Tutorial 09 — Graph Traversal

CS2040S Semester 1 2022/2023

By Wu Biao, adapted from previous slides

Set real display name

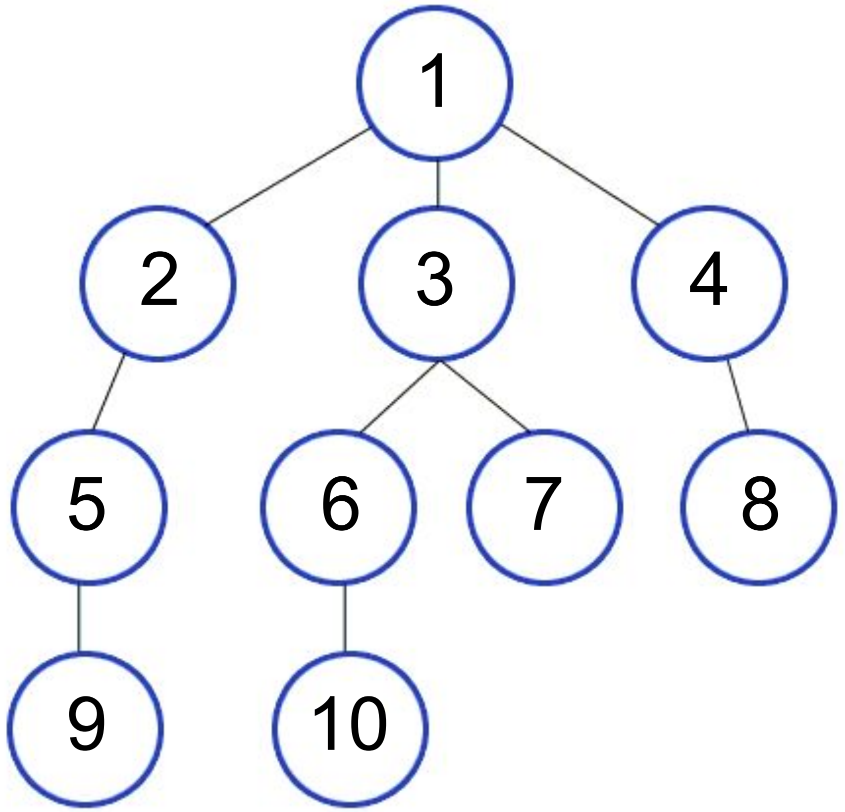
https://pollev.com/rezwanarefin430

BFS Review

Breadth First Search (BFS)ー Level-order Traversal

Likes to proceed *radially*!

It will go deeper **only when it has cleared the current level/radius**.



Note: Order of visitation is marked by number within vertex.

Breadth First Search — Level Order Traversal

For BFS, we use a queue to capture the sequence of vertices to visit!

```
queue<int> q;
visited[src_id] = true;
q.push(src_id);
while (!q.empty()) {
    int v_id = q.front(); q.pop();
    cout << v_id << endl; // Visit op
    for (int & nb_id: adjList[v_id]) {
        if (visited[nb_id]) continue;
        q.push(nb_id);
        visited[nb_id] = true;
    }
}
```

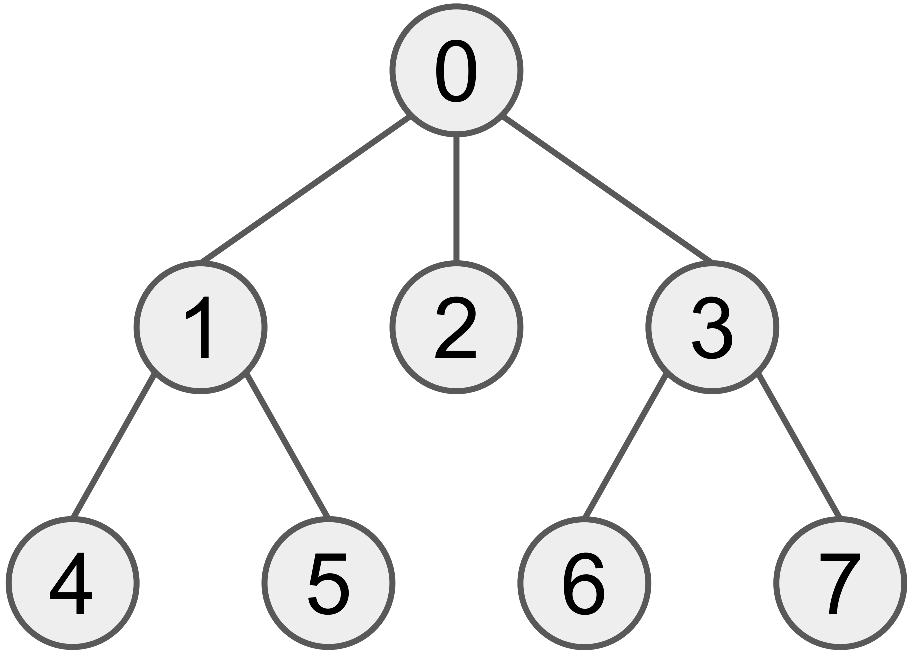
Breadth First Search—Level Order Traversal

