Untitled

19 December 2025 18:47

Phase 1: The Anatomy (The "What")

1. **Introduction & Representation:** Learn Adjacency Matrix vs. Adjacency List.
2. **DFS Basics:** Learn how to find a single path between two nodes.
3. **All Paths:** Understand how to use backtracking to print every possible route.

Phase 2: Global Properties (The "Where")

4. **Connected Components:** The key to "Islands" and "Perfect Friends." This teaches you how to iterate over an unvisited graph.
5. **Is Graph Connected:** A simple check once you understand components.
6. **Number of Islands:** The classic 2D grid application of DFS.

Phase 3: The "Level-Order" Shift

7. **Breadth First Search (BFS):** Shift from "going deep" to "going wide."
8. **Dijkstra's Algorithm:** This is essentially "BFS on steroids" for weighted graphs.
9. **Spread Infection:** A real-world application of BFS (calculating time/levels).

Phase 4: Structure & Logic

10. **Cyclic vs. Bipartite:** Learning the constraints of a graph.
11. **Topological Sort:** Crucial for dependency problems (like Course Schedule).
12. **Prim's Algorithm:** How to connect everything with the minimum cost.

Phase 5: Exhaustive Search (The Hard Stuff)

13. **Hamiltonian Path & Cycles:** Visiting every node exactly once.
14. **Knight's Tour:** A specialized recursion problem using graph logic.
15. **Iterative DFS:** Mastering the stack manually to avoid StackOverflow errors.

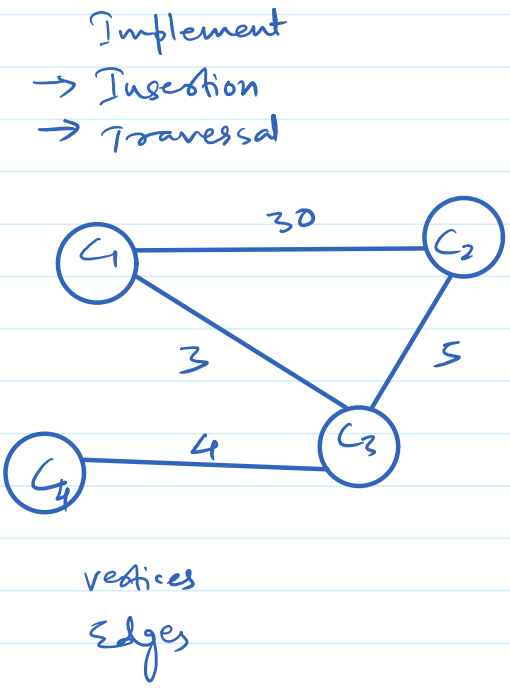
---

13 April 2022 14:36

How to make graph?

It has got
i) Vertices
ii) Edges

Edges can be weighted.
Cities and roads, or network based information.



What all do we find/calculate in Graphs? What all are we going to learn?

Q 1) Are two vertices connected? (Does there exist a path between two far vertices?)
Q 2) Print all the paths
Q 3) smallest path in terms of no of edges(BFS)
Q 4) smallest path in terms of weight? (dijkstra's)
Q 5) Minimum spanning tree (prims and kruskal's)
Q 6) Topological sort (in directed graphs)

But before that, we need to learn how to make graphs to represent maps.

Section 1: Create graphs
Section 2: Learn DFS
Section 3:

Section 1: Create Graph

16 August 2022 10:08

Tip:
Coding is all about understanding what that line means in English.

Same as mathematics, most of the things becomes easy when you start understanding things of new language in the language you're comfortable in.

So it's easy if you take it slowly. Learning is a slow process, and we must enjoy it.

Create Graph

13 April 2022 15:01

Ways to represent:

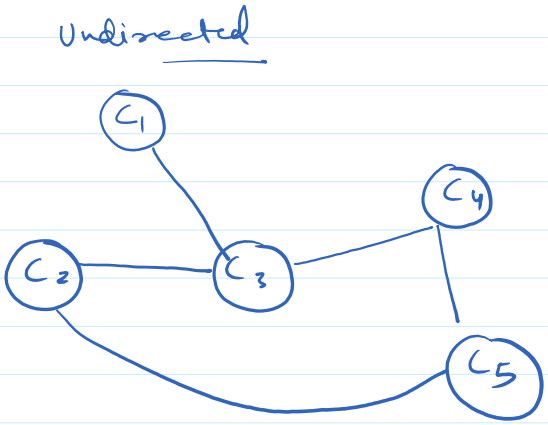
1. Adjacency matrix

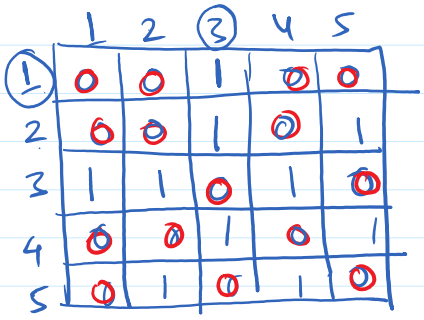
Make a 2-d matrix, rows and col index denotes the vertices and the corresponding arr[r][c] denotes the edge, and you can store the weights over there.

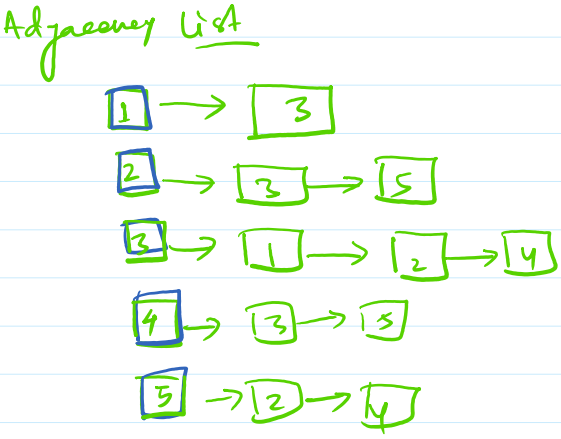
When no of vertices < 10000, then only we can make an array.

2. Adjacency list(popular method)

Array list of edges
No of lists will be equal to no of vertices.



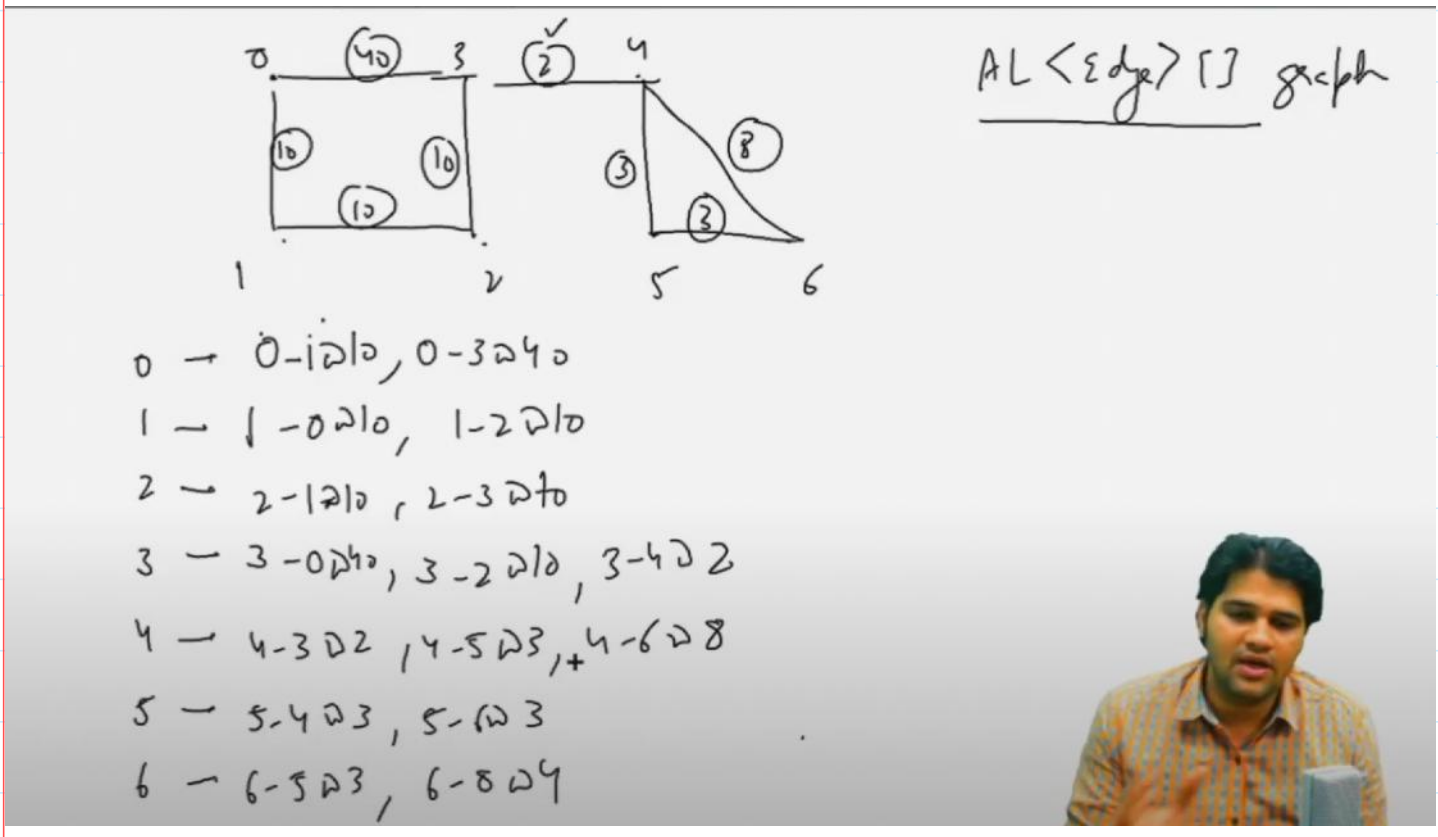




0-1@10, 0-3@40

1-0@10, 1-2@10

Source - Neighbour @ weight



---

1. The Analogy: A Cabinet of Folders

Imagine you are building a filing system for 7 people (your vertices).

- **The Array (`[]`):** This is the **Cabinet**. It has exactly 7 drawers, numbered 0 to 6. You can't add more drawers later; the size is fixed.
- **The ArrayList (`ArrayList<Edge>`):** This is a **Folder** inside a drawer. Since one person might have many friends (edges), the folder needs to be able to expand as you add more papers.

So, `ArrayList<Edge>[] graph` literally translates to: *"A fixed-size cabinet where every drawer contains an expandable folder of edges."*

---

2. Breaking Down the Syntax

Let's look at the line piece by piece:

`ArrayList<Edge>[]` **(The Variable Type)**

- `ArrayList<Edge>` : This tells Java what is going to live *inside* each slot of the array.
- `[]` : This tells Java that `graph` is an **Array**.

`graph` **(The Name)**

- This is just the name of your cabinet.

`new ArrayList[vces]` **(The Initialization)**

- `new` : This allocates memory for the cabinet.
- `ArrayList` : This says "make slots big enough to hold an ArrayList."
- `[vces]` : This says "make exactly 7 slots."

Just like Integer[] arr = new Integer[3];

1. The Syntax

To create an array of Integers without filling it immediately, you must specify the **size**:

Java

```
Integer[] arr = new Integer[3];
```

2. What `new Integer[3]` does on the right side

When Java executes `new Integer[3]`, it goes to the computer's memory and says: *"I need a continuous block of space large enough to hold **3 addresses** (pointers) to Integer objects."*

At this exact moment, the array looks like this:

- `arr[0] = null`
- `arr[1] = null`
- `arr[2] = null`

Even though you used the word `Integer`, you haven't actually created any numbers yet. You have only created the **slots** to hold them.

---

3. Comparing the "Right Side" of both examples

Let's look at your `Integer` array vs. the `Graph` array side-by-side:

Code	What is being built on the Right Side?	What's in the slots initially?
`new Integer[3]`	A row of 3 empty "reserved seats" for numbers.	`null`, `null`, `null`
`new ArrayList[7]`	A row of 7 empty "reserved seats" for Lists.	`null`, `null`, ... (7 times)

Export to Sheets

4. Why the Graph needs a loop but Integers might not

This is the "aha!" moment:

- **For the Integer array:** You can just say `arr[0] = 10;`. Java is smart enough to take the number `10`, wrap it in an object, and put it in the slot.
- **For the Graph array:** You cannot just say `graph[0].add(...)`. Because `graph[0]` is **null**, there is no "List" there yet to receive the data. You have to physically create the list first: `graph[0] = new ArrayList<Edge>();`

Summary

- `Integer[] arr = {1, 3, null};` → You built the shelf **and** put items on it.
- `Integer[] arr = new Integer[3];` → You built the shelf, but it is **empty**.

Untitled

18 December 2025 08:46

so graph is array of arraylist of edges

Spot on! That is exactly the right way to visualize it.

To make it 100% clear, let's look at the "hierarchy" of this structure from the top down:

1. **The Array (The "Rows"):** A fixed-size structure where each index represents a **Vertex** (0, 1, 2...).
2. **The ArrayList (The "Columns"):** Inside each array slot sits an `ArrayList`. This list can grow or shrink depending on how many neighbors that vertex has.
3. **The Edge (The "Data"):** Inside the `ArrayList` are the actual `Edge` objects containing the `src`, `nbr`, and `wt`.

Untitled

17 December 2025 21:37

for Integer[] arr = new Integer[3]

but for ArrayList<Edge>[] graph = new ArrayList<Edge>[3]

why is it wrong

This is one of the most frustrating "gotchas" in Java. It feels inconsistent because it *is* inconsistent.

The short answer: **Java does not allow the creation of "Generic Arrays."**

1. The Comparison

To see why one works and the other fails, we have to look at what happens at **Runtime** (when the code is actually running).

- `Integer[] arr = new Integer[3]` Java knows exactly what an `Integer` is. At runtime, the array "guards the door." If you try to put a `String` into this array, the array will catch it and throw an error. This is called being **reified**.
- `ArrayList<Edge>[] graph = new ArrayList<Edge>[3]` Because of something called **Type Erasure**, Java "forgets" the `<Edge>` part after the code is compiled. At runtime, it just sees `ArrayList[]`.

---

2. The Conflict (The "Why")

Java has a safety rule: **Arrays must know their component type at runtime.** If Java allowed `new ArrayList<Edge>[3]`, you could trick the system like this:

1. Create an array meant for `ArrayList<Edge>`.
2. Because of Type Erasure, the array thinks it's just a general `ArrayList[]`.
3. You could accidentally (or purposely) put an `ArrayList<String>` into the array.
4. The array wouldn't stop you because it "forgot" it was only supposed to hold `Edges`.

To prevent this confusion and potential crashes, Java simply **forbids** the syntax `new ArrayList<Edge>[...]` entirely.

---

3. The Correct Way (The "Workaround")

To make it work, you have to create a "Raw" array and then assign it to your Generic variable.

Java

```
// Right side: We create a "Raw" ArrayList array (No <Edge> here)
// Left side: We tell the compiler "Treat these as ArrayLists of Edges"
ArrayList<Edge>[] graph = new ArrayList[7];
```

Summary Table

Statement	Type	Allowed?	Why?
`new Integer[3]`	Simple Object	**YES**	Type is known at runtime.
`new ArrayList<Edge>[3]`	Generic Object	**NO**	Type info is "erased" at runtime; Java can't "guard the door."
`new ArrayList[3]`	Raw Object	**YES**	This is the "old school" way that Java allows for compatibility.

Export to Sheets

The result: You are forced to use the "Raw" version on the right side, and then use your `for` loop to manually put the `<Edge>` folders into each slot.

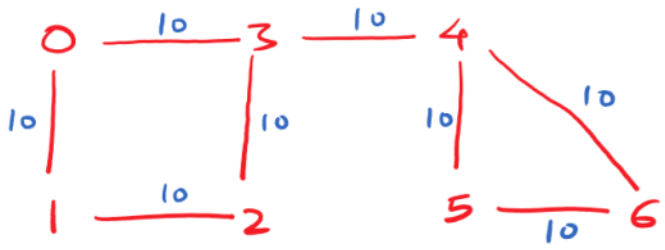
Section 2: DFS

16 August 2022 10:07

DFS

Untitled

18 December 2025 09:04



0 → 01-10, 03-10

1 → 10-10, 12-10

2 → 21-10, 23-10

3 → 32-10, 30-10, 34-10

4 → 43-10, 45-10, 46-10

5 → 54-10, 56-10

6 → 65-10, 64-10

https://github.com/AlgoMagnet/0or1/blob/main/Graph/Implementation1.java

```
Graph hardcoded with 7 vertices.
Vertex 0 -> [0-1 @ 10] [0-3 @ 10]
Vertex 1 -> [1-0 @ 10] [1-2 @ 10]
Vertex 2 -> [2-1 @ 10] [2-3 @ 10]
Vertex 3 -> [3-2 @ 10] [3-0 @ 10] [3-4 @ 10]
Vertex 4 -> [4-3 @ 10] [4-5 @ 10] [4-6 @ 10]
Vertex 5 -> [5-4 @ 10] [5-6 @ 10]
Vertex 6 -> [6-5 @ 10] [6-4 @ 10]
```

Also learn how to write display()

Untitled

20 December 2025 09:01

If I directly want to print edge

Then I'llhave to do this

```
public static void display(ArrayList<Edge>[] graph){
    for(int i=0; i<7; i++){
        System.out.print("graph["+i+"] ->");
        for(Edge e: graph[i]){
            System.out.print(e);
        }
        System.out.println();
    }
}
```

It looks like you are seeing the default memory addresses (hashcodes) of your `Edge` objects instead of the actual data they contain. This happens because the `Edge` class hasn't overridden the default `toString()` method from the `Object` class.

To see the source, destination, and weight of your edges, you need to add a `toString()` method inside your `Edge` class.

1. Fix: Override `toString()`

Update your `Edge` class definition like this:

Java

```java
static class Edge {
    int src;
    int nbr; // or dest
    int wt;

    Edge(int src, int nbr, int wt) {
        this.src = src;
        this.nbr = nbr;
        this.wt = wt;
    }

    @Override
    public String toString() {
        // This formats how the edge appears when printed
        return " (" + src + "-" + nbr + " @ " + wt + ")";
    }
}
```

2. Why this is happening

In Java, when you call `System.out.print(e)`, the JVM looks for a `toString()` method in your `Edge` class.

- **Without the override:** It uses the `Object` class version, which outputs `ClassName@HashCode`.
- **With the override:** It uses your custom string, turning those memory addresses into readable data like `(0-1 @ 10)`.

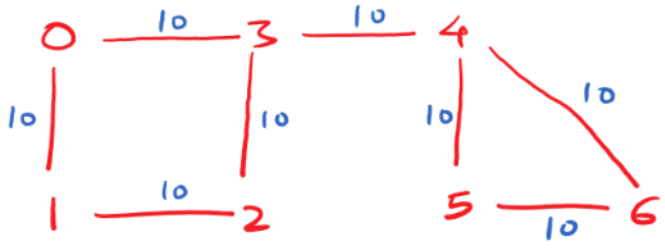
Untitled

14 April 2022 10:05

How to Find Path in Graphs using Depth First Search]

The expectation from this function is to find the path from src to dest in the given graph.
if present then it will return true, else will return false.

Is there any path present from 0 to 6?



