graph as children nodes in a tree, one could see the reason why we could consider the algorithm as a sort of **postorder DFS traversal** in a tree.



**Complexity**

* Time Complexity: \mathcal{O}(|E| \log{\frac{|E|}{|V|}})O(∣*E*∣log∣*V*∣∣*E*∣​) where |E|∣*E*∣ is the number of edges (flights) in the input.
  + As one can see from the above algorithm, during the DFS process, we would traverse each edge once. Therefore, the complexity of the DFS function would be |E|∣*E*∣.
  + However, before the DFS, we need to sort the outgoing edges for each vertex. And this, unfortunately, dominates the overall complexity.
  + It is though tricky to estimate the complexity of sorting, which depends on the structure of the input graph.
  + In the worst case where the graph is not balanced, i.e. the connections are concentered in a single airport. Imagine the graph is of star shape, in this case, the JFK airport would assume half of the flights (since we still need the return flight). As a result, the sorting operation on this airport would be exceptionally expensive, i.e. N \log{N}*N*log*N*, where N = \frac{|E|}{2}*N*=2∣*E*∣​. And this would be the final complexity as well, since it dominates the rest of the calculation.
  + Let us consider a less bad case, or an average case, where the graph is less clustered, i.e. each node has the equal number of outgoing flights. Under this assumption, each airport would have \frac{|E|}{(2\cdot|V|)}(2⋅∣*V*∣)∣*E*∣​ number of flights (still we need the return flights). Again, we can plug it into the N \log N*N*log*N* minimal sorting complexity. In addition, this time, we need to take into consideration all airports, rather than the superhub (JFK) in the above case. As a result, we have |V| \cdot (N \log N)∣*V*∣⋅(*N*log*N*), where N = \frac{|E|}{2\cdot|V|}*N*=2⋅∣*V*∣∣*E*∣​. If we expand the formula, we will obtain the complexity of the average case as \mathcal{O}(\frac{|E|}{2} \log{\frac{|E|}{2\cdot|V|}}) = \mathcal{O}(|E| \log{\frac{|E|}{|V|}})O(2∣*E*∣​log2⋅∣*V*∣∣*E*∣​)=O(∣*E*∣log∣*V*∣∣*E*∣​)
* Space Complexity: \mathcal{O}(|V| + |E|)O(∣*V*∣+∣*E*∣) where |V|∣*V*∣ is the number of airports and |E|∣*E*∣ is the number of flights.
  + In the algorithm, we use the graph, which would require the space of |V| + |E|∣*V*∣+∣*E*∣.
  + Since we applied recursion in the algorithm, which would incur additional memory consumption in the function call stack. The maximum depth of the recursion would be exactly the number of flights in the input, i.e. |E|∣*E*∣.
  + As a result, the total space complexity of the algorithm would be \mathcal{O}(|V| + 2\cdot|E|) = \mathcal{O}(|V| + |E|)O(∣*V*∣+2⋅∣*E*∣)=O(∣*V*∣+∣*E*∣).

**Evaluate Division**

You are given an array of variable pairs equations and an array of real numbers values, where equations[i] = [Ai, Bi] and values[i] represent the equation Ai / Bi = values[i]. Each Ai or Bi is a string that represents a single variable.

You are also given some queries, where queries[j] = [Cj, Dj] represents the jth query where you must find the answer for Cj / Dj = ?.

Return the answers to all queries. If a single answer cannot be determined, return -1.0.

**Note:** The input is always valid. You may assume that evaluating the queries will not result in division by zero and that there is no contradiction.

**Example 1:**

**Input:** equations = [["a","b"],["b","c"]], values = [2.0,3.0], queries = [["a","c"],["b","a"],["a","e"],["a","a"],["x","x"]]

**Output:** [6.00000,0.50000,-1.00000,1.00000,-1.00000]

**Explanation:**

Given: a / b = 2.0, b / c = 3.0

queries are: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ?

return: [6.0, 0.5, -1.0, 1.0, -1.0 ]

**Example 2:**

**Input:** equations = [["a","b"],["b","c"],["bc","cd"]], values = [1.5,2.5,5.0], queries = [["a","c"],["c","b"],["bc","cd"],["cd","bc"]]

**Output:** [3.75000,0.40000,5.00000,0.20000]

**Example 3:**

**Input:** equations = [["a","b"]], values = [0.5], queries = [["a","b"],["b","a"],["a","c"],["x","y"]]

**Output:** [0.50000,2.00000,-1.00000,-1.00000]

**Constraints:**

* 1 <= equations.length <= 20
* equations[i].length == 2
* 1 <= Ai.length, Bi.length <= 5
* values.length == equations.length
* 0.0 < values[i] <= 20.0
* 1 <= queries.length <= 20
* queries[i].length == 2
* 1 <= Cj.length, Dj.length <= 5
* Ai, Bi, Cj, Dj consist of lower case English letters and digits.

Do you recognize this as a graph problem?

## Solution

#### **Overview**

As revealed by the hints, the problem can be solved with two important data structures, namely Graph and Union-Find.

In the following sections, we will explain how to solve the problem respectively with regards to the data structures.

#### **Approach 1: Path Search in Graph**

**Intuition**

First, let us look at the example given in the problem description. Given two equations, namely \frac{a}{b} = 2, \space \frac{b}{c} = 3*ba*​=2, *cb*​=3, we could derive the following equations:

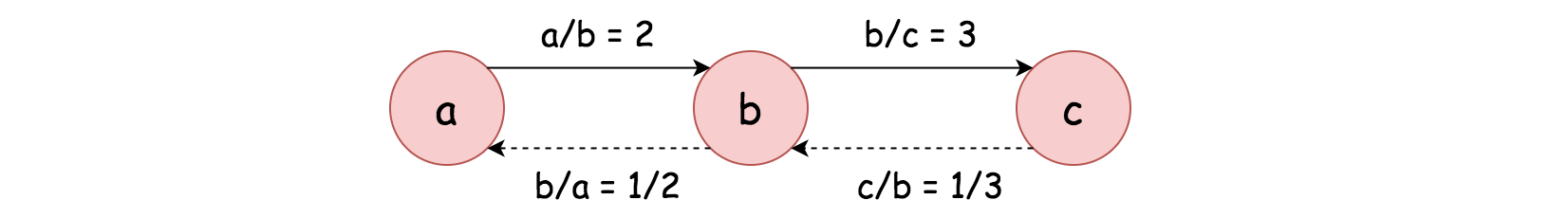
* 1). \frac{b}{a} = \frac{1}{2}, \space \frac{c}{b} = \frac{1}{3}*ab*​=21​, *bc*​=31​
* 2). \frac{a}{c} = \frac{a}{b} \cdot \frac{b}{c} = 2 \cdot 3 = 6*ca*​=*ba*​⋅*cb*​=2⋅3=6

Each division implies the reverse of the division, which is how we derive the equations in **(1)**. While by **chaining up** equations, we could obtain new equations in **(2)**.

We could reformulate the equations with the graph data structure, where each variable can be represented as a **node** in the graph, and the division relationship between variables can be modeled as **edge** with direction and weight.

The direction of edge indicates the order of division, and the weight of edge indicates the result of division.

With the above formulation, we then can convert the initial equations into the following graph:



To evaluate a query (e.g. \frac{a}{c}=?*ca*​=?) is equivalent to performing two tasks on the graph: 1). find if there exists a path between the two entities. 2). if so, calculate the cumulative products along the paths.

In the above example (\frac{a}{c}=?*ca*​=?), we could find a path between them, and the cumulative products are 66. As a result, we can conclude that the result of \frac{a}{c}*ca*​ is 2 \cdot 3 = 62⋅3=6.

**Algorithm**

As one can see, we just transform the problem into a **path searching** problem in a graph.

More precisely, we can reinterpret the problem as "given two nodes, we are asked to check if there exists a path between them. If so, we should return the cumulative products along the path as the result.

Given the above problem statement, it seems intuitive that one could apply the backtracking algorithm, or sometimes people might call it DFS (Depth-First Search).

Essentially, we can break down the algorithm into two steps overall:

* Step 1). we build the graph out of the list of input equations.
  + Each equation corresponds to two edges in the graph.
* Step 2). once the graph is built, we then can evaluate the query one by one.
  + The evaluation of the query is done via searching the path between the given two variables.
  + Other than the above searching operation, we need to handle two exceptional cases as follows:
  + Case 1): if either of the nodes does not exist in the graph, i.e. the variables did not appear in any of the input equations, then we can assert that no path exists.
  + Case 2): if the origin and the destination are the same node, i.e. \frac{a}{a}*aa*​, we can assume that there exists an invisible self-loop path for each node and the result is one.

Here we give one sample implementation on the backtracking algorithm.

|  |
| --- |
| class Solution {  public double[] calcEquation(List<List<String>> equations, double[] values,  List<List<String>> queries) {  HashMap<String, HashMap<String, Double>> graph = new HashMap<>();  // Step 1). build the graph from the equations  for (int i = 0; i < equations.size(); i++) {  List<String> equation = equations.get(i);  String dividend = equation.get(0), divisor = equation.get(1);  double quotient = values[i];  if (!graph.containsKey(dividend))  graph.put(dividend, new HashMap<String, Double>());  if (!graph.containsKey(divisor))  graph.put(divisor, new HashMap<String, Double>());  graph.get(dividend).put(divisor, quotient);  graph.get(divisor).put(dividend, 1 / quotient);  }  // Step 2). Evaluate each query via bactracking (DFS)  // by verifying if there exists a path from dividend to divisor  double[] results = new double[queries.size()];  for (int i = 0; i < queries.size(); i++) {  List<String> query = queries.get(i);  String dividend = query.get(0), divisor = query.get(1);  if (!graph.containsKey(dividend) || !graph.containsKey(divisor))  results[i] = -1.0;  else if (dividend == divisor)  results[i] = 1.0;  else {  HashSet<String> visited = new HashSet<>();  results[i] = backtrackEvaluate(graph, dividend, divisor, 1, visited);  }  }  return results;  }  private double backtrackEvaluate(HashMap<String, HashMap<String, Double>> graph, String currNode, String targetNode, double accProduct, Set<String> visited) {  // mark the visit  visited.add(currNode);  double ret = -1.0;  Map<String, Double> neighbors = graph.get(currNode);  if (neighbors.containsKey(targetNode))  ret = accProduct \* neighbors.get(targetNode);  else {  for (Map.Entry<String, Double> pair : neighbors.entrySet()) {  String nextNode = pair.getKey();  if (visited.contains(nextNode))  continue;  ret = backtrackEvaluate(graph, nextNode, targetNode,  accProduct \* pair.getValue(), visited);  if (ret != -1.0)  break;  }  }  // unmark the visit, for the next backtracking  visited.remove(currNode);  return ret;  }  } |

Note: with the built graph, one could also apply the **BFS** (Breadth-First Search) algorithm, as opposed to the DFS algorithm we employed.

However, the essence of the solution remains the same, i.e. we are searching for a path in a graph.

**Complexity Analysis**

Let N*N* be the number of input equations and M*M* be the number of queries.

* Time Complexity: \mathcal{O}(M \cdot N)O(*M*⋅*N*)
  + First of all, we iterate through the equations to build a graph. Each equation takes \mathcal{O}(1)O(1) time to process. Therefore, this step will take \mathcal{O}(N)O(*N*) time in total.
  + For each query, we need to traverse the graph. In the worst case, we might need to traverse the entire graph, which could take \mathcal{O}(N)O(*N*). Hence, in total, the evaluation of queries could take M \cdot \mathcal{O}(N) = \mathcal{O}(M \cdot N)*M*⋅O(*N*)=O(*M*⋅*N*).
  + To sum up, the overall time complexity of the algorithm is \mathcal{O}(N) + \mathcal{O}(M \cdot N) = \mathcal{O}(M \cdot N)O(*N*)+O(*M*⋅*N*)=O(*M*⋅*N*)
* Space Complexity: \mathcal{O}(N)O(*N*)
  + We build a graph out the equations. In the worst case where there is no overlapping among the equations, we would have N*N* edges and 2N2*N* nodes in the graph. Therefore, the sapce complexity of the graph is \mathcal{O}(N + 2N) = \mathcal{O}(3N) = \mathcal{O}(N)O(*N*+2*N*)=O(3*N*)=O(*N*).
  + Since we employ the recursion in the backtracking, we would consume additional memory in the function call stack, which could amount to \mathcal{O}(N)O(*N*) space.
  + In addition, we used a set visited to keep track of the nodes we visited during the backtracking. The space complexity of the visited set would be \mathcal{O}(N)O(*N*).
  + To sum up, the overall space complexity of the algorithm is \mathcal{O}(N) + \mathcal{O}(N) + \mathcal{O}(N) = \mathcal{O}(N)O(*N*)+O(*N*)+O(*N*)=O(*N*).
  + Note that we did not take into account the space needed to hold the results. Otherwise, the total space complexity would be \mathcal{O}(N + M)O(*N*+*M*).

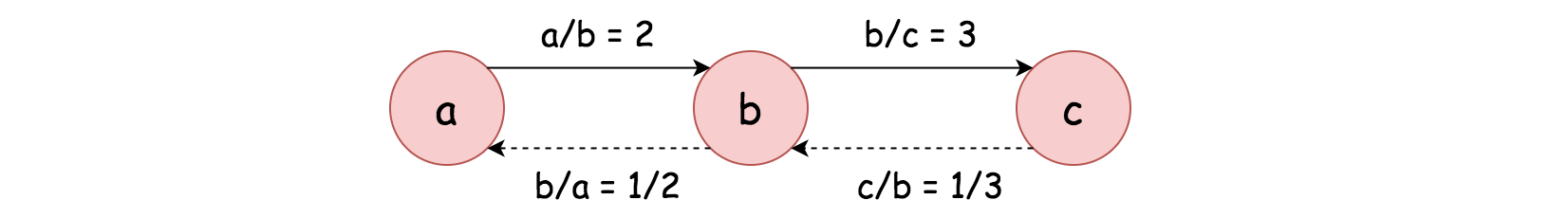
#### **Approach 2: Union-Find with Weights**

**Intuition**

As we mentioned before, the problem can also be solved with the **Union-Find** data structure and algorithm.

As a reminder, the [Union-Find](https://en.wikipedia.org/wiki/Disjoint-set_data_structure) data structure, also known as Disjoint Set, is used to track a set of elements partitioned into a number of disjoint (non-overlapping) subsets. The Union-Find data structure is often applied to solve the **graph partition** problem, where we partition a graph into a set of inter-connected subgraph.

Given the above description, it is not immediately evident that how we can apply the algorithm in this problem. To get familiar with the Union-Find data structure, we would recommend one to check out another problem called [Largest Component Size by Common Factor](https://leetcode.com/problems/largest-component-size-by-common-factor/), where we can apply the Union-Find algorithm in a more **canonical** way.



One thing is clear though. Thanks to the characteristic of the Union-Find data structure, we can easily determine whether the nodes of a and c belong to the same group in the above graph, which accomplishes the first task that we need to perform, i.e. determining if there exists a path between two nodes.

However, one important task is missing, which is how can we calculate the cumulative product along the path, with the Union-Find data structure.

As a spoiler alert, it suffices to adapt the Union-Find data structure and algorithm a little bit.

**Customized Union-Find Data Structure**

The name of Union-Find data structure is originated from the fact that it mainly consists of two operations: Union() and Find() defined as follows:

* Find(x): get the identity of the group that the element x belongs to.
* Union(x, y): merge the two groups that the two elements belong to respectively.

Now, here are the adaptions that we will do. Or more precisely, here are a few **behaviors** that our customized Union-Find data structure would possess at the end.

First of all, essentially we will build a table which contains an entry for each node in the graph, with the help of Union-Find.

The entry is defined as key -> (group\_id, weight). For example, initially, given a node a, its entry in the table would look like 'a' -> ('a', 1), where the first 'a'indicates the id of the node, the second 'a' indicates the id of the group that the node belongs to, and finally the value 1 indicates the weight of the node.

With the above definitions, the tasks become simple. Given two nodes (variables a and b) with entries as (a\_group\_id, a\_weight) and (b\_group\_id, b\_weight) respectively, to evaluate the query of \frac{a}{b} = ?*ba*​=?, we only need to perform the following two calculations:

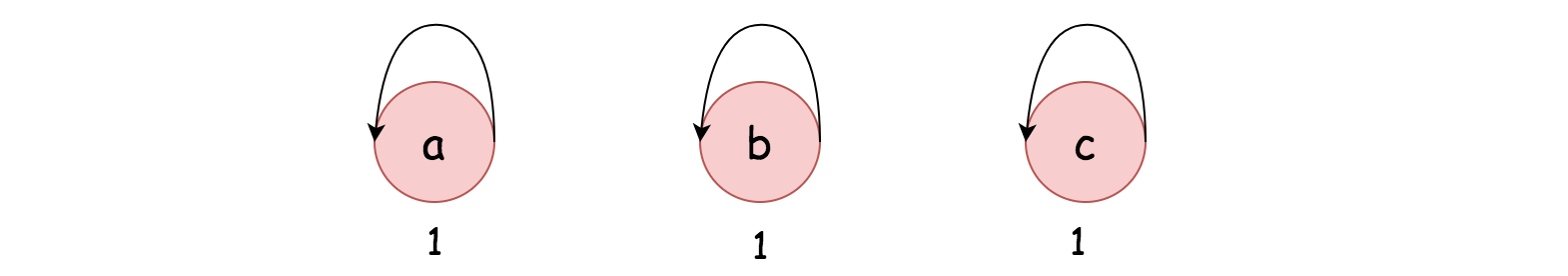
* a\_group\_id == b\_group\_id: If so, they belong to the same group, i.e. there exists a path between them.
* a\_weight / b\_weight: If the above condition holds, by dividing over the **relative** weights that are assigned to the variables, we then can obtain the result of \frac{a}{b}*ba*​ at the end.

Now it all boils down to how we can build the above table with the help of Union-Find algorithm.

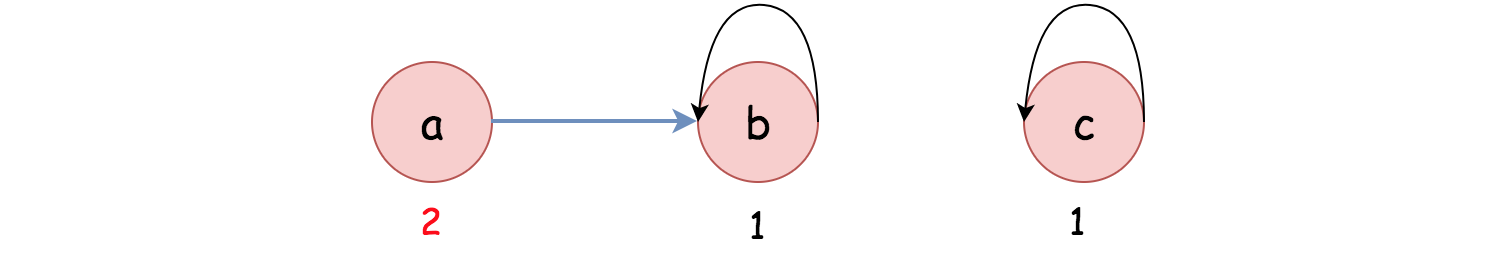
Again, let us look at the same example we presented before.

We have two equations as input, namely \frac{a}{b} = 2, \space \frac{b}{c} = 3*ba*​=2, *cb*​=3.

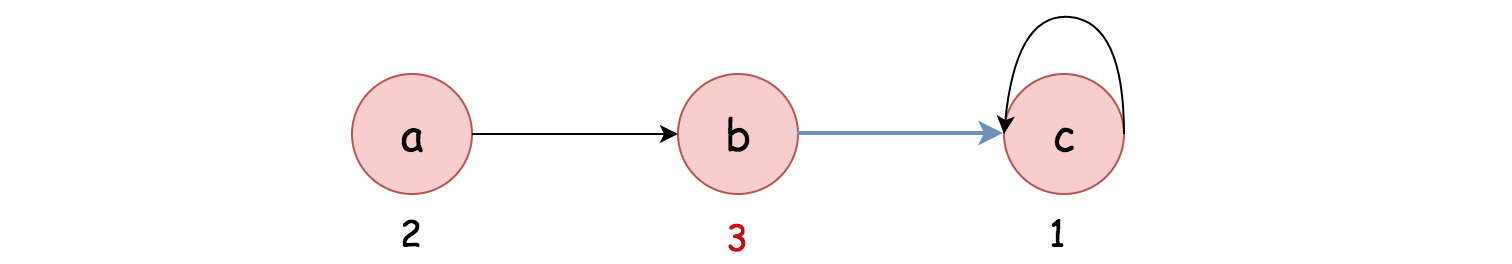
* Initially, the entries for each variable would look like the following, where the group\_id of each variable is the variable itself and the weight of each variable is 1. Each variable forms a group on its own, since there is no relationship among them at the moment.



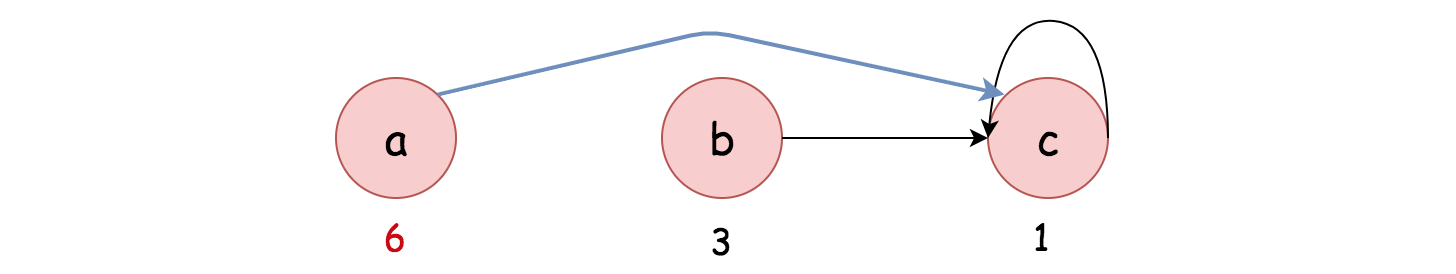
* Now if we process the equation \frac{a}{b}=2*ba*​=2, by joining (**Union** operation) the two groups that the variables a and b belong to, we would obtain the results as shown in the following graph. More precisely, we attach the group of dividend a to the one of the divisor b. Meanwhile, we would also update the relative weight of the group a to reflect the ratio between the two variables.



* Similarly, we continue to process the equation of \frac{b}{c}=3*cb*​=3, by joining (**Union** operation) the groups of b and c together. Similarly, we attach the group of dividend b to the one of divisor c. And also we update the weight of the group b to reflect the ratio between the two variables.



* As one might notice, there is some inconsistency in the above graph, i.e. the group\_id of the variable of a should then be c and the weight of the variable a should be 6 rather than 2. Indeed, these inconsistencies are expected. The **magic** happens when we invoke the **Find** operation on the variable a, where a chain reaction would be triggered to update the group\_id and weight along the chain, as follows:



Once the **lazy** evaluation of find() is triggered, the states of the nodes along the chain would then be updated, and eventually they become consistent.

The mechanism of update is fairly similar with the DFS traversal, as one will see more in detail in the implementation later.

**Algorithm**

Now that we have defined the behaviors for the desired Union-Find data structure, let us put them down into implementation.

The overall interfaces of our Union-Find data structure remain the same. We will implement two functions: find(variable) and union(dividend, divisor, quotient).

* find(variable): the function will return the group\_id that the variable belongs to. Moreover, the function will update the states of variables along the chain, if there is any discrepancy.
* union(dividend, divisor, quotient): this function will attach the group of dividend to that of the divisor, if they are not already the same group. In addition, it needs to update the weight of the dividend variable accordingly, so that the ratio between the dividend and divisor is respected.

