

People Tech – Week 1 assignment

Swetha Kare

Learnings for Array Data Structure

Arrays allow sequential storage of data into the memory and it helps to add elements in contiguous memory locations;

Insertion into arrays is easy if it is added at the end of the current element; if not then we need to shift the elements from their original position till the end which is challenging

Deletion based on index creates a challenge because if we delete an element then we will have to have the positions shift forward as we cannot have any empty places in the array.

Arrays can be both static and dynamic; dynamic arrays are lists in java they keep growing, shrinking as the elements get added or removed. Static arrays have a limited size and adding elements beyond that predefined size is not possible.

Time complexity:

Inserting an element at the end : $O(1)$

Inserting an element at the beginning or in the middle : $O(n)$.

Deleting an element from the end: $O(1)$

Deleting an element from the beginning or in the middle: $O(n)$.

Traversing an array takes $O(n)$

Searching an element in an array takes $O(n)$

Searching an element in binary search takes $O(\log n)$

Space complexity of the array is $O(n)$ it is because the array is allotted with memory even if we use it or not.

If index is known then finding an element in an array is very simple and take $O(1)$ time complexity/

Examples of Array Data Structure used in the real world.

In gaming arrays could be used as maps or grids.

In storing elements one after the other and performing sort operations it is easy to use arrays

Learning for LinkedList Data Structure:

A linked list is a linear data structure where elements (nodes) are connected by pointers. Each node contains data and a reference to the next node. There are different types, including singly linked lists

(single direction traversal), doubly linked lists (bi-directional traversal), and circular linked lists (the last node points back to the first).

Basic linked list operations include:

- Creation: a new node is created and other nodes are linked to it, all the nodes store reference to the next nodes.
- Insertion: Adding a node at the start is $O(1)$ time complexity, to add nodes in the middle or in the end we need to traverse the list and the worst case time complexity to add a node in the middle or end is $O(n)$.
- Deletion: Removing a node from the front is $O(1)$ time complexity while removing the node from the end and at a specified position could be $O(n)$ time complexity.
- Traversal: to obtain a node from the list we need to perform traversal; the time complexity of a traversal operation could be $O(n)$

Comparing arrays and linked lists:

- Arrays allow quick random access ($O(1)$), but insertions and deletions require shifting elements ($O(n)$), and resizing an array requires copying data to a new memory block.
- Linked lists dynamically grow or shrink in memory and are more efficient for frequent insertions and deletions ($O(1)$ at the head), though accessing elements requires traversing the list ($O(n)$).
- 3. Arrays do not occupy a lot of memory whereas linked lists occupy a lot of memory compared to the arrays because they will be storing the next nodes reference.

Hands-On Practice:

[Explore](#)
[Problems](#)
[Contest](#)
[Discuss](#)
[Interview](#)
[Store](#)

Description

Solution

Discuss (999+)

Submissions

i Java

Autocomplete

1. Two Sum

Easy 7925 349 Add to List Share

Given an array `nums`. We define a running sum of an array as `runningSum[i] = sum(nums[0]...nums[i])`.

Return the running sum of `nums`.

Example 1:

Input: `nums = [1,2,3,4]`
Output: `[1,3,6,10]`
Explanation: Running sum is obtained as follows: `[1, 1+2, 1+2+3, 1+2+3+4]`.

Example 2:

Input: `nums = [1,1,1,1,1]`
Output: `[1,2,3,4,5]`
Explanation: Running sum is obtained as follows: `[1, 1+1, 1+1+1, 1+1+1+1, 1+1+1+1+1]`.