0 → 01-10, 03-10
1 → 10-10, 12-10
2 → 21-10, 23-10
3 → 32-10, 30-10, 34-10
4 → 43-10, 45-10, 46-10
5 → 54-10, 56-10
6 → 65-10, 64-10

So as 0 is not equal to 6, so we will check the neighbours of 0, if there is a path from neighbour to the destination.

```java
public static boolean hasPath(ArrayList<Edge>[] graph, int src, int dest, boolean[] visited){
    if(src == dest){
        return true;
    }
    visited[src]= true;
    for(Edge edge: graph[src]){
        if(visited[edge.nbr]==false){
            boolean hasNbrPath = hasPath(graph, edge.nbr, dest,visited);
            if(hasNbrPath == true)
                return true;
        }
    }
    return false;
}
```

Do dry run.

https://github.com/AlgoMagnet/0or1/blob/main/Graph/HasPath.java

This code is a perfect implementation of the **DFS (Depth First Search)** algorithm to solve the "Has Path" problem. You have correctly set up the adjacency list, marked visited nodes to avoid infinite loops, and used recursion to explore the graph.

Here is a breakdown of how your logic works and a small tip to improve your code structure.

How your `hasPath` logic works

1. **Base Case:** If `src == dest`, you've arrived! It returns `true`.
2. **Marking Visited:** `visited[src] = true;` is crucial. Since your graph is undirected (0 connects to 1, and 1 connects to 0), without this, the code would bounce back and forth between 0 and 1 forever.
3. **Recursive Step:** It looks at all neighbors of the current vertex. If a neighbor hasn't been visited, it asks that neighbor: "*Can you find a path to the destination?*"
4. **Backtracking/Return:** If any neighbor returns `true`, the message is passed all the way back up the recursion stack to the main call.

To remember the **Has Path** algorithm (which is a Depth First Search), you should memorize it as a **four-step recursive pattern**. Thinking of it as a "story" or a checklist makes it much easier to rewrite from scratch without looking at the code.

The "Has Path" Memory Algorithm

1. The "Are we there yet?" (Base Case)

Every recursive function needs a way to stop.

- **Logic:** If the current source is the destination, you found it!
- **Code:** `if (src == dest) return true;`

2. The "Breadcrumb" (Preventing Infinite Loops)

Graphs often have cycles (like $0 \to 1 \to 3 \to 0$). If you don't mark where you've been, you'll walk in circles forever.

- **Logic:** Mark the current node as "visited" immediately.
- **Code:** `visited[src] = true;`

3. The "Neighbor Interview" (The Loop)

Ask everyone standing next to you if they know the way.

- **Logic:** Loop through every `edge` connected to the current `src`.
- **Code:** `for (Edge e : graph[src])`

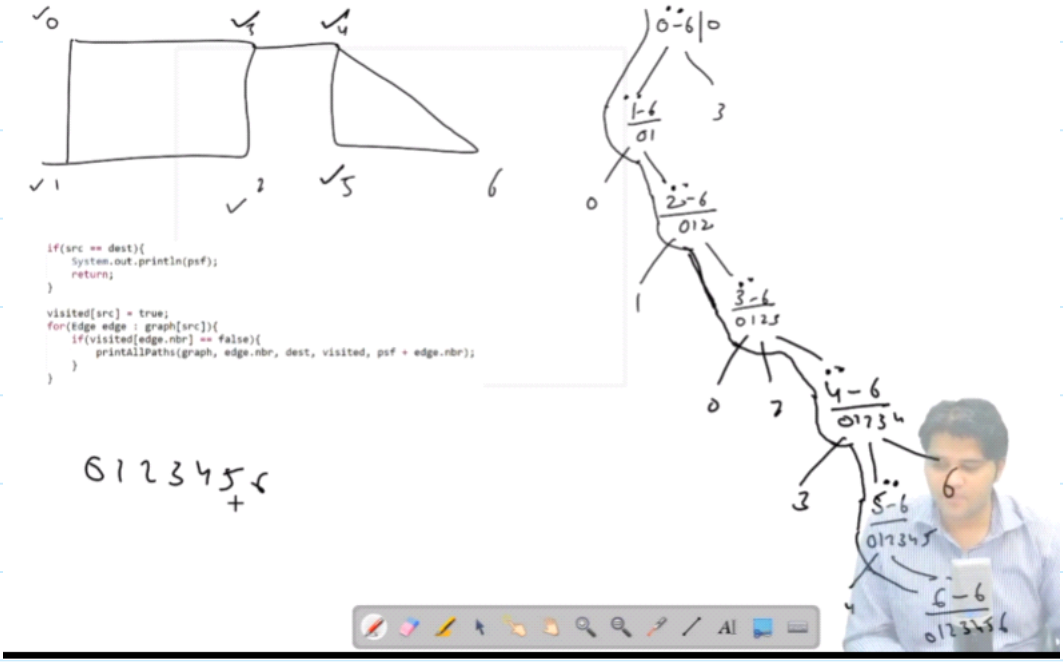
4. The "Check and Pass" (The Recursion)

Only talk to neighbors you haven't visited yet. If even **one** neighbor says they found a path, you immediately tell your "boss" (the previous function call) `true`.

- **Logic:** * If `neighbor` is NOT visited:
  - Ask neighbor to find the path recursively.
  - If they say `true`, you return `true`.
- **Code:** ```java if (!visited[e.nbr]) { if (hasPath(graph, e.nbr, dest, visited)) return true; }

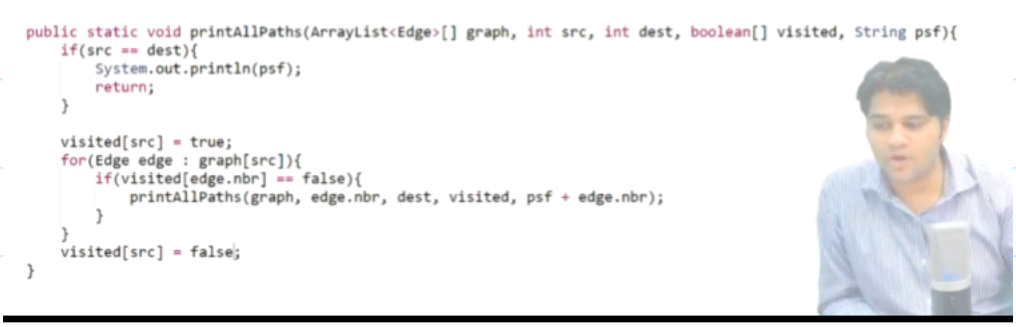
Print all path.

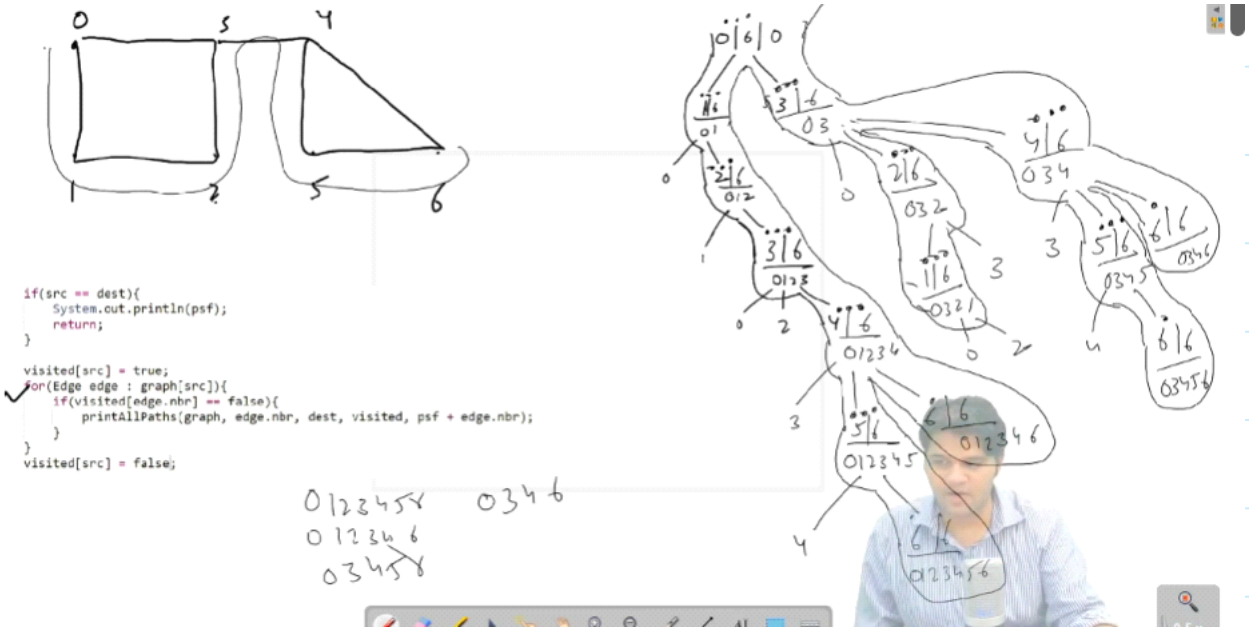
18 December 2025 10:03



It won't be able to send to 3, as it is already visited.

So we have unvisit that node after the loop





If you are asked to write this from scratch, just ask yourself these 4 questions:

1. **Am I there yet?** (`if src == dest`)
2. **Did I mark myself?** (`visited[src] = true`)
3. **Did I check all neighbors?** (`for` loop)
4. **Did I clean up?** (`visited[src] = false`)

Memory Trick: "Post-it Notes"

- **Has Path:** You put a permanent "Visited" sticker on a door.
- **Print All Paths:** You put a Post-it note on the door while you're inside, but you **peel it off** when you walk out of the room.

https://github.com/AlgoMagnet/0or1/blob/main/Graph/PrintAllPaths.java

Untitled

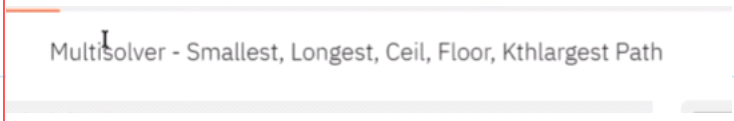
20 December 2025 10:46

How to count the number of path?

Untitled

18 December 2025 12:43

Multisolver - smallest, longest, ceil, floor, kthlargest path



Untitled

29 May 2022 07:59

How do we initialize our variables using identity?

```
static String spath;
static Integer spathwt = Integer.MAX_VALUE;
static String lpath;
static Integer lpathwt = Integer.MIN_VALUE;
static String cpath;
static Integer cpathwt = Integer.MAX_VALUE;
static String fpath;
static Integer fpathwt = Integer.MIN_VALUE;
static PriorityQueue<Pair> pq = new PriorityQueue<>();
public static void multisolver(ArrayList<Edge>[] graph, int src, int dest, boolean[] visited, int criteria, int k, String psf, int
   if(src == dest){
       if(wsf < spathwt){
           spathwt = wsf;
           spath = psf;
       }

       if(wsf > lpathwt){
           lpathwt = wsf;
           lpath = psf;
       }

       if(wsf > criteria && wsf < cpathwt){

       }
       return;
   }
```

How did we decide that smallest path wt should be infinity
And largest path weight should be minus infinity.

If we have to find product of all numbers, then prod =1.

For sum, we do sum =0

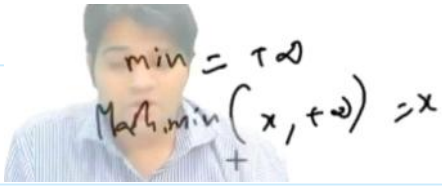
$s = 0$

$x + 0 = x$

Agar kisi operand ke sath 0 interact karega using some operator (+), aur same operand wapas aa jaye, to us operator ki identity hai 0;
+ ki identity hai 0.

Product ki identity hai 1,

So while finding minimum, we initialize with plus infinity.



Kisi bhi number ka min nikaliye plus infinity ke sath, to wahi number aa jayega.

To minimum ki identity hai plus infinity.

Similarly maximum ki identity has minus infinity.

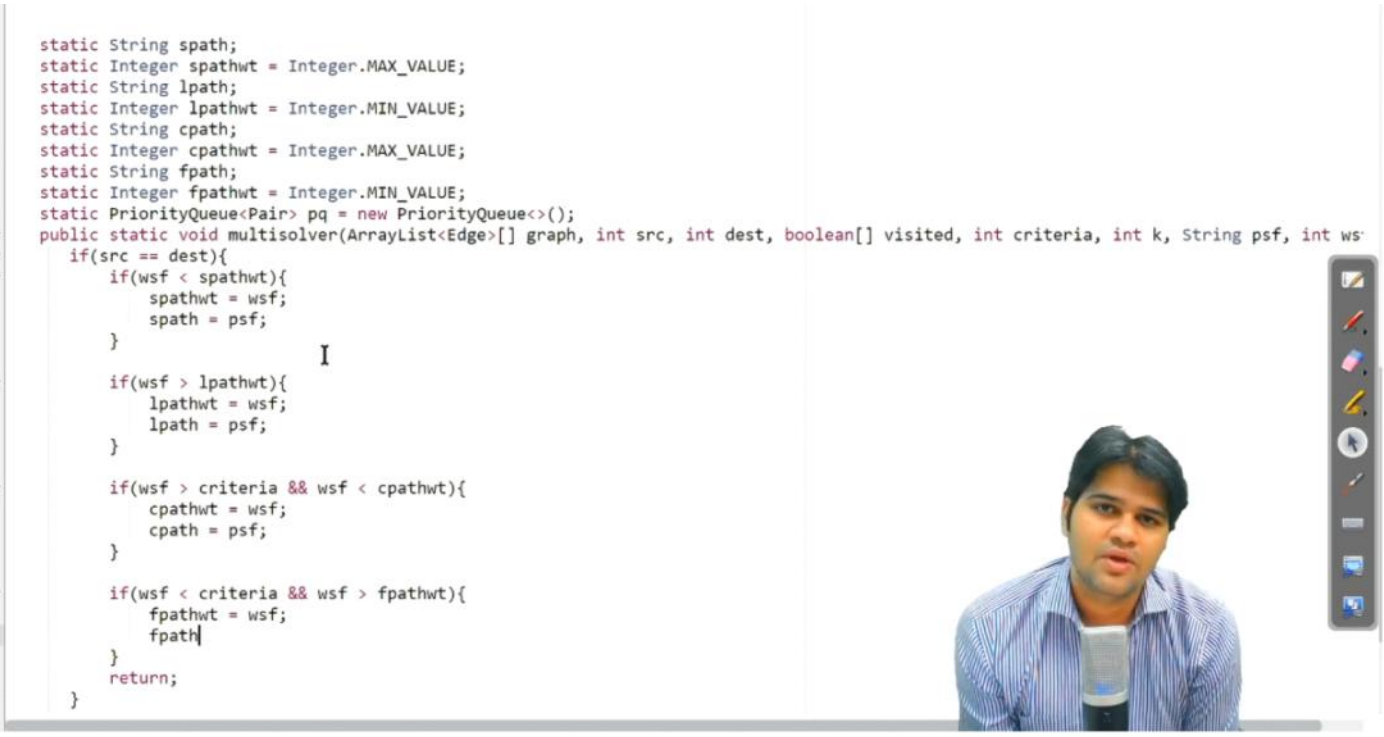
Untitled

29 May 2022 08:03

Ceil of 42 means
42 se just bara

42 se bare walo number me sabse chota

Floor of 42,
42 se chote wale numbers me sabse bara
It means floor is about finding maximum among those numbers smaller than a given number.



```
static String spath;
static Integer spathwt = Integer.MAX_VALUE;
static String lpath;
static Integer lpathwt = Integer.MIN_VALUE;
static String cpath;
static Integer cpathwt = Integer.MAX_VALUE;
static String fpath;
static Integer fpathwt = Integer.MIN_VALUE;
static PriorityQueue<Pair> pq = new PriorityQueue<>();
public static void multisolver(ArrayList<Edge>[] graph, int src, int dest, boolean[] visited, int criteria, int k, String psf, int ws
   if(src == dest){
       if(wsf < spathwt){
           spathwt = wsf;
           spath = psf;
       }

       if(wsf > lpathwt){
           lpathwt = wsf;
           lpath = psf;
       }

       if(wsf > criteria && wsf < cpathwt){
           cpathwt = wsf;
           cpath = psf;
       }

       if(wsf < criteria && wsf > fpathwt){
           fpathwt = wsf;
           fpath
       }
       return;
   }
```

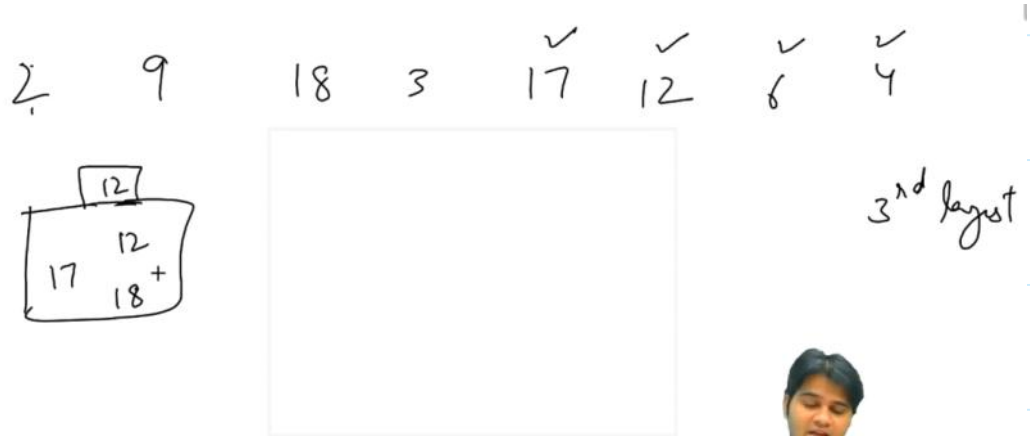
Isi liye fpathwt ko hamne minus infinity se initialize kiya hai

Untitled

29 May 2022 08:06

How to find the kth largest path according to wt.

So we use priority queue.



Team selection method which we have seen before.

Min priority queue of only 3 elements would help.

```
        if(wsf < spathwt){
            spathwt = wsf;
            spath = psf;
        }

        if(wsf > lpathwt){
            lpathwt = wsf;
            lpath = psf;
        }

        if(wsf > criteria && wsf < cpathwt){
            cpathwt = wsf;
            cpath = psf;
        }

        if(wsf < criteria && wsf > fpathwt){
            fpathwt = wsf;
            fpath = psf;
        }

        if(pq.size() < k){
            pq.add(new Pair(wsf, psf));
        } else {
            if(wsf > pq.peek().wsf){
                pq.remove();
                pq.add(new Pair(wsf, psf));
            }
        }
        return;
    }
```

Components of Graph

Untitled

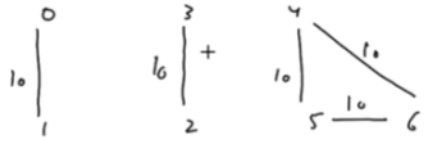
18 December 2025 13:53

Sample Input

```
7
5
0 1 10
2 3 10
4 5 10
5 6 10
4 6 10
```

Sample Output

```
[[0, 1], [2, 3], [4, 5, 6]]
```



https://github.com/AlgoMagnet/0or1/blob/main/Graph/ConnectedComponents.java

The Two Types of "Visited"

Algorithm	Do you Backtrack? (`visited[i]=false`)	Why?
Has Path / Connected Components	**NO**	You just need to know if you can *reach* the node. Once reached, you're done with it.
Print All Paths / Multisolver	**YES**	You need to find *every possible route*. A node might be part of Path A and Path B, so you must "unlock" it for the next route.

The "Island" Analogy

Imagine you are a cartographer mapping islands in the ocean:

- **The Main Loop** is your ship sailing across the sea looking for land.
- When you see **Vertex 0**, you've found an island!
- You send a scout (**DFS/drawTree**) to walk across the whole island and plant flags on every coordinate (0, 1, etc.).
- **If you take the flags back (unvisit):** When your ship sails a bit further and sees Coordinate 1, you won't realize it's the same island you already mapped. You'll waste time mapping it again.
- **By keeping the flags (stay visited):** Your ship sees the flag on Coordinate 1 and says, "Oh, I've already been there, no need to stop," and keeps sailing until it finds a truly new island (like Vertex 2).

The Connected Components Memory Algorithm: L.V.D.