We present a sample implementation of the above two functions in the later section, which is inspired from the post of [WangQiuc](https://leetcode.com/problems/evaluate-division/discuss/270993/Python-BFS-and-UF(detailed-explanation)) in the discussion forum. Concise the implementation might be, it might be tricky to wrap one's head around it. One might want to refer to the step-wise example we showed before.

Once we implement the above two functions, we then solve the problem in two steps:

* Step 1): we iterate through each input equation, and invoke the union(dividend, divisor, quotient) on each of them, in order to build the Union-Find data structure.
* Step 2): we evaluate the query one by one. The evaluation is just as intuitive as our first approach, which can be broken down into the following cases:
  + case 1): Either of the variables did not appear in the input equations. The query is not valid. We then return -1.0 as the result.
  + case 2): If both variables are valid, we then apply the find(variable) to obtain the tuple of (group\_id, weight) for each variable. If they are not of the same group, i.e. there is no chain of division between them, we then return -1.0 as the result.
  + case 3): Finally if both variables are of the same group, then we simply perform the division between their weights as the result.

|  |
| --- |
| class Solution {  public double[] calcEquation(List<List<String>> equations, double[] values,  List<List<String>> queries) {  HashMap<String, Pair<String, Double>> gidWeight = new HashMap<>();  // Step 1). build the union groups  for (int i = 0; i < equations.size(); i++) {  List<String> equation = equations.get(i);  String dividend = equation.get(0), divisor = equation.get(1);  double quotient = values[i];  union(gidWeight, dividend, divisor, quotient);  }  // Step 2). run the evaluation, with "lazy" updates in find() function  double[] results = new double[queries.size()];  for (int i = 0; i < queries.size(); i++) {  List<String> query = queries.get(i);  String dividend = query.get(0), divisor = query.get(1);  if (!gidWeight.containsKey(dividend) || !gidWeight.containsKey(divisor))  // case 1). at least one variable did not appear before  results[i] = -1.0;  else {  Pair<String, Double> dividendEntry = find(gidWeight, dividend);  Pair<String, Double> divisorEntry = find(gidWeight, divisor);  String dividendGid = dividendEntry.getKey();  String divisorGid = divisorEntry.getKey();  Double dividendWeight = dividendEntry.getValue();  Double divisorWeight = divisorEntry.getValue();  if (!dividendGid.equals(divisorGid))  // case 2). the variables do not belong to the same chain/group  results[i] = -1.0;  else  // case 3). there is a chain/path between the variables  results[i] = dividendWeight / divisorWeight;  }  }  return results;  }  private Pair<String, Double> find(HashMap<String, Pair<String, Double>> gidWeight, String nodeId) {  if (!gidWeight.containsKey(nodeId))  gidWeight.put(nodeId, new Pair<String, Double>(nodeId, 1.0));  Pair<String, Double> entry = gidWeight.get(nodeId);  // found inconsistency, trigger chain update  if (!entry.getKey().equals(nodeId)) {  Pair<String, Double> newEntry = find(gidWeight, entry.getKey());  gidWeight.put(nodeId, new Pair<String, Double>(  newEntry.getKey(), entry.getValue() \* newEntry.getValue()));  }  return gidWeight.get(nodeId);  }  private void union(HashMap<String, Pair<String, Double>> gidWeight, String dividend, String divisor, Double value) {  Pair<String, Double> dividendEntry = find(gidWeight, dividend);  Pair<String, Double> divisorEntry = find(gidWeight, divisor);  String dividendGid = dividendEntry.getKey();  String divisorGid = divisorEntry.getKey();  Double dividendWeight = dividendEntry.getValue();  Double divisorWeight = divisorEntry.getValue();  // merge the two groups together,  // by attaching the dividend group to the one of divisor  if (!dividendGid.equals(divisorGid)) {  gidWeight.put(dividendGid, new Pair<String, Double>(divisorGid,  divisorWeight \* value / dividendWeight));  }  }  } |

**Complexity Analysis**

Since we applied the Union-Find data structure in our algorithm, we would like to start with a statement on the time complexity of the data structure, as follows:

**Statement**: If M*M* operations, either Union or Find, are applied to N*N* elements, the total run time is \mathcal{O}(M \cdot \log^{\*}{N})O(*M*⋅log∗*N*), where \log^{\*}log∗ is the [iterated logarithm](https://en.wikipedia.org/wiki/Iterated_logarithm).

One can refer to the [proof of Union-Find complexity](https://en.wikipedia.org/wiki/Proof_of_O(log*n)_time_complexity_of_union%E2%80%93find) for more details.

In our case, the maximum number of elements in the Union-Find data structure is equal to twice of the number of equations, i.e. each equation introduces two new variables.

Let N*N* be the number of input equations and M*M* be the number of queries.

* Time Complexity: \mathcal{O}\big( (M+N) \cdot \log^{\*}N\big)O((*M*+*N*)⋅log∗*N*).
  + First of all, we iterate through each input equation and invoke union() upon it. As a result, the overall time complexity of this step is \mathcal{O}\big(N \cdot \log^{\*}N\big)O(*N*⋅log∗*N*).
  + As the second step, we then evaluate the query one by one. For each evaluation, we invoke the find() function at most twice. Therefore, the overall time complexity of this step is \mathcal{O}\big(M \cdot \log^{\*}N\big)O(*M*⋅log∗*N*).
  + To sum up, the total time complexity of the algorithm is \mathcal{O}\big( (M+N) \cdot \log^{\*}N\big)O((*M*+*N*)⋅log∗*N*).
  + Note, as compared to the DFS/BFS search approach, Union-Find data structure is more **efficient** for the repetitive/redundant query scenario.
  + Once we evaluate a query with Union-Find, all the subsequent repetitive queries or any query that has the overlapping with the previous query in terms of variable group could be evaluated in **constant time**. For instance, in the above example, once the query of \frac{a}{c}*ca*​ is evaluated, if later we want to evaluate \frac{a}{b}*ba*​, we could instantly obtain the states of variables a and b without triggering the chain update again. While for DFS/BFS approaches, the cost of evaluating each query is independent for each other.
* Space Complexity: \mathcal{O}(N)O(*N*)
  + The space complexity of our Union-Find data structure is \mathcal{O}(N)O(*N*) where we maintain a state for each variable.
  + Since the find() function is implemented with recursion, there would be some additional memory consumption in function call stack, which could amount to \mathcal{O}(N)O(*N*).
  + To sum up, the total space complexity of the algorithm is \mathcal{O}(N) + \mathcal{O}(N) = \mathcal{O}(N)O(*N*)+O(*N*)=O(*N*).
  + Again, we did not take into account the space needed to hold the results. Otherwise, the total space complexity would be \mathcal{O}(N + M)O(*N*+*M*).

**Print Binary Tree**

Print a binary tree in an m\*n 2D string array following these rules:

1. The row number m should be equal to the height of the given binary tree.
2. The column number n should always be an odd number.
3. The root node's value (in string format) should be put in the exactly middle of the first row it can be put. The column and the row where the root node belongs will separate the rest space into two parts (**left-bottom part and right-bottom part**). You should print the left subtree in the left-bottom part and print the right subtree in the right-bottom part. The left-bottom part and the right-bottom part should have the same size. Even if one subtree is none while the other is not, you don't need to print anything for the none subtree but still need to leave the space as large as that for the other subtree. However, if two subtrees are none, then you don't need to leave space for both of them.
4. Each unused space should contain an empty string "".
5. Print the subtrees following the same rules.

**Example 1:**

**Input:**

1

/

2

**Output:**

[["", "1", ""],

["2", "", ""]]

**Example 2:**

**Input:**

1

/ \

2 3

\

4

**Output:**

[["", "", "", "1", "", "", ""],

["", "2", "", "", "", "3", ""],

["", "", "4", "", "", "", ""]]

**Example 3:**

**Input:**

1

/ \

2 5

/

3

/

4

**Output:**

[["", "", "", "", "", "", "", "1", "", "", "", "", "", "", ""]

["", "", "", "2", "", "", "", "", "", "", "", "5", "", "", ""]

["", "3", "", "", "", "", "", "", "", "", "", "", "", "", ""]

["4", "", "", "", "", "", "", "", "", "", "", "", "", "", ""]]

**Note:** The height of binary tree is in the range of [1, 10].

## Solution

#### **Approach 1: Recursive Solution**

We start by initializing a res*res* array with the dimensions being height \cdot 2^{height}-1*height*⋅2*height*−1. Here, height*height* refers to the number of levels in the given tree. In order to fill this res*res* array with the required elements, initially, we fill the complete array with "" . After this we make use of a recursive function fill(res, root, i, l, r) which fills the res*res* array such that the current element has to be filled in i^{th}*ith* row, and the column being the middle of the indices l*l* and r*r*, where l*l* and r*r* refer to the left and the right boundaries of the columns in which the current element can be filled.

In every recursive call, we do as follows:

1. If we've reached the end of the tree, i.e. if root==null, return.
2. Determine the column in which the current element(root*root*) needs to be filled, which is the middle of l*l* and r*r*, given by say, j*j*. The row number is same as i*i*. Put the current element at res[i][j]*res*[*i*][*j*].
3. Make the recursive call for the left child of the root*root* using fill(res, root.left, i + 1, l, (l + r) / 2).
4. Make the recursive call for the right child of the root*root* using fill(res, root.right, i + 1, (l + r + 1) / 2, r).

Note, that in the last two recursive calls, we update the row number(level of the tree). This ensures that the child nodes fit into the correct row. We also update the column boundaries appropriately based on the l*l* and r*r* values.

Further, to determine the height*height* also, we make use of recursive funtion getHeight(root), which returns the height of the tree starting from the root*root* node. We traverse into all the branches possible in the tree recursively and find the depth of the longest branch.

At the end, we convert the res*res* array into the required list format, before returning the results.

|  |
| --- |
| public class Solution {  public List<List<String>> printTree(TreeNode root) {  int height = getHeight(root);  String[][] res = new String[height][(1 << height) - 1];  for(String[] arr:res)  Arrays.fill(arr,"");  List<List<String>> ans = new ArrayList<>();  fill(res, root, 0, 0, res[0].length);  for(String[] arr:res)  ans.add(Arrays.asList(arr));  return ans;  }  public void fill(String[][] res, TreeNode root, int i, int l, int r) {  if (root == null)  return;  res[i][(l + r) / 2] = "" + root.val;  fill(res, root.left, i + 1, l, (l + r) / 2);  fill(res, root.right, i + 1, (l + r + 1) / 2, r);  }  public int getHeight(TreeNode root) {  if (root == null)  return 0;  return 1 + Math.max(getHeight(root.left), getHeight(root.right));  }  } |

**Complexity Analysis**

* Time complexity : O(h \cdot 2^h)*O*(*h*⋅2*h*). We need to fill the res*res* array of size h \cdot 2^h - 1*h*⋅2*h*−1. Here, h*h* refers to the height of the given tree.
* Space complexity : O(h \cdot 2^h)*O*(*h*⋅2*h*). res*res* array of size h \cdot 2^h - 1*h*⋅2*h*−1 is used.

#### **Approach 2: Using Queue (BFS)**

**Algorithm**

We can also solve the problem by making use of Breadth First Search's idea. For this, we make use of a class Params*Params* which stores the parameters of a node*node* of the tree, including its value, its level in the tree(i*i*), and the left(l*l*) and right(r*r*) boundaries of the columns in which this element can be filled in the result to be returned.

We start by initializing a res*res* array as in the previous approach. After this, we add the parametrized root*root* of the tree into a queue*queue*. After this, we do the following at every step.

1. Remove an element, p*p*, from the front of the queue*queue*.
2. Add this element at its correct position in the res*res* array given by res[p.i][(p.l + p.r) / 2]*res*[*p*.*i*][(*p*.*l*+*p*.*r*)/2]. Here, the values i*i*, l*l* and r*r* refer to the column/level number, and the left and right boundaries permissible for putting the current node into res*res*. These are obtained from the node's parameters, which have been associated with it before putting it into the queue*queue*.
3. If the left child of p*p* exists, put it at the back of the queue*queue*, in a parametized form, by appropriately updating the level as the next level and the boundaries permissible as well.
4. If the right child of p*p* exists, put it at the back of the queue*queue*, in a parametized form, by appropriately updating the level as the next level and the boundaries permissible as well.
5. Continue steps 1. to 4. till the queue*queue* becomes empty.

At the end, we again convert the res*res* array into the required list format, before returning the results.

|  |
| --- |
| public class Solution  /\*\*  \* Definition for a binary tree node.  \* public class TreeNode {  \* int val;  \* TreeNode left;  \* TreeNode right;  \* TreeNode(int x) { val = x; }  \* }  \*/  public class Solution {  class Params {  Params(TreeNode n, int ii, int ll, int rr) {  root = n;  i = ii;  l = ll;  r = rr;  }  TreeNode root;  int i, l, r;  }  public List < List < String >> printTree(TreeNode root) {  int height = getHeight(root);  System.out.println(height);  String[][] res = new String[height][(1 << height) - 1];  for (String[] arr: res)  Arrays.fill(arr, "");  List < List < String >> ans = new ArrayList < > ();  fill(res, root, 0, 0, res[0].length);  for (String[] arr: res)  ans.add(Arrays.asList(arr));  return ans;  }  public void fill(String[][] res, TreeNode root, int i, int l, int r) {  Queue < Params > queue = new LinkedList();  queue.add(new Params(root, 0, 0, res[0].length));  while (!queue.isEmpty()) {  Params p = queue.remove();  res[p.i][(p.l + p.r) / 2] = "" + p.root.val;  if (p.root.left != null)  queue.add(new Params(p.root.left, p.i + 1, p.l, (p.l + p.r) / 2));  if (p.root.right != null)  queue.add(new Params(p.root.right, p.i + 1, (p.l + p.r + 1) / 2, p.r));  }  }  public int getHeight(TreeNode root) {  Queue < TreeNode > queue = new LinkedList();  queue.add(root);  int height = 0;  while (!queue.isEmpty()) {  height++;  Queue < TreeNode > temp = new LinkedList();  while (!queue.isEmpty()) {  TreeNode node = queue.remove();  if (node.left != null)  temp.add(node.left);  if (node.right != null)  temp.add(node.right);  }  queue = temp;  }  return height;  }  } |

**Complexity Analysis**

* Time complexity : O(h \cdot 2^h)*O*(*h*⋅2*h*). We need to fill the res*res* array of size h \cdot 2^h - 1*h*⋅2*h*−1. Here, h*h* refers to the height of the given tree.
* Space complexity : O(h \cdot 2^h)*O*(*h*⋅2*h*). res*res* array of size h \cdot 2^h - 1*h*⋅2*h*−1 is used.

**Exclusive Time of Functions**

On a **single-threaded** CPU, we execute a program containing n functions. Each function has a unique ID between 0 and n-1.

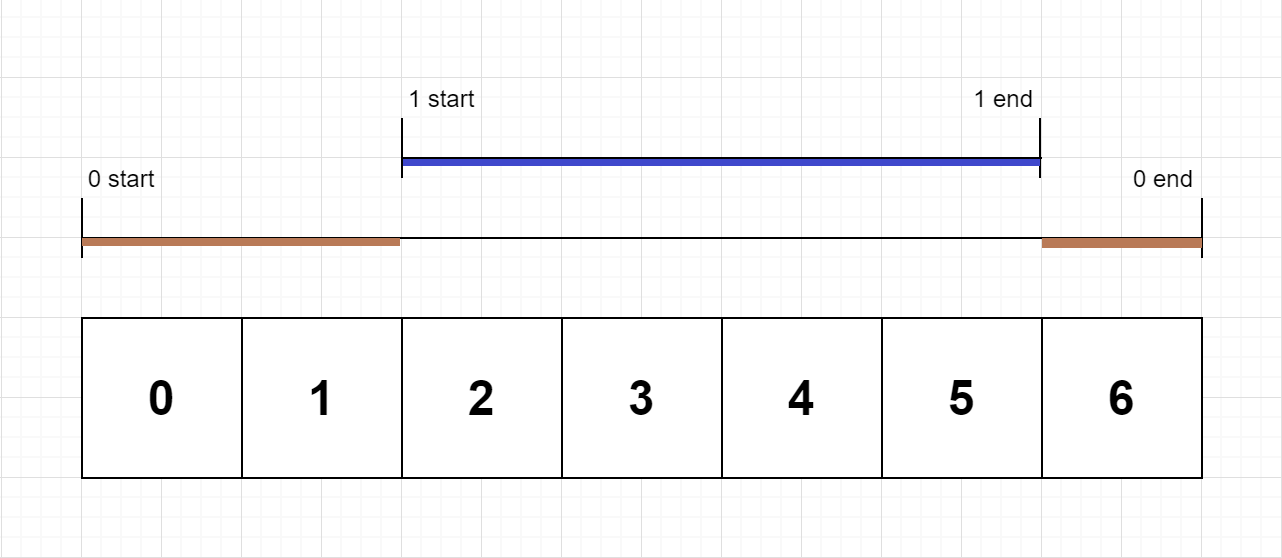
Function calls are **stored in a**[**call stack**](https://en.wikipedia.org/wiki/Call_stack): when a function call starts, its ID is pushed onto the stack, and when a function call ends, its ID is popped off the stack. The function whose ID is at the top of the stack is **the current function being executed**. Each time a function starts or ends, we write a log with the ID, whether it started or ended, and the timestamp.

You are given a list logs, where logs[i] represents the ith log message formatted as a string "{function\_id}:{"start" | "end"}:{timestamp}". For example, "0:start:3" means a function call with function ID 0 **started at the beginning** of timestamp 3, and "1:end:2" means a function call with function ID 1 **ended at the end** of timestamp 2. Note that a function can be called **multiple times, possibly recursively**.

A function's **exclusive time** is the sum of execution times for all function calls in the program. For example, if a function is called twice, one call executing for 2 time units and another call executing for 1 time unit, the **exclusive time** is 2 + 1 = 3.

Return *the****exclusive time****of each function in an array, where the value at the*ith*index represents the exclusive time for the function with ID*i.

**Example 1:**



**Input:** n = 2, logs = ["0:start:0","1:start:2","1:end:5","0:end:6"]

**Output:** [3,4]

**Explanation:**

Function 0 starts at the beginning of time 0, then it executes 2 for units of time and reaches the end of time 1.

Function 1 starts at the beginning of time 2, executes for 4 units of time, and ends at the end of time 5.

Function 0 resumes execution at the beginning of time 6 and executes for 1 unit of time.

So function 0 spends 2 + 1 = 3 units of total time executing, and function 1 spends 4 units of total time executing.

**Example 2:**

**Input:** n = 1, logs = ["0:start:0","0:start:2","0:end:5","0:start:6","0:end:6","0:end:7"]

**Output:** [8]

**Explanation:**

Function 0 starts at the beginning of time 0, executes for 2 units of time, and recursively calls itself.

Function 0 (recursive call) starts at the beginning of time 2 and executes for 4 units of time.

Function 0 (initial call) resumes execution then immediately calls itself again.

Function 0 (2nd recursive call) starts at the beginning of time 6 and executes for 1 unit of time.

Function 0 (initial call) resumes execution at the beginning of time 7 and executes for 1 unit of time.

So function 0 spends 2 + 4 + 1 + 1 = 8 units of total time executing.

**Example 3:**

**Input:** n = 2, logs = ["0:start:0","0:start:2","0:end:5","1:start:6","1:end:6","0:end:7"]

**Output:** [7,1]

**Explanation:**

Function 0 starts at the beginning of time 0, executes for 2 units of time, and recursively calls itself.

Function 0 (recursive call) starts at the beginning of time 2 and executes for 4 units of time.

Function 0 (initial call) resumes execution then immediately calls function 1.

Function 1 starts at the beginning of time 6, executes 1 units of time, and ends at the end of time 6.

Function 0 resumes execution at the beginning of time 6 and executes for 2 units of time.

So function 0 spends 2 + 4 + 1 = 7 units of total time executing, and function 1 spends 1 unit of total time executing.

**Example 4:**

**Input:** n = 2, logs = ["0:start:0","0:start:2","0:end:5","1:start:7","1:end:7","0:end:8"]

**Output:** [8,1]

**Example 5:**

**Input:** n = 1, logs = ["0:start:0","0:end:0"]

**Output:** [1]

**Constraints:**

* 1 <= n <= 100
* 1 <= logs.length <= 500
* 0 <= function\_id < n
* 0 <= timestamp <= 109
* No two start events will happen at the same timestamp.
* No two end events will happen at the same timestamp.
* Each function has an "end" log for each "start" log.

## Solution

#### **Approach #1 Using Stack [Time Limit Exceeded]**

Before starting off with the solution, let's discuss a simple idea. Suppose we have three functions func\_1*func*1​, func\_2*func*2​ and func\_3*func*3​ such that func\_1*func*1​ calls func\_2*func*2​ and then func\_2*func*2​ calls func\_3*func*3​. In this case, func\_3*func*3​ starts at the end and ends first, func\_2*func*2​ starts at 2nd position and ends at the 2nd last step. Similarly, func\_1*func*1​ starts first and ends at the last position. Thus, we can conclude that the function which is entered at the end finishes first and the one which is entered first ends at the last position.

From the above discussion, we can conclude that we can make use of a stack*stack* to solve the given problem. We can start by pushing the first function's id from the given logs*logs* list onto the array. We also keep a track of the current time*time*. We also make use of a res*res* array, such that res[i]*res*[*i*] is to keep a track of the exclusive time spent by the Fucntion with function id i*i* till the current time.

Now, we can move on to the next function in logs*logs*. The start/end time of the next function will obviously be larger than the start time of the function on the stack*stack*. We keep on incrementing the current time*time* and the exclusive time for the function on the top of the stack*stack* till the current time becomes equal to the start/end time of the next function in the logs*logs* list.

Thus, now, we've reached a point, where the control shifts from the last function to a new function, due to a function call(indicated by a start label for the next function), or the last function could exit(indicated by the end label for the next function). Thus, we can no longer continue with the same old function.

If the next function includes a start label, we push this function on the top of the stack*stack*, since the last function would need to be revisited again in the future. On the other hand, if the next function includes an end label, it means the last function on the top of the stack*stack* is terminating.

We also know that an end label indicates that this function executes till the end of the given time. Thus, we need to increment the current time*time* and the exclusive time of the last function as well to account for this fact. Now, we can remove(pop) this function from the stack*stack*. We can continue this process for every function in the logs*logs* list.

At the end, the res*res* array gives the exclusive times for each function.

Summarizing the above process, we need to do the following:

1. Push the function id of the first function in the logs*logs* list on the stack*stack*.
2. Keep incrementing the exlusive time(along with the current time) corresponding to the function on the top of the stack*stack*(in the res*res* array), till the current time equals the start/end time corresponding to the next function in the logs*logs* list.
3. If the next function has a 'start' label, push this function's id onto the stack. Otherwise, increment the last function's exclusive time(along with the current time), and pop the function id from the top of the stack.
4. Repeat steps 2 and 3 till all the functions in the logs*logs* list have been considered.
5. Return the resultant exlcusive time(res*res*).

|  |
| --- |
| public class Solution {  public int[] exclusiveTime(int n, List < String > logs) {  Stack < Integer > stack = new Stack < > ();  int[] res = new int[n];  String[] s = logs.get(0).split(":");  stack.push(Integer.parseInt(s[0]));  int i = 1, time = Integer.parseInt(s[2]);  while (i < logs.size()) {  s = logs.get(i).split(":");  while (time < Integer.parseInt(s[2])) {  res[stack.peek()]++;  time++;  }  if (s[1].equals("start"))  stack.push(Integer.parseInt(s[0]));  else {  res[stack.peek()]++;  time++;  stack.pop();  }  i++;  }  return res;  }  } |

**Complexity Analysis**

* Time complexity : O(t)*O*(*t*). We increment the time till all the functions are done with the execution. Here, t*t* refers to the end time of the last function in the logs*logs* list.
* Space complexity : O(n)*O*(*n*). The stack*stack* can grow upto a depth of atmost n/2*n*/2. Here, n*n* refers to the number of elements in the given logs*logs* list.