Example 3:

```

1 class Solution {
2     public int[] runningSum(int[] nums) {
3
4         int[] prefix = new int[nums.length];
5         prefix[0] = nums[0];
6         for(int i = 1; i < prefix.length; i++){
7             prefix[i] = prefix[i-1] + nums[i];
8         }
9         return prefix;
10    }
11 }

```

Testcase

Run Code Result

Debugger

Accepted

Runtime: 0 ms

Your input

[1,2,3,4]

Output

[1,3,6,10]

Expected

[1,3,6,10]

$O(n)$ time complexity; $O(n)$ space complexity. This is a very simple prefix sum approach

[Explore](#)
[Problems](#)
[Contest](#)
[Discuss](#)
[Interview](#)
[Store](#)

Description

Solution

Discuss (999+)

Submissions

i Java

Autocomplete

1. Two Sum

Easy 58013 2051 Add to List Share

Given an array of integers `nums` and an integer `target`, return *indices* of the two numbers such that they add up to `target`.

You may assume that each input would have **exactly one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

Input: `nums = [2,7,11,15]`, `target = 9`
Output: `[0,1]`
Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

Example 2:

Input: `nums = [3,2,4]`, `target = 6`
Output: `[1,2]`

```

1 class Solution {
2     public int[] twoSum(int[] nums, int target) {
3         int[] temp = new int[2];
4         int sum = 0;
5         Map<Integer,Integer> mp = new HashMap<>();
6         for(int i = 0; i < nums.length; i++){
7             sum = target - nums[i];
8             if(mp.containsKey(sum)){
9                 temp[0] = mp.get(sum);
10                temp[1] = i;
11                return temp;
12            }
13            mp.put(nums[i], i);
14        }
15        return temp;
16    }
17 }
18

```

used a hashmap to find two numbers in an array that add up to a target. It iterates over the array, checking if the complement (target - current number) exists in the hashmap. If it does, it returns the indices of the two numbers; otherwise, it stores the current number and its index in the hashmap.

$O(n)$ time and space complexity.

Explore Problems Contest Discuss Interview Store

26. Remove Duplicates from Sorted Array

Easy 15068 18943 Add to List Share

Given an integer array `nums` sorted in **non-decreasing order**, remove the duplicates **in-place** such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in `nums`*.

Consider the number of unique elements of `nums` to be `k`, to get accepted, you need to do the following things:

- Change the array `nums` such that the first `k` elements of `nums` contain the unique elements in the order they were present in `nums` initially. The remaining elements of `nums` are not important as well as the size of `nums`.
- Return `k`.

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int[] expectedNums = [...]; // The expected answer with correct length
```

```
1 class Solution {
2     public int removeDuplicates(int[] nums) {
3         int len = nums.length;
4         ArrayList<Integer> ar = new ArrayList<>();
5         for(int i = 0; i < nums.length; i++){
6             if(!ar.contains(nums[i])){
7                 ar.add(nums[i]);
8             }
9         }
10        for(int i = 0; i < ar.size(); i++){
11            nums[i] = ar.get(i);
12        }
13        return ar.size();
14    }
15 }
```

removing duplicates from a sorted array by using an ArrayList to store unique elements. It iterates through the array, checks if each element is already in the ArrayList, and adds it if it's not. After processing, it copies the unique elements back into the original array.

time complexity is $O(n^2)$ because it has `ar.contains()` which will check n times for each elements of the n sized array

Space complexity is $O(n)$.

Explore Problems Contest Discuss Interview Store

66. Plus One

Easy 9533 5417 Add to List Share

You are given a **large integer** represented as an integer array `digits`, where each `digits[i]` is the i^{th} digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return *the resulting array of digits*.

Example 1:

```
Input: digits = [1,2,3]
Output: [1,2,4]
Explanation: The array represents the integer 123.
Incrementing by one gives 123 + 1 = 124.
Thus, the result should be [1,2,4].
```