To remember this, think **L.V.D.** (Loop, Visit, Draw).

1. The Loop (The Explorer)

- **Logic**: Run a `for` loop from `0` to `vces - 1`. This ensures no vertex is left behind, even if it has no edges.
- **Action**: If `visited[v]` is `false`, you've discovered a brand-new component!.

2. The Visit (The Collector)

- **Logic**: Inside the loop, create a fresh `ArrayList<Integer>` to hold the members of this new "island.".
- **Action**: Call your DFS helper function (`drawTree`) to fill this list.

3. The Draw (The Scout)

- **Logic**: This is your standard DFS.
- **Action**:
  1. Mark the current vertex as `visited = true`.
  2. Add it to the current component list.
  3. Loop through neighbors.
  4. **Crucial**: **DO NOT BACKTRACK** (don't set visited to false). Once a node is part of a component, it stays there.

Is the Graph Connected?

Once you have your `ArrayList<ArrayList<Integer>> comps`, you can answer a very common follow-up question: **"Is this graph connected?"**

- **Logic**: If `comps.size() == 1`, the entire graph is one big island. It is connected.
- **Logic**: If `comps.size() > 1`, there are multiple islands. It is not connected.

It is completely normal to feel intimidated by `ArrayList<ArrayList<Integer>>`. It looks like a "code mouthful," but the secret to not being scared is to **stop reading the whole thing at once** and start seeing it as a **Box of Folders**.

1. The Visual Solution: "Box of Folders"

Think of the structure from the inside out:

- `Integer` : A single sheet of paper (e.g., the number `5`).
- `ArrayList<Integer>` : A **Folder** containing those sheets (one island of vertices).
- `ArrayList<...>` : A **Box** that holds all those folders together.

So, `ArrayList<ArrayList<Integer>>` is just a **Box of Folders containing Numbers**.

---

2. The Semantic Solution: Use "Type Aliasing" (Mental or Real)

When you see a complex type, rename it in your head to something human-readable.

Instead of seeing...	Think...
`ArrayList<Integer>`	`Component`
`ArrayList<ArrayList<Integer>>`	`AllComponents`

It's a great observation. In recursive functions like `hasPath`, the base case is a **success condition** (reaching the destination). In `drawTree` (which is a Depth First Search), the base case is actually **implicit**.

Here is why you don't see a traditional `if(condition) return;` at the top of the function:

1. The "Implicit" Base Case

In recursion, a base case is simply the condition that stops the function from calling itself further. In your `drawTree` function, the stopping condition is the **for-loop and the `if` check**:

Java

```
for(Edge e: graph[src]){
    if(visited[e.nbr] == false){
        drawTree(graph, e.nbr, visited, currentComp);
    }
}
```

If a node has no unvisited neighbors (meaning all adjacent nodes are already in `visited`), the `if` condition will never be true. The loop finishes, the function ends, and it naturally returns to the previous caller. **The "exhaustion" of unvisited neighbors is the base case.**

---

2. Comparison: `hasPath` vs. `getConnectedComponents`

Feature	`hasPath`	`drawTree` (Get Components)
Goal	Find a specific target node.	Visit **every** reachable node.
Base Case	`if (src == dest) return true;`	When no unvisited neighbors remain.
Logic	Stop as soon as you find the "Exit."	Keep going until you've explored the entire "Room."

3. Why we don't use `if(visited[src])` as a base case

Sometimes people write the base case at the very top like this:

Java

```
public static void drawTree(graph, src, visited, currentComp) {
    if(visited[src]) return; // This is an alternative base case

    visited[src] = true;
    // ... rest of code
}
```

However, your current code uses **pre-emptive checking** (the `if(visited[e.nbr] == false)` inside the loop). This ensures that `drawTree` is **never called** on a node that has already been visited. Because you check *before* you call, you don't need to check *after* you enter the function.

Summary

The base case exists, but it's structural rather than conditional. The recursion stops when:

1. A vertex has no edges (leaf node).
2. All neighbors of the vertex are already marked `true` in the `visited` array.

Exactly! You hit the nail on the head. In recursive programming, these are often referred to as **Look-Ahead** (Proactive) and **Base-Case** (Reactive) strategies.

Here is the breakdown of the two styles:

1. The Proactive Approach (Your Code)

This is what you currently have. You are being "proactive" by checking the condition **before** making the recursive call.

Java

```
for(Edge e: graph[src]) {
    if(visited[e.nbr] == false) { // The "Guard"
        drawTree(graph, e.nbr, visited, currentComp);
    }
}
```

- **How it works:** You only pick up the phone if you know the person is available.
- **Why no base case?** Because the `if` statement acts as a gatekeeper. If there are no unvisited neighbors, the function simply reaches the end of its code and returns.

---

2. The Reactive Approach

In this style, you make the call blindly and let the function handle the "rejection" at the very start.

Java

```
public static void drawTree(...) {
    // REACTIVE BASE CASE
    if (visited[src] == true) {
        return;
    }

    visited[src] = true;
    currentComp.add(src);

    for(Edge e: graph[src]) {
        drawTree(graph, e.nbr, ...); // Blind call
    }
}
```

- **How it works:** You call everyone, and if they are busy (visited), they immediately hang up on you (`return`).
- **Pros/Cons:** This is often "cleaner" to look at, but it results in more function calls being added to the Stack, even if they return immediately.

Visualization of the Flow

In your **Proactive** version, the recursion tree only expands to nodes that actually need processing.

Why `hasPath` feels different

In `hasPath` , even if you use a "proactive" check for the `visited` array, you still need a **Reactive Base Case** for the destination: `if (src == dest) return true;`

This is because "reaching the destination" isn't a violation of a rule (like visiting a node twice); it's the **successful completion** of your goal. Since `drawTree` doesn't have a single "target" (its goal is just to finish), it doesn't need that specific "Success" base case.

```
    public static ArrayList<ArrayList<Integer>>
getConnectedComponents(ArrayList<Edge>[] graph, int
vces, boolean[] visited) {
        ArrayList<ArrayList<Integer>> allComps = new
ArrayList<>();
        for (int i = 0; i < vces; i++) {
            // Even in reactive, we usually check here
so we don't
            // add empty lists to allComps for already
visited nodes.
            if (visited[i] == false) {
                ArrayList<Integer> currentComp = new
ArrayList<>();
                drawTree(graph, i, visited,
currentComp);
                allComps.add(currentComp);
```

```
                }
            }
            return allComps;
        }
        public static void drawTree(ArrayList<Edge>[] graph,
    int src, boolean[] visited, ArrayList<Integer>
    currentComp) {
            // --- REACTIVE BASE CASE ---
            // We "react" to the fact that we might have
    been called
            // on a node that is already processed.
            if (visited[src] == true) {
                return;
            }
            // 1. Mark and Add (The work)
            visited[src] = true;
            currentComp.add(src);
            // 2. Proactive "Blind" Calls
            for (Edge e : graph[src]) {
                // We don't check 'if' the neighbor is
    visited here.
                // We just throw the call and let the next
    function handle it.
                drawTree(graph, e.nbr, visited,
    currentComp);
            }
        }
```

Yes, you absolutely can! To make `hasPath` fully reactive, you move the `visited` check to the top as a base case.

However, there is a **critical logical trap** you have to watch out for when doing this with `hasPath`: the **order** of your base cases matters.

The Reactive `hasPath`

In this version, we make the "blind call" inside the loop and handle the "rejections" (already visited or reached destination) at the very start of the function.

Java

```java
public static boolean hasPath(ArrayList<Edge>[] graph, int src, int dest, boolean
    // 1. REACTIVE BASE CASE: Already visited?
    if (visited[src] == true) {
        return false;
    }

    // 2. SUCCESS BASE CASE: Found the target?
    if (src == dest) {
        return true;
    }

    // 3. Mark Visited
    visited[src] = true;

    // 4. Neighbor Loop (Blind calls)
    for (Edge edge : graph[src]) {
        // No 'if' check here; just call and react to the result
        if (hasPath(graph, edge.nbr, dest, visited)) {
            return true;
        }
    }

    return false;
}
```

⚠️ The Logic Trap: Order of Checks

When writing reactive code, you must be careful about which base case comes first. Look at these two scenarios:

- **Correct Order (Visited check first):** If `src` is already visited, we return `false` immediately. This prevents infinite loops in graphs with cycles.
- **The Trap (Destination check first):** If you check `if(src == dest)` **before** `if(visited[src])`, the code still works for `hasPath`. **But**, if you were writing a function that *counts* paths or finds *all* paths, putting the destination check first might lead to double-counting or infinite recursion depending on how you reset your `visited` array.

Comparison: Proactive vs. Reactive `hasPath`

Feature	Proactive (Your Original)	Reactive (New Version)
Efficiency	**Higher.** Doesn't waste time setting up a function call for a node we already know is visited.	**Lower.** Creates a full Stack Frame just to immediately return `false`.
Readability	Slightly "noisier" due to nested `if` statements inside the loop.	Very "clean." The loop only focuses on the recursion logic.
Stack Overflow Risk	Lower.	Slightly higher on extremely deep/dense graphs because of the "useless" calls.

Export to Sheets

Which one should you use?

In competitive programming or interviews:

- **Proactive** is generally preferred for Graphs because it prevents the stack from filling up with unnecessary calls, especially in very large graphs (like 10^5 nodes).
- **Reactive** is often used in Trees or simpler recursive problems (like Flood Fill in a 2D matrix) because the code looks more elegant.

No of islands

16 April 2022 08:41

It is an application problem of get connected components:

Here it is not given in form of graphs, it's just a 2d matrix.

```
void drawtreeforcomponent(vector<vector<char>>grid, int i, int j, int** visited)
{
    if(i<0 || j< 0 || i>= grid.size() || j>=grid[0].size() || grid[i][j]=='0' || visited[i][j]==1 )
    {
        return ;
    }

    visited[i][j]=1;
    drawtreeforcomponent(grid, i-1, j, visited);
    drawtreeforcomponent(grid, i, j+1, visited);
    drawtreeforcomponent(grid, i, j-1, visited);
    drawtreeforcomponent(grid, i+1, j, visited);
}

class Solution {
public:

    int numIslands(vector<vector<char>>& grid) {
        int row=grid.size();
        int col=grid[0].size();
        int** visited = (int**)calloc( sizeof(int*), row);
        for (int i = 0; i < row; i++)
        visited[i] = (int*)calloc( sizeof(int), col);
        int count=0;
        for(int i=0;i<row; i++)
        {
            for(int j=0;j<col; j++)
            {
                if(grid[i][j]=='1' && visited[i][j]==0)
                {
                    drawtreeforcomponent(grid, i, j, visited);
                    count++;
                }
            }
        }
        return count;
    }
};
```

47 / 49 test cases passed. Status: **Time Limit Exceeded**

Submitted: **36 minutes ago**

Last executed input: [["0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0"…

```
void drawtreeforcomponent(vector<vector<char>>grid, int i, int j, int** visited)
{
   if(i<0 || j< 0 || i>= grid.size() || j>=grid[0].size() || grid[i][j]=='0' || visited[i]
[j]==1 )
   {
      return ;
   }

   visited[i][j]=1;
   drawtreeforcomponent(grid, i-1, j, visited);
   drawtreeforcomponent(grid, i, j+1, visited);
   drawtreeforcomponent(grid, i, j-1, visited);
   drawtreeforcomponent(grid, i+1, j, visited);
}

class Solution {
public:

   int numIslands(vector<vector<char>>& grid) {
      int row=grid.size();
      int col=grid[0].size();
      int** visited = (int**)calloc( sizeof(int*), row);
      for (int i = 0; i < row; i++)
      visited[i] = (int*)calloc( sizeof(int), col);
      int count=0;
      for(int i=0;i<row; i++)
      {
         for(int j=0;j<col; j++)
         {
            if(grid[i][j]=='1' && visited[i][j]==0)
```

```
            {
                drawtreeforcomponent(grid, i, j, visited);
                count++;
            }
        }
    }
    return count;
  }
};
```

Then after making slight change in the code, it ran.

There are two techniques: proactive and reactive
Soch soch ke call lagao, ya fir pehle call laga lo aur base case me baad me sambhal lo

```
1
2
3  class Solution {
4  public:
5      void drawtreeforcomponent(vector<vector<char>>& grid, int i, int j)
6      {
7          if(i<0 || j< 0 || i>= grid.size() || j>=grid[0].size() ||  grid[i][j]=='0' || grid[i][j] == '2')
8          {
9              return ;
10         }
11         grid[i][j]='2';
12         drawtreeforcomponent(grid, i-1, j);
13         drawtreeforcomponent(grid, i, j+1);
14         drawtreeforcomponent(grid, i, j-1);
15         drawtreeforcomponent(grid, i+1, j);
16     }
17     int numIslands(vector<vector<char>>& grid) {
18         int row=grid.size();
19         int col=grid[0].size();
20
21         int count=0;
22         for(int i=0;i<row; i++)
23         {
24             for(int j=0;j<col; j++)
25             {
26                 if(grid[i][j]=='1')
27                 {
28                     drawtreeforcomponent(grid, i, j);
29                     count++;
30                 }
31             }
32         }
33         return count;
34     }
35 };
```

Submission Detail

49 / 49 test cases passed.

Status: **Accepted**

Runtime: **41 ms**
Memory Usage: **12.3 MB**

Submitted: **0 minutes ago**

```
class Solution {
public:
  void drawtreeforcomponent(vector<vector<char>>& grid, int i, int j)
  {
    if(i<0 || j< 0 || i>= grid.size() || j>=grid[0].size() ||  grid[i][j]=='0' || grid[i]
[j] == '2')
    {
      return ;
    }
    grid[i][j]='2';
    drawtreeforcomponent(grid, i-1, j);
    drawtreeforcomponent(grid, i, j+1);
    drawtreeforcomponent(grid, i, j-1);
    drawtreeforcomponent(grid, i+1, j);
  }
  int numIslands(vector<vector<char>>& grid) {
    int row=grid.size();
    int col=grid[0].size();

    int count=0;
    for(int i=0;i<row; i++)
    {
      for(int j=0;j<col; j++)
      {
        if(grid[i][j]=='1')
        {
          drawtreeforcomponent(grid, i, j);
          count++;
        }
      }
    }
```

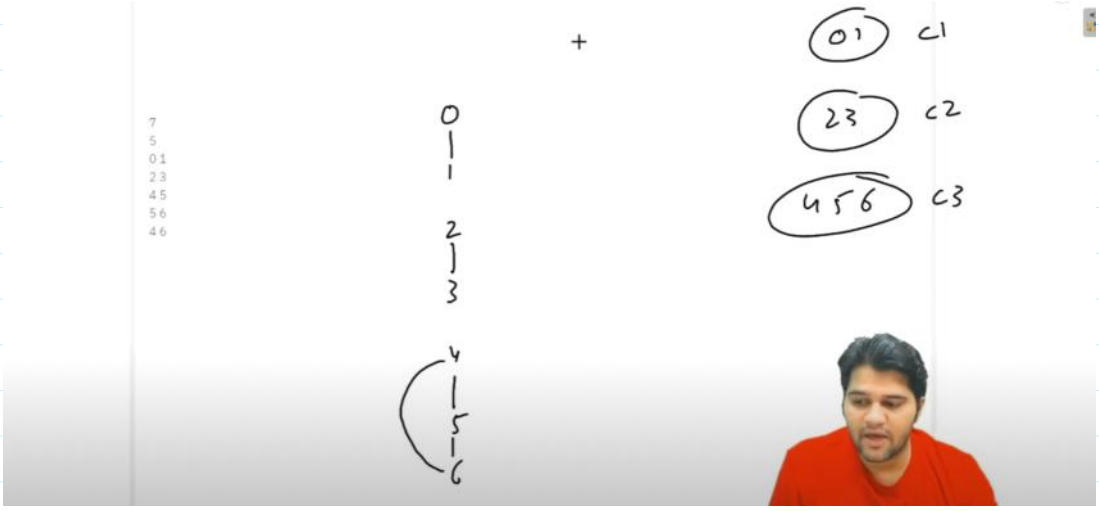
```
        return count;
    }
};
```

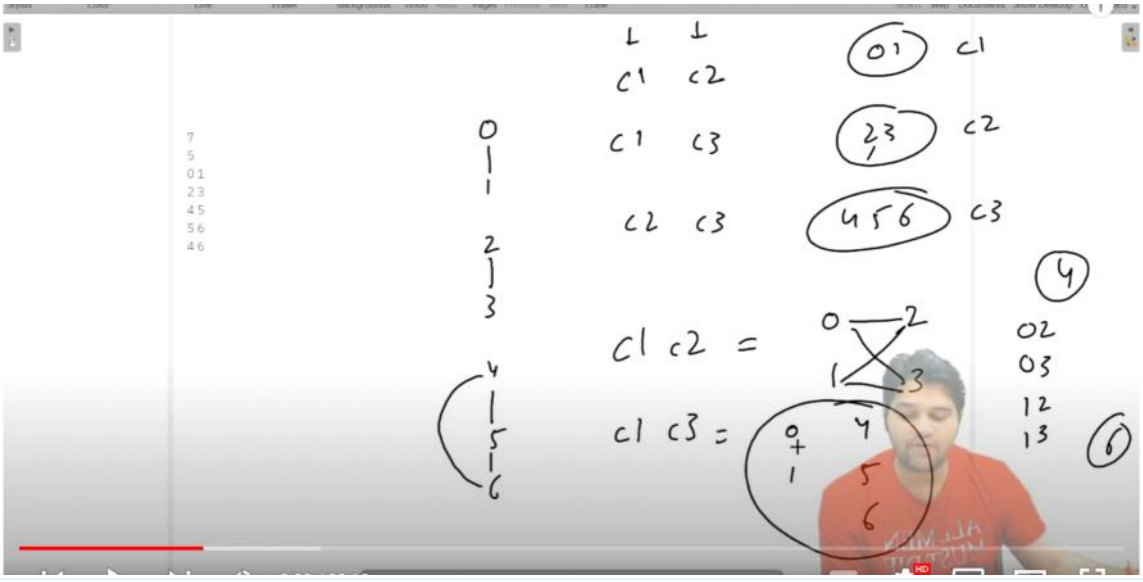
Perfect friend

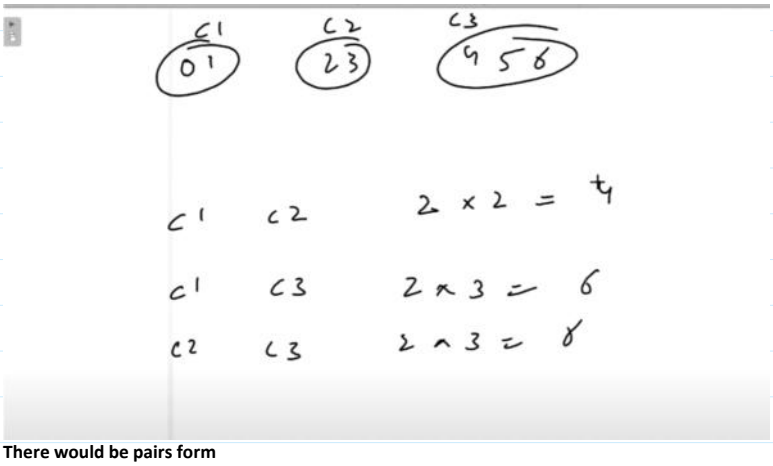
16 April 2022 14:27

If I tell you that you have to use components of graph concept, then the difficulty level of this problem is nothing. The real art is in finding out what concepts one has to use in order to solve a problem. So give yourself a surprise.

1. You are given a number n (representing the number of students). Each student will have an id from 0 to n - 1.
2. You are given a number k (representing the number of clubs)
3. In the next k lines, two numbers are given separated by a space. The numbers are ids of students belonging to same club.
4. You have to find in how many ways can we select a pair of students such that both students are from different clubs.