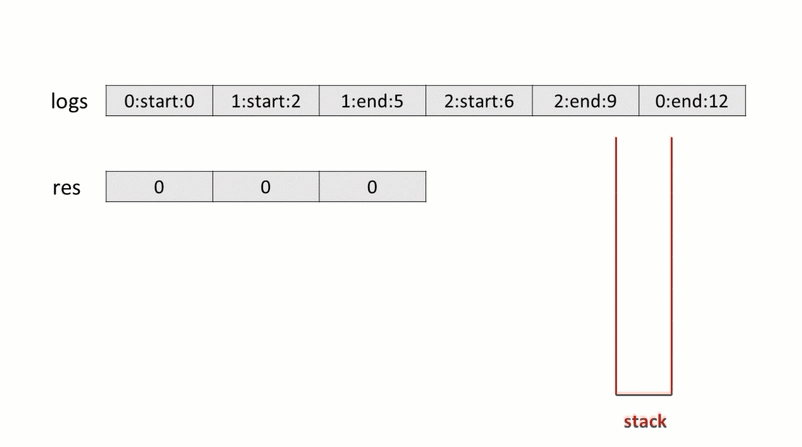
#### **Approach #2 Better Approach [Accepted]**

**Algorithm**

In the last approach, for every function on the top of the stack*stack*, we incremented the current time and the exclusive time of this same function till the current time became equal to the start/end time of the next function.

Instead of doing this incrementing step by step, we can directly use the difference between the next function's start/stop time and the current function's start/stop time. The rest of the process remains the same as in the last approach.

The following animation illustrates the process.



|  |
| --- |
| public class Solution {  public int[] exclusiveTime(int n, List < String > logs) {  Stack < Integer > stack = new Stack < > ();  int[] res = new int[n];  String[] s = logs.get(0).split(":");  stack.push(Integer.parseInt(s[0]));  int i = 1, prev = Integer.parseInt(s[2]);  while (i < logs.size()) {  s = logs.get(i).split(":");  if (s[1].equals("start")) {  if (!stack.isEmpty())  res[stack.peek()] += Integer.parseInt(s[2]) - prev;  stack.push(Integer.parseInt(s[0]));  prev = Integer.parseInt(s[2]);  } else {  res[stack.peek()] += Integer.parseInt(s[2]) - prev + 1;  stack.pop();  prev = Integer.parseInt(s[2]) + 1;  }  i++;  }  return res;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). We iterate over the entire logs*logs* array just once. Here, n*n* refers to the number of elements in the logs*logs* list.
* Space complexity : The stack*stack* can grow upto a depth of atmost n/2*n*/2. Here, n*n* refers to the number of elements in the given logs*logs* list.

**Employee Free Time**

We are given a list schedule of employees, which represents the working time for each employee.

Each employee has a list of non-overlapping Intervals, and these intervals are in sorted order.

Return the list of finite intervals representing **common, positive-length free time** for *all* employees, also in sorted order.

(Even though we are representing Intervals in the form [x, y], the objects inside are Intervals, not lists or arrays. For example, schedule[0][0].start = 1, schedule[0][0].end = 2, and schedule[0][0][0] is not defined).  Also, we wouldn't include intervals like [5, 5] in our answer, as they have zero length.

**Example 1:**

**Input:** schedule = [[[1,2],[5,6]],[[1,3]],[[4,10]]]

**Output:** [[3,4]]

**Explanation:** There are a total of three employees, and all common

free time intervals would be [-inf, 1], [3, 4], [10, inf].

We discard any intervals that contain inf as they aren't finite.

**Example 2:**

**Input:** schedule = [[[1,3],[6,7]],[[2,4]],[[2,5],[9,12]]]

**Output:** [[5,6],[7,9]]

**Constraints:**

* 1 <= schedule.length , schedule[i].length <= 50
* 0 <= schedule[i].start < schedule[i].end <= 10^8

   Hide Hint #1

Take all the intervals and do an "events" (or "line sweep") approach - an event of (x, OPEN) increases the number of active intervals, while (x, CLOSE) decreases it. Processing in sorted order from left to right, if the number of active intervals is zero, then you crossed a region of common free time.

#### **Approach #1: Events (Line Sweep) [Accepted]**

**Intuition**

If some interval overlaps any interval (for any employee), then it won't be included in the answer. So we could reduce our problem to the following: given a set of intervals, find all places where there are no intervals.

To do this, we can use an "events" approach present in other interval problems. For each interval [s, e], we can think of this as two events: balance++ when time = s, and balance-- when time = e. We want to know the regions where balance == 0.

**Algorithm**

For each interval, create two events as described above, and sort the events. Now for each event occuring at time t, if the balance is 0, then the preceding segment [prev, t] did not have any intervals present, where prev is the previous value of t.

|  |
| --- |
| class Solution {  public List<Interval> employeeFreeTime(List<List<Interval>> avails) {  int OPEN = 0, CLOSE = 1;  List<int[]> events = new ArrayList();  for (List<Interval> employee: avails)  for (Interval iv: employee) {  events.add(new int[]{iv.start, OPEN});  events.add(new int[]{iv.end, CLOSE});  }  Collections.sort(events, (a, b) -> a[0] != b[0] ? a[0]-b[0] : a[1]-b[1]);  List<Interval> ans = new ArrayList();  int prev = -1, bal = 0;  for (int[] event: events) {  // event[0] = time, event[1] = command  if (bal == 0 && prev >= 0)  ans.add(new Interval(prev, event[0]));  bal += event[1] == OPEN ? 1 : -1;  prev = event[0];  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(C\log C)*O*(*C*log*C*), where C*C* is the number of intervals across all employees.
* Space Complexity: O(C)*O*(*C*).

#### **Approach #2: Priority Queue [Accepted]**

**Intuition**

Say we are at some time where no employee is working. That work-free period will last until the next time some employee has to work.

So let's maintain a heap of the next time an employee has to work, and it's associated job. When we process the next time from the heap, we can add the next job for that employee.

**Algorithm**

Keep track of the latest time anchor that we don't know of a job overlapping that time.

When we process the earliest occurring job not yet processed, it occurs at time t, by employee e\_id, and it was that employee's e\_jx'th job. If anchor < t, then there was a free interval Interval(anchor, t).

|  |
| --- |
| class Solution {  public List<Interval> employeeFreeTime(List<List<Interval>> avails) {  List<Interval> ans = new ArrayList();  PriorityQueue<Job> pq = new PriorityQueue<Job>((a, b) ->  avails.get(a.eid).get(a.index).start -  avails.get(b.eid).get(b.index).start);  int ei = 0, anchor = Integer.MAX\_VALUE;  for (List<Interval> employee: avails) {  pq.offer(new Job(ei++, 0));  anchor = Math.min(anchor, employee.get(0).start);  }  while (!pq.isEmpty()) {  Job job = pq.poll();  Interval iv = avails.get(job.eid).get(job.index);  if (anchor < iv.start)  ans.add(new Interval(anchor, iv.start));  anchor = Math.max(anchor, iv.end);  if (++job.index < avails.get(job.eid).size())  pq.offer(job);  }  return ans;  }  }  class Job {  int eid, index;  Job(int e, int i) {  eid = e;  index = i;  }  } |

**Complexity Analysis**

* Time Complexity: O(C\log N)*O*(*C*log*N*), where N*N* is the number of employees, and C*C* is the number of jobs across all employees. The maximum size of the heap is N*N*, so each push and pop operation is O(\log N)*O*(log*N*), and there are O(C)*O*(*C*) such operations.
* Space Complexity: O(N)*O*(*N*) in additional space complexity.

**24 Game**

You have 4 cards each containing a number from 1 to 9. You need to judge whether they could operated through \*, /, +, -, (, ) to get the value of 24.

**Example 1:**

**Input:** [4, 1, 8, 7]

**Output:** True

**Explanation:** (8-4) \* (7-1) = 24

**Example 2:**

**Input:** [1, 2, 1, 2]

**Output:** False

**Note:**

1. The division operator / represents real division, not integer division. For example, 4 / (1 - 2/3) = 12.
2. Every operation done is between two numbers. In particular, we cannot use - as a unary operator. For example, with [1, 1, 1, 1] as input, the expression -1 - 1 - 1 - 1 is not allowed.
3. You cannot concatenate numbers together. For example, if the input is [1, 2, 1, 2], we cannot write this as 12 + 12.

#### **Approach #1: Backtracking [Accepted]**

**Intuition and Algorithm**

There are only 4 cards and only 4 operations that can be performed. Even when all operations do not commute, that gives us an upper bound of 12 \* 6 \* 2 \* 4 \* 4 \* 4 = 921612∗6∗2∗4∗4∗4=9216 possibilities, which makes it feasible to just try them all. Specifically, we choose two numbers (with order) in 12 ways and perform one of 4 operations (12 \* 4). Then, with 3 remaining numbers, we choose 2 of them and perform one of 4 operations (6 \* 4). Finally we have two numbers left and make a final choice of 2 \* 4 possibilities.

We will perform 3 binary operations (+, -, \*, / are the operations) on either our numbers or resulting numbers. Because - and / do not commute, we must be careful to consider both a / b and b / a.

For every way to remove two numbers a, b in our list, and for each possible result they can make, like a+b, a/b, etc., we will recursively solve the problem on this smaller list of numbers.

|  |
| --- |
| class Solution {  public boolean judgePoint24(int[] nums) {  ArrayList A = new ArrayList<Double>();  for (int v: nums) A.add((double) v);  return solve(A);  }  private boolean solve(ArrayList<Double> nums) {  if (nums.size() == 0) return false;  if (nums.size() == 1) return Math.abs(nums.get(0) - 24) < 1e-6;  for (int i = 0; i < nums.size(); i++) {  for (int j = 0; j < nums.size(); j++) {  if (i != j) {  ArrayList<Double> nums2 = new ArrayList<Double>();  for (int k = 0; k < nums.size(); k++) if (k != i && k != j) {  nums2.add(nums.get(k));  }  for (int k = 0; k < 4; k++) {  if (k < 2 && j > i) continue;  if (k == 0) nums2.add(nums.get(i) + nums.get(j));  if (k == 1) nums2.add(nums.get(i) \* nums.get(j));  if (k == 2) nums2.add(nums.get(i) - nums.get(j));  if (k == 3) {  if (nums.get(j) != 0) {  nums2.add(nums.get(i) / nums.get(j));  } else {  continue;  }  }  if (solve(nums2)) return true;  nums2.remove(nums2.size() - 1);  }  }  }  }  return false;  }  } |

**Complexity Analysis**

* Time Complexity: O(1)*O*(1). There is a hard limit of 9216 possibilities, and we do O(1)*O*(1) work for each of them.
* Space Complexity: O(1)*O*(1). Our intermediate arrays are at most 4 elements, and the number made is bounded by an O(1)*O*(1) factor.

**Max Area of Island**

Given a non-empty 2D array grid of 0's and 1's, an **island** is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

Find the maximum area of an island in the given 2D array. (If there is no island, the maximum area is 0.)

**Example 1:**

[[0,0,1,0,0,0,0,1,0,0,0,0,0],

[0,0,0,0,0,0,0,1,1,1,0,0,0],

[0,1,1,0,1,0,0,0,0,0,0,0,0],

[0,1,0,0,1,1,0,0,**1**,0,**1**,0,0],

[0,1,0,0,1,1,0,0,**1**,**1**,**1**,0,0],

[0,0,0,0,0,0,0,0,0,0,**1**,0,0],

[0,0,0,0,0,0,0,1,1,1,0,0,0],

[0,0,0,0,0,0,0,1,1,0,0,0,0]]

Given the above grid, return 6. Note the answer is not 11, because the island must be connected 4-directionally.

**Example 2:**

[[0,0,0,0,0,0,0,0]]

Given the above grid, return 0.

**Note:** The length of each dimension in the given grid does not exceed 50.

#### **Approach #1: Depth-First Search (Recursive) [Accepted]**

**Intuition and Algorithm**

We want to know the area of each connected shape in the grid, then take the maximum of these.

If we are on a land square and explore every square connected to it 4-directionally (and recursively squares connected to those squares, and so on), then the total number of squares explored will be the area of that connected shape.

To ensure we don't count squares in a shape more than once, let's use seen to keep track of squares we haven't visited before. It will also prevent us from counting the same shape more than once.

|  |
| --- |
| class Solution {  int[][] grid;  boolean[][] seen;  public int area(int r, int c) {  if (r < 0 || r >= grid.length || c < 0 || c >= grid[0].length ||  seen[r][c] || grid[r][c] == 0)  return 0;  seen[r][c] = true;  return (1 + area(r+1, c) + area(r-1, c)  + area(r, c-1) + area(r, c+1));  }  public int maxAreaOfIsland(int[][] grid) {  this.grid = grid;  seen = new boolean[grid.length][grid[0].length];  int ans = 0;  for (int r = 0; r < grid.length; r++) {  for (int c = 0; c < grid[0].length; c++) {  ans = Math.max(ans, area(r, c));  }  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(R\*C)*O*(*R*∗*C*), where R*R* is the number of rows in the given grid, and C*C* is the number of columns. We visit every square once.
* Space complexity: O(R\*C)*O*(*R*∗*C*), the space used by seen to keep track of visited squares, and the space used by the call stack during our recursion.

#### **Approach #2: Depth-First Search (Iterative) [Accepted]**

**Intuition and Algorithm**

We can try the same approach using a stack based, (or "iterative") depth-first search.

Here, seen will represent squares that have either been visited or are added to our list of squares to visit (stack). For every starting land square that hasn't been visited, we will explore 4-directionally around it, adding land squares that haven't been added to seen to our stack.

On the side, we'll keep a count shape of the total number of squares seen during the exploration of this shape. We'll want the running max of these counts.

|  |
| --- |
| class Solution {  public int maxAreaOfIsland(int[][] grid) {  boolean[][] seen = new boolean[grid.length][grid[0].length];  int[] dr = new int[]{1, -1, 0, 0};  int[] dc = new int[]{0, 0, 1, -1};  int ans = 0;  for (int r0 = 0; r0 < grid.length; r0++) {  for (int c0 = 0; c0 < grid[0].length; c0++) {  if (grid[r0][c0] == 1 && !seen[r0][c0]) {  int shape = 0;  Stack<int[]> stack = new Stack();  stack.push(new int[]{r0, c0});  seen[r0][c0] = true;  while (!stack.empty()) {  int[] node = stack.pop();  int r = node[0], c = node[1];  shape++;  for (int k = 0; k < 4; k++) {  int nr = r + dr[k];  int nc = c + dc[k];  if (0 <= nr && nr < grid.length &&  0 <= nc && nc < grid[0].length &&  grid[nr][nc] == 1 && !seen[nr][nc]) {  stack.push(new int[]{nr, nc});  seen[nr][nc] = true;  }  }  }  ans = Math.max(ans, shape);  }  }  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(R\*C)*O*(*R*∗*C*), where R*R* is the number of rows in the given grid, and C*C* is the number of columns. We visit every square once.
* Space complexity: O(R\*C)*O*(*R*∗*C*), the space used by seen to keep track of visited squares, and the space used by stack.

**Bus Routes**

You are given an array routes representing bus routes where routes[i] is a bus route that the ith bus repeats forever.

* For example, if routes[0] = [1, 5, 7], this means that the 0th bus travels in the sequence 1 -> 5 -> 7 -> 1 -> 5 -> 7 -> 1 -> ... forever.

You will start at the bus stop source (You are not on any bus initially), and you want to go to the bus stop target. You can travel between bus stops by buses only.

Return *the least number of buses you must take to travel from*source*to*target. Return -1 if it is not possible.

**Example 1:**

**Input:** routes = [[1,2,7],[3,6,7]], source = 1, target = 6

**Output:** 2

**Explanation:** The best strategy is take the first bus to the bus stop 7, then take the second bus to the bus stop 6.

**Example 2:**

**Input:** routes = [[7,12],[4,5,15],[6],[15,19],[9,12,13]], source = 15, target = 12

**Output:** -1

**Constraints:**

* 1 <= routes.length <= 500.
* 1 <= routes[i].length <= 105
* All the values of routes[i] are **unique**.
* sum(routes[i].length) <= 105
* 0 <= routes[i][j] < 106
* 0 <= source, target < 106

#### **Approach #1: Breadth First Search [Accepted]**

**Intuition**

Instead of thinking of the stops as nodes (of a graph), think of the buses as nodes. We want to take the least number of buses, which is a shortest path problem, conducive to using a breadth-first search.

**Algorithm**

We perform a breadth first search on bus numbers. When we start at S, originally we might be able to board many buses, and if we end at T we may have many targets for our goal state.

One difficulty is to efficiently decide whether two buses are connected by an edge. They are connected if they share at least one bus stop. Whether two lists share a common value can be done by set intersection (HashSet), or by sorting each list and using a two pointer approach.

To make our search easy, we will annotate the depth of each node: info[0] = node, info[1] = depth.

|  |
| --- |
| import java.awt.Point;  class Solution {  public int numBusesToDestination(int[][] routes, int S, int T) {  if (S==T) return 0;  int N = routes.length;  List<List<Integer>> graph = new ArrayList();  for (int i = 0; i < N; ++i) {  Arrays.sort(routes[i]);  graph.add(new ArrayList());  }  Set<Integer> seen = new HashSet();  Set<Integer> targets = new HashSet();  Queue<Point> queue = new ArrayDeque();  // Build the graph. Two buses are connected if  // they share at least one bus stop.  for (int i = 0; i < N; ++i)  for (int j = i+1; j < N; ++j)  if (intersect(routes[i], routes[j])) {  graph.get(i).add(j);  graph.get(j).add(i);  }  // Initialize seen, queue, targets.  // seen represents whether a node has ever been enqueued to queue.  // queue handles our breadth first search.  // targets is the set of goal states we have.  for (int i = 0; i < N; ++i) {  if (Arrays.binarySearch(routes[i], S) >= 0) {  seen.add(i);  queue.offer(new Point(i, 0));  }  if (Arrays.binarySearch(routes[i], T) >= 0)  targets.add(i);  }  while (!queue.isEmpty()) {  Point info = queue.poll();  int node = info.x, depth = info.y;  if (targets.contains(node)) return depth+1;  for (Integer nei: graph.get(node)) {  if (!seen.contains(nei)) {  seen.add(nei);  queue.offer(new Point(nei, depth+1));  }  }  }  return -1;  }  public boolean intersect(int[] A, int[] B) {  int i = 0, j = 0;  while (i < A.length && j < B.length) {  if (A[i] == B[j]) return true;  if (A[i] < B[j]) i++; else j++;  }  return false;  }  } |

**Complexity Analysis**

* Time Complexity: Let N*N* denote the number of buses, and b\_i*bi*​ be the number of stops on the i*i*th bus.
  + To create the graph, in Python we do O(\sum (N - i) b\_i)*O*(∑(*N*−*i*)*bi*​) work (we can improve this by checking for which of r1, r2 is smaller), while in Java we did a O(\sum b\_i \log b\_i)*O*(∑*bi*​log*bi*​) sorting step, plus our searches are O(N \sum b\_i)*O*(*N*∑*bi*​) work.
  + Our (breadth-first) search is on N*N* nodes, and each node could have N*N* edges, so it is O(N^2)*O*(*N*2).
* Space Complexity: O(N^2 + \sum b\_i)*O*(*N*2+∑*bi*​) additional space complexity, the size of graph and routes. In Java, our space complexity is O(N^2)*O*(*N*2) because we do not have an equivalent of routes. Dual-pivot quicksort (as used in Arrays.sort(int[])) is an in-place algorithm, so in Java we did not increase our space complexity by sorting.

**Shortest Bridge**

In a given 2D binary array A, there are two islands.  (An island is a 4-directionally connected group of 1s not connected to any other 1s.)

Now, we may change 0s to 1s so as to connect the two islands together to form 1 island.

Return the smallest number of 0s that must be flipped.  (It is guaranteed that the answer is at least 1.)

**Example 1:**

**Input:** A = [[0,1],[1,0]]

**Output:** 1

**Example 2:**

**Input:** A = [[0,1,0],[0,0,0],[0,0,1]]

**Output:** 2

**Example 3:**

**Input:** A = [[1,1,1,1,1],[1,0,0,0,1],[1,0,1,0,1],[1,0,0,0,1],[1,1,1,1,1]]

**Output:** 1

**Constraints:**

* 2 <= A.length == A[0].length <= 100
* A[i][j] == 0 or A[i][j] == 1

## Solution

#### **Approach 1: Find and Grow**

**Intuition**

Conceptually, our method is very straightforward: find both islands, then for one of the islands, keep "growing" it by 1 until we touch the second island.

We can use a depth-first search to find the islands, and a breadth-first search to "grow" one of them. This leads to a verbose but correct solution.

**Algorithm**

To find both islands, look for a square with a 1 we haven't visited, and dfs to get the component of that region. Do this twice. After, we have two components source and target.

To find the shortest bridge, do a BFS from the nodes source. When we reach any node in target, we will have found the shortest distance.

Please see the code for more implementation details.

|  |
| --- |
| class Solution {  public int shortestBridge(int[][] A) {  int R = A.length, C = A[0].length;  int[][] colors = getComponents(A);  Queue<Node> queue = new LinkedList();  Set<Integer> target = new HashSet();  for (int r = 0; r < R; ++r)  for (int c = 0; c < C; ++c) {  if (colors[r][c] == 1) {  queue.add(new Node(r, c, 0));  } else if (colors[r][c] == 2) {  target.add(r \* C + c);  }  }  while (!queue.isEmpty()) {  Node node = queue.poll();  if (target.contains(node.r \* C + node.c))  return node.depth - 1;  for (int nei: neighbors(A, node.r, node.c)) {  int nr = nei / C, nc = nei % C;  if (colors[nr][nc] != 1) {  queue.add(new Node(nr, nc, node.depth + 1));  colors[nr][nc] = 1;  }  }  }  throw null;  }  public int[][] getComponents(int[][] A) {  int R = A.length, C = A[0].length;  int[][] colors = new int[R][C];  int t = 0;  for (int r0 = 0; r0 < R; ++r0)  for (int c0 = 0; c0 < C; ++c0)  if (colors[r0][c0] == 0 && A[r0][c0] == 1) {  // Start dfs  Stack<Integer> stack = new Stack();  stack.push(r0 \* C + c0);  colors[r0][c0] = ++t;  while (!stack.isEmpty()) {  int node = stack.pop();  int r = node / C, c = node % C;  for (int nei: neighbors(A, r, c)) {  int nr = nei / C, nc = nei % C;  if (A[nr][nc] == 1 && colors[nr][nc] == 0) {  colors[nr][nc] = t;  stack.push(nr \* C + nc);  }  }  }  }  return colors;  }  public List<Integer> neighbors(int[][] A, int r, int c) {  int R = A.length, C = A[0].length;  List<Integer> ans = new ArrayList();  if (0 <= r-1) ans.add((r-1) \* R + c);  if (0 <= c-1) ans.add(r \* R + (c-1));  if (r+1 < R) ans.add((r+1) \* R + c);  if (c+1 < C) ans.add(r \* R + (c+1));  return ans;  }  }  class Node {  int r, c, depth;  Node(int r, int c, int d) {  this.r = r;  this.c = c;  depth = d;  }  } |

**Complexity Analysis**

* Time Complexity: O(\mathcal{A})*O*(A), where \mathcal{A}A is the content of A.
* Space Complexity: O(\mathcal{A})*O*(A).