There are other ways of writing the logic for visitation tracking, such as this:

```
queue<int> q;
q.push(src_id);
while (!q.empty()) {
    int v_id = q.front(); q.pop();
    if (visited[v_id]) continue;
    visited[v_id] = true;
    cout << v_id << endl; // Visit op
    for (int & nb_id: adjList[v_id]) {
        q.push(nb_id);
    }
}
```

Breadth First Search — Level Order Traversal

```
queue<int> q;
visited[src_id] = true;
q.push(src_id);
while (!q.empty()) {
    int v_id = q.front(); q.pop();
    cout << v_id << endl; // Visit op
    for (int & nb_id: adjList[v_id]) {
        if (visited[nb_id]) continue;
        q.push(nb_id);
        visited[nb_id] = true;
    }
}
```



Output:

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

Review Harder Topics

Question 1

In decreasing order of difficulty, the following topics are usually found to be challenging for students:

1. https://visualgo.net/en/dfsbfs?slide=7-1 to 7-3: Detecting back edge/cycle in the graph.
2. https://visualgo.net/en/dfsbfs?slide=7-10 to 7-11: Toposort, revisited in Q3 & Q4.
3. https://visualgo.net/en/dfsbfs?slide=7-6 to 7-9: Check your understanding about the $O(V(V+E))$ versus just $O(V+E)$ analysis again.
4. https://visualgo.net/en/dfsbfs?slide=8: All other more advanced graph traversal topics not under the main focus of CS2040/C are optional and will not be part of CS2040/C final assessment.

DFS Complexity Analysis — Test yourself!

Claim: The time complexity of DFS is $O(V+E)$.

True or False?

DFS Complexity Analysis — Test yourself!

Claim: The time complexity of DFS is $O(V+E)$.

True! Visits each vertex once, and each edge once.

DFS Complexity Analysis — Test yourself!

Claim: When finding connected components (CC), we perform DFS once per connected component.

True or False?

DFS Complexity Analysis — Test yourself!

Claim: When finding connected components (CC), we perform DFS once per connected component.

True!

We will revisit this later.

DFS Complexity Analysis — Test yourself!

Claim: There are at most V CCs in a graph

True or False?

DFS Complexity Analysis — Test yourself!

Claim: There are at most V CCs in a graph

True!

Extreme case when every vertex is an isolated component.

i.e. graph has no edges.

DFS Complexity Analysis — Test yourself!

Claim: When we run DFS to detect every CC, the total time complexity is $\mathrm{V} \times \mathrm{O(V+E)} = \mathrm{O(V^2+VE)}$ because we run DFS on every vertex in the graph.

True or False?

DFS Complexity Analysis — Test yourself!

Claim: When we run DFS to detect every CC, the total time complexity is $\mathrm{V} \times \mathrm{O(V+E)} = \mathrm{O(V^2+VE)}$ because we run DFS on every vertex in the graph.

False!

It is still $\mathrm{O(V+E)}$! Why? Recall that we will only run DFS once per CC. After DFS ran in a CC, all its vertices will be marked as visited. Each vertex and each edge is still visited O(1) times.

Relevant discussion

I have V integers distributed across C arrays.

$1 \leq C \leq V$

Sort each array using $O(N \log N)$ sort. What is the total time complexity?