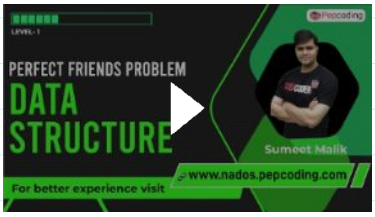
There would be pairs form
C1 : c2
C1 : C3
C2 : C3

```
150     int pairs=0;
151     for(int i=0;i<comp.size(); i++)
152     {
153         for(int j=i+1; j< comp.size(); j++)
154         {
155             int count=comp[i].size() * comp[j].size();
156             pairs +=count;
157         }
158     }
159     cout<<pairs<<endl;
160     return 0;
161 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

16

[Perfect Friends Problem using Graphs | Data Structures in JAVA]

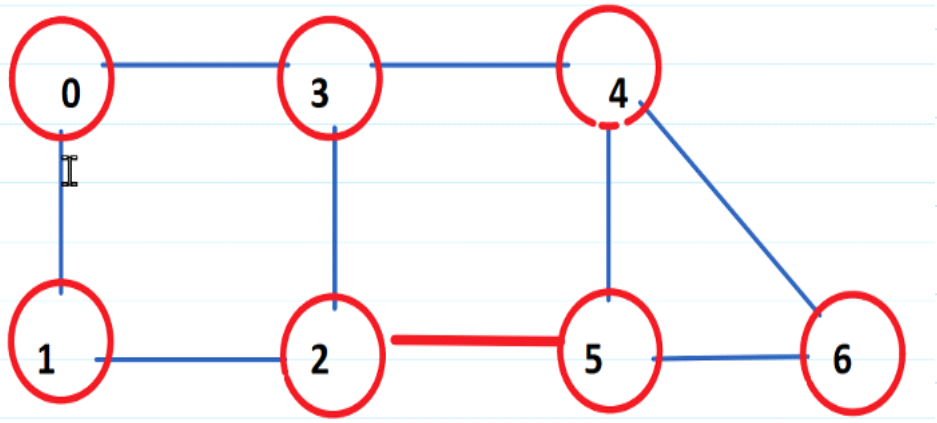


Untitled

16 April 2022 09:36

Hamiltonian path and cycle

You're given a graph and a source



Hamiltonian path is a path which starts from the src and visits all the vertices without visiting them twice.
If "0" would have been the source, then few of the hamiltonian paths are
0-> 1->2->3->4->5->6

Now when does a hamiltonian path becomes a hamiltonian cycle,
When the first and the last vertex have the direct edge
0123465 is also one hamiltonian path
0125643 is hamiltonian path and also a cycle
0346521 is also a path and a cycle

Untitled

29 May 2022 08:42

It is just like print all path (there we used to have src and dest, and jate hue visit karte the aur ate hue unvisit karte the, base case hota tha ki destination pahuch gye)

Bas is baar yahi hoga ki base case badal jayegi, ki sari vertex visit ho gyi

Base case me ye check karna parega ki sari vertices visited ho gayi ki nahi

Isme bool visited nahi use karenge, bas visited set use karenge, baki sab kuch to same hi rehta hai.

Untitled

17 April 2022 11:45

```
struct node* getnewnode()…
struct graph* createGraph(int v)…
void printGraph(struct graph* gr1)…
void insert(struct graph* gr1, int src, int dest, int weight)…

void Hamiltonian(struct graph* gr1, int src, set<int> visited, string str, int original_src)
{
    if(visited.size() == gr1->noOfVertices-1)
    {
        cout<<str;;
        int closing_edge_found=0;
        struct node* temp=gr1->array[src].head;
        while(temp!=NULL)
        {
            if(temp->dest==original_src){
            closing_edge_found=1;
            break;
            }
            temp=temp->next;
        }
        if(closing_edge_found==1)
            cout<<"*"<<endl;
        else
            cout<<"."<<endl;
        return;
    }
    visited.insert(src);
    struct node* temp=gr1->array[src].head;
    while(temp!=NULL)
    {
        if(visited.find(temp->dest) == visited.end()){
           string s= to_string(temp->dest);
            Hamiltonian(gr1, temp->dest, visited, str + s, original_src);
        }
        temp=temp->next;// if you forget this line in linked list traversal,
                        // I can't save you from sleepless nights
    }
    visited.erase(src);
}
int main()
{
    int V = 7;
    struct graph* gr1;// gr1 is my first graph of type struct graph
    gr1= createGraph(V); // creates a graph with 5 vertices.

    insert(gr1, 0, 1, 10);// insert 0 -> 1 and 1-> 0 in graph gr1
    insert(gr1, 1, 2, 10);
    insert(gr1, 2, 3, 10);
    insert(gr1, 0, 3, 40);
    insert(gr1, 2, 5, 10);
    insert(gr1, 3, 4, 2);
    insert(gr1, 4, 5, 3);
    insert(gr1, 5, 6, 3);
    insert(gr1, 4, 6, 8);

    set<int> visited;
    string stri;
    stri.push_back('0');
    Hamiltonian(gr1, 0, visited, stri, 0);
    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
[Running] cd "c:\Users\ankit\Documents\Trees\" && g++ gold.cpp -o gold && "c:\Users\ankit\Documents\Trees\"gold
0346521*
0125643*
0123465.
0123456.
```

```
#include<bits/stdc++.h>
using namespace std;
struct node{
    int src;
    int dest;
    int weight;
    struct node* next;
```

```
};
struct arraylist{
    struct node* head;
};
struct graph{
    int noOfVertices;
    struct arraylist* array;
};
struct node* getnewnode()
{
    struct node* temp;
    temp=(struct node*)malloc(sizeof(struct node));
    temp->next=NULL;
    return temp;
}
struct graph* createGraph(int v)
{
    struct graph* gr1;
    gr1=(struct graph*)malloc(sizeof(struct graph));
    gr1->noOfVertices=v;
    gr1->array=(struct arraylist*)malloc(sizeof(struct
arraylist)*v);
    for(int i=0;i<v;i++)
    {
        gr1->array[i].head=NULL;
    }
    return gr1;
}
void printGraph(struct graph* gr1)
{
    for(int v=0;v<gr1->noOfVertices;v++)
    {
        struct node* temp=gr1->array[v].head;
        if(temp==NULL)
        return;
        else{
            printf("[%d]: ",v);
        while(temp->next!=NULL)
        {
            printf("(%d-> %d @ %d)---->", temp->src,
temp->dest, temp->weight);
            temp=temp->next;
        }
        printf("(%d-> %d @ %d) \n", temp->src, temp->
dest, temp->weight);
        }
    }
}
void insert(struct graph* gr1, int src, int dest, int
weight)
{
    struct node* temp=getnewnode();
    temp->src=src;
    temp->dest=dest;
    temp->weight=weight;
    if(gr1->array[src].head==NULL)
    {
        gr1->array[src].head=temp;
    }
    else
    {
        temp->next=gr1->array[src].head;
        gr1->array[src].head=temp;
    }
   temp=getnewnode();
    temp->src=dest;
    temp->dest=src;
```

```
    temp->weight=weight;
    if(gr1->array[dest].head==NULL)
    {
        gr1->array[dest].head=temp;
    }
    else
    {
        temp->next=gr1->array[dest].head;
        gr1->array[dest].head=temp;
    }
}
void Hamiltonian(struct graph* gr1, int src, set<int>
visited, string str, int original_src)
{
    if(visited.size() == gr1->noOfVertices-1)
    {
        cout<<str;;
        int closing_edge_found=0;
        struct node* temp=gr1->array[src].head;
        while(temp!=NULL)
        {
            if(temp->dest==original_src){
            closing_edge_found=1;
            break;
            }
            temp=temp->next;
        }
        if(closing_edge_found==1)
            cout<<"*"<<endl;
        else
            cout<<"."<<endl;
        return;
    }
    visited.insert(src);
    struct node* temp=gr1->array[src].head;
    while(temp!=NULL)
    {
        if(visited.find(temp->dest) == visited.end()){
           string s= to_string(temp->dest);
            Hamiltonian(gr1, temp->dest, visited, str
+ s, original_src);
        }
        temp=temp->next;// if you forget this line in
linked list traversal,
                        // I can't save you from
sleepless nights
    }
    visited.erase(src);
}
int main()
{
    int V = 7;
    struct graph* gr1;// gr1 is my first graph of type
struct graph
    gr1= createGraph(V); // creates a graph with 5
vertices.

    insert(gr1, 0, 1, 10);// insert 0 -> 1 and 1-> 0
in graph gr1
    insert(gr1, 1, 2, 10);
    insert(gr1, 2, 3, 10);
    insert(gr1, 0, 3, 40);
    insert(gr1, 2, 5, 10);
    insert(gr1, 3, 4, 2);
    insert(gr1, 4, 5, 3);
    insert(gr1, 5, 6, 3);
    insert(gr1, 4, 6, 8);
```

```
    set<int> visited;
    string stri;
    stri.push_back('0');
    Hamiltonian(gr1, 0, visited, stri, 0);
    return 0;
}
```

Untitled

29 May 2022 08:59

```
void Hamiltonian(struct graph* gr1, int src, set<int> visited, string str, int
{
    if(visited.size() == gr1->noOfVertices-1)
    {
        cout<<str;;
```

A very small point to keep in mind is:

Visited hamesha 'path so far" se ek step piche chalta rehta hai

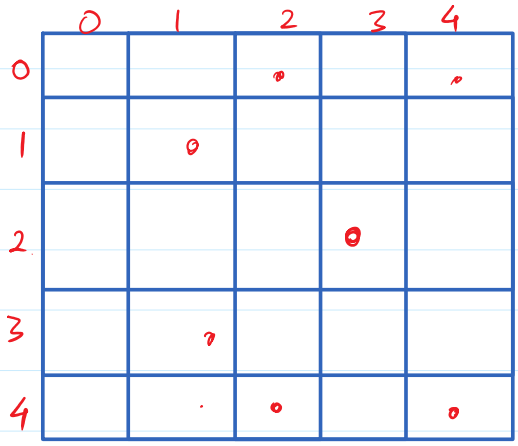
Visited me hum tabhi vertex dalte hai, jab hum hamiltonian function ke andar ate hai.

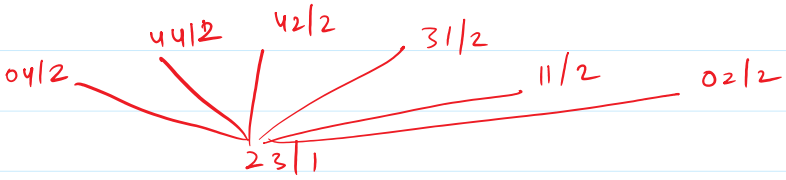
Function ke andar ane ke baad visited set me add hota hai, isiliye humne ye dekha ki agar visited ki size total no of vertex se bas ek kam hona chahie kyunki abhi to add nahi hua hai uske andar, bas wo call hua hai.

[Hamiltonian Path & Cycles in Graphs and Graph Theory]

Knights tour

17 April 2022 11:47



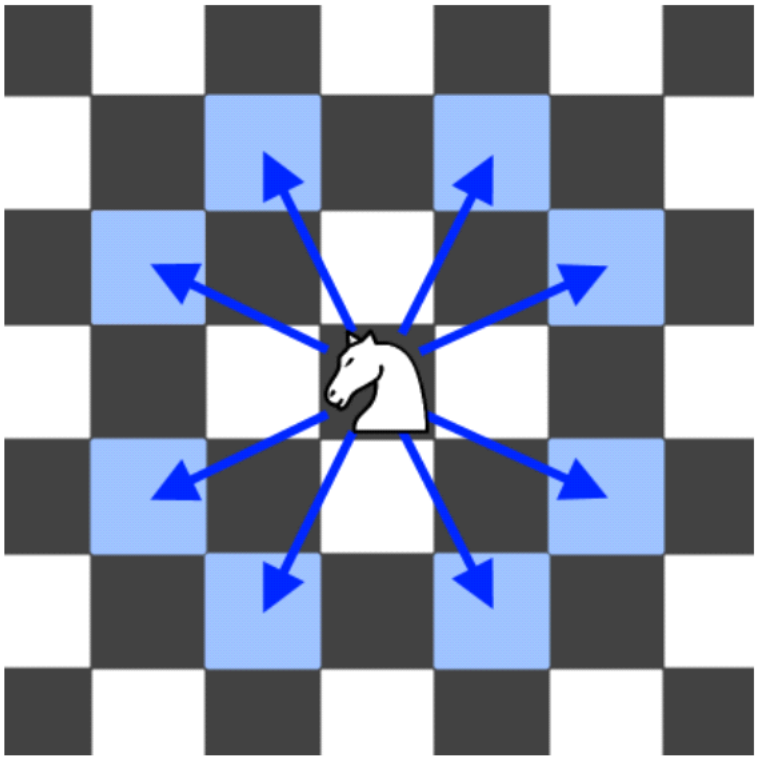


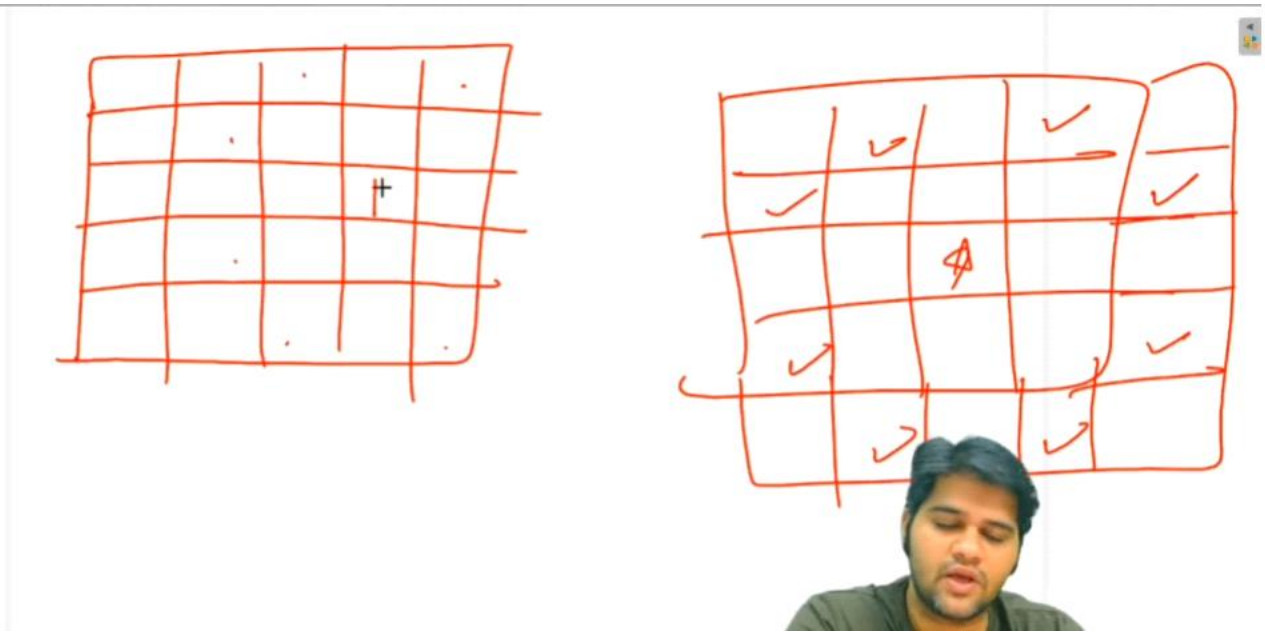
Untitled

29 May 2022 09:03

n*n ka chess board diya gaya hai
Aur ek starting point di gayi hai

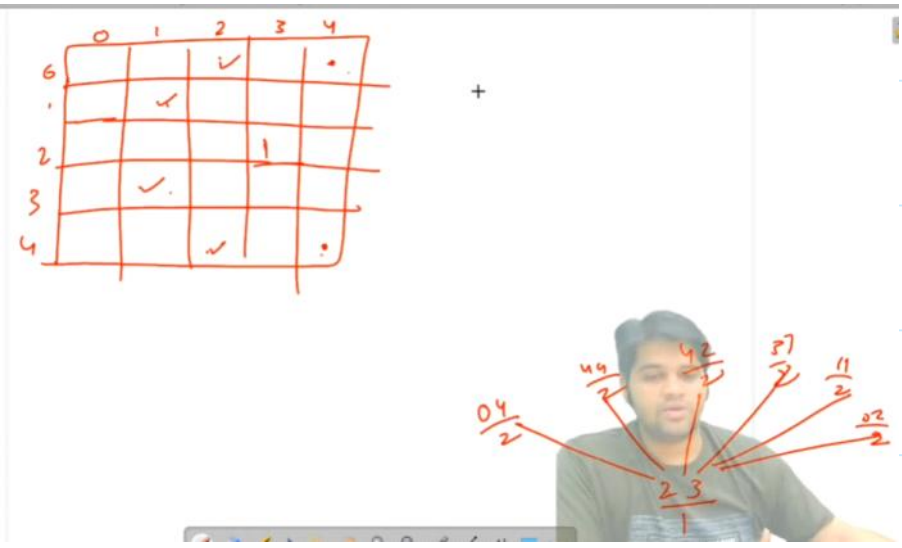
How knight moves.





Kya knight ke paas aisa rasta hai to visit all the cell without visiting any cell twice. And print all the travel paths.

Kaha kaha second move chal sakte ho



Level and options wali baat hai, jaha pe khare ho waha se kaha kaha ja sakte ho