**Russian Doll Envelopes**

You have a number of envelopes with widths and heights given as a pair of integers (w, h). One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.

What is the maximum number of envelopes can you Russian doll? (put one inside other)

**Note:**  
Rotation is not allowed.

**Example:**

**Input:** [[5,4],[6,4],[6,7],[2,3]]

**Output:** 3

**Explanation: T**he maximum number of envelopes you can Russian doll is 3 ([2,3] => [5,4] => [6,7]).

## Solution

#### **Intuition**

The problem boils down to a two dimensional version of the longest increasing subsequence problem (LIS).

We must find the longest sequence seq such that the elements in seq[i+1] are greater than the corresponding elements in seq[i] (this means that seq[i] can fit into seq[i+1]).

The problem we run into is that the items we are given come in arbitrary order - we can't just run a standard LIS algorithm because we're allowed to rearrange our data. How can we order our data in a way such that our LIS algorithm will always find the best answer?

**Notes on the LIS algorithm**

You can find the longest increasing subsequence problem with a solution [here](https://leetcode.com/problems/longest-increasing-subsequence/). If you're not familiar with the O(N \log N)*O*(*N*log*N*) algorithm please go visit that question as it's a prerequisite for this one.

For the sake of completeness here's a brief explanation on how the LIS algorithm used below works:

dp is an array such that dp[i] is the smallest element that ends an increasing subsequence of length i + 1. Whenever we encounter a new element e, we binary search inside dp to find the largest index i such that e can end that subsequence. We then update dp[i] with e.

The length of the LIS is the same as the length of dp, as if dp has an index i, then it must have a subsequence of length i+1.

#### **Approach 1: Sort + Longest Increasing Subsequence**

**Algorithm**

We answer the question from the intuition by sorting. Let's pretend that we found the best arrangement of envelopes. We know that each envelope must be increasing in w, thus our best arrangement has to be a subsequence of all our envelopes sorted on w.

After we sort our envelopes, we can simply find the length of the longest increasing subsequence on the second dimension (h). Note that we use a clever trick to solve some edge cases:

Consider an input [[1, 3], [1, 4], [1, 5], [2, 3]]. If we simply sort and extract the second dimension we get [3, 4, 5, 3], which implies that we can fit three envelopes (3, 4, 5). The problem is that we can only fit one envelope, since envelopes that are equal in the first dimension can't be put into each other.

In order fix this, we don't just sort increasing in the first dimension - we also sort decreasing on the second dimension, so two envelopes that are equal in the first dimension can never be in the same increasing subsequence.

Now when we sort and extract the second element from the input we get [5, 4, 3, 3], which correctly reflects an LIS of one.

**Implementation**

|  |
| --- |
| class Solution {  public int lengthOfLIS(int[] nums) {  int[] dp = new int[nums.length];  int len = 0;  for (int num : nums) {  int i = Arrays.binarySearch(dp, 0, len, num);  if (i < 0) {  i = -(i + 1);  }  dp[i] = num;  if (i == len) {  len++;  }  }  return len;  }  public int maxEnvelopes(int[][] envelopes) {  // sort on increasing in first dimension and decreasing in second  Arrays.sort(envelopes, new Comparator<int[]>() {  public int compare(int[] arr1, int[] arr2) {  if (arr1[0] == arr2[0]) {  return arr2[1] - arr1[1];  } else {  return arr1[0] - arr2[0];  }  }  });  // extract the second dimension and run LIS  int[] secondDim = new int[envelopes.length];  for (int i = 0; i < envelopes.length; ++i) secondDim[i] = envelopes[i][1];  return lengthOfLIS(secondDim);  }  } |

**Complexity Analysis**

* Time complexity : O(N \log N)*O*(*N*log*N*), where N*N* is the length of the input. Both sorting the array and finding the LIS happen in O(N \log N)*O*(*N*log*N*)
* Space complexity : O(N)*O*(*N*). Our lis function requires an array dp which goes up to size N*N*. Also the sorting algorithm we use may also take additional space.

**Maximum Vacation Days**

LeetCode wants to give one of its best employees the option to travel among **N** cities to collect algorithm problems. But all work and no play makes Jack a dull boy, you could take vacations in some particular cities and weeks. Your job is to schedule the traveling to maximize the number of vacation days you could take, but there are certain rules and restrictions you need to follow.

**Rules and restrictions:**

1. You can only travel among **N** cities, represented by indexes from 0 to N-1. Initially, you are in the city indexed 0 on **Monday**.
2. The cities are connected by flights. The flights are represented as a **N\*N** matrix (not necessary symmetrical), called **flights** representing the airline status from the city i to the city j. If there is no flight from the city i to the city j, **flights[i][j] = 0**; Otherwise, **flights[i][j] = 1**. Also, **flights[i][i] = 0** for all i.
3. You totally have **K** weeks (**each week has 7 days**) to travel. You can only take flights at most once **per day** and can only take flights on each week's **Monday** morning. Since flight time is so short, we don't consider the impact of flight time.
4. For each city, you can only have restricted vacation days in different weeks, given an **N\*K** matrix called **days** representing this relationship. For the value of **days[i][j]**, it represents the maximum days you could take vacation in the city **i** in the week **j**.

You're given the **flights** matrix and **days** matrix, and you need to output the maximum vacation days you could take during **K** weeks.

**Example 1:**

**Input:**flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[1,3,1],[6,0,3],[3,3,3]]

**Output:** 12

**Explanation:**   
Ans = 6 + 3 + 3 = 12.

One of the best strategies is:

1st week : fly from city 0 to city 1 on Monday, and play 6 days and work 1 day.   
(Although you start at city 0, we could also fly to and start at other cities since it is Monday.)

2nd week : fly from city 1 to city 2 on Monday, and play 3 days and work 4 days.

3rd week : stay at city 2, and play 3 days and work 4 days.

**Example 2:**

**Input:**flights = [[0,0,0],[0,0,0],[0,0,0]], days = [[1,1,1],[7,7,7],[7,7,7]]

**Output:** 3

**Explanation:**   
Ans = 1 + 1 + 1 = 3.

Since there is no flights enable you to move to another city, you have to stay at city 0 for the whole 3 weeks.   
For each week, you only have one day to play and six days to work.   
So the maximum number of vacation days is 3.

**Example 3:**

**Input:**flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[7,0,0],[0,7,0],[0,0,7]]

**Output:** 21

**Explanation:**  
Ans = 7 + 7 + 7 = 21

One of the best strategies is:

1st week : stay at city 0, and play 7 days.

2nd week : fly from city 0 to city 1 on Monday, and play 7 days.

3rd week : fly from city 1 to city 2 on Monday, and play 7 days.

**Note:**

1. **N and K** are positive integers, which are in the range of [1, 100].
2. In the matrix **flights**, all the values are integers in the range of [0, 1].
3. In the matrix **days**, all the values are integers in the range [0, 7].
4. You could stay at a city beyond the number of vacation days, but you should **work** on the extra days, which won't be counted as vacation days.
5. If you fly from the city A to the city B and take the vacation on that day, the deduction towards vacation days will count towards the vacation days of city B in that week.
6. We don't consider the impact of flight hours towards the calculation of vacation days.

 Hide Hint #1

First try to understand the problem carefully and then take some example and solve it on a paper.

   Hide Hint #2

Can you interpret the given input as a graph? Which graph traversal technique is suitable here?

   Hide Hint #3

Can we use some space to avoid redundant function calls?

## Solution

#### **Approach #1 Using Depth First Search [Time Limit Exceeded]**

**Algorithm**

In the brute force approach, we make use of a recursive function dfs*dfs*, which returns the number of vacations which can be taken startring from cur\\_city*cur*\_*city* as the current city and weekno*weekno* as the starting week.

In every function call, we traverse over all the cities(represented by i*i*) and find out all the cities which are connected to the current city, cur\\_city*cur*\_*city*. Such a city is represented by a 1 at the corresponding flights[cur\\_city][i]*flights*[*cur*\_*city*][*i*] position. Now, for the current city, we can either travel to the city which is connected to it or we can stay in the same city. Let's say the city to which we change our location from the current city be represented by j*j*. Thus, after changing the city, we need to find the number of vacations which we can take from the new city as the current city and the incremented week as the new starting week. This count of vacations can be represented as: days[j][weekno] + dfs(flights, days, j, weekno + 1)*days*[*j*][*weekno*]+*dfs*(*flights*,*days*,*j*,*weekno*+1).

Thus, for the current city, we obtain a number of vacations by choosing different cities as the next cities. Out of all of these vacations coming from different cities, we can find out the maximum number of vacations that need to be returned for every dfs*dfs* function call.

|  |
| --- |
| public class Solution {  public int maxVacationDays(int[][] flights, int[][] days) {  return dfs(flights, days, 0, 0);  }  public int dfs(int[][] flights, int[][] days, int cur\_city, int weekno) {  if (weekno == days[0].length)  return 0;  int maxvac = 0;  for (int i = 0; i < flights.length; i++) {  if (flights[cur\_city][i] == 1 || i == cur\_city) {  int vac = days[i][weekno] + dfs(flights, days, i, weekno + 1);  maxvac = Math.max(maxvac, vac);  }  }  return maxvac;  }  } |

**Complexity Analysis**

* Time complexity : O(n^k)*O*(*nk*). Depth of Recursion tree will be k*k* and each node contains n*n* branches in the worst case. Here n*n* represents the number of cities and k*k* is the total number of weeks.
* Space complexity : O(k)*O*(*k*). The depth of the recursion tree is k*k*.

#### **Approach #2 Using DFS with memoization [Accepted]:**

**Algorithm**

In the last approach, we make a number of redundant function calls, since the same function call of the form dfs(flights, days, cur\_city, weekno) can be made multiple number of times with the same cur\\_city*cur*\_*city* and weekno*weekno*. These redundant calls can be pruned off if we make use of memoization.

In order to remove these redundant function calls, we make use of a 2-D memoization array memo*memo*. In this array, memo[i][j]*memo*[*i*][*j*] is used to store the number of vacactions that can be taken using the i^{th}*ith* city as the current city and the j^{th}*jth* week as the starting week. This result is equivalent to that obtained using the function call: dfs(flights, days, i, j). Thus, if the memo*memo* entry corresponding to the current function call already contains a valid value, we can directly obtain the result from this array instead of going deeper into recursion.

|  |
| --- |
| public class Solution {  public int maxVacationDays(int[][] flights, int[][] days) {  int[][] memo = new int[flights.length][days[0].length];  for (int[] l: memo)  Arrays.fill(l, Integer.MIN\_VALUE);  return dfs(flights, days, 0, 0, memo);  }  public int dfs(int[][] flights, int[][] days, int cur\_city, int weekno, int[][] memo) {  if (weekno == days[0].length)  return 0;  if (memo[cur\_city][weekno] != Integer.MIN\_VALUE)  return memo[cur\_city][weekno];  int maxvac = 0;  for (int i = 0; i < flights.length; i++) {  if (flights[cur\_city][i] == 1 || i == cur\_city) {  int vac = days[i][weekno] + dfs(flights, days, i, weekno + 1, memo);  maxvac = Math.max(maxvac, vac);  }  }  memo[cur\_city][weekno] = maxvac;  return maxvac;  }  } |

**Complexity Analysis**

* Time complexity : O(n^2k)*O*(*n*2*k*). memo*memo* array of size n\*k*n*∗*k* is filled and each cell filling takes O(n) time .
* Space complexity : O(n\*k)*O*(*n*∗*k*). memo*memo* array of size n\*k*n*∗*k* is used. Here n*n* represents the number of cities and k*k* is the total number of weeks.

#### **Approach #3 Using 2-D Dynamic Programming [Accepted]:**

**Algorithm**

The idea behind this approach is as follows. The maximum number of vacations that can be taken given we start from the i^{th}*ith* city in the j^{th}*jth* week is not dependent on the the vacations that can be taken in the earlier weeks. It only depends on the number of vacations that can be taken in the upcoming weeks and also on the connections between the various cities(flights*flights*).

Therefore, we can make use of a 2-D dp*dp*, in which dp[i][k]*dp*[*i*][*k*] represents the maximum number of vacations which can be taken starting from the i^{th}*ith* city in the k^{th}*kth* week. This dp*dp* is filled in the backward manner(in terms of the week number).

While filling up the entry for dp[i][k]*dp*[*i*][*k*], we need to consider the following cases:

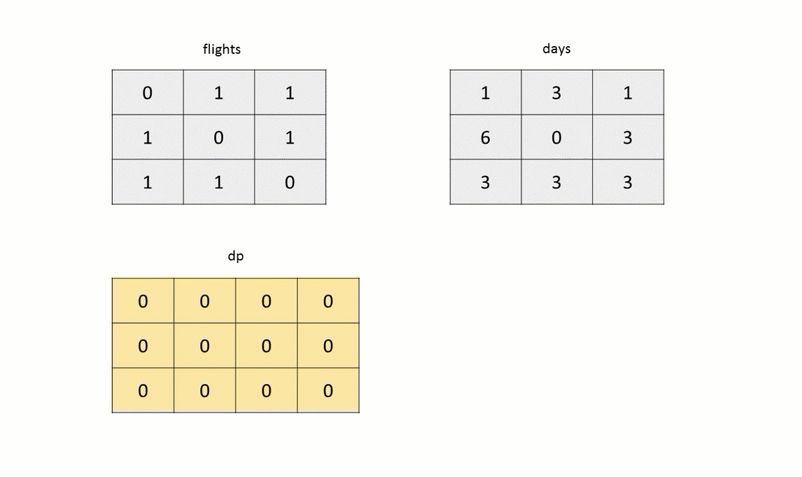
1. We start from the i^{th}*ith* city in the k^{th}*kth* week and stay in the same city for the (k+1)^{th}(*k*+1)*th* week. Thus, the factor to be considered for updating the dp[i][k]*dp*[*i*][*k*] entry will be given by: days[i][k] + dp[i, k+1]*days*[*i*][*k*]+*dp*[*i*,*k*+1].
2. We start from the i^{th}*ith* city in the k^{th}*kth* week and move to the j^{th}*jth* city in the (k+1)^{th}(*k*+1)*th* week. But, for changing the city in this manner, we need to be able to move from the i^{th}*ith* city to the j^{th}*jth* city i.e. flights[i][j]*flights*[*i*][*j*] should be 1 for such i*i* and j*j*.

But, while changing the city from i^{th}*ith* city in the k^{th}*kth* week, we can move to any j^{th}*jth* city such that a connection exists between the i^{th}*ith* city and the j^{th}*jth* city i.e. flights[i][j]=1*flights*[*i*][*j*]=1. But, in order to maximize the number of vacations that can be taken starting from the i^{th}*ith* city in the k^{th}*kth* week, we need to choose the destination city that leads to maximum no. of vacations. Thus, the factor to be considered here, is given by: \text{max}days[j][k] + days[j, k+1]max*days*[*j*][*k*]+*days*[*j*,*k*+1], for all i*i*, j*j*, k*k* satisfying flights[i][j] = 1*flights*[*i*][*j*]=1, 0 ≤ i,j ≤ n, where n*n* refers to the number of cities.

At the end, we need to find the maximum out of these two factors to update the dp[i][k]*dp*[*i*][*k*] value.

In order to fill the dp*dp* values, we start by filling the entries for the last week and proceed backwards. At last, the value of dp[0][0]*dp*[0][0] gives the required result.

The following animation illustrates the process of filling the dp*dp* array.



|  |
| --- |
| public class Solution {  public int maxVacationDays(int[][] flights, int[][] days) {  if (days.length == 0 || flights.length == 0) return 0;  int[][] dp = new int[days.length][days[0].length + 1];  for (int week = days[0].length - 1; week >= 0; week--) {  for (int cur\_city = 0; cur\_city < days.length; cur\_city++) {  dp[cur\_city][week] = days[cur\_city][week] + dp[cur\_city][week + 1];  for (int dest\_city = 0; dest\_city < days.length; dest\_city++) {  if (flights[cur\_city][dest\_city] == 1) {  dp[cur\_city][week] = Math.max(days[dest\_city][week] + dp[dest\_city][week + 1], dp[cur\_city][week]);  }  }  }  }  return dp[0][0];  }  } |

**Complexity Analysis**

* Time complexity : O(n^2k)*O*(*n*2*k*). dp*dp* array of size n\*k*n*∗*k* is filled and each cell filling takes O(n) time. Here n*n* represents the number of cities and k*k* is the total number of weeks.
* Space complexity : O(n\*k)*O*(*n*∗*k*). dp*dp* array of size n\*k*n*∗*k* is used.

#### **Approach #4 Using 1-D Dynamic Programming [Accepted]:**

**Algorithm**

As can be observed in the previous approach, in order to update the dp*dp* entries for i^{th}*ith* week, we only need the values corresponding to (i+1)^{th}(*i*+1)*th* week along with the days*days* and flights*flights* array. Thus, instead of using a 2-D dp*dp* array, we can omit the dimension corresponding to the weeks and make use of a 1-D dp*dp* array.

Now, dp[i]*dp*[*i*] is used to store the number of vacations that provided that we start from the i^{th}*ith* city in the current week. The procedure remains the same as that of the previous approach, except that we make the updations in the same dp*dp* row again and again. In order to store the dp*dp* values corresponding to the current week temporarily, we make use of a temp*temp* array so that the original dp*dp* entries corresponding to week+1*week*+1 aren't altered.

|  |
| --- |
| public class Solution {  public int maxVacationDays(int[][] flights, int[][] days) {  if (days.length == 0 || flights.length == 0) return 0;  int[] dp = new int[days.length];  for (int week = days[0].length - 1; week >= 0; week--) {  int[] temp = new int[days.length];  for (int cur\_city = 0; cur\_city < days.length; cur\_city++) {  temp[cur\_city] = days[cur\_city][week] + dp[cur\_city];  for (int dest\_city = 0; dest\_city < days.length; dest\_city++) {  if (flights[cur\_city][dest\_city] == 1) {  temp[cur\_city] = Math.max(days[dest\_city][week] + dp[dest\_city], temp[cur\_city]);  }  }  }  dp = temp;  }  return dp[0];  }  } |

**Complexity Analysis**

* Time complexity : O(n^2k)*O*(*n*2*k*). dp*dp* array of size n\*k*n*∗*k* is filled and each cell filling takes O(n) time. Here n*n* represents the number of cities and k*k* is the total number of weeks.
* Space complexity : O(k)*O*(*k*). dp*dp* array of size nk*nk* is used.

**Cherry Pickup**

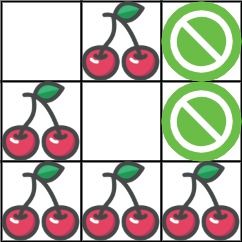
You are given an n x n grid representing a field of cherries, each cell is one of three possible integers.

* 0 means the cell is empty, so you can pass through,
* 1 means the cell contains a cherry that you can pick up and pass through, or
* -1 means the cell contains a thorn that blocks your way.

Return *the maximum number of cherries you can collect by following the rules below*:

* Starting at the position (0, 0) and reaching (n - 1, n - 1) by moving right or down through valid path cells (cells with value 0 or 1).
* After reaching (n - 1, n - 1), returning to (0, 0) by moving left or up through valid path cells.
* When passing through a path cell containing a cherry, you pick it up, and the cell becomes an empty cell 0.
* If there is no valid path between (0, 0) and (n - 1, n - 1), then no cherries can be collected.

**Example 1:**



**Input:** grid = [[0,1,-1],[1,0,-1],[1,1,1]]

**Output:** 5

**Explanation:** The player started at (0, 0) and went down, down, right right to reach (2, 2).

4 cherries were picked up during this single trip, and the matrix becomes [[0,1,-1],[0,0,-1],[0,0,0]].

Then, the player went left, up, up, left to return home, picking up one more cherry.

The total number of cherries picked up is 5, and this is the maximum possible.

**Example 2:**

**Input:** grid = [[1,1,-1],[1,-1,1],[-1,1,1]]

**Output:** 0

**Constraints:**

* n == grid.length
* n == grid[i].length
* 1 <= n <= 50
* grid[i][j] is -1, 0, or 1.
* grid[0][0] != -1
* grid[n - 1][n - 1] != -1

#### **Approach #1: Greedy [Wrong Answer]**

**Intuition**

Let's find the most cherries we can pick up with one path, pick them up, then find the most cherries we can pick up with a second path on the remaining field.

Though a counter example might be hard to think of, this approach fails to find the best answer to this case:

11100

00101

10100

00100

00111

**Algorithm**

We can use dynamic programming to find the most number of cherries dp[i][j] that can be picked up from any location (i, j) to the bottom right corner. This is a classic question very similar to [Minimum Path Sum](https://leetcode.com/problems/minimum-path-sum/description/), refer to the link if you are not familiar with this type of question.

After, we can find an first path that maximizes the number of cherries taken by using our completed dp as an oracle for deciding where to move. We'll choose the move that allows us to pick up more cherries (based on comparing dp[i+1][j] and dp[i][j+1]).

After taking the cherries from that path (and removing it from the grid), we'll take the cherries again.

|  |
| --- |
| class Solution {  public int cherryPickup(int[][] grid) {  int ans = 0;  int[][] path = bestPath(grid);  if (path == null) return 0;  for (int[] step: path) {  ans += grid[step[0]][step[1]];  grid[step[0]][step[1]] = 0;  }  for (int[] step: bestPath(grid))  ans += grid[step[0]][step[1]];  return ans;  }  public int[][] bestPath(int[][] grid) {  int N = grid.length;  int[][] dp = new int[N][N];  for (int[] row: dp) Arrays.fill(row, Integer.MIN\_VALUE);  dp[N-1][N-1] = grid[N-1][N-1];  for (int i = N-1; i >= 0; --i) {  for (int j = N-1; j >= 0; --j) {  if (grid[i][j] >= 0 && (i != N-1 || j != N-1)) {  dp[i][j] = Math.max(i+1 < N ? dp[i+1][j] : Integer.MIN\_VALUE,  j+1 < N ? dp[i][j+1] : Integer.MIN\_VALUE);  dp[i][j] += grid[i][j];  }  }  }  if (dp[0][0] < 0) return null;  int[][] ans = new int[2\*N - 1][2];  int i = 0, j = 0, t = 0;  while (i != N-1 || j != N-1) {  if (j+1 == N || i+1 < N && dp[i+1][j] >= dp[i][j+1]) i++;  else j++;  ans[t][0] = i;  ans[t][1] = j;  t++;  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N^2)*O*(*N*2), where N*N* is the length of grid. Our dynamic programming consists of two for-loops of length N.
* Space Complexity: O(N^2)*O*(*N*2), the size of dp.

#### **Approach #2: Dynamic Programming (Top Down) [Accepted]**

**Intuition**

Instead of walking from end to beginning, let's reverse the second leg of the path, so we are only considering two paths from the beginning to the end.

Notice after t steps, each position (r, c) we could be, is on the line r + c = t. So if we have two people at positions (r1, c1) and (r2, c2), then r2 = r1 + c1 - c2. That means the variables r1, c1, c2 uniquely determine 2 people who have walked the same r1 + c1 number of steps. This sets us up for dynamic programming quite nicely.

**Algorithm**

Let dp[r1][c1][c2] be the most number of cherries obtained by two people starting at (r1, c1) and (r2, c2) and walking towards (N-1, N-1) picking up cherries, where r2 = r1+c1-c2.

If grid[r1][c1] and grid[r2][c2] are not thorns, then the value of dp[r1][c1][c2] is (grid[r1][c1] + grid[r2][c2]), plus the maximum of dp[r1+1][c1][c2], dp[r1][c1+1][c2], dp[r1+1][c1][c2+1], dp[r1][c1+1][c2+1] as appropriate. We should also be careful to not double count in case (r1, c1) == (r2, c2).