$O(V \log V)$ because $a \log a + b \log b \leq (a + b) \log (a + b)$.

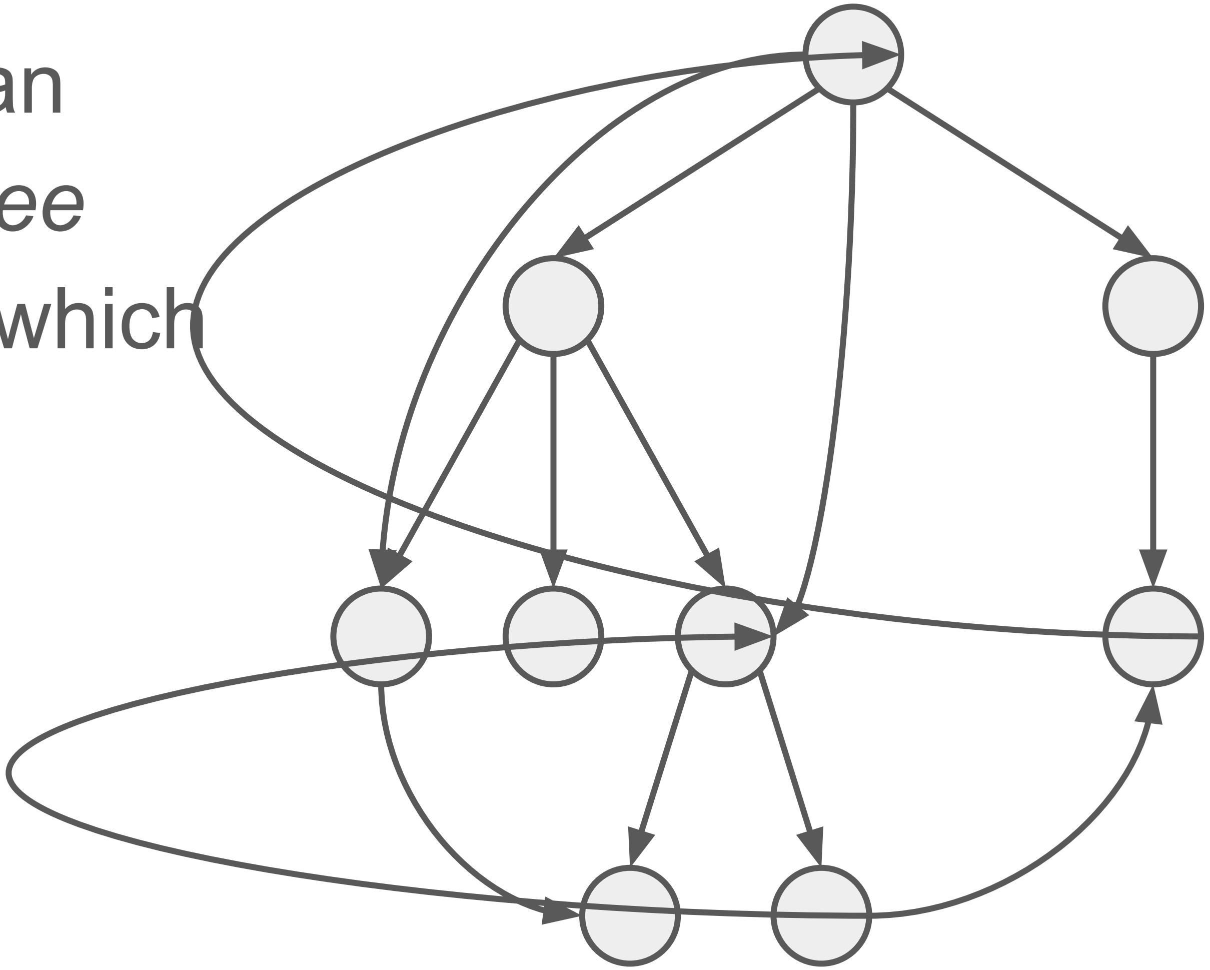
BFS Complexity Analysis — Test yourself!

Does all the previous analysis for DFS also applies to BFS?

DFS Spanning Tree

[Credits: SG IOI Training Team]

DFS visitation order entails an implicit **directed** *spanning tree* **rooted** at the source vertex which DFS ran from.