```
        int c = scn.nextInt();

        int[][] chess = new int[n][n];
        printKnightsTour(chess, r, c, 1);
    }

    public static void printKnightsTour(int[][] chess, int r, int c,
        if(r < 0 || c < 0 || r >= chess.length || c >= chess.length
            return;
        } else if(move == chess.length * chess.length){
            chess[r][c] = move;
            displayBoard(chess);
            chess[r][c] = 0;
            return;
        }


        chess[r][c] = move;
        printKnightsTour(chess, r - 2, c + 1, move + 1);
        printKnightsTour(chess, r - 1, c + 2, move + 1);
        printKnightsTour(chess, r + 1, c + 2, move + 1);
        printKnightsTour(chess, r + 2, c + 1, move + 1);
        printKnightsTour(chess, r + 2, c - 1, move + 1);
        printKnightsTour(chess, r + 1, c - 2, move + 1);
        printKnightsTour(chess, r - 1, c - 2, move + 1);
        printKnightsTour(chess, r - 2, c - 1, move + 1);
        chess[r][c] = 0;
    }
```

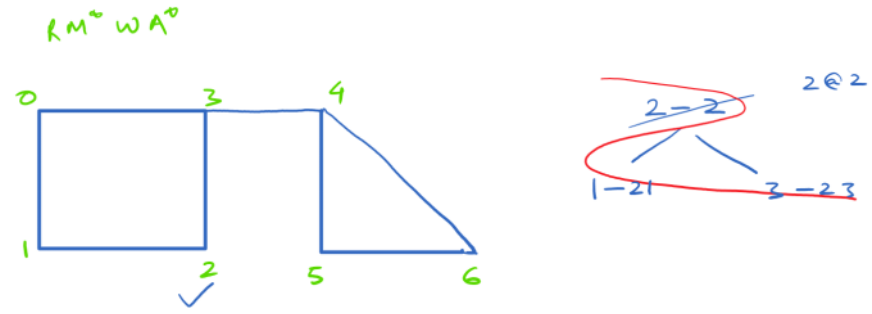
Section 4: BFS

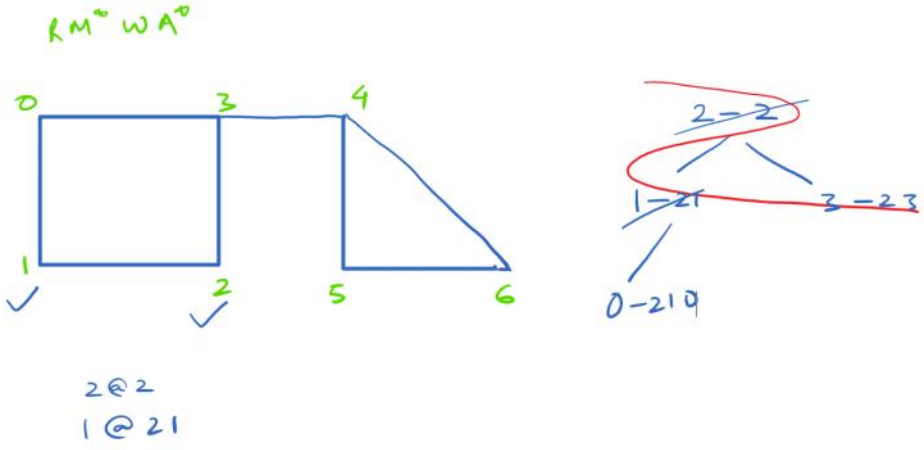
16 August 2022 10:19

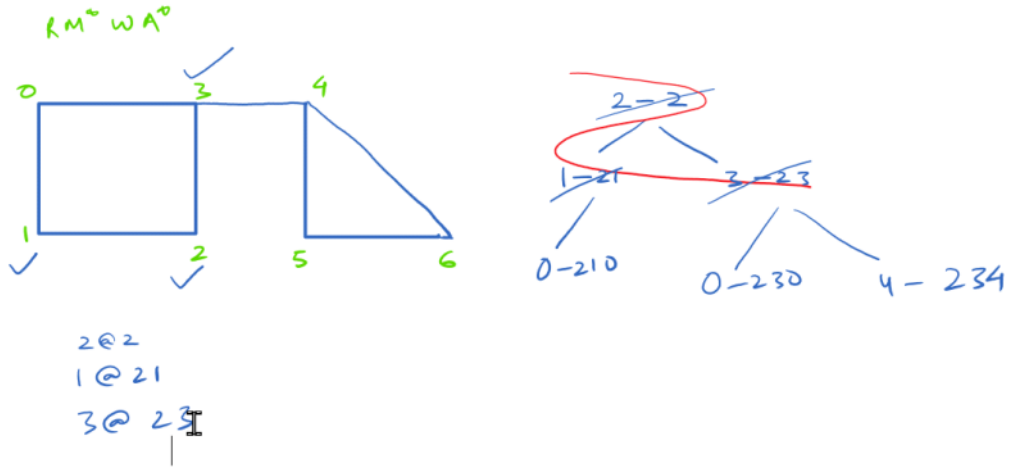
https://github.com/AlgoMagnet/0or1/blob/main/Graph/BFS.java
BFS me hum queue use karenge as it is like level order like tree.

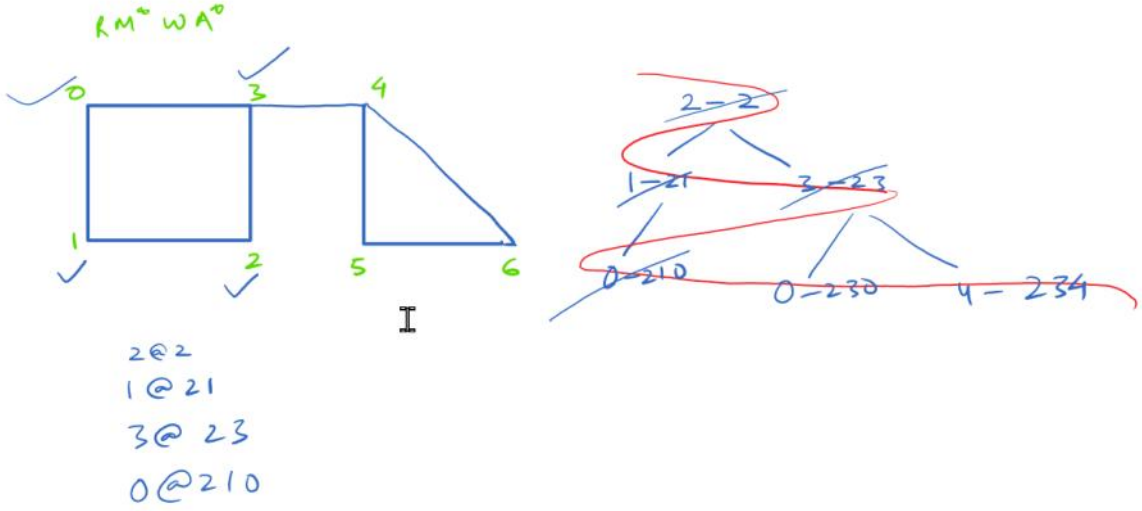
Source and source tak ka path daal do''

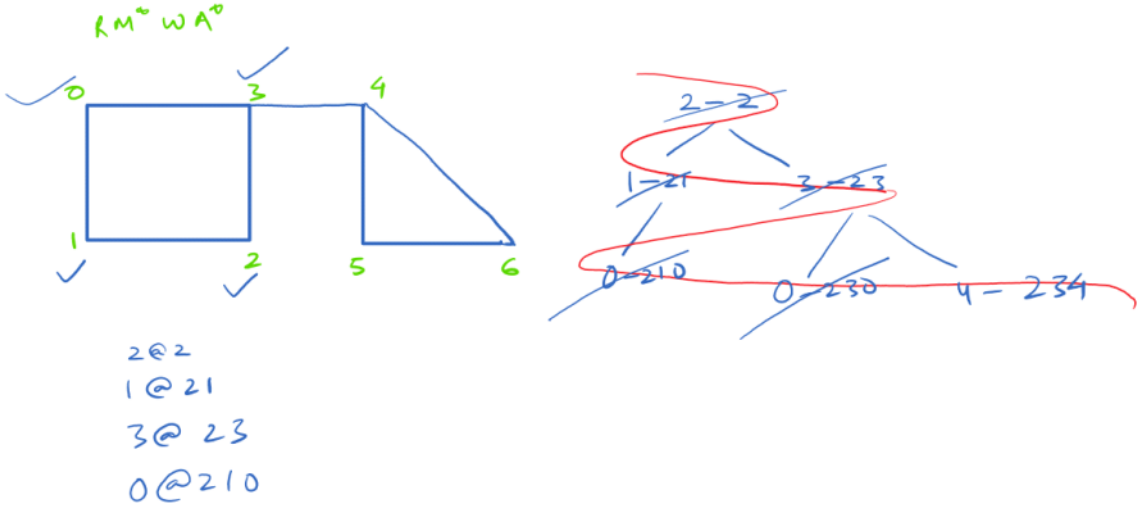
Aur purane raste me naya neighbour add karte jana hai



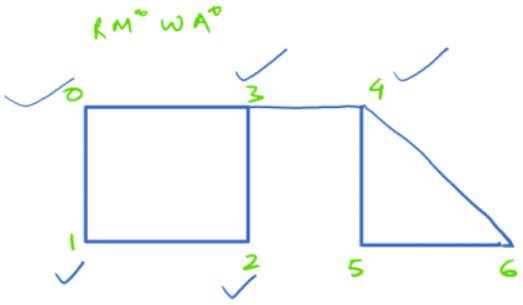


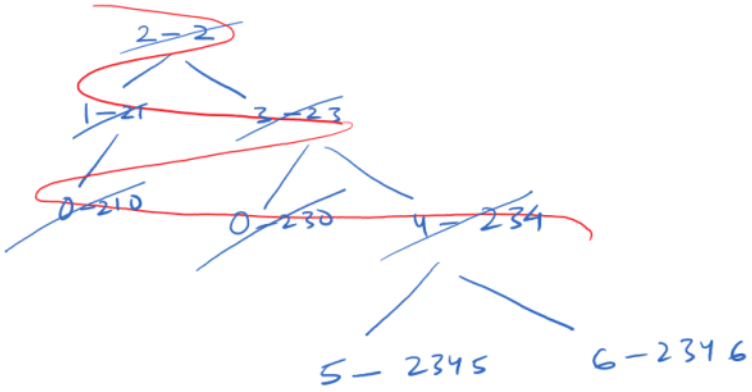




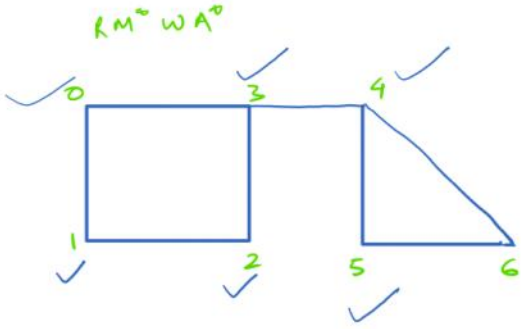


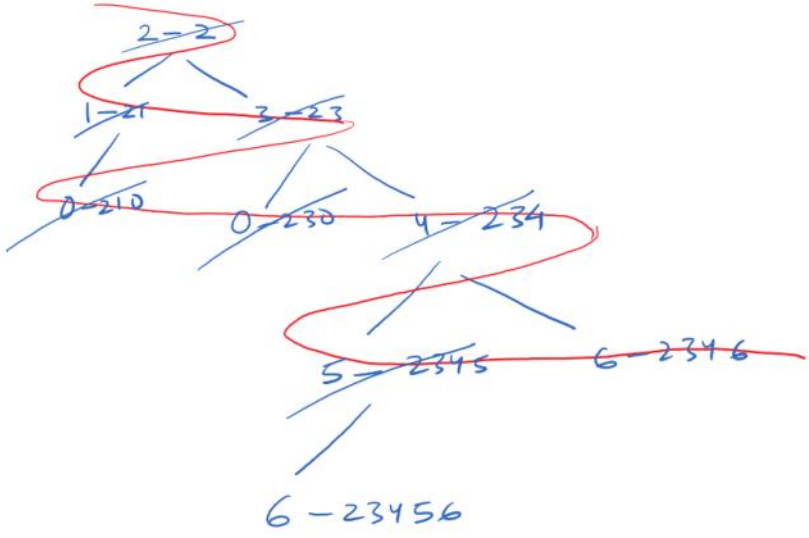
This time we didn't go beyond M* as the vertex 0 was already marked, so don't move ahead.



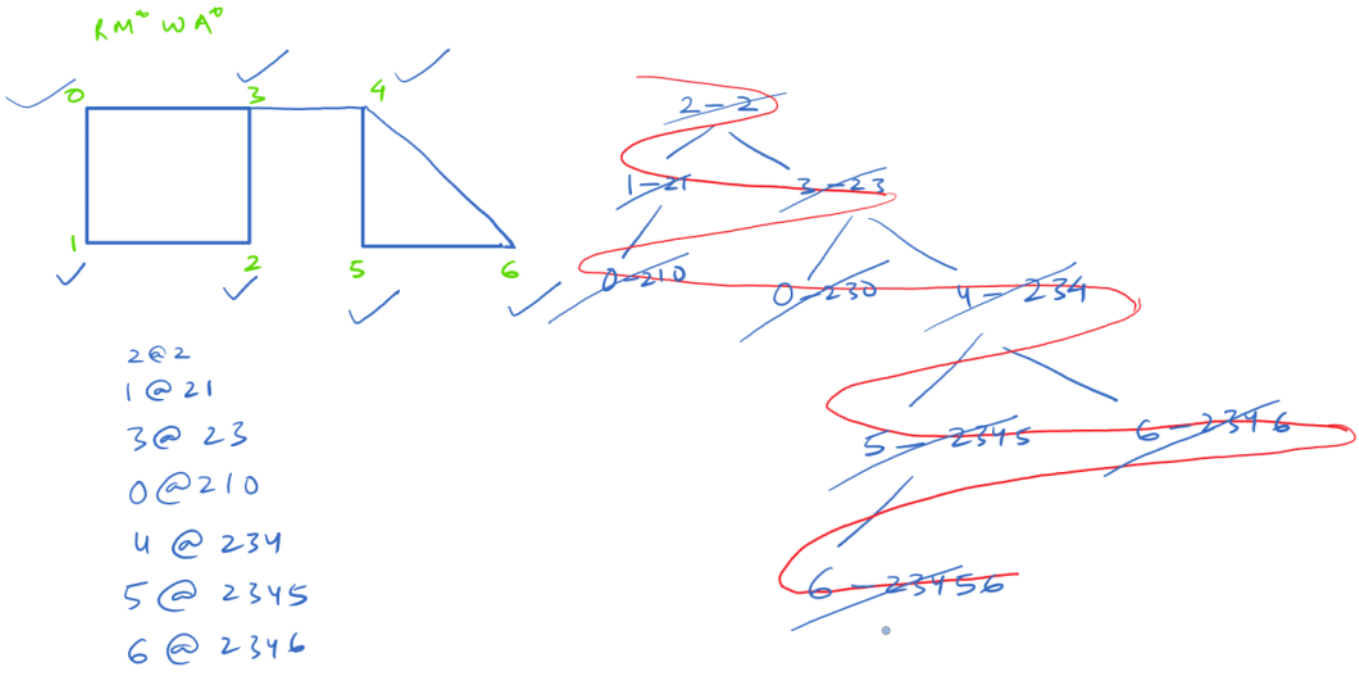


2@2
1@21
3@23
0@210
4@234



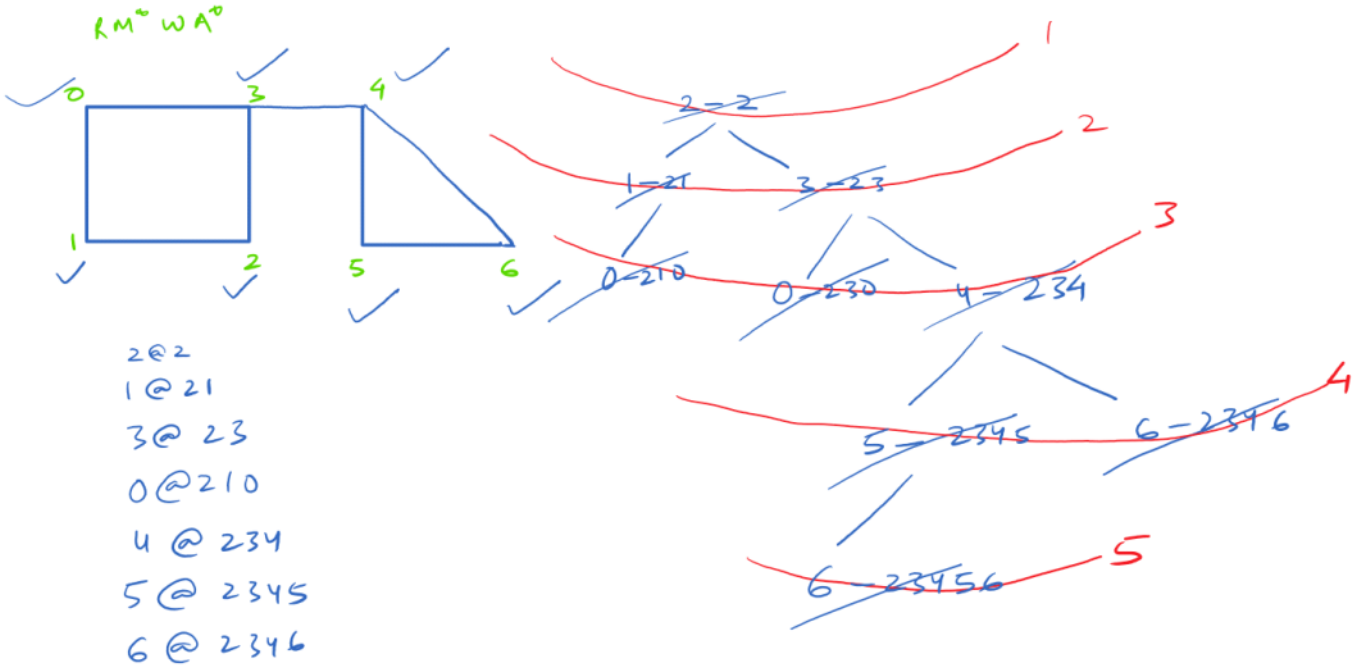


2@2
1@21
3@23
0@210
4@234
5@2345

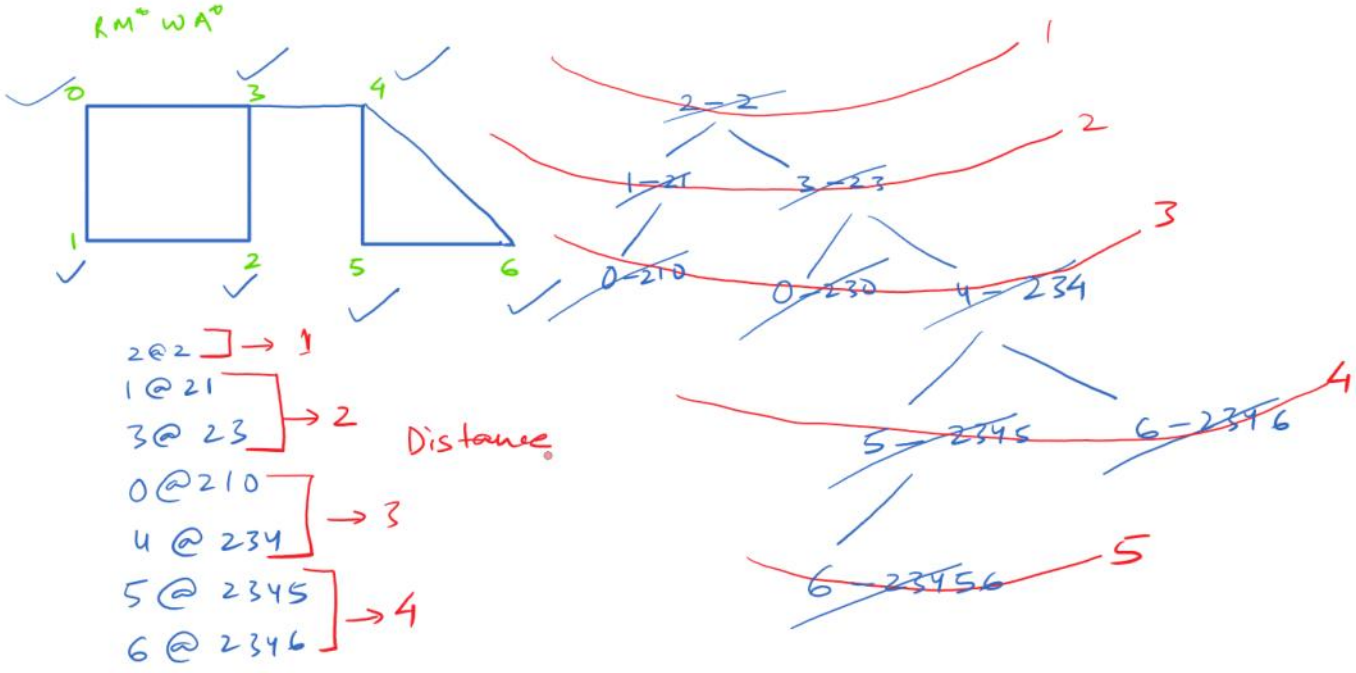


Pehle level pe ek duri wale
Fir do duri wle
Fir teen duri wale
Fir 4 duri wale
Fir 5 duri wale.

Radius me grow karta hai



Notice how the BFS grows in the radius.



BFS

18 April 2022 09:27

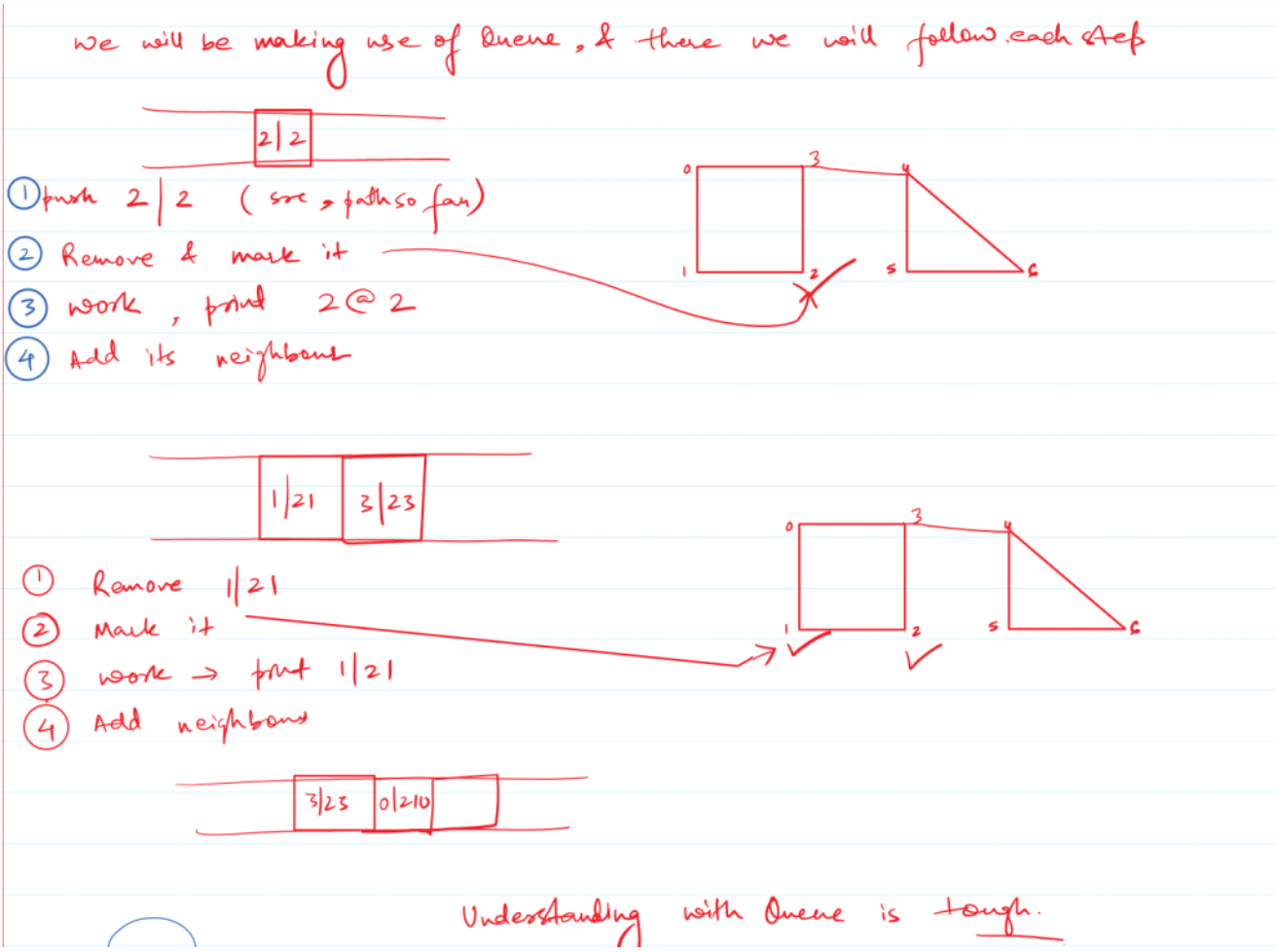
Jaha euler path pe travel karte the, unko hum DFS kehte hai, abhi dekho BFS.

The travel in the Euler path is DFS with recursion like finding all path and hamiltonian path

BFS is like level order where we move by increasing the radius.

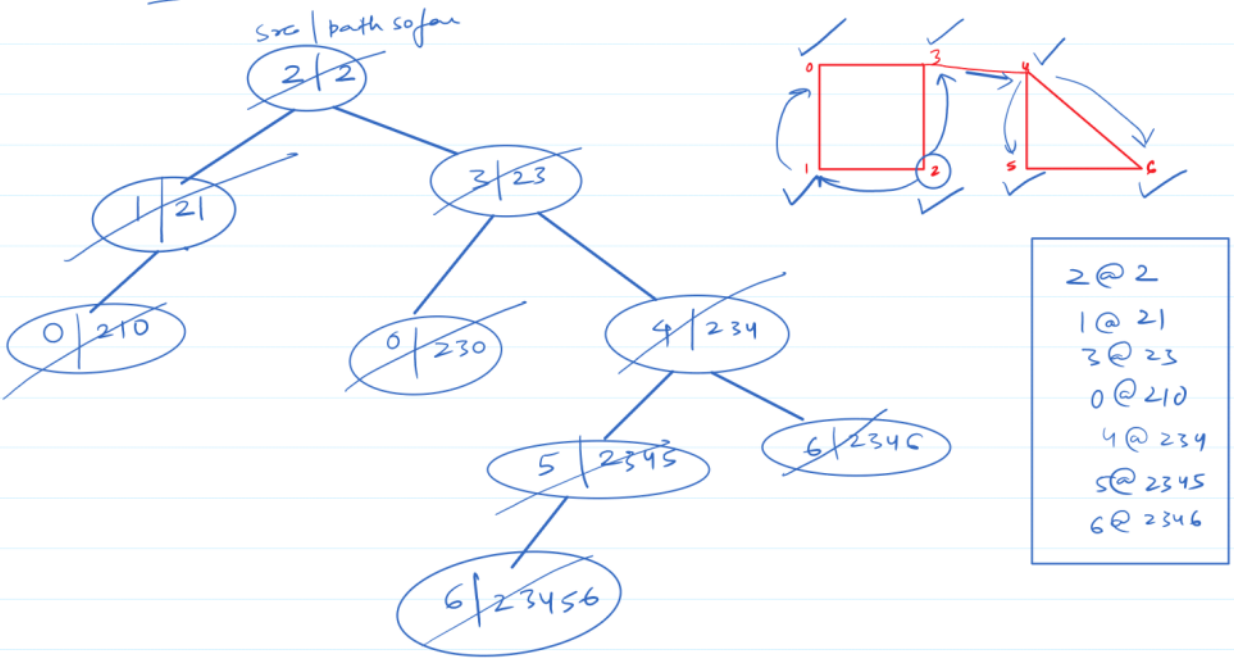
Let's say the given source is 2 and we have to do BFS
So remember this trick: RM*WA*
[Remove, Mark star, Work, Add star]

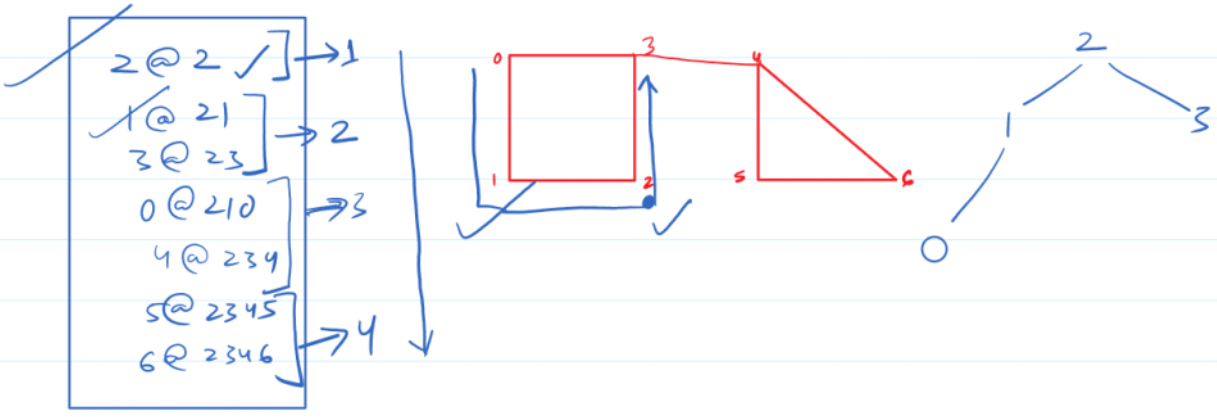
Remove from the queue,
Mark it visited , * means if it already visited, then don't do WA*
Work means print
And Add* means adding neighbours which are unmarked.



Understanding with Queue is tough.

RMWA*





Untitled

20 December 2025 19:34

Why we use queue.removeFirst()

That's a great observation. In Java, the choice between `remove()` and `removeFirst()` often comes down to which **Interface** or **Implementation** you are using for your queue.

Here is the breakdown of why you might see `removeFirst()` instead of the standard `remove()`.

---

1. The Class Type: `Deque` vs. `Queue`

In many modern Java implementations (like BFS algorithms), we use `ArrayDeque` or `LinkedList`. These classes implement the `Deque` (Double-Ended Queue) interface.

- `queue.remove()`: This comes from the standard `Queue` interface. It is designed to always remove from the head.
- `queue.removeFirst()`: This comes from the `Deque` interface. It is more explicit.

Why use `removeFirst()`? When working with a `Deque`, using `removeFirst()` and `addLast()` makes it crystal clear to anyone reading your code that you are treating the collection as a **FIFO (First-In-First-Out)** queue. It differentiates the action from `removeLast()`, which would turn the structure into a Stack.

2. Readability and Intent

While they often do the exact same thing under the hood, `removeFirst()` is more "self-documenting."

Method	Source Interface	Behavior
`remove()`	`Queue`	Removes the head. Throws exception if empty.