Why did we say it was the maximum of dp[r+1][c1][c2] etc.? It corresponds to the 4 possibilities for person 1 and 2 moving down and right:

* Person 1 down and person 2 down: dp[r1+1][c1][c2];
* Person 1 right and person 2 down: dp[r1][c1+1][c2];
* Person 1 down and person 2 right: dp[r1+1][c1][c2+1];
* Person 1 right and person 2 right: dp[r1][c1+1][c2+1];

|  |
| --- |
| class Solution {  int[][][] memo;  int[][] grid;  int N;  public int cherryPickup(int[][] grid) {  this.grid = grid;  N = grid.length;  memo = new int[N][N][N];  for (int[][] layer: memo)  for (int[] row: layer)  Arrays.fill(row, Integer.MIN\_VALUE);  return Math.max(0, dp(0, 0, 0));  }  public int dp(int r1, int c1, int c2) {  int r2 = r1 + c1 - c2;  if (N == r1 || N == r2 || N == c1 || N == c2 ||  grid[r1][c1] == -1 || grid[r2][c2] == -1) {  return -999999;  } else if (r1 == N-1 && c1 == N-1) {  return grid[r1][c1];  } else if (memo[r1][c1][c2] != Integer.MIN\_VALUE) {  return memo[r1][c1][c2];  } else {  int ans = grid[r1][c1];  if (c1 != c2) ans += grid[r2][c2];  ans += Math.max(Math.max(dp(r1, c1+1, c2+1), dp(r1+1, c1, c2+1)),  Math.max(dp(r1, c1+1, c2), dp(r1+1, c1, c2)));  memo[r1][c1][c2] = ans;  return ans;  }  }  } |

**Complexity Analysis**

* Time Complexity: O(N^3)*O*(*N*3), where N*N* is the length of grid. Our dynamic programming has O(N^3)*O*(*N*3) states.
* Space Complexity: O(N^3)*O*(*N*3), the size of memo.

#### **Approach #3: Dynamic Programming (Bottom Up) [Accepted]**

**Intuition**

Like in Approach #2, we have the idea of dynamic programming.

Say r1 + c1 = t is the t-th layer. Since our recursion only references the next layer, we only need to keep two layers in memory at a time.

**Algorithm**

At time t, let dp[c1][c2] be the most cherries that we can pick up for two people going from (0, 0) to (r1, c1) and (0, 0) to (r2, c2), where r1 = t-c1, r2 = t-c2. Our dynamic program proceeds similarly to Approach #2.

|  |
| --- |
| class Solution {  public int cherryPickup(int[][] grid) {  int N = grid.length;  int[][] dp = new int[N][N];  for (int[] row: dp) Arrays.fill(row, Integer.MIN\_VALUE);  dp[0][0] = grid[0][0];  for (int t = 1; t <= 2\*N - 2; ++t) {  int[][] dp2 = new int[N][N];  for (int[] row: dp2) Arrays.fill(row, Integer.MIN\_VALUE);  for (int i = Math.max(0, t-(N-1)); i <= Math.min(N-1, t); ++i) {  for (int j = Math.max(0, t-(N-1)); j <= Math.min(N-1, t); ++j) {  if (grid[i][t-i] == -1 || grid[j][t-j] == -1) continue;  int val = grid[i][t-i];  if (i != j) val += grid[j][t-j];  for (int pi = i-1; pi <= i; ++pi)  for (int pj = j-1; pj <= j; ++pj)  if (pi >= 0 && pj >= 0)  dp2[i][j] = Math.max(dp2[i][j], dp[pi][pj] + val);  }  }  dp = dp2;  }  return Math.max(0, dp[N-1][N-1]);  }  } |

**Complexity Analysis**

* Time Complexity: O(N^3)*O*(*N*3), where N*N* is the length of grid. We have three for-loops of size O(N)*O*(*N*).
* Space Complexity: O(N^2)*O*(*N*2), the sizes of dp and dp2.

**Design Snake Game**

Design a [Snake game](https://en.wikipedia.org/wiki/Snake_(video_game)) that is played on a device with screen size height x width. [Play the game online](http://patorjk.com/games/snake/) if you are not familiar with the game.

The snake is initially positioned at the top left corner (0, 0) with a length of 1 unit.

You are given an array food where food[i] = (ri, ci) is the row and column position of a piece of food that the snake can eat. When a snake eats a piece of food, its length and the game's score both increase by 1.

Each piece of food appears one by one on the screen, meaning the second piece of food will not appear until the snake eats the first piece of food.

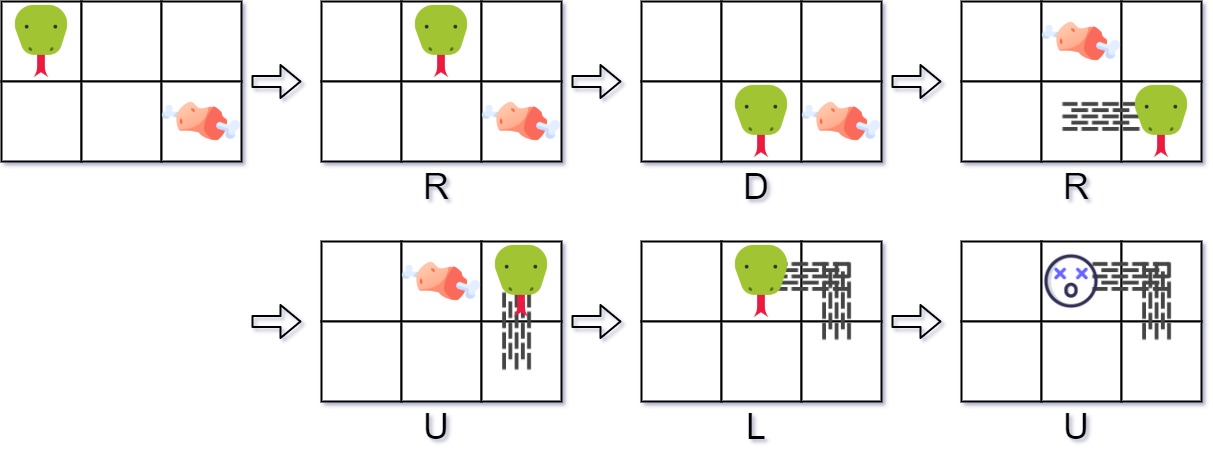
When a piece of food appears on the screen, it is **guaranteed** that it will not appear on a block occupied by the snake.

The game is over if the snake goes out of bounds (hits a wall) or if its head occupies a space that its body occupies **after** moving (i.e. a snake of length 4 cannot run into itself).

Implement the SnakeGame class:

* SnakeGame(int width, int height, int[][] food) Initializes the object with a screen of size height x width and the positions of the food.
* int move(String direction) Returns the score of the game after applying one direction move by the snake. If the game is over, return -1.

**Example 1:**



**Input**

["SnakeGame", "move", "move", "move", "move", "move", "move"]

[[3, 2, [[1, 2], [0, 1]]], ["R"], ["D"], ["R"], ["U"], ["L"], ["U"]]

**Output**

[null, 0, 0, 1, 1, 2, -1]

**Explanation**

SnakeGame snakeGame = new SnakeGame(3, 2, [[1, 2], [0, 1]]);

snakeGame.move("R"); // return 0

snakeGame.move("D"); // return 0

snakeGame.move("R"); // return 1, snake eats the first piece of food. The second piece of food appears

// at (0, 1).

snakeGame.move("U"); // return 1

snakeGame.move("L"); // return 2, snake eats the second food. No more food appears.

snakeGame.move("U"); // return -1, game over because snake collides with border

**Constraints:**

* 1 <= width, height <= 104
* 1 <= food.length <= 50
* food[i].length == 2
* 0 <= ri < height
* 0 <= ci < width
* direction.length == 1
* direction is 'U', 'D', 'L', or 'R'.
* At most 104 calls will be made to move.

## Solution

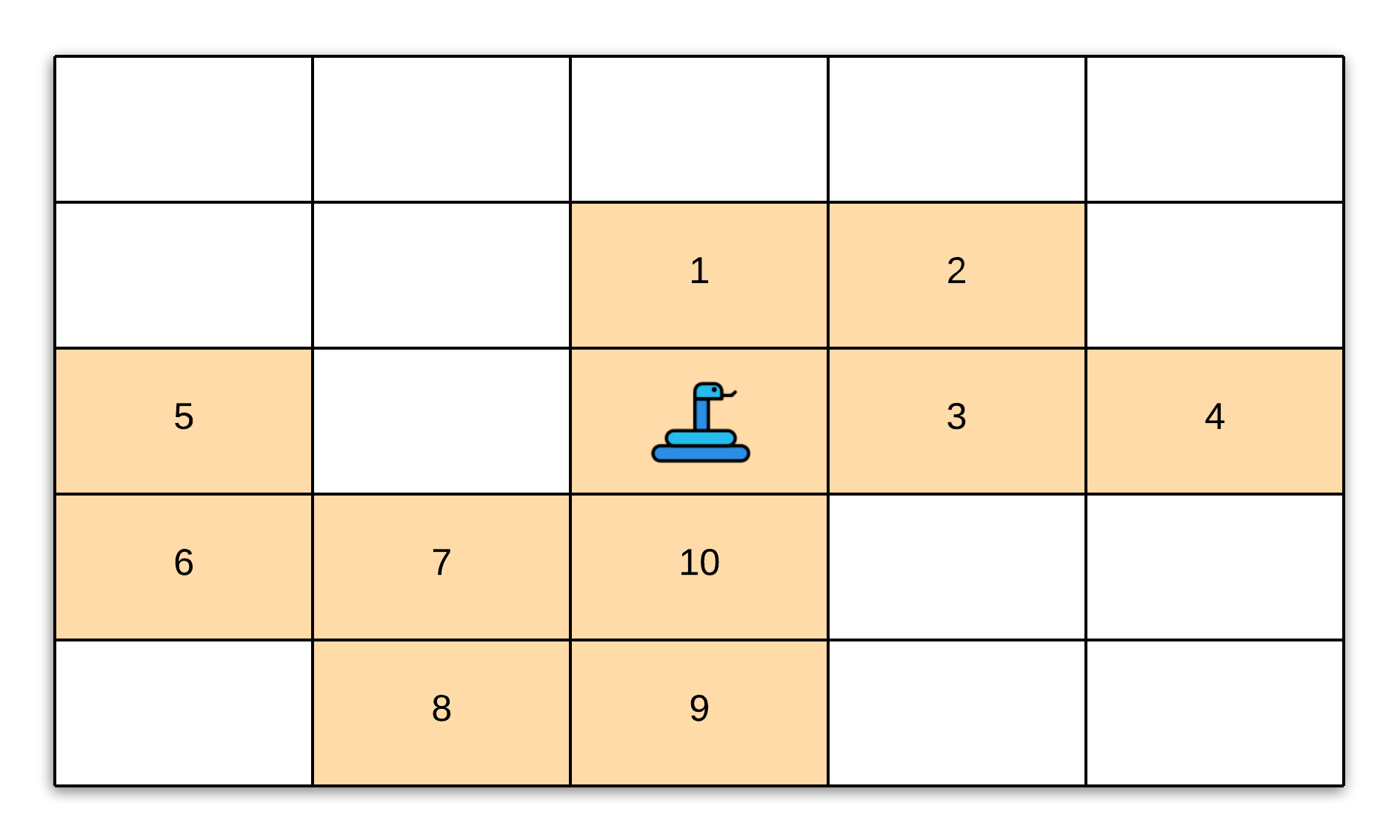
#### **Overview**

Who doesn't feel nostalgic while thinking about the famous Snake video game? It used to be (and still is) the goto video game on phones and other platforms for so many of us and there are countless variations of the game out there. The version that this problem talks about is the most basic one. And this being a design problem makes things more interesting!

Let's go over the details in the problem statement once.

* We're given the width and height of the grid over which the snake moves.
* Additionally, we are also given the list of grid positions where the food would appear one after the other. Just like the traditional snake, the next food item only appears once the current one is consumed.
* Consuming a piece of food increasses the length of the snake by one. In terms of our problem statement, the length of the snake is increased by one more cell from the grid with each cell being of unit length and width.
* The snake can move in four directions U, D, L, and R. Everytime the snake has to be moved, the move() function would be called and this is the only function we need to focus on in this question.
* The game ends when either of these conditions happens:
  + The snake becomes too long to potentially fit inside the grid or
  + The snake hits one of the boundaries which would happen in the previous case as well.
  + The snake bites itself i.e. when the head of the snake collides with its body in the next move.

The problem statement doesn't have any follow up statements, but we're going to discuss a follow-up to this question where the wall becomes infinite i.e. the snake can move across walls and the only condition then for the game to end is when the snake crashes into itself on the grid.



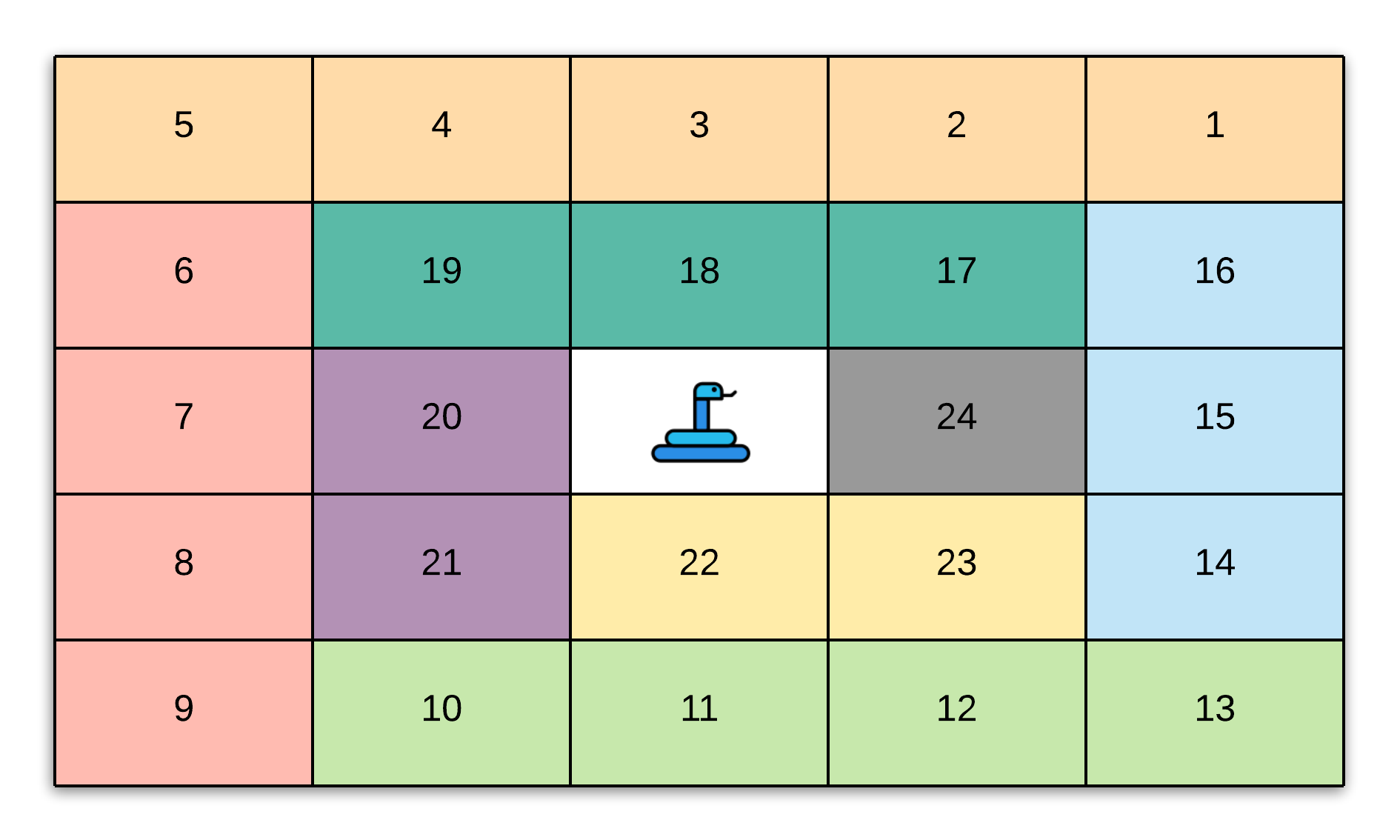
#### **Approach: Queue and Hash Set**

**Intuition**

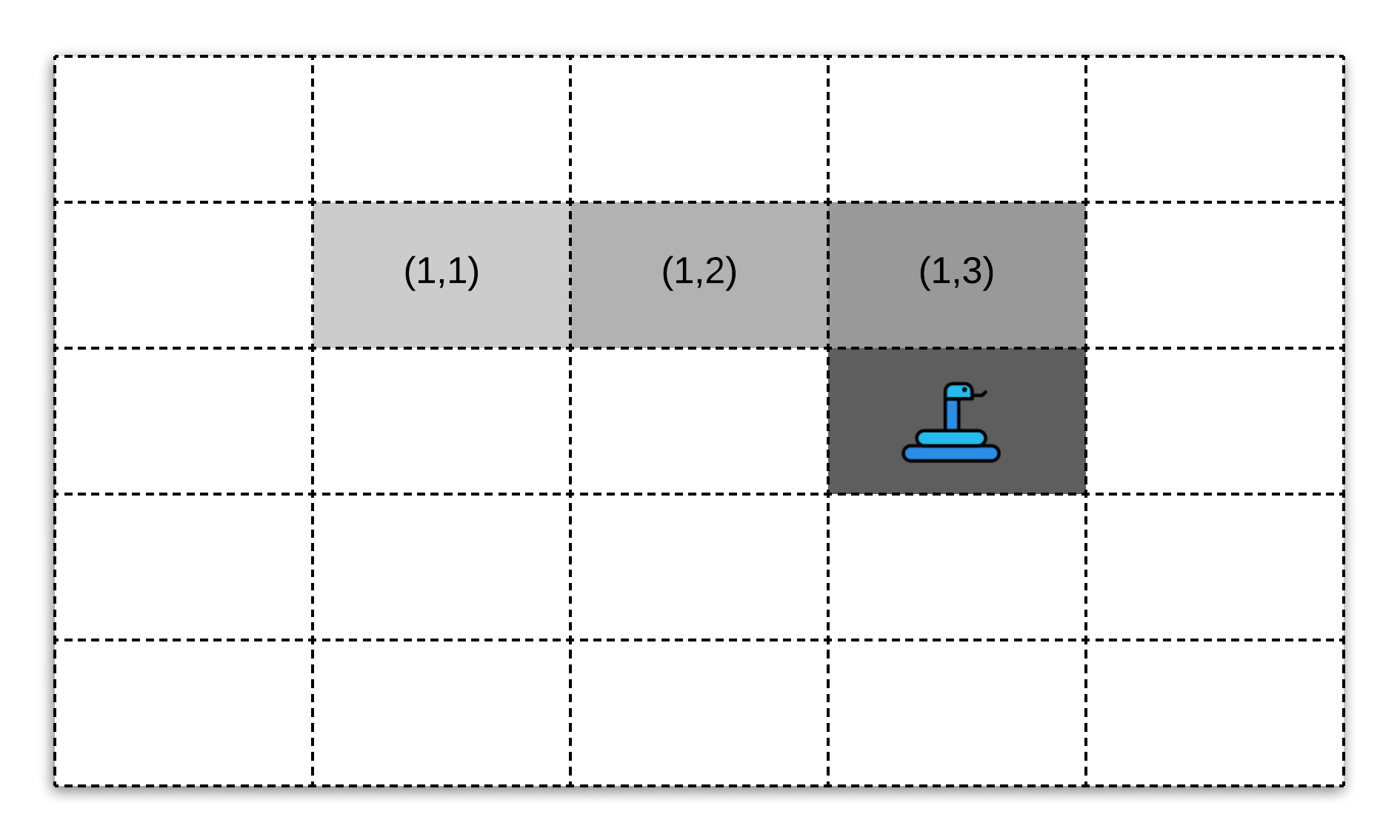
Let's start by thinking about how we want to store the snake?

In terms of the grid, a snake is just an **ordered** collection of cells.

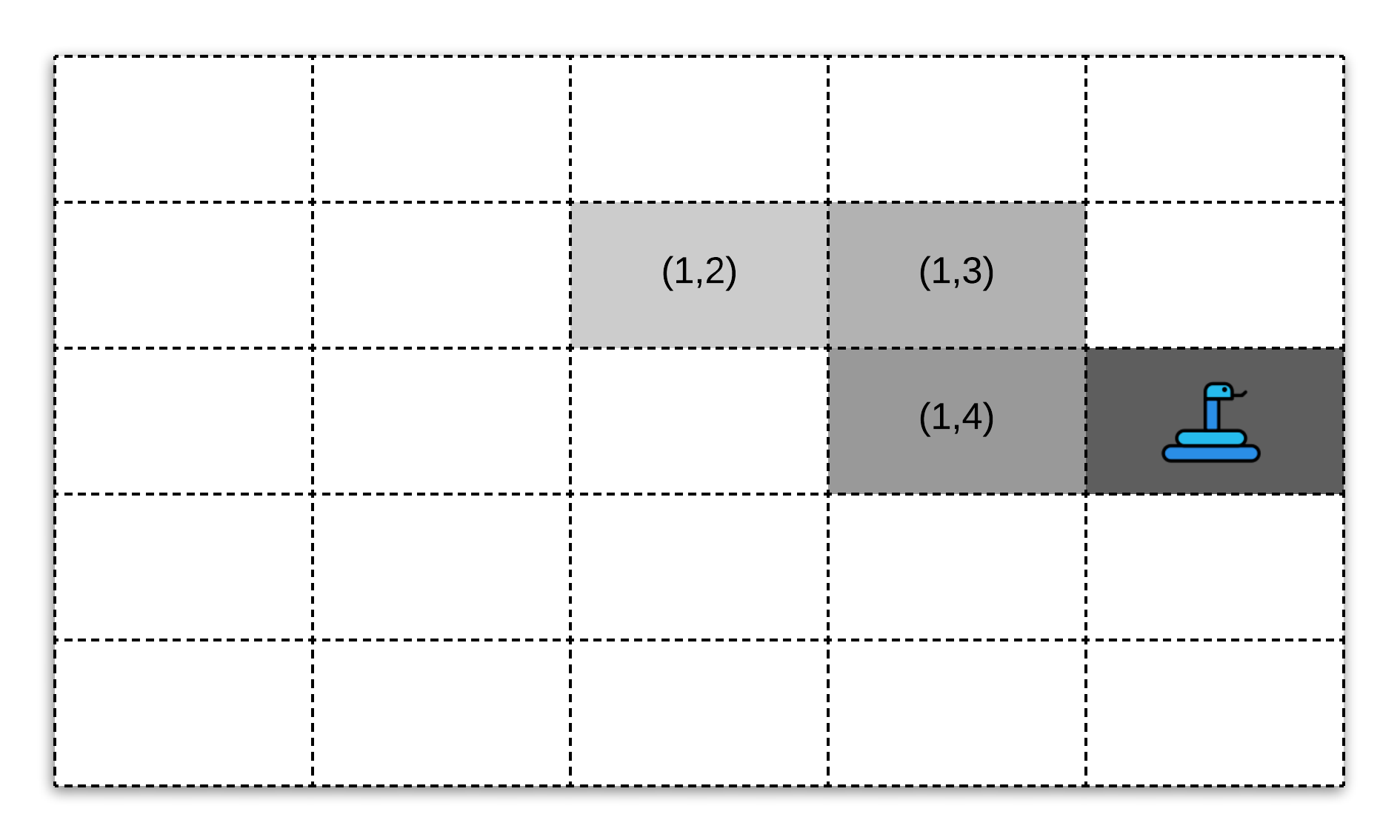
We can technically use an array to store the cells representing a snake. However, we would need to instantiate an array the size of width \* height of the grid since a snake can be composed of all the the cells of the grid in the worst case. A spiral kind of a snake. Let's look at such a snake occupying the grid.