```
1 class Solution {
2     public int[] plusOne(int[] digits) {
3         for(int i = digits.length-1; i >= 0; i--){
4             if(digits[i] != 9){
5                 digits[i]++;
6                 return digits;
7             }
8         }
9         int newdigit[] = new int[digits.length+1];
10        newdigit[0] = 1;
11        return newdigit;
12    }
13 }
```

1. incrementing a number represented by an array of digits. It starts from the last digit, adding 1 to it. If the digit is 9, it sets it to 0 and moves to the next digit. If all digits are 9 (e.g., 999), a new

array is created with an additional digit to handle the carry (e.g., 1000).

Time Complexity and space complexity is $O(n)$.

Learning for Stacks:

Stacks are for LIFO operations; can be implemented with any data structure

A stack is a data structure that follows the Last In, First Out (LIFO) principle, meaning the most recently added item is the first to be removed. It operates with key operations such as push, pop, peek, isEmpty, and Size.

Stacks can be implemented using arrays, linked lists, deques, or queues. With arrays, the stack's size may be fixed or dynamic, with simple and fast access but potential resizing overhead. Linked lists offer dynamic sizing with efficient insertions and deletions, though they require extra memory for node references. Deques provide optimized operations for adding and removing from one end, while two queues can simulate stack behavior with more complex operations.

The time complexity for stack operations is generally $O(1)$, meaning they are performed in constant time. The space complexity is $O(n)$, where n is the number of elements, reflecting the memory needed to store the stack's items.

Stacks are used in various applications, including managing function calls, evaluating expressions, handling undo/redo operations, backtracking algorithms, and checking balanced parentheses. They are simple to implement and efficient for many problems but have limited direct access to elements and can involve memory overhead depending on the implementation.

Learning for Queues:

A queue is a data structure that follows the First In, First Out principle, meaning the first item added is the first to be removed. It operates with key operations such as enqueue to add an item to the back, dequeue to remove an item from the front, peek to view the front item without removing it, isEmpty to check if the queue is empty, and Size to return the number of items.

Queues can be implemented using arrays, linked lists, or deques. With arrays, the queue's size may be fixed or dynamic, offering simple and fast access but potentially requiring resizing. Linked lists provide dynamic sizing with efficient insertions and deletions at both ends, though they require extra memory for node references. Deques, which allow additions and removals from both ends, can be used to implement queues with optimized operations.

The time complexity for queue operations is generally $O(1)$, meaning they are performed in constant time. The space complexity is $O(n)$, where n is the number of elements, reflecting the memory needed to store the queue's items.

Queues are used in various applications, including task scheduling, managing requests in web servers, handling print jobs, and implementing breadth-first search in algorithms. They are simple to implement and efficient for managing ordered data but can involve memory overhead depending on the implementation.

Learning for Hash Tables:

A hash table is a data structure that provides fast access to data using a mechanism known as hashing. It works by mapping keys to values via a hash function, which computes an index into an array where the value is stored. The key operations are insertion, where a key-value pair is added; deletion, where a key-value pair is removed; and retrieval, where a value is accessed using a key.

Hash tables can be implemented using arrays and various collision resolution techniques, such as chaining (where each array index points to a linked list of entries) or open addressing (where colliding elements are placed in the next available slot according to a probing sequence). Chaining allows hash tables to handle collisions more flexibly, while open addressing often provides better cache performance but requires careful handling of collisions and deletions.

The time complexity for average-case operations such as insertion, deletion, and retrieval is generally $O(1)$, meaning they are performed in constant time, assuming a good hash function and minimal collisions. However, in the worst case, due to collisions, operations can degrade to $O(n)$, where n is the number of elements.

The space complexity is $O(n)$, where n is the number of elements, reflecting the memory needed to store the hash table's entries.

Hash tables are widely used in various applications, including implementing associative arrays, database indexing, caching, and sets. They offer efficient access and management of data but can suffer from issues related to hash function quality and collision handling.

Practiced a few easy level problems on leetcode for stacks, queues, hashtables, Linkedlist.

Description
Solution
Discuss (999+)
Submissions
Java
Autocomplete
i {}

141. Linked List Cycle

Easy 15700 1401 Add to List Share

Given `head`, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to. **Note that `pos` is not passed as a parameter.**

Return `true` if there is a cycle in the linked list. Otherwise, return `false`.

Example 1:

```

1  /**
2   * Definition for singly-linked list.
3   * class ListNode {
4   *     int val;
5   *     ListNode next;
6   *     ListNode(int x) {
7   *         val = x;
8   *         next = null;
9   *     }
10  */
11
12  public class Solution {
13  public boolean hasCycle(ListNode head) {
14      ListNode slow = head;
15      ListNode fast = head;
16      if(head == null){
17          return false;
18      }
19      while(fast.next != null && fast.next.next != null){
20          fast = fast.next.next;
21          slow = slow.next;
22          if(fast == slow){
23              return true;
24          }
25      }
26      return false;
27  }
28  }
29  
```

Description
Solution
Discuss (999+)
Submissions
Java
Autocomplete
i

205. Isomorphic Strings

Easy 9156 2125 Add to List Share

Given two strings `s` and `t`, determine if they are isomorphic.

Two strings `s` and `t` are isomorphic if the characters in `s` can be replaced to get `t`.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

Example 1:

Input: `s = "egg", t = "add"`

Output: `true`

Explanation:

The strings `s` and `t` can be made identical by:

- Mapping `'e'` to `'a'`.
- Mapping `'g'` to `'d'`.

```

1  import java.util.HashMap;
2
3  class Solution {
4  public boolean isIsomorphic(String s, String t) {
5      HashMap<Character, Character> mp = new HashMap<>();
6      if (s.length() != t.length()) {
7          return false;
8      }
9      for (int i = 0; i < s.length(); i++) {
10         char str = s.charAt(i);
11         char ttr = t.charAt(i);
12         if (mp.containsKey(str)) {
13             char item = mp.get(str);
14             if (item != ttr) {
15                 return false;
16             }
17         } else if (mp.containsValue(ttr)) {
18             for (char key : mp.keySet()) {
19                 if (mp.get(key) == ttr) {
20                     return false;
21                 }
22             }
23         } else {
24             mp.put(str, ttr);
25         }
26     }
27     return true;
28 }
29 }
30

```

Your previous code was restored from your local storage. [Reset to default](#)

Explore
Problems
Contest
Discuss
Interview
Store

0

Description
Solution
Discuss (999+)
Submissions
Java
Autocomplete

20. Valid Parentheses

Easy 24362 1805 Add to List Share

Given a string `s` containing just the characters `'('`, `'>`, `'{'`, `'>`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.
- Every close bracket has a corresponding open bracket of the same type.

Example 1:

Input: `s = "()"`

Output: `true`

Example 2:

Input: `s = "()[]{}"`

Output: `true`

```

1  class Solution {
2  public boolean isValid(String s) {
3      Stack<Character> st = new Stack<>();
4      for(int i = 0; i < s.length(); i++){
5          char c = s.charAt(i);
6          if(c == '(' || c == '{' || c == '['){
7              st.push(c);
8          }else{
9              if(st.isEmpty()){
10                 return false;
11             }
12             char top = st.pop();
13             if(c == ')' && top != '(' || c == '}' && top != '{' || c == ']' && top != '['){
14                 return false;
15             }
16             }
17         }
18     }
19     return st.isEmpty() ? true : false;
20 }
21

```

Algorithms:

Insertion Sort is a straightforward sorting algorithm that works by gradually building up a sorted portion of the array. It starts with the first element and iteratively inserts each subsequent element into its correct position within the sorted portion. The algorithm involves comparing the current element to those in the sorted section and shifting elements as needed to make space for the new element. Its time complexity is $O(n^2)$ in average and worst cases, making it less efficient for large datasets, but it performs well for small or nearly sorted arrays. Its space complexity is $O(1)$, as it sorts in place without requiring extra space beyond the input array. Insertion Sort is often used in applications where the dataset is small or partially sorted, such as in simple database sorting or as part of more complex algorithms that handle small partitions of data.

Merge Sort is a more advanced sorting algorithm that follows a divide-and-conquer approach. It divides the array into smaller subarrays, recursively sorts each subarray, and then merges them back together in a sorted manner. This method ensures that the entire array is sorted efficiently. Merge Sort has a time complexity of $O(n \log n)$ across average, worst, and best cases, making it much more suitable for large datasets compared to Insertion Sort. However, it requires additional space proportional to the size of the input array, giving it a space complexity of $O(n)$. Merge Sort is particularly useful for sorting large volumes of data, such as in data processing applications, and is also well-suited for linked lists and external sorting where data is too large to fit into memory all at once.

Applications

Insertion Sort:

- Small Datasets:** Ideal for small arrays or lists where its simplicity and in-place sorting are advantageous.

Merge Sort:

- **Large Datasets:** Efficient for sorting large volumes of data, such as in database management systems or large-scale data processing applications.

In summary, Insertion Sort is efficient for small or nearly sorted data, while Merge Sort is better suited for large datasets and scenarios requiring stable sorting.

Description
Solution
Discuss (999+)
Submissions

21. Merge Two Sorted Lists

Easy 22074 2157 Add to List Share

You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.

Example 1:

```

1  class Solution {
2      public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
3          ListNode newlist = new ListNode(-1);
4          ListNode savelist = newlist;
5          ListNode temp1 = new ListNode(-1);
6
7          while(list1 != null && list2 != null){
8              if(list1.val >= list2.val){
9                  newlist.next = list1;
10                 newlist = newlist.next;
11                 list1 = list1.next;
12             }else{
13                 newlist.next = list2;
14                 newlist = newlist.next;
15                 list2 = list2.next;
16             }
17         }
18         if(list1 != null){
19             newlist.next = list1;
20         }
21         if(list2 != null){
22             newlist.next = list2;
23         }
24         return savelist.next;
25     }
26 }

```

Your previous code was restored from your local storage. [Reset to default](#)

Dijkstra's Algorithm finds the shortest path from a source vertex to all other vertices in a graph with non-negative edge weights. It uses a priority queue to explore the closest vertex, updating the shortest known distances as it proceeds. This algorithm is efficient with a time complexity of $O((V + E) \log V)$, where V is the number of vertices and E is the number of edges.

Dijkstra Algorithm

Difficulty: Medium Accuracy: 50.83% Submissions: 179K+ Points: 4

Given a weighted, undirected and connected graph of V vertices and an adjacency list `adj` where `adj[i]` is a list of lists containing two integers where the **first** integer of each list `j` denotes there is **edge** between `i` and `j`, second integers corresponds to the **weight** of that edge. You are given the source vertex `S` and You to Find the shortest distance of all the vertex's from the source vertex `S`. You have to return a list of integers denoting shortest distance between **each node** and Source vertex `S`.

Note: The Graph doesn't contain any negative weight cycle.

The structure of adjacency list is as follows :-

For Example : `adj = { {1, 1}, {2, 6} } . {2, 3}, {0, 1} } . {1, 3}, {0, 6} }`

Here `adj[i]` contains a list which contains all the nodes which are connected to the `i`th vertex. Here `adj[0] = {{1, 1}, {2, 6}}` means there are **two** nodes connected to the `0`th node, **node 1** with an **edge weight** of **1** and **node 2** with an **edge weight** of **6** and similarly for other vertices.

```

60 int dist, node;
61 item(int dist, int node) {
62     this.dist = dist;
63     this.node = node;
64 }
65
66
67 class Solution {
68     static int[] dijkstra(int V, ArrayList<ArrayList<Integer>>> a,
69         PriorityQueue<item> q = new PriorityQueue<>((a, b) -> Integer.compare(a.dist, b.dist)),
70         int[] dist = new int[V];
71         Arrays.fill(dist, Integer.MAX_VALUE);
72         dist[S] = 0;
73         int[] parent = new int[V];
74         parent[S] = 0;
75
76         while (!q.isEmpty()) {
77             item it = q.poll();
78             int par = it.node;
79             for (ArrayList<Integer> ite : adj.get(par)) {
80                 if (it.dist + ite.get(1) < dist[ite.get(0)]) {
81                     dist[ite.get(0)] = it.dist + ite.get(1);
82                     parent[ite.get(0)] = par;
83                     q.add(new item(it.dist + ite.get(1), ite.get(0)));
84                 }
85             }
86         }
87         return dist;
88     }
89 }

```

Bellman-Ford Algorithm computes the shortest paths from a source vertex to all other vertices in a graph, handling negative edge weights. It works by repeatedly relaxing all edges and can detect negative-weight cycles. The time complexity is $O(VE)$, making it less efficient than Dijkstra's for graphs with many

edges.

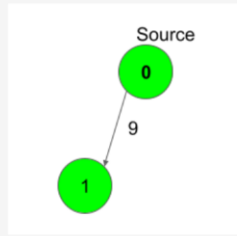
Distance from the Source (Bellman-Ford Algorithm)

Difficulty: Medium Accuracy: 48.11% Submissions: 135K+ Points: 4

Given a weighted and directed graph of V vertices and E edges, Find the shortest distance of all the vertex's from the source vertex S . If a vertex can't be reached from the S then mark the distance as 10^8 . Note: If the Graph contains a negative cycle then return an array consisting of only -1 .

Example 1:

Input:



```
55 int cost;
56 item(int node, int cost){
57     this.node = node;
58     this.cost = cost;
59 }
60 }
61 class Solution {
62     static int[] bellman_ford(int V, ArrayList<ArrayList<Integer>> edges, int S) {
63         int[] dis = new int[V];
64         Arrays.fill(dis, 100000000);
65         dis[S] = 0;
66         for(int i = 0; i < V-1; i++){
67             for(ArrayList<Integer> ar : edges){
68                 int u = ar.get(0);
69                 int v = ar.get(1);
70                 int wt = ar.get(2);
71                 if(dis[u] != 100000000 && dis[u] + wt < dis[v]){
72                     dis[v] = dis[u] + wt;
73                 }
74             }
75         }
76         for(ArrayList<Integer> ar : edges){
77             int u = ar.get(0);
78             int v = ar.get(1);
79             int wt = ar.get(2);
80             if(dis[u] != 100000000 && dis[u] + wt < dis[v]){
81                 int[] temp = new int[V];
82                 temp[0] = -1;
83                 return temp;
84             }
85         }
86         return dis;
87     }
88 }
```

Breadth-First Search (BFS) explores a graph level by level, starting from a source node and visiting all its neighbors before moving to the next level. It is used for finding the shortest path in an unweighted graph and has a time complexity of $O(V + E)$.

Description

Solution

Discuss (999+)

Submissions

733. Flood Fill

Easy 8515 886 Add to List Share

An image is represented by an $m \times n$ integer grid `image` where `image[i][j]` represents the pixel value of the image.

You are also given three integers `sr`, `sc`, and `color`. You should perform a **flood fill** on the image starting from the pixel `image[sr][sc]`.

To perform a **flood fill**, consider the starting pixel, plus any pixels connected **4-directionally** to the starting pixel of the same color as the starting pixel, plus any pixels connected **4-directionally** to those pixels (also with the same color), and so on. Replace the color of all of the aforementioned pixels with `color`.

Return the modified image after performing the flood fill.

Example 1:

Input: `image = [[1,1,1],[1,1,0],[1,0,1]]`, `sr = 1`, `sc = 1`, `color = 2`

Output: `[[2,2,2],[2,2,0],[2,0,1]]`

Explanation:

```
15 for (int j = 0; j < image[0].length; j++) {
16     nimage[i][j] = image[i][j];
17 }
18
19
20 int inicolor = image[sr][sc];
21 int[] dx = {-1, 0, +1, 0};
22 int[] dy = {0, -1, 0, +1};
23 Queue<item> q = new LinkedList<>();
24 q.add(new item(sr, sc));
25 nimage[sr][sc] = color;
26
27 while (!q.isEmpty()) {
28     item it = q.poll();
29     int row = it.x;
30     int col = it.y;
31     for (int i = 0; i < 4; i++) {
32         int nrow = row + dx[i];
33         int ncol = col + dy[i];
34         if (nrow >= 0 && nrow < image.length && ncol >= 0 && ncol <
35             image[0].length) {
36             if (color != nimage[nrow][ncol] && nimage[nrow][ncol] == inicolor)
37                 q.add(new item(nrow, ncol));
38             nimage[nrow][ncol] = color;
39         }
40     }
41 }
42 return nimage;
43 }
44 }
```

Depth-First Search (DFS) explores as far as possible along each branch before backtracking. It is used for tasks such as pathfinding, cycle detection, and topological sorting. Its time complexity is $O(V + E)$, and it

can be implemented using recursion or a stack.

Explore

Problems

Contest

Discuss

Interview

Store

547. Number of Provinces

Medium

9863

369

Add to List

Share


There are n cities. Some of them are connected, while some are not. If city a is connected directly with city b , and city b is connected directly with city c , then city a is connected indirectly with city c .

A **province** is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an $n \times n$ matrix `isConnected` where `isConnected[i][j] = 1` if the i^{th} city and the j^{th} city are directly connected, and `isConnected[i][j] = 0` otherwise.

Return the total number of **provinces**.

Example 1:



Java

Autocomplete