`removeFirst()`	`Deque`	Removes the first element. Throws exception if empty.
`poll()`	`Queue`	Removes the head. Returns `null` if empty (safer).
`pollFirst()`	`Deque`	Removes the first element. Returns `null` if empty.

What all to remember for syntax.

1. How to create a pair class with vertex and path so far

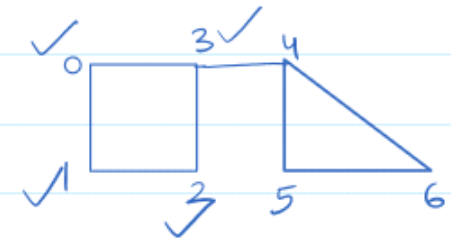
2. How to declare a queue using ArrayDeque
3. Using removeFirst in queue
4. How to get length of ArrayList<Edge>[] graph.
5. This will be helpful in declaring boolean visited array.
6. How to do Mark* (use of continue)

Cycle detection in graph

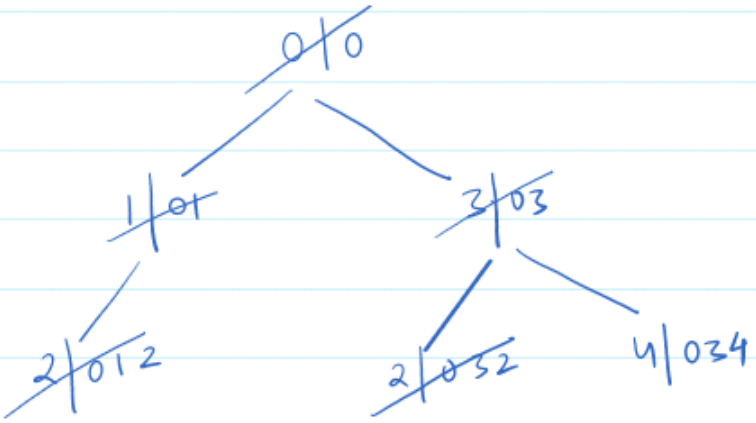
2022 09:56

Cycle detection in Graph

- Can use both BFS & DFS to find the cycle.



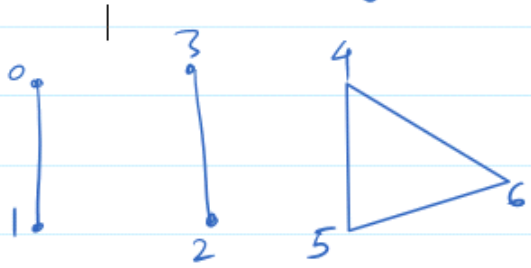
RM* A



As 2 is already Marked.

As 2 is already visited which we can see while trying to mark after removal, it means that there is an existing path from 0 to 2, signifying that there is a cycle present, so return true.

Here is one more thing to consider,
what if graph is having components,
E.g.



Then BFT, when applied on 0 will not go for other components, so,
we must go through each non-visited node in order to check for a cycle.

https://github.com/AlgoMagnet/0or1/blob/main/Graph/IsCyclic.java

Untitled

23 December 2025 15:13

What is a Bipartite Graph?

A graph is bipartite if you can divide its vertices into two sets (e.g., Red and Blue) such that **no two adjacent vertices have the same color.** * **Key Fact:** Every non-cyclic graph is bipartite. If there is a cycle, the graph is bipartite **only if the cycle length is even.**

---

https://github.com/AlgoMagnet/0or1/blob/main/Graph/IsBipartite.java

---

Logic Breakdown (RMWA)

1. **Remove:** Take the node out of the queue.
2. **Mark:** * If the node is already colored, check if its existing color is the same as the color we intended to give it (`rem.color`).
   - If the colors **mismatch**, it means we reached this node via two different paths that require different colors (an **odd cycle**). The graph is not Bipartite.
3. **Add:** Add neighbors to the queue and pass the **inverted color** (`1 - rem.color`).
   - If current is 0, neighbor gets 1.
   - If current is 1, neighbor gets 0.

Summary Checklist

- **Even Cycle:** Graph is Bipartite.
- **Odd Cycle:** Graph is NOT Bipartite.
- **No Cycle:** Graph is Bipartite.

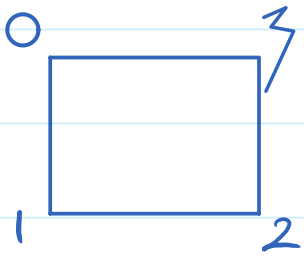
Bipartite

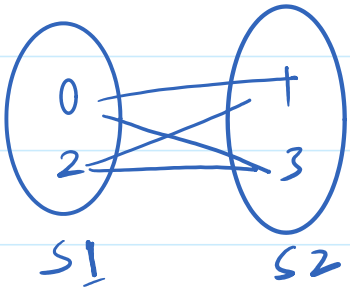
18 April 2022 11:39

Is graph bipartite?

Bipartite graph definition:
If it is possible to divide vertices of graph into
two mutually exclusive (dono set ka intersection kuch nahi)
and exhaustive subsets (union lene pe sare aa jaye)

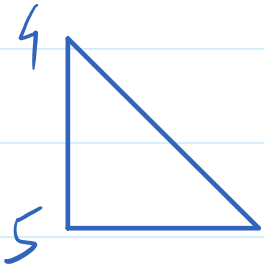
Such that all the edges are across sets, no edges within the set.





over here

→ All edges are across the set and not within the set.

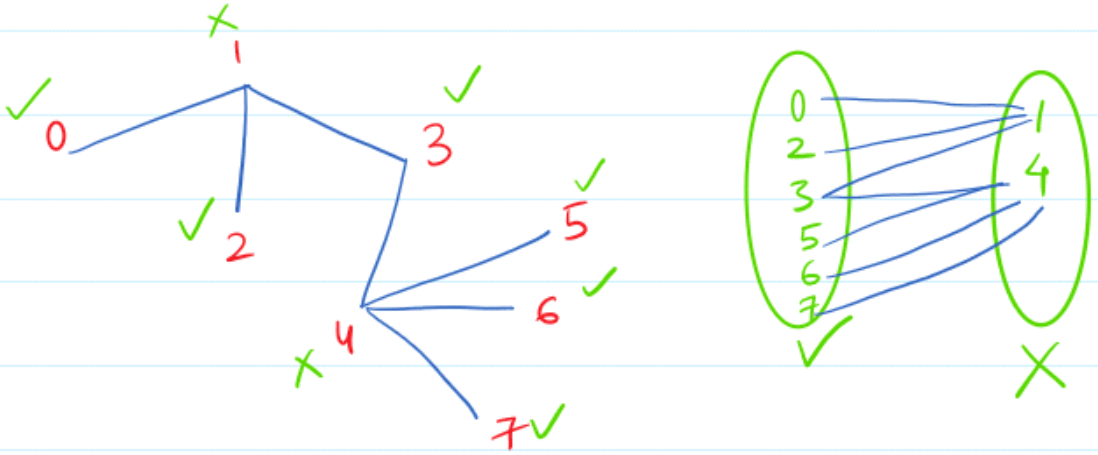


It is not bipartite as vertices can't be divided into two subsets where edges would not be only across edges.

18 April 2022 11:53

If there is no cycle, then bipartite,

if there is cycle,
Then if it is even length: then bipartite,
But if it is odd length: then not bipartite.



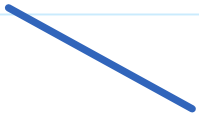
So making each alternate level as (tick) and (cross), while moving in BFT (breadth first traversal), we can arrange the vertices in two subsets and so can prove this bipartite.

In short

Acyclic graph → Bipartite

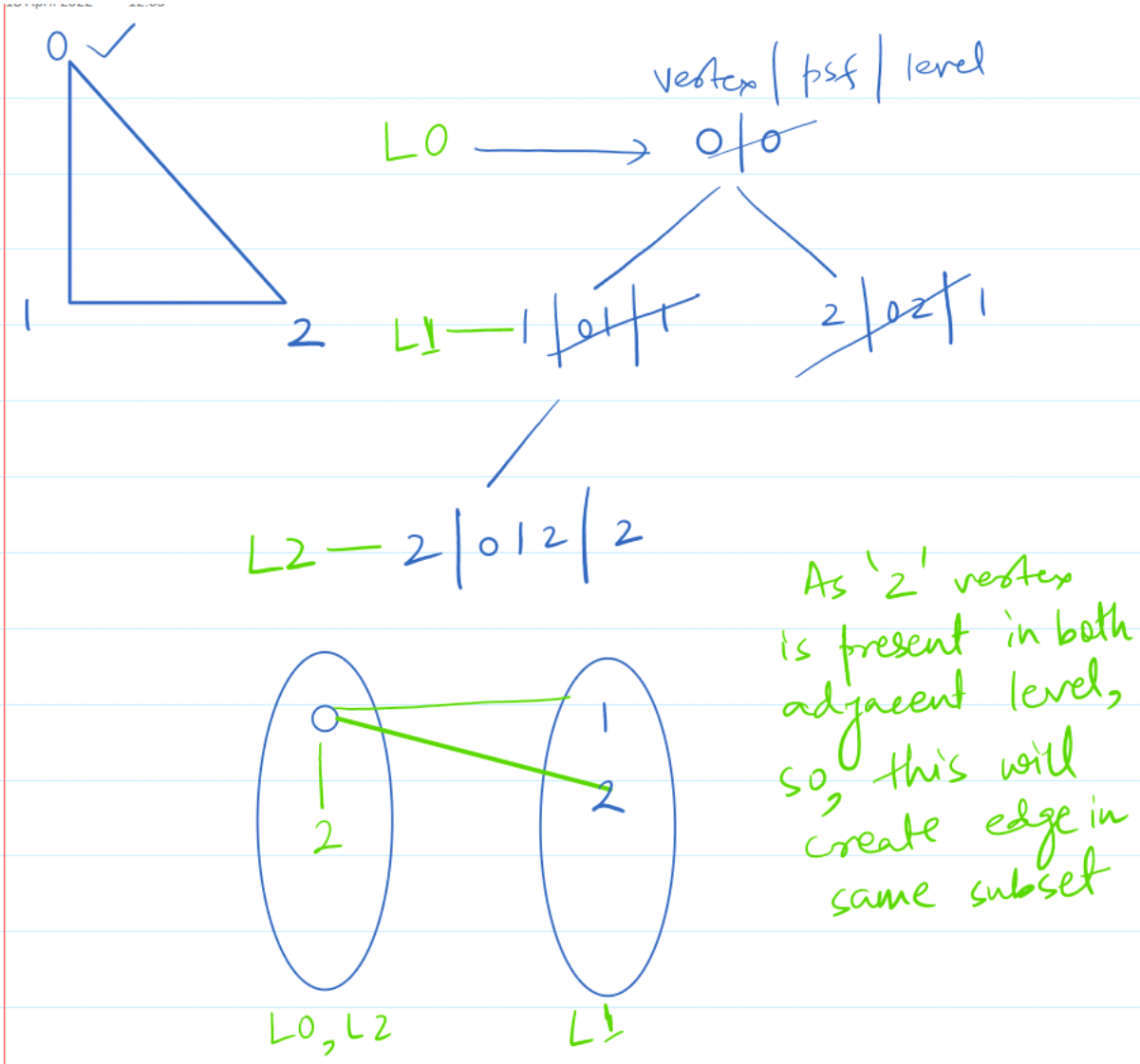
cyclic graph → Even len → Bipartite

→ Odd len → Not Bipartite



Untitled

18 April 2022 12:03



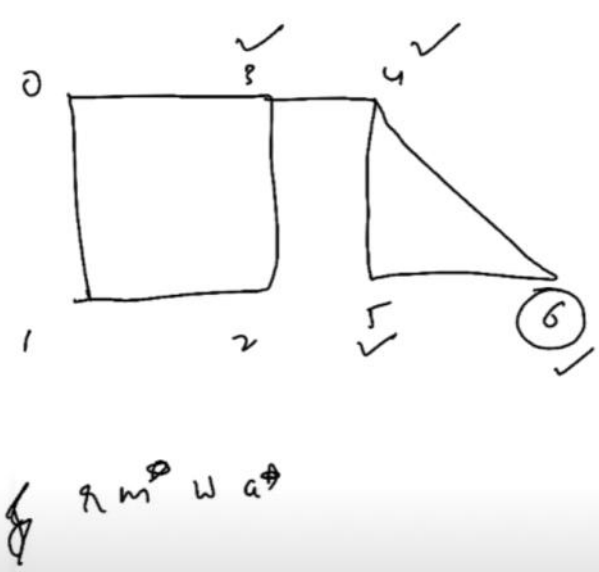
So the vertices should be not in adjacent level, but should be alternate in order to fulfill the criteria.

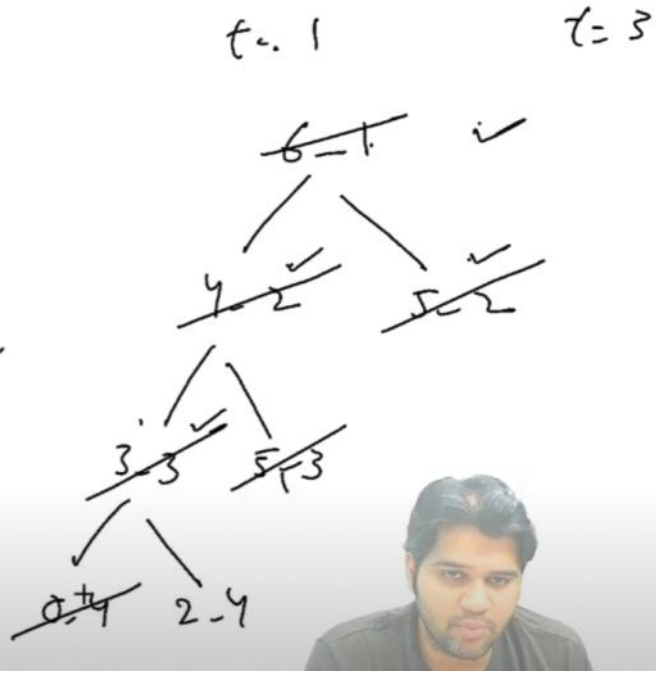
Untitled

29 May 2022 11:35

Spread infection

Padh ke pata nahi chalta ki BFS lagana hai, yahi samajhna hai

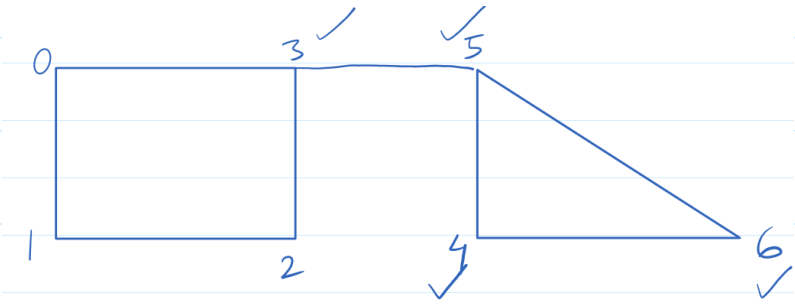


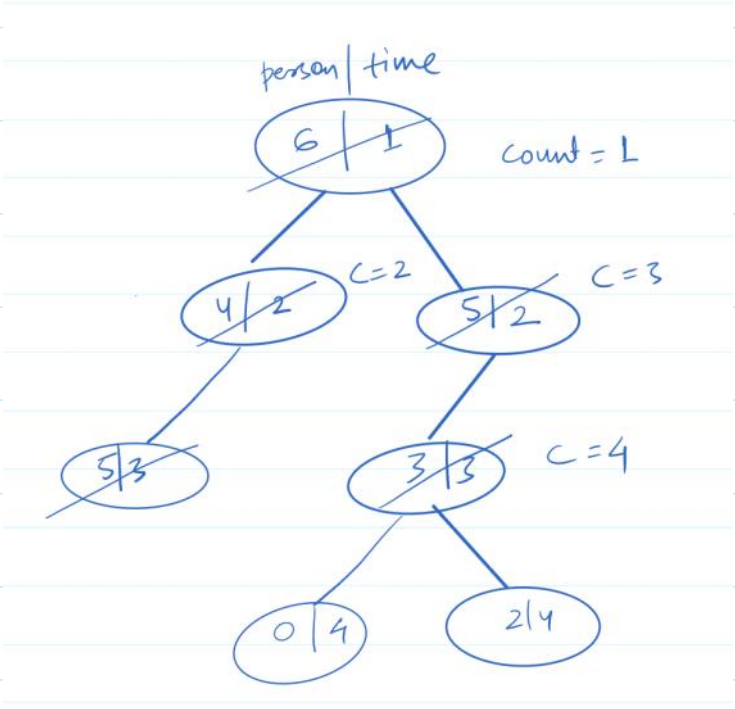


Spread infection

18 April 2022 14:14

Given that 6 is infected at t=1, then find
out how many people would be infected at t=3