This structure is highly unlikely given the random nature of food items appearing on the grid. However, we would need an array the size of the grid to be able to hold this big a snake. The breaking point for an array is when we have to move the snake from one position to another. Let's see what happens to the snake when it moves by one in a direction. The result overall would be the same with some minor changes based on the direction.



In the above figure, we have a snake that occupies 4 cells across the grid or in other words, is of length 4. The snake can be represented by the following collection of cells: [(1,1), (1,2), (1,3), (2,3)]. Now say we have the snake move in the right direction i.e. R. The snake now would look like this across the grid.



Now here, after moving one step to the right, the snake is represented by the cells [(1,2), (1,3), (2,3), (2,4)].

In order to achieve this with an array, we would have to move all the cells around per move which is not exactly ideal. We can build some complicated logic around the movement of the snake in an array but that won't be worth the fixed space complexity that an array would occupy.

Let's see what data structure would naturally fit our requirements for the snake. There are two basic requirements we need to satisfy:

1. Dynamically add new cells to the snake's body and
2. Move the snake in constant amount of time across the grid.

Let's look at the snake representation between moves from the example above to understand what really is happening here and that will help us get to the data structure we need to use for solving this problem.

**Move with No Food**

We already have an example for such a move so we will simply be looking at the snake representation on the grid to understand what's really happening here.

Before the move, the snake was occupying the following cells of the grid in the specified order:

(1,1), (1,2), (1,3), (2,3)

and after the move, the snake was occupying the following positions on the grid:

(1,2), (1,3), (2,3), (2,4)

If you think about this from a **sliding window** perspective, we simply moves the window one step forward i.e. we removed the **tail** of the window and added a new **head** to the window. The tail in this case was (1,2) and the new head being (2,4).

**Move with Food Consumption**

Now let's look at a move by the snake wherein they consume a food item and grow in length. Suppose the move was the same as before and the spot (2,4) contained a food item. The snake head from the previous example, was at (2,3) on the grid. So, a move to the right would cause them to consume this food item thus extending their overall length by one. So now, instead of occupying 4 cells on the grid, the snake would occupy 5 cells. Let's concretely look at the snake representations before and after the move.

Before the move, the snake was occupying the following cells of the grid in the specified order:

(1,1), (1,2), (1,3), (2,3)

and after the move, the snake was occupying the following positions on the grid:

(1,1), (1,2), (1,3), (2,3), (2,4)

Here, we simply added a new **head** to the snake with the head being the cell (2,4). The tail remained the same in this case. These are the only two possibilities for moves that can happen other than the termination conditions for the game. Based on them, let's see what operations out data structure needs to support concretely for us to be able to perform these moves efficiently.

Our abstract data structure needs to support the following operations efficiently.

1. Grow in size dynamically. Note that we never **shrink** in size. The snake can stay the same size as before or grow in size due to the consumption of a food item on the grid. But they can't shrink in size.
2. Maintain a specified ordering of cells in order to represent the snake.
3. Extract the tail cell and potentially add a new head cell to the ordering of cells to represent the updated snake post a move. This is the most important operation of all and this points to a very specific data structure.

Based on the third operation, we can see that the **Queue** would be a good data structure to use since we need to have quick access to the first and last elements of an ordered list and a queue gives us exactly that.

A queue is an abstract data structure with some specified properties which meets our requirements. It can be represented by an array or a linked list. For our purposes, since we need a data structure with dynamic sizing, we would go with a linked-list based implementation for a queue rather than an array since we don't want to pre-allocate any memory for the array and only allocate on the fly. A linked list would be a great fit here since we don't require random access to cells of the snake.

**Algorithm**

1. Initialize a queue containing a single cell (0,0) which is the initial position of the snake at the beginning of the game. Note that we will be doing this in the constructor of the class and not in the move function.
2. The fist thing we need to do inside the move function is to compute the **new head** based on the direction of the move. As we saw in the intuition section, irrespective of the kind of move, we will always get a new head. We need the new head position to determine if the snake has hit a boundary and hence, terminate the game.
3. Let's first discuss the termination conditions before moving on to the modifications we would make to our queue data structure.
   1. The first condition is if the snake cross either of the boundaries of the grid after the mode, then we terminate. So for this, we simply check if the new head (new\_head) satisfies new\_head[0] < 0 or new\_head[0] > height or new\_head[1] < 0 or new\_head[1] > width.
   2. The second condition is if the snake bites itself after the move. An important thing to remember here is that the current tail of the snake is **not** a part of the snake's body. If the move doesn't involve a food, then the tail gets updated (removed) as we have seen. If this is a food move, then the snake cannot bite itself because the food cannot appear on any of the cells occupied by the snake (according to the problem statement).

In order to check if the snake bites itself we need to check if the new head already exists in our queue or not. This can turn out to be an \mathcal{O}(N)O(*N*) operation and that would be costly. So, at the expense of memory, we can also use an additional dictionary data structure to keep the positions of the snake. This dictionary will only be used for this particular check. We can't do with just a dictionary because a dictionary doesn't have an ordered list of elements and we need the ordering for our implementation.

1. If none of the termination conditions have been met, then we will continue to update our queue with the new head and potentially remove the old tail. If the new head lands on a position which contains food, then we simply add the new head to our queue representing the snake. We won't pop the tail in this case since the length of the snake has increased by 1.
2. After each move, we return the length of the snake if this was a valid move. Else, we return -1 to indicate that the game is over.

|  |
| --- |
| class SnakeGame {  HashMap<Pair<Integer, Integer>, Boolean> snakeMap;  Deque<Pair<Integer, Integer>> snake;  int[][] food;  int foodIndex;  int width;  int height;  /\*\*  \* Initialize your data structure here.  \*  \* @param width - screen width  \* @param height - screen height  \* @param food - A list of food positions E.g food = [[1,1], [1,0]] means the first food is  \* positioned at [1,1], the second is at [1,0].  \*/  public SnakeGame(int width, int height, int[][] food) {  this.width = width;  this.height = height;  this.food = food;  this.snakeMap = new HashMap<Pair<Integer, Integer>, Boolean>();  this.snakeMap.put(new Pair<Integer, Integer>(0,0), true); // intially at [0][0]  this.snake = new LinkedList<Pair<Integer, Integer>>();  this.snake.offerLast(new Pair<Integer, Integer>(0,0));  }  /\*\*  \* Moves the snake.  \*  \* @param direction - 'U' = Up, 'L' = Left, 'R' = Right, 'D' = Down  \* @return The game's score after the move. Return -1 if game over. Game over when snake crosses  \* the screen boundary or bites its body.  \*/  public int move(String direction) {  Pair<Integer, Integer> snakeCell = this.snake.peekFirst();  int newHeadRow = snakeCell.getKey();  int newHeadColumn = snakeCell.getValue();  switch (direction) {  case "U":  newHeadRow--;  break;  case "D":  newHeadRow++;  break;  case "L":  newHeadColumn--;  break;  case "R":  newHeadColumn++;  break;  }  Pair<Integer, Integer> newHead = new Pair<Integer, Integer>(newHeadRow, newHeadColumn);  Pair<Integer, Integer> currentTail = this.snake.peekLast();  // Boundary conditions.  boolean crossesBoundary1 = newHeadRow < 0 || newHeadRow >= this.height;  boolean crossesBoundary2 = newHeadColumn < 0 || newHeadColumn >= this.width;  // Checking if the snake bites itself.  boolean bitesItself = this.snakeMap.containsKey(newHead) && !(newHead.getKey() == currentTail.getKey() && newHead.getValue() == currentTail.getValue());    // If any of the terminal conditions are satisfied, then we exit with rcode -1.  if (crossesBoundary1 || crossesBoundary2 || bitesItself) {  return -1;  }  // If there's an available food item and it is on the cell occupied by the snake after the move,  // eat it.  if ((this.foodIndex < this.food.length)  && (this.food[this.foodIndex][0] == newHeadRow)  && (this.food[this.foodIndex][1] == newHeadColumn)) {  this.foodIndex++;  } else {  this.snake.pollLast();  this.snakeMap.remove(currentTail);  }  // A new head always gets added  this.snake.addFirst(newHead);  // Also add the head to the set  this.snakeMap.put(newHead, true);  return this.snake.size() - 1;  }    }  /\*\*  \* Your SnakeGame object will be instantiated and called as such: SnakeGame obj = new  \* SnakeGame(width, height, food); int param\_1 = obj.move(direction);  \*/ |

**Complexity Analysis**

Let W*W* represent the width of the grid and H*H* represent the height of the grid. Also, let N*N* represent the number of food items in the list.

* Time Complexity:
  + The time complexity of the move function is \mathcal{O}(1)O(1).
  + The time taken to calculate bites\_itself is constant since we are using a dictionary to search for the element.
  + The time taken to add and remove an element from the queue is also constant.
* Space Complexity:
  + The space complexity is \mathcal{O}(W \times H + N)O(*W*×*H*+*N*)
  + \mathcal{O}(N)O(*N*) is used by the food data structure.
  + \mathcal{O}(W \times H)O(*W*×*H*) is used by the snake and the snake\_set data structures. At most, we can have snake that occupies all the cells of the grid as explained in the beginning of the article.

**Design Hit Counter**

Design a hit counter which counts the number of hits received in the past 5 minutes.

Each function accepts a timestamp parameter (in seconds granularity) and you may assume that calls are being made to the system in chronological order (ie, the timestamp is monotonically increasing). You may assume that the earliest timestamp starts at 1.

It is possible that several hits arrive roughly at the same time.

**Example:**

HitCounter counter = new HitCounter();

// hit at timestamp 1.

counter.hit(1);

// hit at timestamp 2.

counter.hit(2);

// hit at timestamp 3.

counter.hit(3);

// get hits at timestamp 4, should return 3.

counter.getHits(4);

// hit at timestamp 300.

counter.hit(300);

// get hits at timestamp 300, should return 4.

counter.getHits(300);

// get hits at timestamp 301, should return 3.

counter.getHits(301);

**Follow up:**  
What if the number of hits per second could be very large? Does your design scale?

#### **Approach #1: Using Queue**

**Intuition**

A key observation here is that all the timestamps that we will encounter are going to be in increasing order. Also as the timestamps' value increases we have to ignore the timestamps that occurred previously (having a difference of 300 or more with the latest timestamp). This is the case of considering the elements in the FIFO manner (First in first out) which is best solved by using the "queue" data structure.

**Algorithm**

We will add each timestamp to the queue in the hit method and will remove all the timestamps with difference greater than or equal to 300 from the queue inside getHits. The answer returned from the getHits method is then simply the size of the queue.

Below is the implementation of this approach.

|  |
| --- |
| class HitCounter {  private Queue<Integer> hits;  /\*\* Initialize your data structure here. \*/  public HitCounter() {  this.hits = new LinkedList<Integer>();  }    /\*\* Record a hit.  @param timestamp - The current timestamp (in seconds granularity). \*/  public void hit(int timestamp) {  this.hits.add(timestamp);  }    /\*\* Return the number of hits in the past 5 minutes.  @param timestamp - The current timestamp (in seconds granularity). \*/  public int getHits(int timestamp) {  while (!this.hits.isEmpty()) {  int diff = timestamp - this.hits.peek();  if (diff >= 300) this.hits.remove();  else break;  }  return this.hits.size();  }  } |

**Complexity Analysis**

* Time Complexity
  + hit - Since inserting a value in the queue takes place in O(1)*O*(1) time, hence hit method works in O(1)*O*(1).
  + getHits - Assuming a total of n*n* values present in the queue at a time and the total number of timestamps encountered throughout is N*N*. In the worst case scenario, we might end up removing all the entries from the queue in getHits method if the difference in timestamp is greater than or equal to 300. Hence in the worst case, a "single" call to the getHits method can take O(n)*O*(*n*) time. However, we must notice that each timestamp is processed only twice (first while adding the timestamp in the queue in hit method and second while removing the timestamp from the queue in the getHits method). Hence if the total number of timestamps encountered throughout is N*N*, the overall time taken by getHits method is O(N)*O*(*N*). This results in an amortized time complexity of O(1)*O*(1) for a single call to getHits method.
* Space Complexity: Considering the total timestamps encountered throughout to be N*N*, the queue can have upto N*N* elements, hence overall space complexity of this approach is O(N)*O*(*N*).

#### **Approach #2: Using Deque with Pairs**

Consider the follow up, where we have multiple hits arriving at the "same" timestamps. We can optimize Approach 1 even further. In this approach, we'll not only keep the timestamps but will also keep the count of the timestamps as well. For example, if we call hit method 5 times for timestamp = 1, the queue in case of Approach 1 will look like [1, 1, 1, 1, 1]. This will lead to 5 removals in the getHits method when we remove the value 1 from the queue. To avoid this repetitive removals of the same value, in Approach 2, we'll store the value as (1, 5) where the first value 1 is the timestamp and the second value 5 is the count. For this, we'll use the "deque" data structure which allows us to insert and delete values from both the ends of the queue.

**Algorithm**

The updated algorithm in Approach 2 is as follows.

* If we encounter the hit for the same timestamp, instead of appending a new entry in the deque, we simply increment the count of the latest timestamp.
* In order to keep the track of total number of elements (for the last 300 seconds), we also use a variable total which we initialize to 0 and keep updating as we add or remove the elements from the queue. We increment the value of total by 1 when hit method is called and we decrement by the value of total by the count of the timestamp that we remove from the queue.

Below is the implementation of this approach.

|  |
| --- |
| class HitCounter {  private int total;  private Deque<Pair<Integer, Integer>> hits;  /\*\* Initialize your data structure here. \*/  public HitCounter() {  // Initialize total to 0  this.total = 0;  this.hits = new LinkedList<Pair<Integer, Integer>>();  }    /\*\* Record a hit.  @param timestamp - The current timestamp (in seconds granularity). \*/  public void hit(int timestamp) {  if (this.hits.isEmpty() || this.hits.getLast().getKey() != timestamp) {  // Insert the new timestamp with count = 1  this.hits.add(new Pair<Integer, Integer>(timestamp, 1));  } else {  // Update the count of latest timestamp by incrementing the count by 1  // Obtain the current count of the latest timestamp  int prevCount = this.hits.getLast().getValue();  // Remove the last pair of (timestamp, count) from the deque  this.hits.removeLast();  // Insert a new pair of (timestamp, updated count) in the deque  this.hits.add(new Pair<Integer, Integer>(timestamp, prevCount + 1));  }  // Increment total  this.total++;  }    /\*\* Return the number of hits in the past 5 minutes.  @param timestamp - The current timestamp (in seconds granularity). \*/  public int getHits(int timestamp) {  while (!this.hits.isEmpty()) {  int diff = timestamp - this.hits.getFirst().getKey();  if (diff >= 300) {  // Decrement total by the count of the oldest timestamp  this.total -= this.hits.getFirst().getValue();  this.hits.removeFirst();  }  else break;  }  return this.total;  }  } |

**Complexity Analysis**

In the worst case, when there are not many repetitions, the time complexity and space complexity of Approach 2 is the same as Approach 1. However in case we have repetitions (say k repetitions of a particular ith timestamp), the time complexity and space complexities are as follows.

* Time Complexity:
  + hit - O(1)*O*(1).
  + getHits - If there are a total of n*n* pairs present in the deque, worst case time complexity can be O(n)*O*(*n*). However, by clubbing all the timestamps with same value together, for the ith timestamp with k repetitions, the time complexity is O(1)*O*(1) as here, instead of removing all those k repetitions, we only remove a single entry from the deque.
* Space complexity: If there are a total of N*N* elements that we encountered throughout, the space complexity is O(N)*O*(*N*) (similar to Approach 1). However, in the case of repetitions, the space required for storing those k values O(1)*O*(1).

**Insert Delete GetRandom O(1) - Duplicates allowed**

Design a data structure that supports all following operations in *average* **O(1)** time.

**Note: Duplicate elements are allowed.**

1. insert(val): Inserts an item val to the collection.
2. remove(val): Removes an item val from the collection if present.
3. getRandom: Returns a random element from current collection of elements. The probability of each element being returned is **linearly related** to the number of same value the collection contains.

**Example:**

// Init an empty collection.

RandomizedCollection collection = new RandomizedCollection();

// Inserts 1 to the collection. Returns true as the collection did not contain 1.

collection.insert(1);

// Inserts another 1 to the collection. Returns false as the collection contained 1. Collection now contains [1,1].

collection.insert(1);

// Inserts 2 to the collection, returns true. Collection now contains [1,1,2].

collection.insert(2);

// getRandom should return 1 with the probability 2/3, and returns 2 with the probability 1/3.

collection.getRandom();

// Removes 1 from the collection, returns true. Collection now contains [1,2].

collection.remove(1);

// getRandom should return 1 and 2 both equally likely.

collection.getRandom();

## Solution

#### **Intuition**

We must support three operations with duplicates:

1. insert
2. remove
3. getRandom

To getRandom in O(1)*O*(1) and have it scale linearly with the number of copies of a value. The simplest solution is to store all values in a list. Once all values are stored, all we have to do is pick a random index.

We don't care about the order of our elements, so insert can be done in O(1)*O*(1) using a dynamic array (ArrayList in Java or list in Python).

The issue we run into is how to go about an O(1) remove. Generally we learn that removing an element from an array takes a place in O(N)*O*(*N*), unless it is the last element in which case it is O(1)*O*(1).

The key here is that we don't care about order. For the purposes of this problem, if we want to remove the element at the ith index, we can simply swap the ith element and the last element, and perform an O(1)*O*(1) pop (technically we don't have to swap, we just have to copy the last element into index i because it's popped anyway).

With this in mind, the most difficult part of the problem becomes finding the index of the element we have to remove. All we have to do is have an accompanying data structure that maps the element values to their index.

#### **Approach 1: ArrayList + HashMap**

**Algorithm**

We will keep a list to store all our elements. In order to make finding the index of elements we want to remove O(1)*O*(1), we will use a HashMap or dictionary to map values to all indices that have those values. To make this work each value will be mapped to a set of indices. The tricky part is properly updating the HashMap as we modify the list.

* insert: Append the element to the list and add the index to HashMap[element].
* remove: This is the tricky part. We find the index of the element using the HashMap. We use the trick discussed in the intuition to remove the element from the list in O(1)*O*(1). Since the last element in the list gets moved around, we have to update its value in the HashMap. We also have to get rid of the index of the element we removed from the HashMap.
* getRandom: Sample a random element from the list.

**Implementation**

|  |
| --- |
| public class RandomizedCollection {  ArrayList<Integer> lst;  HashMap<Integer, Set<Integer>> idx;  java.util.Random rand = new java.util.Random();  /\*\* Initialize your data structure here. \*/  public RandomizedCollection() {  lst = new ArrayList<Integer>();  idx = new HashMap<Integer, Set<Integer>>();  }  /\*\* Inserts a value to the collection. Returns true if the collection did not already contain the specified element. \*/  public boolean insert(int val) {  if (!idx.containsKey(val)) idx.put(val, new LinkedHashSet<Integer>());  idx.get(val).add(lst.size());  lst.add(val);  return idx.get(val).size() == 1;  }  /\*\* Removes a value from the collection. Returns true if the collection contained the specified element. \*/  public boolean remove(int val) {  if (!idx.containsKey(val) || idx.get(val).size() == 0) return false;  int remove\_idx = idx.get(val).iterator().next();  idx.get(val).remove(remove\_idx);  int last = lst.get(lst.size() - 1);  lst.set(remove\_idx, last);  idx.get(last).add(remove\_idx);  idx.get(last).remove(lst.size() - 1);  lst.remove(lst.size() - 1);  return true;  }  /\*\* Get a random element from the collection. \*/  public int getRandom() {  return lst.get(rand.nextInt(lst.size()));  }  } |

**Complexity Analysis**

* Time complexity : O(N)*O*(*N*), with N*N* being the number of operations. All of our operations are O(1)*O*(1), giving N \* O(1) = O(N)*N*∗*O*(1)=*O*(*N*).
* Space complexity : O(N)*O*(*N*), with N*N* being the number of operations. The worst case scenario is if we get N*N* add operations, in which case our ArrayList and our HashMap grow to size N*N*.

**Number of Islands II**

A 2d grid map of m rows and n columns is initially filled with water. We may perform an *addLand* operation which turns the water at position (row, col) into a land. Given a list of positions to operate, **count the number of islands after each *addLand* operation**. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

**Example:**

**Input:** m = 3, n = 3, positions = [[0,0], [0,1], [1,2], [2,1]]

**Output:** [1,1,2,3]

**Explanation:**

Initially, the 2d grid grid is filled with water. (Assume 0 represents water and 1 represents land).

0 0 0

0 0 0

0 0 0

Operation #1: addLand(0, 0) turns the water at grid[0][0] into a land.

1 0 0

0 0 0 Number of islands = 1

0 0 0

Operation #2: addLand(0, 1) turns the water at grid[0][1] into a land.

1 1 0

0 0 0 Number of islands = 1

0 0 0

Operation #3: addLand(1, 2) turns the water at grid[1][2] into a land.

1 1 0

0 0 1 Number of islands = 2

0 0 0

Operation #4: addLand(2, 1) turns the water at grid[2][1] into a land.

1 1 0

0 0 1 Number of islands = 3

0 1 0

**Follow up:**

Can you do it in time complexity O(k log mn), where k is the length of the positions?

#### **Approach #1 (Brute force) [Time Limit Exceeded]**

**Algorithm**

Reuse the code for Problem 200: [Number of Islands](https://leetcode.com/problems/number-of-islands/description/), for each addLand operation, just call the numIslands function of Problem 200 to get the number of islands after performing that operation.

|  |
| --- |
| class Solution {  void dfs(char[][] grid, int r, int c, boolean[][] visited) {  int nr = grid.length;  int nc = grid[0].length;  if (r < 0 || c < 0 || r >= nr || c >= nc || grid[r][c] == '0' || visited[r][c]) {  return;  }  visited[r][c] = true;  dfs(grid, r - 1, c, visited);  dfs(grid, r + 1, c, visited);  dfs(grid, r, c - 1, visited);  dfs(grid, r, c + 1, visited);  }  public int numIslands(char[][] grid) {  if (grid == null || grid.length == 0) {  return 0;  }  int nr = grid.length;  int nc = grid[0].length;  boolean[][] visited = new boolean[nr][nc];  for (boolean[] row : visited) {  Arrays.fill(row, false);  }  int num\_islands = 0;  for (int r = 0; r < nr; ++r) {  for (int c = 0; c < nc; ++c) {  if (grid[r][c] == '1' && !visited[r][c]) {  ++num\_islands;  dfs(grid, r, c, visited);  }  }  }  return num\_islands;  }  public List<Integer> numIslands2(int m, int n, int[][] positions) {  List<Integer> ans = new ArrayList<>();  char[][] grid = new char[m][n];  for (char[] row : grid) {  Arrays.fill(row, '0');  }  for (int[] pos : positions) {  grid[pos[0]][pos[1]] = '1';  ans.add(numIslands(grid));  }  return ans;  }  } |

**Complexity Analysis**

* Time complexity : O(L \times m \times n)*O*(*L*×*m*×*n*) where L*L* is the number of operations, m*m* is the number of rows and n*n* is the number of columns.
* Space complexity : O(m \times n)*O*(*m*×*n*) for the grid and visited 2D arrays.