```
8  for (int i = 0; i < isConnected.length; i++) {
9      adjacencyList.add(new ArrayList<Integer>());
10 }
11
12 // Populate the adjacency list
13 for (int i = 0; i < isConnected.length; i++) {
14     for (int j = i + 1; j < isConnected[i].length; j++) {
15         if (isConnected[i][j] == 1 && i != j) {
16             adjacencyList.get(i).add(j);
17             adjacencyList.get(j).add(i);
18         }
19     }
20 }
21 int[] visited = new int[isConnected.length];
22 int cnt = 0;
23
24 // Perform DFS for each unvisited node
25 for (int i = 0; i < isConnected.length; i++) {
26     if (visited[i] == 0) {
27         cnt++;
28         dfs(adjacencyList, visited, i);
29     }
30 }
31 return cnt;
32
33 public void dfs(ArrayList<ArrayList<Integer>> adj, int[] vis, int node) {
34     vis[node] = 1;
35     for (int it : adj.get(node)) {
36         if (vis[it] == 0) {
37             dfs(adj, vis, it);
38         }
39     }
```

Fractional Knapsack Problem is a variation of the knapsack problem where items can be divided into smaller parts. It involves maximizing the total value in a knapsack with a weight limit, allowing fractional amounts of items to be included. It is solved using a greedy algorithm, making it efficient for finding optimal solutions when item fractions are allowed.

Fractional Knapsack

Difficulty: Medium Accuracy: 32.46% Submissions: 275K+ Points: 4

Given weights and values of n items, we need to put these items in a knapsack of capacity w to get the **maximum** total value in the knapsack. Return a double value representing the maximum value in knapsack.

Note: Unlike 0/1 knapsack, you are **allowed** to break the item here. The details of structure/class is defined in the comments above the given function.

Examples:

Input: $n = 3, w = 50, \text{value}[] = [60, 100, 120], \text{weight}[] = [10, 20, 30]$

Output: 240.000000

Explanation: Take the item with value 60 and weight 10, value 100 and weight 20 and split the third item with value 120 and weight 30, to fit it into weight 20. so it becomes $(120/30)*20=80$, so the total value becomes $60+100+80=240.0$ Thus, total maximum value of item we can have is 240.00 from the given capacity of sack.

```
50     else if (r1 > r2) return -1;
51     else return 0;
52 }
53 }
54
55 class Solution
56 {
57     //Function to get the maximum total value in the knapsack.
58     double fractionalKnapsack(int W, Item arr[], int n)
59     {
60         // Your code here
61         weightsort ac = new weightsort();
62         Arrays.sort(arr,ac);
63         int curWeight = 0;
64         double finalvalue = 0.0;
65
66         for (int i = 0; i < n; i++) {
67             if (curWeight + arr[i].weight <= W) {
68                 curWeight += arr[i].weight;
69                 finalvalue += arr[i].value;
70             }
71             else {
72                 int remain = W - curWeight;
73                 finalvalue += ((double)arr[i].value / (double)arr[i].weight
74                     * remain);
75                 break;
76             }
77         }
78         return finalvalue;
79     }
80 }
81
82 //return val;
```