```
public static Pair {
    int v;
    int time;

    Pair(int v, int time){
        this.v = v;
        this.time = time;
    }
}
```

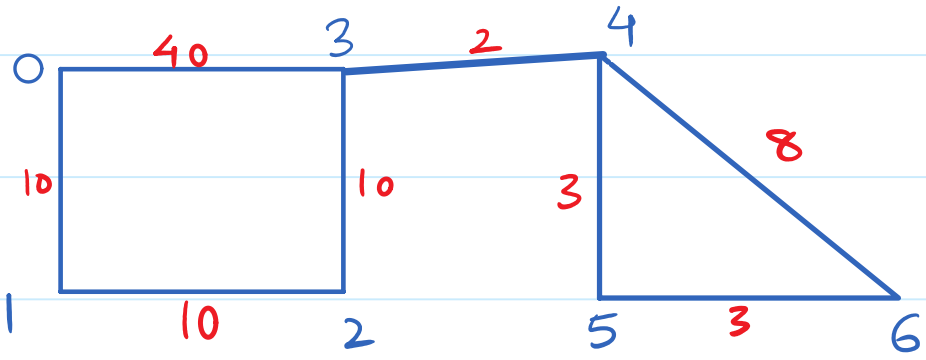
```
// write your code here
ArrayDeque<Pair> q = new ArrayDeque<>();
q.add(new Pair(src, 1));
int[] visited = new int[vtces];
int count = 0;

while(q.size() > 0){
    Pair rem = q.removeFirst();

    if(visited[rem.v] > 0){
        continue;
    }
    visited[rem.v] = rem.time;
    if(rem.time > t){
        break;
    }

    count++;

    for(Edge e: graph[rem.v]){
        if(visited[e.nbr] == 0){
            q.add(new Pair(e.nbr, rem.time + 1));
        }
    }
}
```

We have made the visited array which contains 0 in the beginning which means not visited, otherwise, we are pushing the time in the visited array with every removal of vertices from the queue.

Basically not making it boolean, and making it int so that it can store the time.

Dijsktra's algorithm

18 April 2022 14:26



Given a weighted graph and the src,
Find shortest path in terms of weight from src to all the vertices.

Pro tip: if you have to find the shortest path in terms of edges, then always use BFT. Why? because it increases in terms of radius...

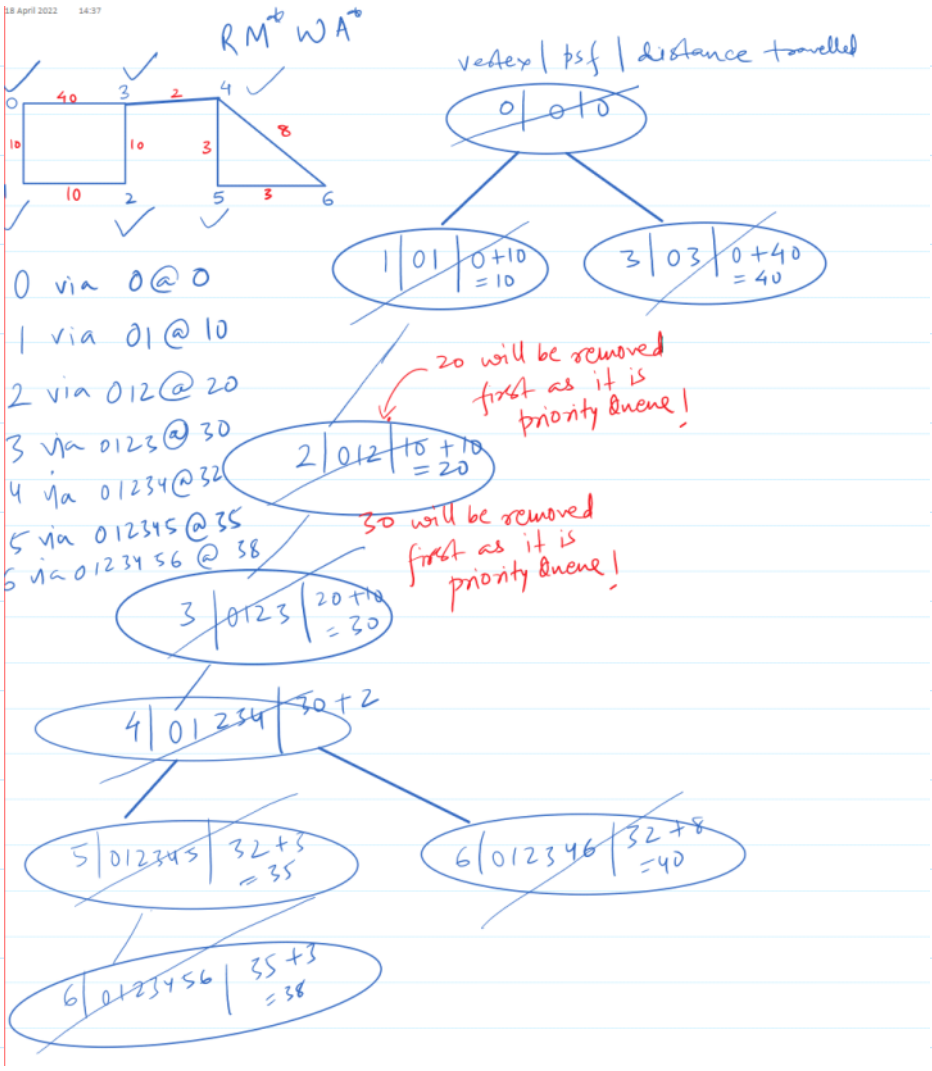
Dijkstra's is similar to BFT but here instead of queue, we will use priority queue.

Priority queue:

I// Priority Queue uses this to bring the pair with 'less value' (weight) to the top

Less value has high priority.

https://github.com/AlgoMagnet/0or1/blob/main/Graph/DijkstraAlgo.java



Last me do 40 bach jayenge, fir hum koi sa bhi pehle nikal sakte hai

Untitled

18 April 2022 14:50

You will have to add the comparable function as you have added pair inside the priority queue, but how will it decide which one is big and which one is small. So we need to implement comparable interface in the class.

In Java, the `Comparable` interface is used to define the **natural ordering** of an object. Without it, if you put two `Pair` objects into a `PriorityQueue`, Java wouldn't know which one is "smaller" or "larger."

Think of `Comparable` as giving your object the ability to say: *"This is how I compare myself to others of my kind."*

---

1. The `compareTo` Contract

The `compareTo` method returns an **integer**, and its value tells Java how to sort the objects:

Return Value	Meaning	Action in Priority Queue
Negative (`< 0`)	`this` is smaller than `other`.	`this` moves toward the front (top).
Zero (`== 0`)	`this` is equal to `other`.	No change in relative priority.
Positive (`> 0`)	`this` is larger than `other`.	`other` stays in front of `this`.

2. Breakdown of your code

In your Dijkstra implementation, you want a **Min-Priority Queue** (the smallest weight should come out first).

Java

```
public int compareTo(Pair other) {
    // If this.wsf is 10 and other.wsf is 40:
    // 10 - 40 = -30 (Negative) -> This pair is "smaller" and gets priority.
    return this.wsf - other.wsf;
}
```

What if you wanted a Max-Priority Queue?

If you wanted the **heaviest** path to come out first, you would simply swap the subtraction:

`return other.wsf - this.wsf;`

---

3. Why use `this.wsf - other.wsf`?

It is a shorthand trick.

- If `this.wsf` is smaller, the result is **negative**.
- If `this.wsf` is larger, the result is **positive**.

A Note of Caution: While subtraction is common in competitive programming, it can cause **Integer Overflow** if you have very large positive and negative numbers. A safer (though more verbose) way to write it is:

Java