#### **Approach #2: (Ad hoc) [Accepted]**

**Algorithm**

Use a HashMap to map index of a land to its island\_ID (starting from 0). For each addLand operation at position (row, col), check if its adjacent neighbors are in the HashMap or not and put the island\_ID of identified neighbors into a set (where each element is unique):

* if the set is empty, then the new land at position (row, col) forms a new island (monotonically increasing island\_ID by 1);
* if the set contains only one island\_ID, then the new land belongs to an existing island and island\_ID remains unchanged;
* if the set contains more than one island\_ID, then the new land bridges these separate islands into one island, we need to iterate through the HashMap to update this information (time consuming!) and decrease the number of island appropriately.

|  |
| --- |
| class Solution {  public List<Integer> numIslands2(int m, int n, int[][] positions) {  List<Integer> ans = new ArrayList<>();  HashMap<Integer, Integer> land2id = new HashMap<Integer, Integer>();  int num\_islands = 0;  int island\_id = 0;  for (int[] pos : positions) {  int r = pos[0], c = pos[1];  Set<Integer> overlap = new HashSet<Integer>();  if (r - 1 >= 0 && land2id.containsKey((r-1) \* n + c)) {  overlap.add(land2id.get((r-1) \* n + c));  }  if (r + 1 < m && land2id.containsKey((r+1) \* n + c)) {  overlap.add(land2id.get((r+1) \* n + c));  }  if (c - 1 >= 0 && land2id.containsKey(r \* n + c - 1)) {  overlap.add(land2id.get(r \* n + c - 1));  }  if (c + 1 < n && land2id.containsKey(r \* n + c + 1)) {  overlap.add(land2id.get(r \* n + c + 1));  }  if (overlap.isEmpty()) {  ++num\_islands;  land2id.put(r \* n + c, island\_id++);  } else if (overlap.size() == 1) {  land2id.put(r \* n + c, overlap.iterator().next());  } else {  int root\_id = overlap.iterator().next();  for (Map.Entry<Integer, Integer> entry : land2id.entrySet()) {  int k = entry.getKey();  int id = entry.getValue();  if (overlap.contains(id)) {  land2id.put(k, root\_id);  }  }  land2id.put(r \* n + c, root\_id);  num\_islands -= (overlap.size() - 1);  }  ans.add(num\_islands);  }  return ans;  }  } |

**Complexity Analysis**

* Time complexity : O(L^2)*O*(*L*2), for each operation, we have to traverse the entire HashMap to update island id and the number of operations is L*L*.
* Space complexity : O(L)*O*(*L*) for the HashMap.

P.S. C++ solution was accepted with 1409 ms runtime, but Java solution got an TLE (Time Limit Exceeded).

#### **Approach #3: Union Find (aka Disjoint Set) [Accepted]**

**Intuition**

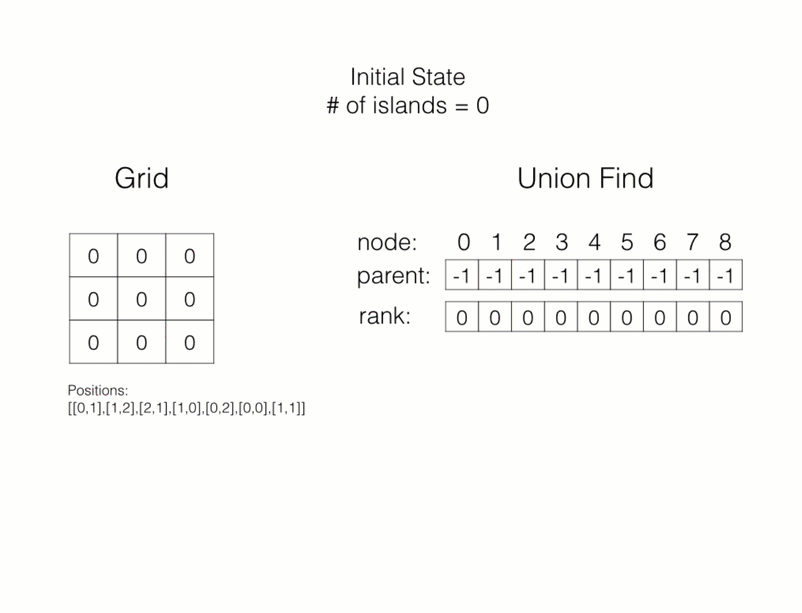
Treat the 2d grid map as an undirected graph (formatted as adjacency matrix) and there is an edge between two horizontally or vertically adjacent nodes of value 1, then the problem reduces to finding the number of connected components in the graph after each addLand operation.

**Algorithm**

Make use of a Union Find data structure of size m\*n to store all the nodes in the graph and initially each node's parent value is set to -1 to represent an empty graph. Our goal is to update Union Find with lands added by addLand operation and union lands belong to the same island.

For each addLand operation at position (row, col), union it with its adjacent neighbors if they belongs to some islands, if none of its neighbors belong to any islands, then initialize the new position as a new island (set parent value to itself) within Union Find.

For detailed description of Union Find (implemented with path compression and union by rank), you can refer to this [article](https://leetcode.com/articles/redundant-connection/).



|  |
| --- |
| class Solution {  class UnionFind {  int count; // # of connected components  int[] parent;  int[] rank;  public UnionFind(char[][] grid) { // for problem 200  count = 0;  int m = grid.length;  int n = grid[0].length;  parent = new int[m \* n];  rank = new int[m \* n];  for (int i = 0; i < m; ++i) {  for (int j = 0; j < n; ++j) {  if (grid[i][j] == '1') {  parent[i \* n + j] = i \* n + j;  ++count;  }  rank[i \* n + j] = 0;  }  }  }  public UnionFind(int N) { // for problem 305 and others  count = 0;  parent = new int[N];  rank = new int[N];  for (int i = 0; i < N; ++i) {  parent[i] = -1;  rank[i] = 0;  }  }  public boolean isValid(int i) { // for problem 305  return parent[i] >= 0;  }  public void setParent(int i) {  parent[i] = i;  ++count;  }  public int find(int i) { // path compression  if (parent[i] != i) parent[i] = find(parent[i]);  return parent[i];  }  public void union(int x, int y) { // union with rank  int rootx = find(x);  int rooty = find(y);  if (rootx != rooty) {  if (rank[rootx] > rank[rooty]) {  parent[rooty] = rootx;  } else if (rank[rootx] < rank[rooty]) {  parent[rootx] = rooty;  } else {  parent[rooty] = rootx; rank[rootx] += 1;  }  --count;  }  }  public int getCount() {  return count;  }  }  public List<Integer> numIslands2(int m, int n, int[][] positions) {  List<Integer> ans = new ArrayList<>();  UnionFind uf = new UnionFind(m \* n);  for (int[] pos : positions) {  int r = pos[0], c = pos[1];  List<Integer> overlap = new ArrayList<>();  if (r - 1 >= 0 && uf.isValid((r-1) \* n + c)) overlap.add((r-1) \* n + c);  if (r + 1 < m && uf.isValid((r+1) \* n + c)) overlap.add((r+1) \* n + c);  if (c - 1 >= 0 && uf.isValid(r \* n + c - 1)) overlap.add(r \* n + c - 1);  if (c + 1 < n && uf.isValid(r \* n + c + 1)) overlap.add(r \* n + c + 1);  int index = r \* n + c;  uf.setParent(index);  for (int i : overlap) uf.union(i, index);  ans.add(uf.getCount());  }  return ans;  }  } |

**Complexity Analysis**

* Time complexity : O(m \times n + L)*O*(*m*×*n*+*L*) where L*L* is the number of operations, m*m* is the number of rows and n*n* is the number of columns. it takes O(m \times n)*O*(*m*×*n*) to initialize UnionFind, and O(L)*O*(*L*) to process positions. Note that Union operation takes essentially constant time[[1]](https://leetcode.com/problems/number-of-islands-ii/solution/#fn1) when UnionFind is implemented with both path compression and union by rank.
* Space complexity : O(m \times n)*O*(*m*×*n*) as required by UnionFind data structure.

**Footnotes**

1. <https://en.wikipedia.org/wiki/Disjoint-set_data_structure> [↩︎](https://leetcode.com/problems/number-of-islands-ii/solution/#fnref1)

**Random Pick Index**

Given an array of integers with possible duplicates, randomly output the index of a given target number. You can assume that the given target number must exist in the array.

**Note:**  
The array size can be very large. Solution that uses too much extra space will not pass the judge.

**Example:**

int[] nums = new int[] {1,2,3,3,3};

Solution solution = new Solution(nums);

// pick(3) should return either index 2, 3, or 4 randomly. Each index should have equal probability of returning.

solution.pick(3);

// pick(1) should return 0. Since in the array only nums[0] is equal to 1.

solution.pick(1);

#### **Approach 1: Brute force**

**Intuition**

In this approach, we will simply store the indices of all the numbers equal to target that are present in the array and will return an index at random. The drawback of this approach is, every time the pick method is called, an extra space is required, which is dependent on the number of occurences of the number target in the given array.

**Algorithm**

Below is the implementation of the above mentioned approach.

|  |
| --- |
| class Solution {  private int[] nums;    private Random rand;    public Solution(int[] nums) {  // Do not allocate extra space for the nums array  this.nums = nums;  this.rand = new Random();  }    public int pick(int target) {  List<Integer> indices = new ArrayList<>();  int n = this.nums.length;  int count = 0;  int idx = 0;  for (int i = 0; i < n; ++i) {  if (this.nums[i] == target) {  indices.add(i);  }  }  int l = indices.size();  // pick an index at random  int randomIndex = indices.get(rand.nextInt(l));  return randomIndex;  }  } |

**Complexity Analysis**

* Time Complexity
  + pick - If N*N* represents the size of the nums array, the time complexity of pick method is O(N)*O*(*N*).
* Space Complexity: O(N)*O*(*N*) (space required by indices in pick method).

#### **Approach 2: Caching results using a hashmap**

**Intuition**

In this approach, instead of extracting all the positions in the pick method where a given target is present, we'll pre-compute it. Hence for each of the number present in the array nums, we'll store all the positions (indices) where this number occurs. This will avoid unnecessary creation of indices array in case pick method is given a target value which is repeated. Since we are storing all the indices for each of the number, hence the space requirement for this approach is huge.

**Algorithm**

Below is the implementation of the above mentioned approach.

|  |
| --- |
| class Solution {  private HashMap<Integer, List<Integer>> indices;  private Random rand;    public Solution(int[] nums) {  this.rand = new Random();  this.indices = new HashMap<Integer, List<Integer>>();  int l = nums.length;  for (int i = 0; i < l; ++i) {  if (!this.indices.containsKey(nums[i])) {  this.indices.put(nums[i], new ArrayList<>());  }  this.indices.get(nums[i]).add(i);  }  }    public int pick(int target) {  int l = indices.get(target).size();  // pick an index at random  int randomIndex = indices.get(target).get(rand.nextInt(l));  return randomIndex;  }  } |

**Complexity Analysis**

* Time Complexity
  + If N*N* represents the size of the nums array, building indices takes O(N)*O*(*N*) time.
  + pick method takes O(1)*O*(1) time.
* Space Complexity: O(N)*O*(*N*) required for indices.

#### **Approach 3: Reservoir sampling**

**Intuition**

[Reservoir sampling](https://en.wikipedia.org/wiki/Reservoir_sampling) is a technique which is used to generate numbers randomly when we have a large pool of numbers. As mentioned in the note for this question, the array size can be large, hence it is a reasonable choice to use Reservoir Sampling. Consider an array of size n*n* from which we need to chose a number randomly. Consider these numbers to be coming in the form of a stream, hence at each step, we have to take the decision of whether or not to choose a given number, such that the overall probability of each number being chosen is same (\frac{1}{n}*n*1​ in this case). If we have a total of n*n* numbers and we pick the i^{th}*ith* number, this implies that we do not pick any number further from index (i + 1)(*i*+1) to n*n*. In terms of probability, this can be represented as

\frac{1}{i} \* \frac{i}{i + 1} \* \frac{i + 1}{i + 2} ... \* \frac{n - 1}{n}*i*1​∗*i*+1*i*​∗*i*+2*i*+1​...∗*nn*−1​

This can be interpreted as

* Picking the i^{th}*ith* number from the list of i*i* numbers
* Not picking the (i + 1)^{th}(*i*+1)*th* number from the list of (i + 1)(*i*+1) numbers. Hence picking any of the remaining i*i* numbers.
* And so on ...
* Not picking the (n)^{th}(*n*)*th* number from the list of (n)(*n*) numbers. Hence picking any of the remaining (n - 1)(*n*−1) numbers.

Upon simplifying the above expression, we can see that the probability of chosing any number at the i^{th}*ith* step comes out to be \frac{1}{n}*n*1​. Hence we can say that reservoir sampling allows us to choose any number uniformly at random from the list of n*n* numbers.

Note that for any i*i*, the decision of whether or not to choose this i^{th}*ith* number depends on the first term of the above mentioned expression, which is \frac{1}{i}*i*1​. In general, if count represents the total number of numbers we have from which we need to chose a random number uniformly, we chose such a number with probability \frac{1}{count}*count*1​. This is what we will be doing here.

**Algorithm**

Below is the implementation of the above mentioned approach.

|  |
| --- |
| class Solution {  private int[] nums;  private Random rand;    public Solution(int[] nums) {  this.nums = nums;  this.rand = new Random();  }    public int pick(int target) {  int n = this.nums.length;  int count = 0;  int idx = 0;  for (int i = 0; i < n; ++i) {  // if nums[i] is equal to target, i is a potential candidate  // which needs to be chosen uniformly at random  if (this.nums[i] == target) {  // increment the count of total candidates  // available to be chosen uniformly at random  count++;  // we pick the current number with probability 1 / count (reservoir sampling)  if (rand.nextInt(count) == 0) {  idx = i;  }  }  }  return idx;  }  } |

**Complexity Analysis**

* Time Complexity
  + If N*N* represents the size of the nums array, pick method takes O(N)*O*(*N*) time
* Space Complexity: O(1)*O*(1)

**Solve the Equation**

Solve a given equation and return the value of x in the form of string "x=#value". The equation contains only '+', '-' operation, the variable x and its coefficient.

If there is no solution for the equation, return "No solution".

If there are infinite solutions for the equation, return "Infinite solutions".

If there is exactly one solution for the equation, we ensure that the value of x is an integer.

**Example 1:**

**Input:** "x+5-3+x=6+x-2"

**Output:** "x=2"

**Example 2:**

**Input:** "x=x"

**Output:** "Infinite solutions"

**Example 3:**

**Input:** "2x=x"

**Output:** "x=0"

**Example 4:**

**Input:** "2x+3x-6x=x+2"

**Output:** "x=-1"

**Example 5:**

**Input:** "x=x+2"

**Output:** "No solution"

## Solution

#### **Approach #1 Partioning Coefficients [Accepted]**

In the current approach, we start by splitting the given equation*equation* based on = sign. This way, we've separated the left and right hand side of this equation. Once this is done, we need to extract the individual elements(i.e. x's and the numbers) from both sides of the equation. To do so, we make use of breakIt function, in which we traverse over the given equation(either left hand side or right hand side), and put the separated parts into an array.

Now, the idea is as follows. We treat the given equation as if we're bringing all the x's on the left hand side and all the rest of the numbers on the right hand side as done below for an example.

x+5-3+x=6+x-2

x+x-x=6-2-5+3

Thus, every x in the left hand side of the given equation is treated as positive, while that on the right hand side is treated as negative, in the current implementation.

Likewise, every number on the left hand side is treated as negative, while that on the right hand side is treated as positive. Thus, by doing so, we obtain all the x's in the new lhs*lhs* and all the numbers in the new rhs*rhs* of the original equation.

Further, in case of an x, we also need to find its corresponding coefficients in order to evaluate the final effective coefficient of x on the left hand side. We also evaluate the final effective number on the right hand side as well.

Now, in case of a unique solution, the ratio of the effective rhs*rhs* and lhs*lhs* gives the required result. In case of infinite solutions, both the effective lhs*lhs* and rhs*rhs* turns out to be zero e.g. x+1=x+1. In case of no solution, the coefficient of x(lhs*lhs*) turns out to be zero, but the effective number on the rhs*rhs* is non-zero.

|  |
| --- |
| public class Solution {  public String coeff(String x) {  if (x.length() > 1 && x.charAt(x.length() - 2) >= '0' && x.charAt(x.length() - 2) <= '9')  return x.replace("x", "");  return x.replace("x", "1");  }  public String solveEquation(String equation) {  String[] lr = equation.split("=");  int lhs = 0, rhs = 0;  for (String x: breakIt(lr[0])) {  if (x.indexOf("x") >= 0) {  lhs += Integer.parseInt(coeff(x));  } else  rhs -= Integer.parseInt(x);  }  for (String x: breakIt(lr[1])) {  if (x.indexOf("x") >= 0)  lhs -= Integer.parseInt(coeff(x));  else  rhs += Integer.parseInt(x);  }  if (lhs == 0) {  if (rhs == 0)  return "Infinite solutions";  else  return "No solution";  }  return "x=" + rhs / lhs;  }  public List < String > breakIt(String s) {  List < String > res = new ArrayList < > ();  String r = "";  for (int i = 0; i < s.length(); i++) {  if (s.charAt(i) == '+' || s.charAt(i) == '-') {  if (r.length() > 0)  res.add(r);  r = "" + s.charAt(i);  } else  r += s.charAt(i);  }  res.add(r);  return res;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). Generating coefficients and findinn lhs*lhs* and rhs*rhs* will take O(n)*O*(*n*).
* Space complexity : O(n)*O*(*n*). ArrayList res*res* size can grow upto n*n*.

#### **Approach #2 Using regex for spliting [Accepted]**

**Algorithm**

In the last approach, we made use of a new function breakIt to obtain the individual components of either the left hand side or the right hand side. Instead of doing so, we can also make use of splitting based on + or - sign, to obtain the individual elements. The rest of the process remains the same as in the last approach.

In order to do the splitting, we make use of an expression derived from regular expressions(regex). Simply speaking, regex is a functionality used to match a target string based on some given criteria. The ?=n quantifier, in regex, matches any string that is followed by a specific string n*n*. What it's saying is that the captured match must be followed by n*n* but the n*n* itself isn't captured.

By making use of this kind of expression in the split functionality, we make sure that the partitions are obtained such that the + or - sign remains along with the parts(numbers or coefficients) even after the splitting.

|  |
| --- |
| public class Solution {  public String coeff(String x) {  if (x.length() > 1 && x.charAt(x.length() - 2) >= '0' && x.charAt(x.length() - 2) <= '9')  return x.replace("x", "");  return x.replace("x", "1");  }  public String solveEquation(String equation) {  String[] lr = equation.split("=");  int lhs = 0, rhs = 0;  for (String x: lr[0].split("(?=\\+)|(?=-)")) {  if (x.indexOf("x") >= 0) {  lhs += Integer.parseInt(coeff(x));  } else  rhs -= Integer.parseInt(x);  }  for (String x: lr[1].split("(?=\\+)|(?=-)")) {  if (x.indexOf("x") >= 0)  lhs -= Integer.parseInt(coeff(x));  else  rhs += Integer.parseInt(x);  }  if (lhs == 0) {  if (rhs == 0)  return "Infinite solutions";  else  return "No solution";  } else  return "x=" + rhs / lhs;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). Generating coefficients and finding *lhs* and *rhs* will take *O*(*n*).
* Space complexity : O(n)*O*(*n*). ArrayList res*res* size can grow upto n*n*.

**Construct Quad Tree**

Given a n \* n matrix grid of 0's and 1's only. We want to represent the grid with a Quad-Tree.

Return *the root of the Quad-Tree* representing the grid.

Notice that you can assign the value of a node to **True** or **False** when isLeaf is **False**, and both are **accepted** in the answer.

A Quad-Tree is a tree data structure in which each internal node has exactly four children. Besides, each node has two attributes:

* val: True if the node represents a grid of 1's or False if the node represents a grid of 0's.
* isLeaf: True if the node is leaf node on the tree or False if the node has the four children.

class Node {

public boolean val;

    public boolean isLeaf;

    public Node topLeft;

    public Node topRight;

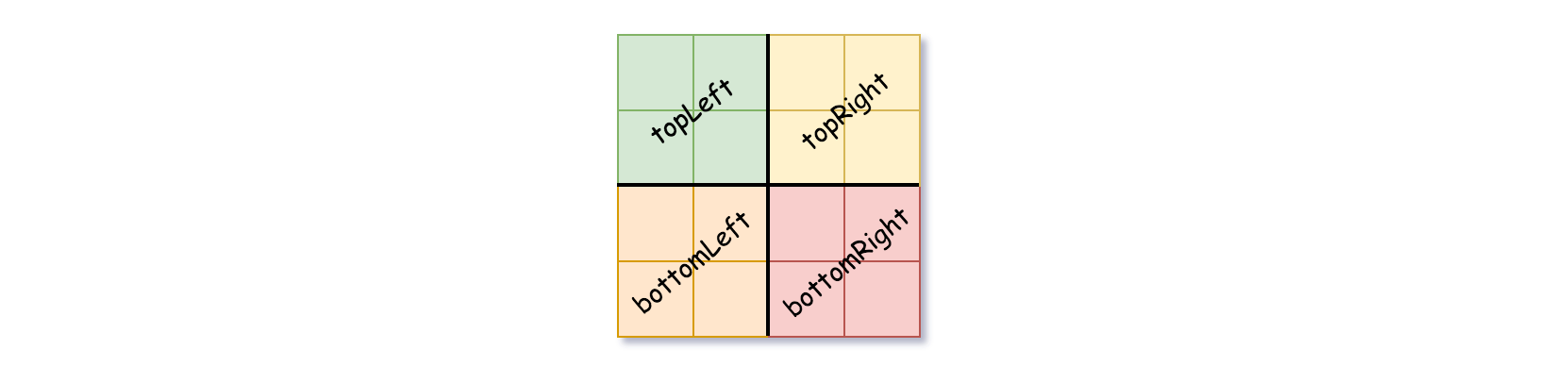
    public Node bottomLeft;

    public Node bottomRight;

}

We can construct a Quad-Tree from a two-dimensional area using the following steps:

1. If the current grid has the same value (i.e all 1's or all 0's) set isLeaf True and set val to the value of the grid and set the four children to Null and stop.
2. If the current grid has different values, set isLeaf to False and set val to any value and divide the current grid into four sub-grids as shown in the photo.
3. Recurse for each of the children with the proper sub-grid.



If you want to know more about the Quad-Tree, you can refer to the [wiki](https://en.wikipedia.org/wiki/Quadtree).

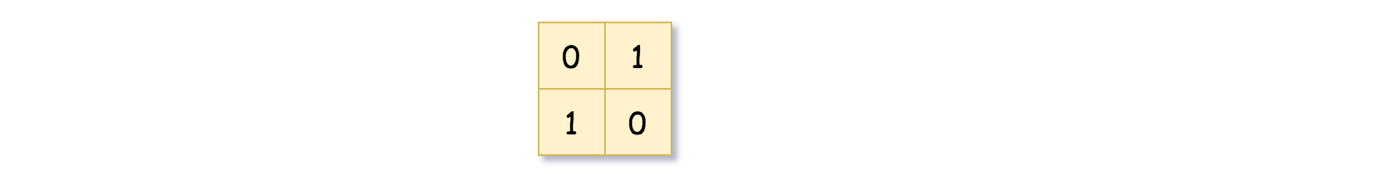
**Quad-Tree format:**

The output represents the serialized format of a Quad-Tree using level order traversal, where null signifies a path terminator where no node exists below.

It is very similar to the serialization of the binary tree. The only difference is that the node is represented as a list [isLeaf, val].

If the value of isLeaf or val is True we represent it as **1** in the list [isLeaf, val] and if the value of isLeaf or val is False we represent it as **0**.