```
public int compareTo(Pair other) {
    if (this.wsf < other.wsf) return -1;
    if (this.wsf > other.wsf) return 1;
    return 0;
}
```

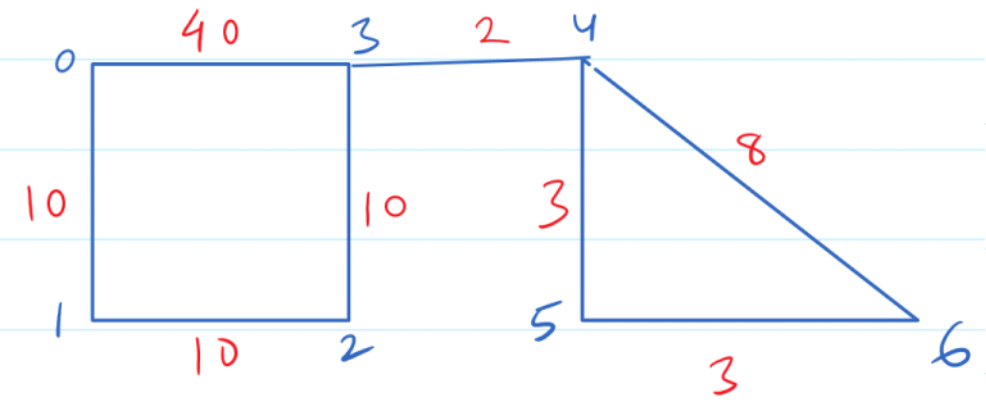
Prim's algorithms

18 April 2022 14:51

https://github.com/AlgoMagnet/0or1/blob/main/Graph/PrimsAlgo.java

This algorithm is based on minimum spanning tree.

Minimum no of wires to connect the PCs



There can be more than one spanning tree,
minimum spanning tree will have minimum sum of the weights.

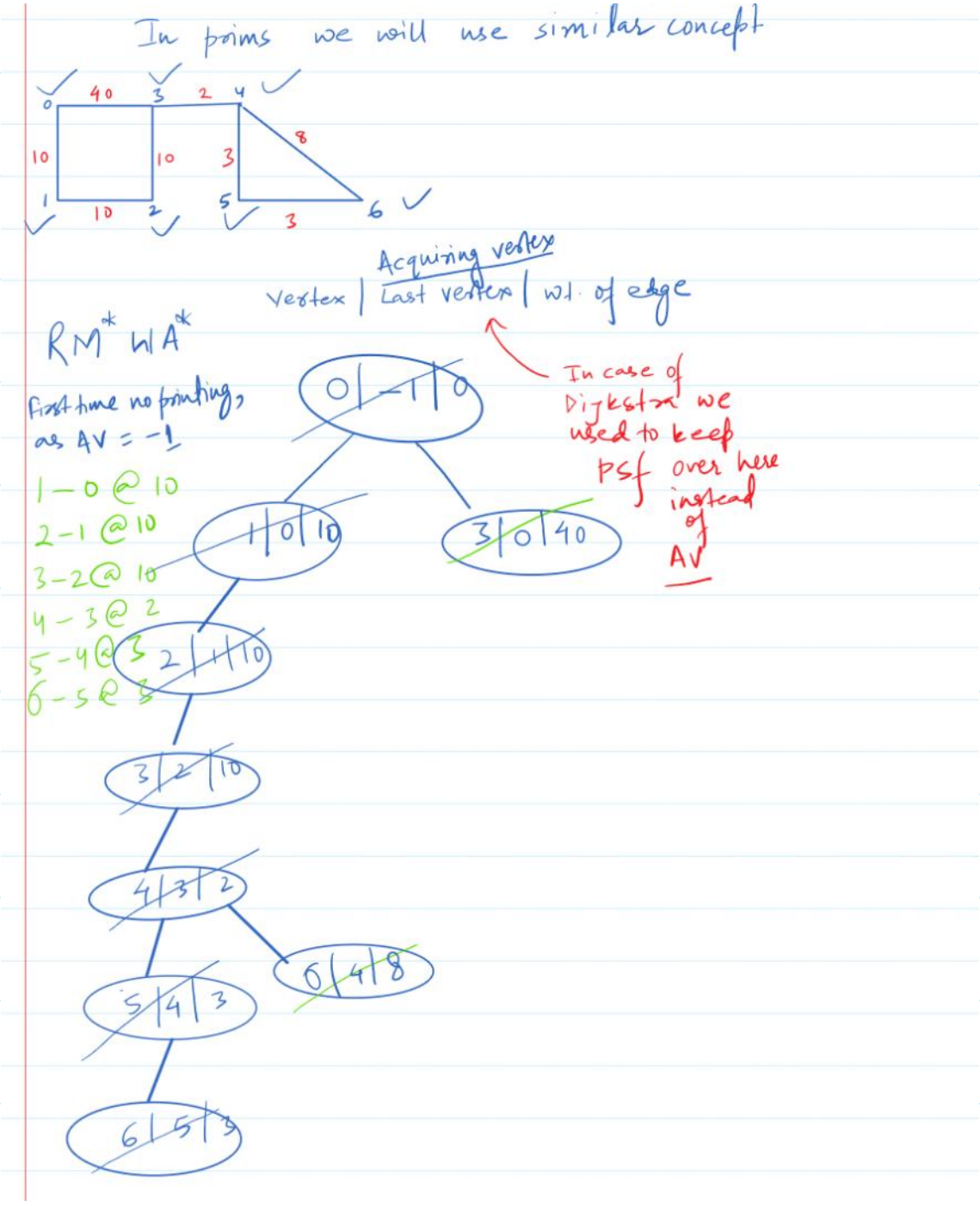
MST is a subgraph
→Tree (acyclic)
→spanning (Means all vertices included)

Untitled

18 April 2022 15:27

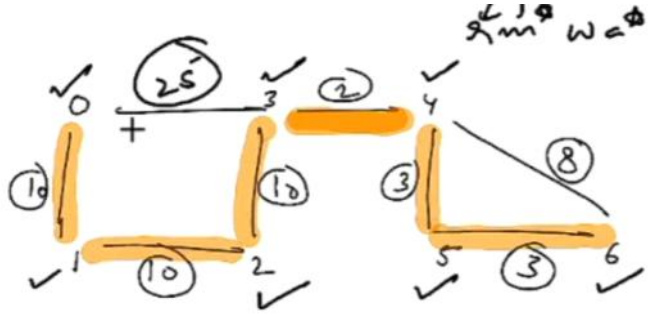
Prim's algorithm is like Dijsktra
Dijkstra was like BFT where we replaced queue with Priority queue.

Pehli vertex imaginary vertex hai -1, jiske beech me kuch nahi hai 0 wt ki edge hai, asliyat me hai nahi

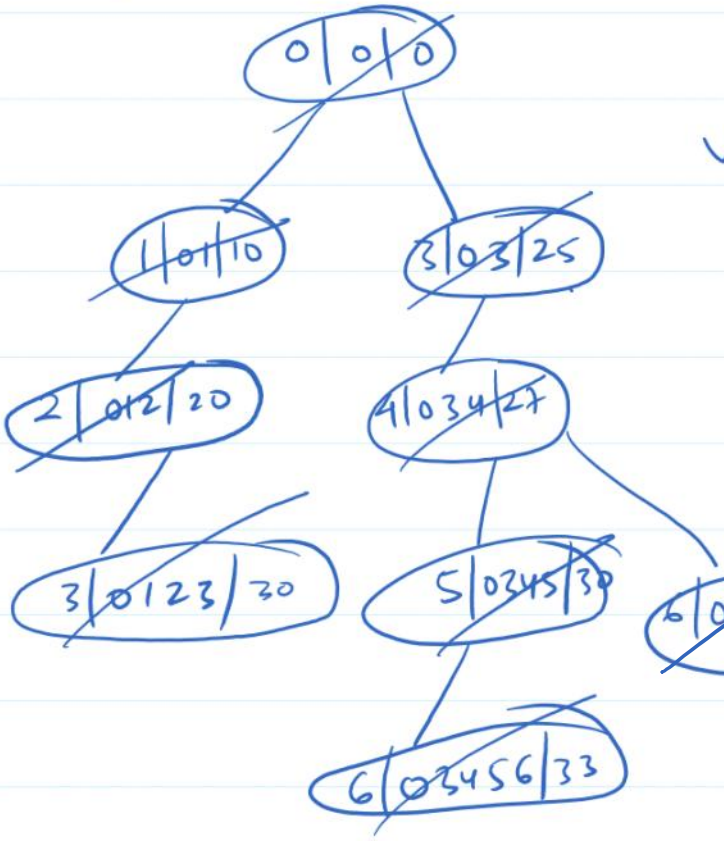


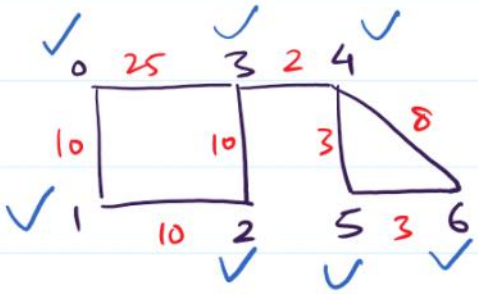
18 April 2022 15:55

Do the dry run of both dijskstras and prims on this graph and see how both are different



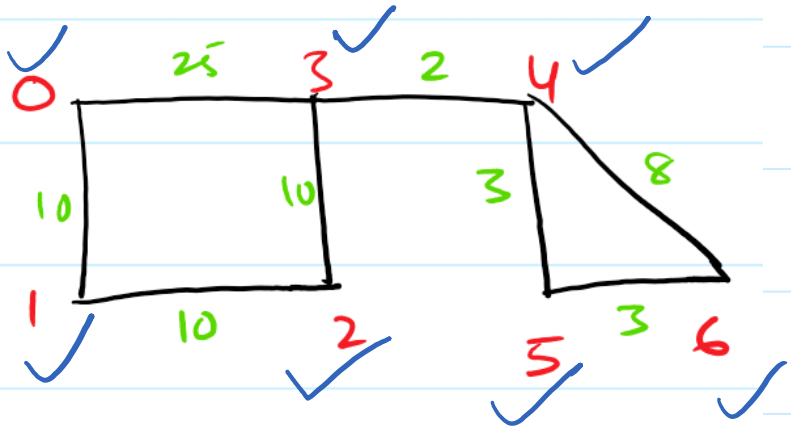
Dijkstra's (RM^b WA^b)



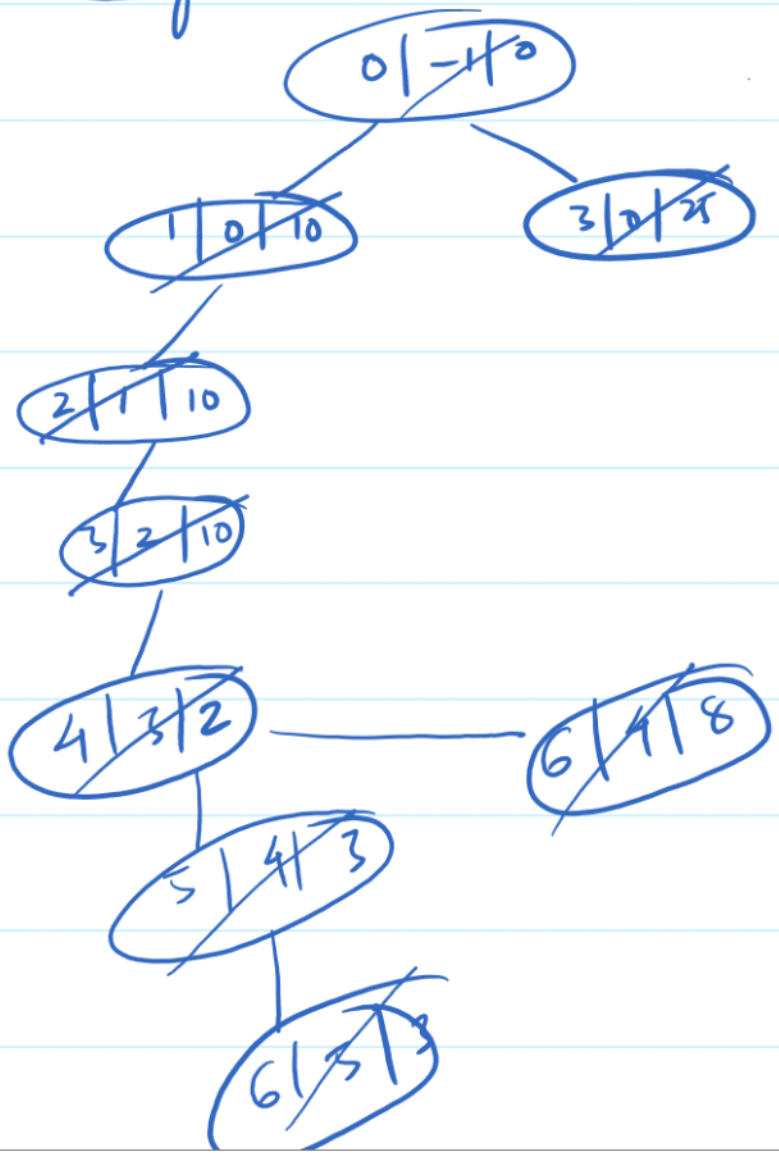


0 via 0@0
1 via 01@10
2 via 012@20
3 via 03@25
4 via 034@27
5 via 0345@30
6 via 03456@33

29 May 2022 14:52



prims Algo



1-0@10
2-1@10
3-2@10
4-3@2
5-4@3
6-5@3

Topological sort

29 May 2022 15:13

1. What is Topological Sort?

Topological Sort is a linear ordering of vertices such that for every directed edge uv (from node u to node v), node u appears before v in the ordering. As your notes emphasize, this only works in a **Directed Acyclic Graph (DAG)**.

- **Directed**: The edges have arrows showing a specific direction.
- **Acyclic**: There are no cycles (you cannot follow the arrows and end up back where you started).

2. The "Work Dependency" Concept

Your notes describe Topological Sort as a way to solve **work dependency** problems.

- **Analogy**: If you have several files to compile and some depend on others, Topological Sort tells you the valid sequence to finish the task.
- **Rule**: If file A is needed to compile file B, file A must be processed first.

3. How the DFS Algorithm Works (from your notes)

The images indicate that we use **Depth First Traversal (DFS)** to achieve this sort:

- **The Process**: We go as deep as possible into the graph.
- **The "Finish" Line**: A node is only "finished" once all the nodes it points to have been visited.
- **The Stack**: Because we want to know which files to compile first, we use a Stack to catch nodes as they finish their recursion.

https://github.com/AlgoMagnet/0or1/blob/main/Graph/TopologicalSort.java

2. The Code Structure: "DFS + Stack"

You can remember the implementation by breaking it into two distinct parts:

Part A: The Manager (Wrapper)

Remember that you need a **Visited Array** and a **Stack**.

- **Why Visited?** To ensure we don't get stuck in loops and only process each node once.
- **Why a Loop?** To handle "Disconnected Components"—nodes that aren't reachable from the first source node.

Part B: The Recursive Scout (DFS)

This is exactly like your `drawTree` function, but with **one extra line at the bottom**:

1. **Mark** current node as visited.
2. **Scout** all neighbors recursively.
3. **Push to Stack** *after* the recursive calls return.

3. The "Post-Order" Trick

The key to remembering *why* we use a Stack is the concept of **Reverse Order**.

- In DFS, the node that has **no dependencies** (nothing left to visit) finishes its function call last.
- If you push it onto a **Stack**, the "last to finish" becomes the "first to be popped".
- **Mnemonic:** *"Finish the dependencies first, then put me on the stack"*.

Iterative DFS

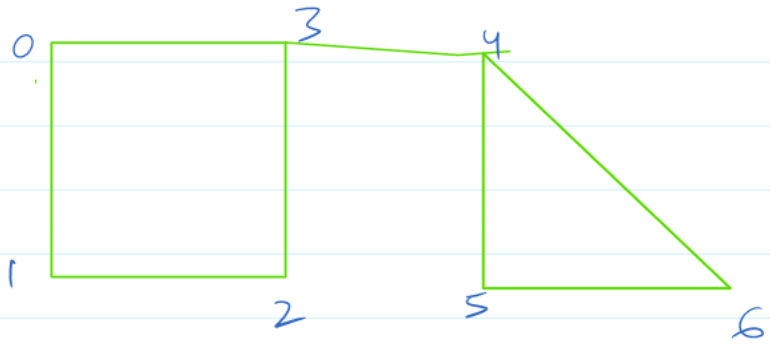
18 April 2022 17:12

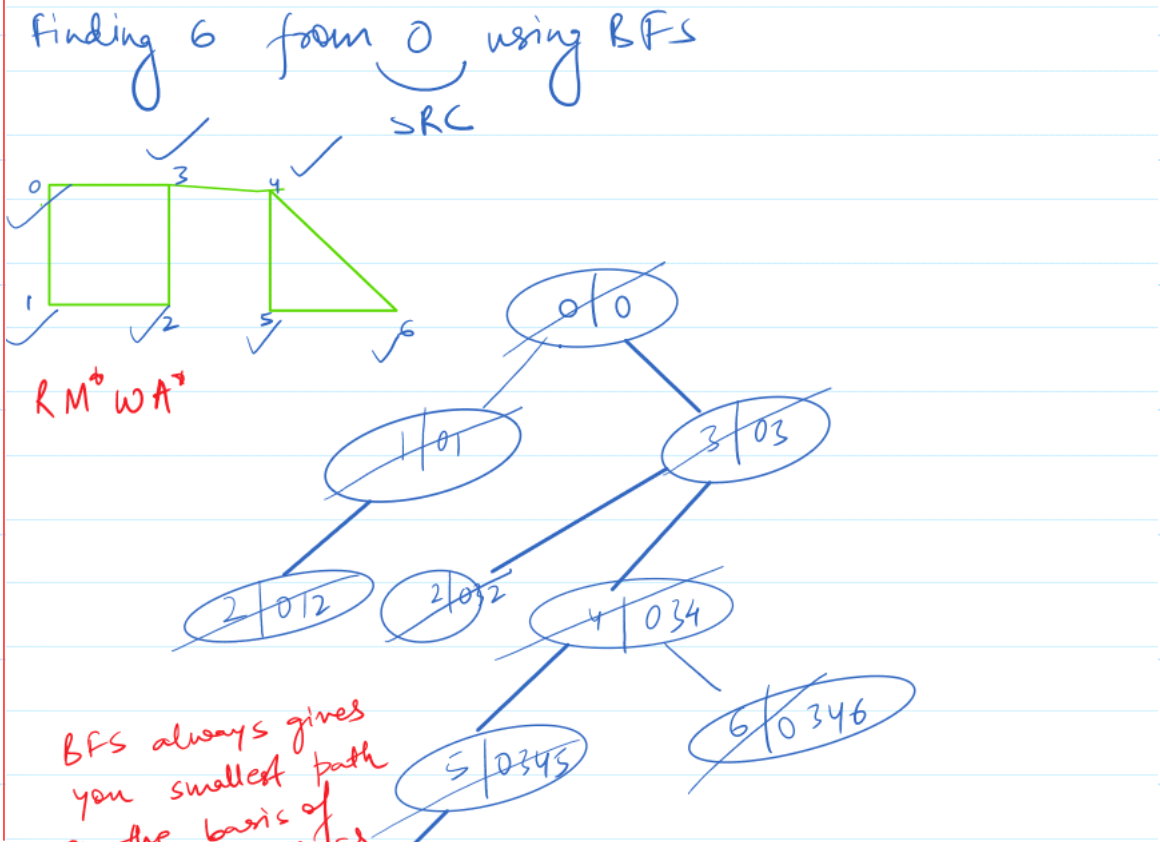
When we had recursive DFS, then why do we need iterative one.

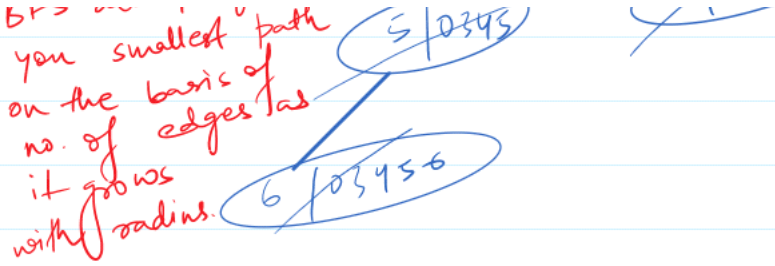
To convert your **BFS** code into an **Iterative DFS** code, you essentially replace the **Queue** with a **Stack**. While BFS explores level by level (radius-based), Iterative DFS explores as deep as possible along one branch before backtracking.

Key Differences in Iterative DFS

- **Data Structure:** Uses a `Stack` instead of an `ArrayDeque` (Queue).
- **Behavior:** The **RMWA** (Remove, Mark, Work, Add) logic remains the same, but because a stack is **Last-In, First-Out (LIFO)**, the last neighbor added to the stack is the first one processed, creating the "depth-first" effect.

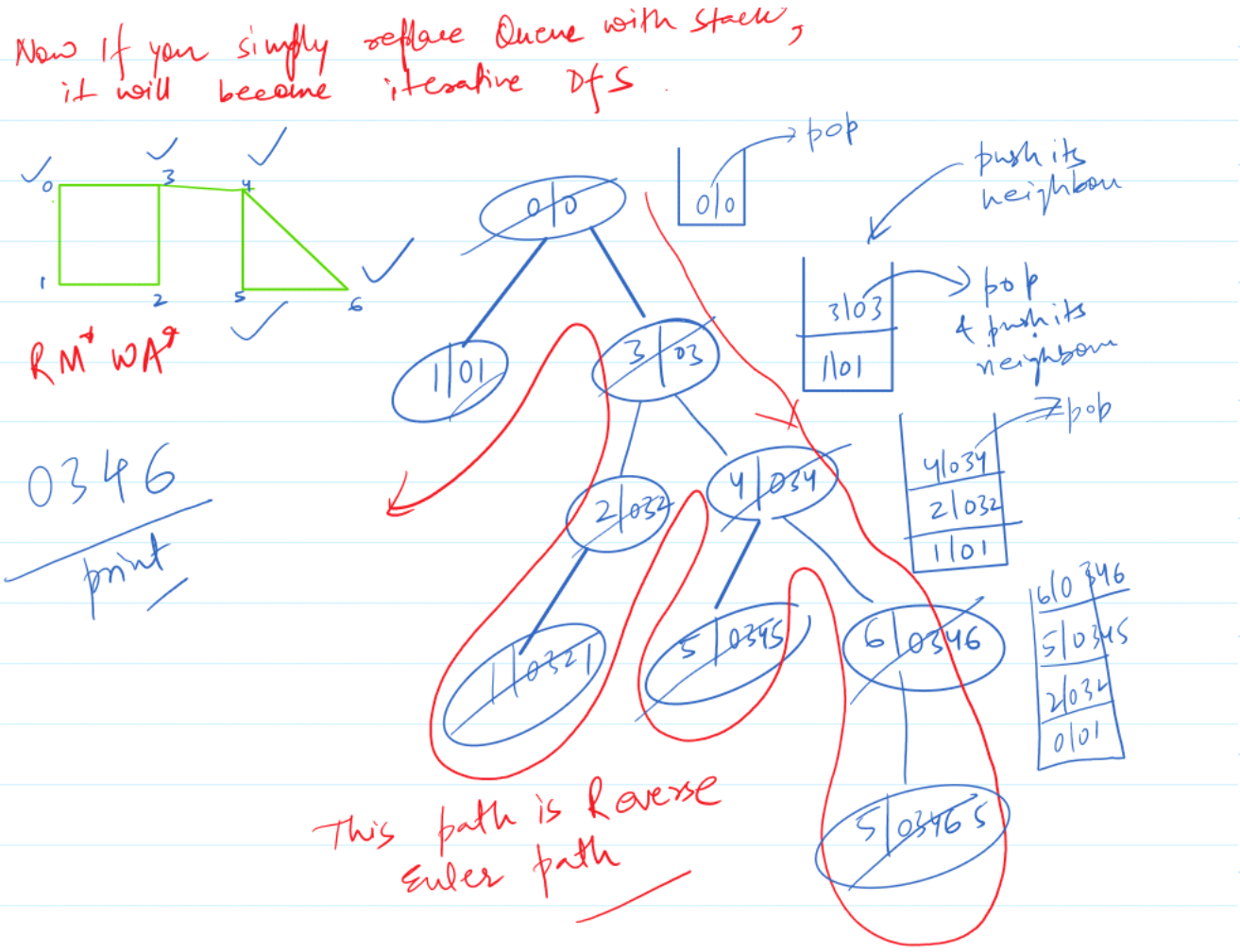





you smallest path
on the basis of
no. of edges as
it grows
with nodes.
5 0345
6 03456

Untitled

18 April 2022 17:18



Jab wo 3 pe tha to uske paas do option thi, ki wo apne sibling pe jaye ya apne child pe jaye, par stack ke use ke karan pehle wo children pe gaya.

To wo deep gaya pehle. Sibling pe nahi jayegi, apne children pe jayegi, isi ko kehte hai reverse euler.

[Iterative Depth First Traversal | Iterative DFS in Graphs | DFS Code]


LEVEL- 1
ITERATIVE DEAPTH
FIRST TRAVERSAL
DFS CODE
Pepcoding
Sumeet Malik
www.nados.pepcoding.com
For better experience visit

Untitled

18 April 2022 17:26

When the graph is having many vertices but is linear.

So for let's say 100000 vertices, in case of recursion we will use the stack frame of memory. It is function call stack and recursion will make new frames in call stack

So instead of that in iterative dfs, we are simply making stack in the heap section of the memory, thus we are not using the function call stack.

So basically we are simulating the stack in the heap which was happening in the function call stack.