**Example 1:**

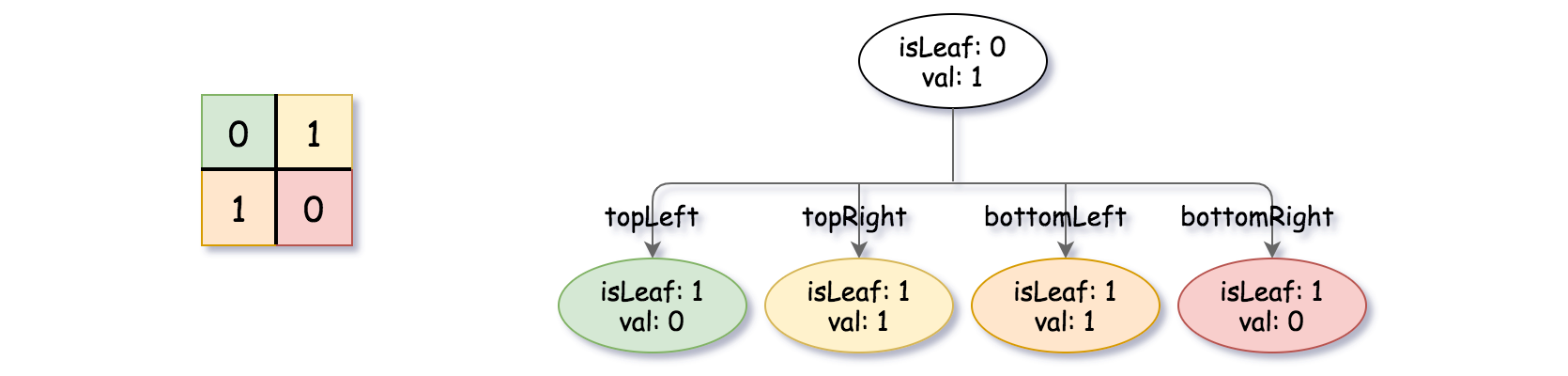


**Input:** grid = [[0,1],[1,0]]

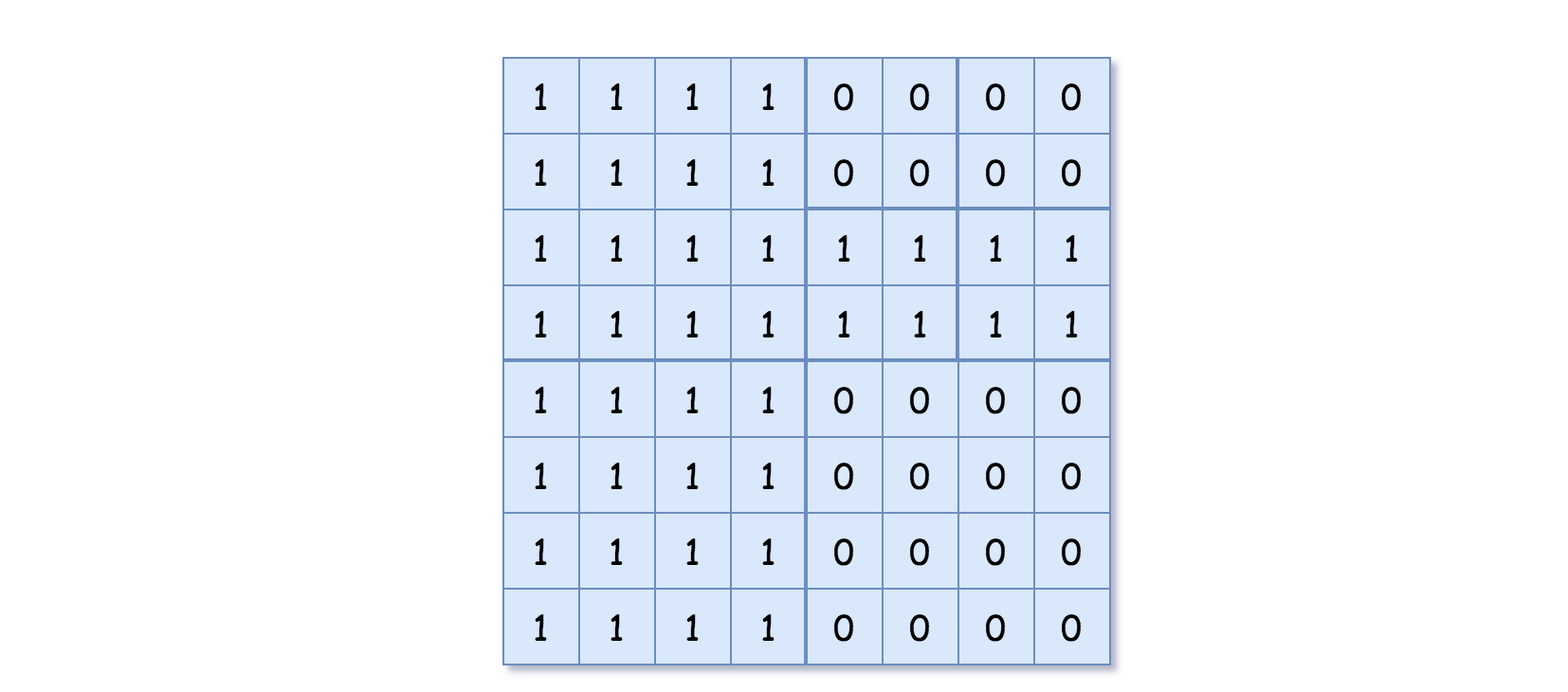
**Output:** [[0,1],[1,0],[1,1],[1,1],[1,0]]

**Explanation:** The explanation of this example is shown below:

Notice that 0 represnts False and 1 represents True in the photo representing the Quad-Tree.



**Example 2:**



**Input:** grid = [[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,0,0],[1,1,1,1,1,1,1,1],[1,1,1,1,1,1,1,1],[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,0,0]]

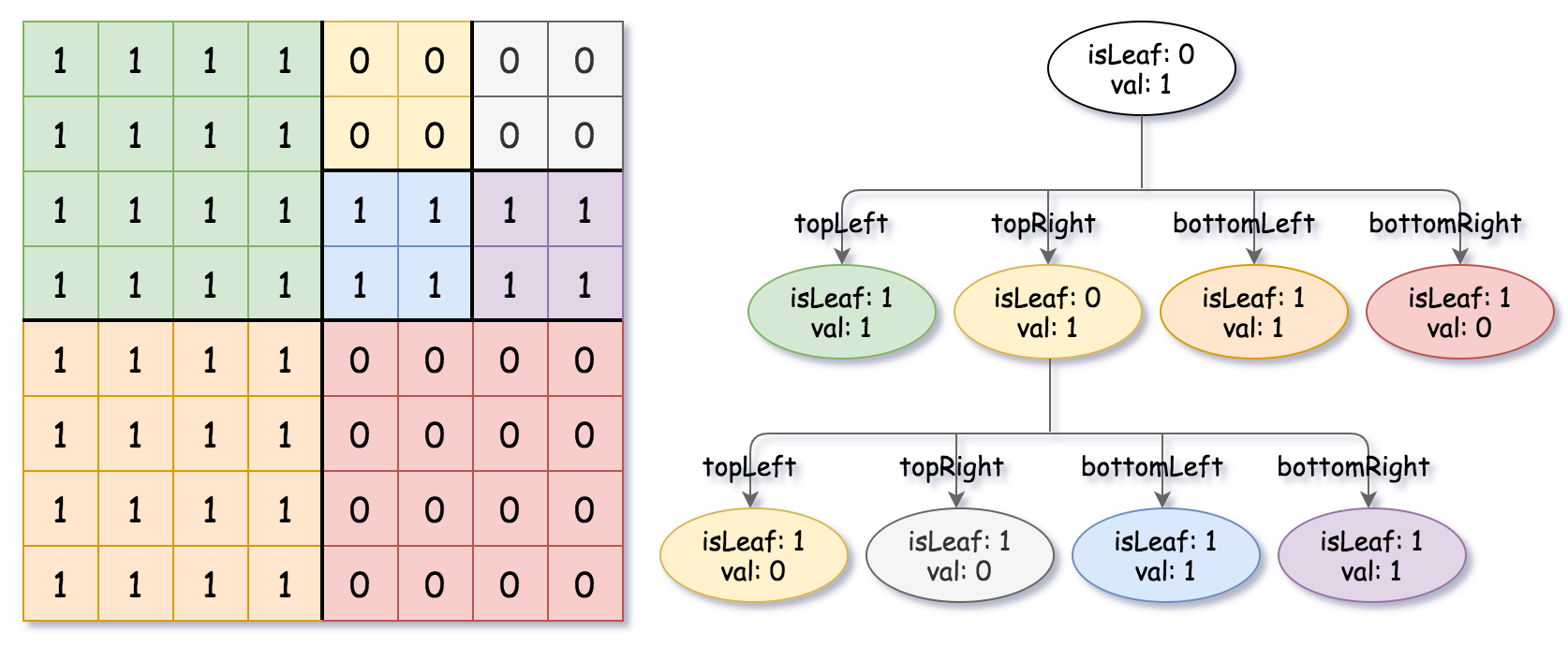
**Output:** [[0,1],[1,1],[0,1],[1,1],[1,0],null,null,null,null,[1,0],[1,0],[1,1],[1,1]]

**Explanation:** All values in the grid are not the same. We divide the grid into four sub-grids.

The topLeft, bottomLeft and bottomRight each has the same value.

The topRight have different values so we divide it into 4 sub-grids where each has the same value.

Explanation is shown in the photo below:



**Example 3:**

**Input:** grid = [[1,1],[1,1]]

**Output:** [[1,1]]

**Example 4:**

**Input:** grid = [[0]]

**Output:** [[1,0]]

**Example 5:**

**Input:** grid = [[1,1,0,0],[1,1,0,0],[0,0,1,1],[0,0,1,1]]

**Output:** [[0,1],[1,1],[1,0],[1,0],[1,1]]

**Constraints:**

* n == grid.length == grid[i].length
* n == 2^x where 0 <= x <= 6

**Random Pick with Weight**

You are given an array of positive integers w where w[i] describes the weight of ithindex (0-indexed).

We need to call the function pickIndex() which **randomly** returns an integer in the range [0, w.length - 1]. pickIndex() should return the integer proportional to its weight in the w array. For example, for w = [1, 3], the probability of picking the index 0 is 1 / (1 + 3) = 0.25 (i.e 25%) while the probability of picking the index 1 is 3 / (1 + 3) = 0.75 (i.e 75%).

More formally, the probability of picking index i is w[i] / sum(w).

**Example 1:**

**Input**

["Solution","pickIndex"]

[[[1]],[]]

**Output**

[null,0]

**Explanation**

Solution solution = new Solution([1]);

solution.pickIndex(); // return 0. Since there is only one single element on the array the only option is to return the first element.

**Example 2:**

**Input**

["Solution","pickIndex","pickIndex","pickIndex","pickIndex","pickIndex"]

[[[1,3]],[],[],[],[],[]]

**Output**

[null,1,1,1,1,0]

**Explanation**

Solution solution = new Solution([1, 3]);

solution.pickIndex(); // return 1. It's returning the second element (index = 1) that has probability of 3/4.

solution.pickIndex(); // return 1

solution.pickIndex(); // return 1

solution.pickIndex(); // return 1

solution.pickIndex(); // return 0. It's returning the first element (index = 0) that has probability of 1/4.

Since this is a randomization problem, multiple answers are allowed so the following outputs can be considered correct :

[null,1,1,1,1,0]

[null,1,1,1,1,1]

[null,1,1,1,0,0]

[null,1,1,1,0,1]

[null,1,0,1,0,0]

......

and so on.

**Constraints:**

* 1 <= w.length <= 10000
* 1 <= w[i] <= 10^5
* pickIndex will be called at most 10000 times.

## Solution

#### **Overview**

This is actually a very practical problem which appears often in the scenario where we need to do **sampling** over a set of data.

Nowadays, people talk a lot about machine learning algorithms. As many would reckon, one of the basic operations involved in training a machine learning algorithm (e.g. Decision Tree) is to sample a batch of data and feed them into the model, rather than taking the entire data set. There are several rationales behind doing sampling over data, which we will not cover in detail, since it is not the focus of this article.

If one is interested, one can refer to our Explore card of [Machine Learning 101](https://leetcode.com/explore/learn/card/machine-learning-101/) which gives an overview on the fundamental concepts of machine learning, as well as the Explore card of [Decision Tree](https://leetcode.com/explore/learn/card/decision-tree/) which explains in detail on how to construct a decision tree algorithm.

Now, given the above background, hopefully one is convinced that this is an interesting problem, and it is definitely worth solving.

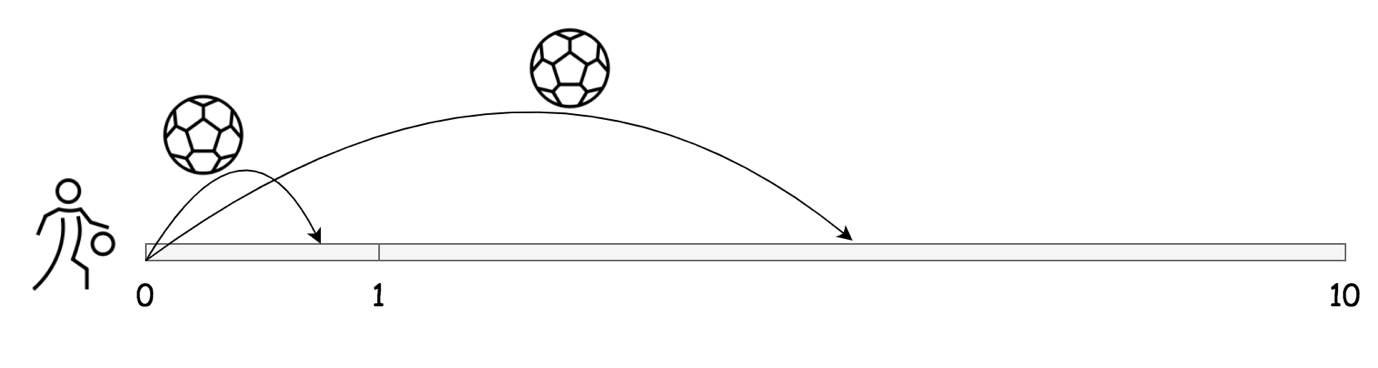
**Intuition**

Given a list of positive values, we are asked to randomly pick up a value based on the weight of each value. To put it simple, the task is to do **sampling with weight**.

Let us look at a simple example. Given an input list of values [1, 9], when we pick up a number out of it, the chance is that 9 times out of 10 we should pick the number 9 as the answer.

In other words, the **probability** that a number got picked is proportional to the value of the number, with regards to the total sum of all numbers.

To understand the problem better, let us imagine that there is a line in the space, we then project each number into the line according to its value, i.e. a large number would occupy a broader range on the line compared to a small number. For example, the range for the number 9 should be exactly nine times as the range for the number 1.



Now, let us throw a ball ***randomly*** onto the line, then it is safe to say there is a good chance that the ball will fall into the range occupied by the number 9. In fact, if we repeat this experiment for a large number of times, then statistically speaking, 9 out of 10 times the ball will fall into the range for the number 9.

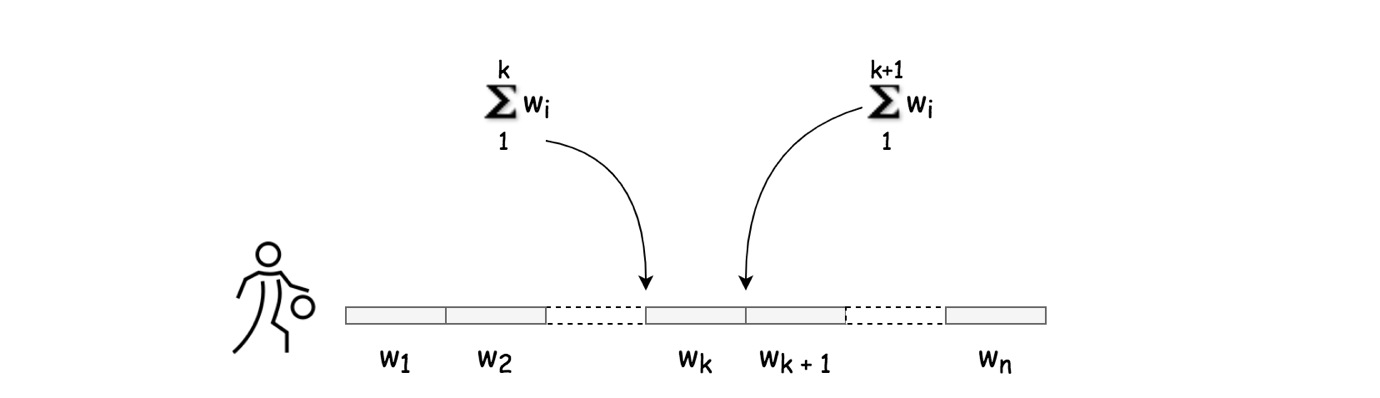
Voila. That is the intuition behind this problem.

**Simulation**

So to solve the problem, we can simply **simulate** the aforementioned experiment with a computer program.

First of all, let us construct the line in the experiment by **chaining up** all values together.

Let us denote a list of numbers as [w\_1, w\_2, w\_3, ..., w\_n][*w*1​,*w*2​,*w*3​,...,*wn*​]. Starting from the beginning of the line, we then can represent the **offsets** for each range K*K* as (\sum\_{1}^{K}{w\_i}, \sum\_{1}^{K+1}{w\_i})(∑1*K*​*wi*​,∑1*K*+1​*wi*​), as shown in the following graph:



As many of you might recognize now, the offsets of the ranges are actually the [prefix sums](https://en.wikipedia.org/wiki/Prefix_sum) from a sequence of numbers. For each number in a sequence, its corresponding prefix sum, also known as **cumulative sum**, is the sum of all previous numbers in the sequence plus the number itself.

As an observation from the definition of prefix sums, one can see that the list of prefix sums would be strictly monotonically increasing, if all numbers are positive.

To throw a ball on the line is to find an offset to place the ball. Let us call this offset **target**.

Once we randomly generate the target offset, the task is now boiled down to finding the range that this target falls into.

Let us rephrase the problem now, given a list of offsets (i.e. prefix sums) and a target offset, our task is to fit the target offset into the list so that the ascending order is maintained.

#### **Approach 1: Prefix Sums with Linear Search**

**Intuition**

If one comes across this problem during an interview, one can consider the problem almost resolved, once one reduces the original problem down to the problem of inserting an element into a sorted list.

Concerning the above problem, arguably the most intuitive solution would be **linear search**. Many of you might have already thought one step ahead, by noticing that the input list is sorted, which is a sign to apply a more advanced search algorithm called **binary search**.

Let us do one thing at one time. In this approach, we will first focus on the linear search algorithm so that we could work out other implementation details. In the next approach, we will then improve upon this approach with a binary search algorithm.

So far, there is one little detail that we haven't discussed, which is how to randomly generate a target offset for the ball. By "randomly", we should ensure that each point on the line has an equal opportunity to be the target offset for the ball.

In most of the programming languages, we have some random() function that generates a random value between 0 and 1. We can **scale up** this randomly-generated value to the entire range of the line, by multiplying it with the size of the range. At the end, we could use this scaled random value as our target offset.

As an alternative solution, sometimes one might find a randomInteger(range) function that could generate a random integer from a given range. One could then directly use the output of this function as our target offset.

Here, we adopt the random() function, since it could also work for the case where the weights are float values.

**Algorithm**

We now should have all the elements at hand for the implementation.

* First of all, before picking an index, we should first set up the playground, by generating a list of prefix sums from a given list of numbers. The best place to do so would be in the constructor of the class, so that we don't have to generate it again and again at the invocation of pickIndex() function.
  + In the constructor, we should also keep the total sum of the input numbers, so that later we could use this total sum to scale up the random number.
* For the pickIndex() function, here are the steps that we should perform.
  + Firstly, we generate a random number between 0 and 1. We then scale up this number, which will serve as our target offset.
  + We then scan through the prefix sums that we generated before by linear search, to find the first prefix sum that is larger than our target offset.
  + And the index of this prefix sum would be exactly the right place that the target should fall into. We return the index as the result of pickIndex() function.

|  |
| --- |
| class Solution {  private int[] prefixSums;  private int totalSum;  public Solution(int[] w) {  this.prefixSums = new int[w.length];  int prefixSum = 0;  for (int i = 0; i < w.length; ++i) {  prefixSum += w[i];  this.prefixSums[i] = prefixSum;  }  this.totalSum = prefixSum;  }  public int pickIndex() {  double target = this.totalSum \* Math.random();  int i = 0;  // run a linear search to find the target zone  for (; i < this.prefixSums.length; ++i) {  if (target < this.prefixSums[i])  return i;  }  // to have a return statement, though this should never happen.  return i - 1;  }  } |

**Complexity Analysis**

Let N*N* be the length of the input list.

* Time Complexity
  + For the constructor function, the time complexity would be \mathcal{O}(N)O(*N*), which is due to the construction of the prefix sums.
  + For the pickIndex() function, its time complexity would be \mathcal{O}(N)O(*N*) as well, since we did a linear search on the prefix sums.
* Space Complexity
  + For the constructor function, the space complexity would be \mathcal{O}(N)O(*N*), which is again due to the construction of the prefix sums.
  + For the pickIndex() function, its space complexity would be \mathcal{O}(1)O(1), since it uses constant memory. Note, here we consider the prefix sums that it operates on, as the input of the function.

#### **Approach 2: Prefix Sums with Binary Search**

**Intuition**

As we promised before, we could improve the above approach by replacing the linear search with the **binary search**, which then can reduce the time complexity of the pickIndex() function from \mathcal{O}(N)O(*N*) to \mathcal{O}(\log{N})O(log*N*).

As a reminder, the condition to apply binary search on a list is that the list should be sorted, either in ascending or descending order. For the list of prefix sums that we search on, this condition is guaranteed, as we discussed before.

**Algorithm**

We could base our implementation largely on the previous approach. In fact, the only place we need to modify is the pickIndex() function, where we replace the linear search with the binary search.

As a reminder, there exist built-in functions of binary search in almost all programming languages. If one comes across this problem during the interview, it might be acceptable to use any of the built-in functions.

On the other hand, the interviewers might insist on implementing a binary search by hand. It would be good to prepare for this request as well.

There are several code patterns to implement a binary search algorithm, which we cover in the Explore card of [Binary Search algorithm](https://leetcode.com/explore/learn/card/binary-search/). One can refer to the card for more details.

|  |
| --- |
| class Solution {  private int[] prefixSums;  private int totalSum;  public Solution(int[] w) {  this.prefixSums = new int[w.length];  int prefixSum = 0;  for (int i = 0; i < w.length; ++i) {  prefixSum += w[i];  this.prefixSums[i] = prefixSum;  }  this.totalSum = prefixSum;  }  public int pickIndex() {  double target = this.totalSum \* Math.random();  // run a binary search to find the target zone  int low = 0, high = this.prefixSums.length;  while (low < high) {  // better to avoid the overflow  int mid = low + (high - low) / 2;  if (target > this.prefixSums[mid])  low = mid + 1;  else  high = mid;  }  return low;  }  } |

**Complexity Analysis**

Let N*N* be the length of the input list.

* Time Complexity
  + For the constructor function, the time complexity would be \mathcal{O}(N)O(*N*), which is due to the construction of the prefix sums.
  + For the pickIndex() function, this time its time complexity would be \mathcal{O}(\log{N})O(log*N*), since we did a binary search on the prefix sums.
* Space Complexity
  + For the constructor function, the space complexity remains \mathcal{O}(N)O(*N*), which is again due to the construction of the prefix sums.
  + For the pickIndex() function, its space complexity would be \mathcal{O}(1)O(1), since it uses constant memory. Note, here we consider the prefix sums that it operates on, as the input of the function.