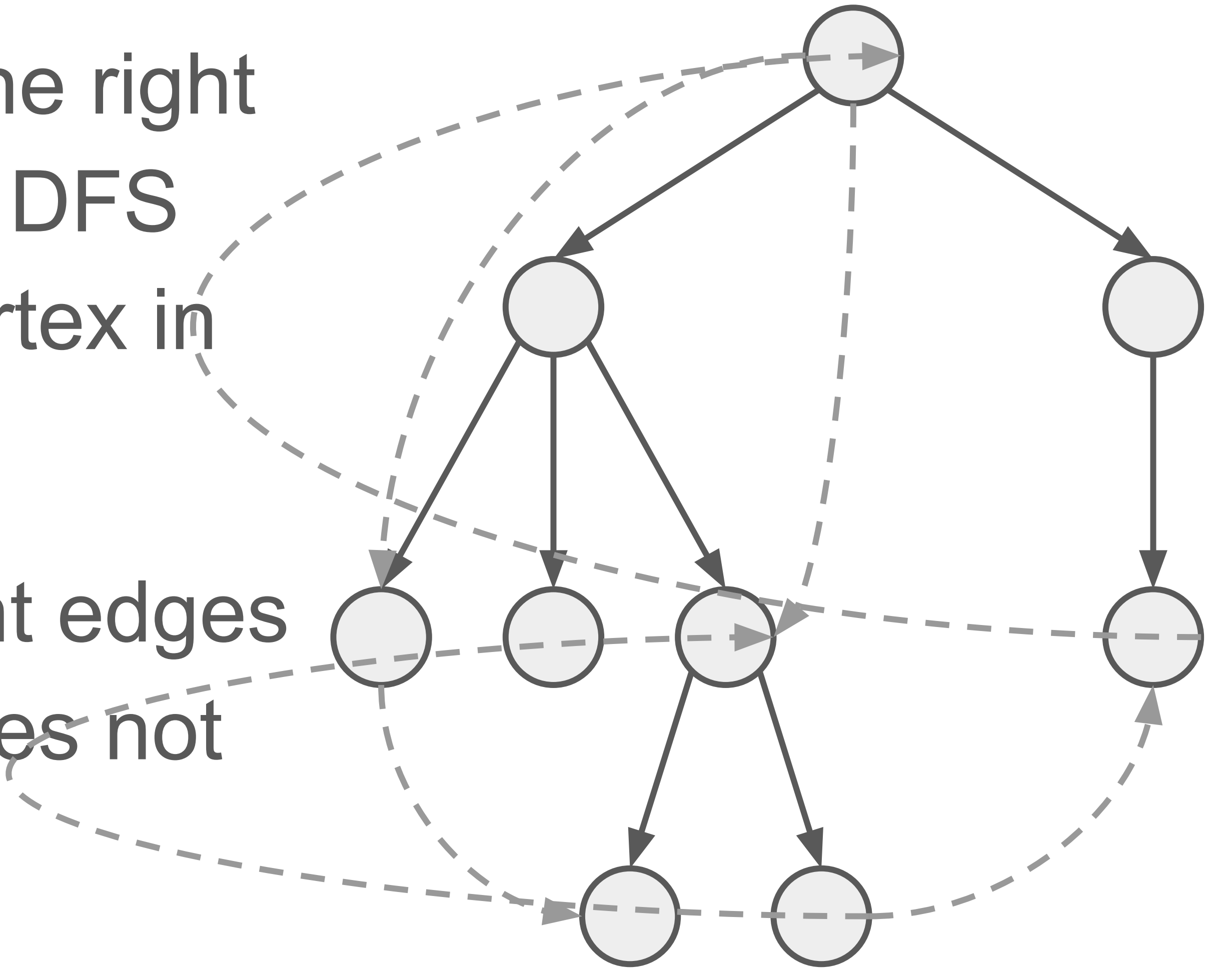


DFS Spanning Tree

[Credits: SG IOI Training Team]

For instance, we can have the right DFS spanning tree if we run DFS starting from the topmost vertex in the graph.

Note: Dashed lines represent edges untraversed by DFS. i.e edges not in the spanning tree



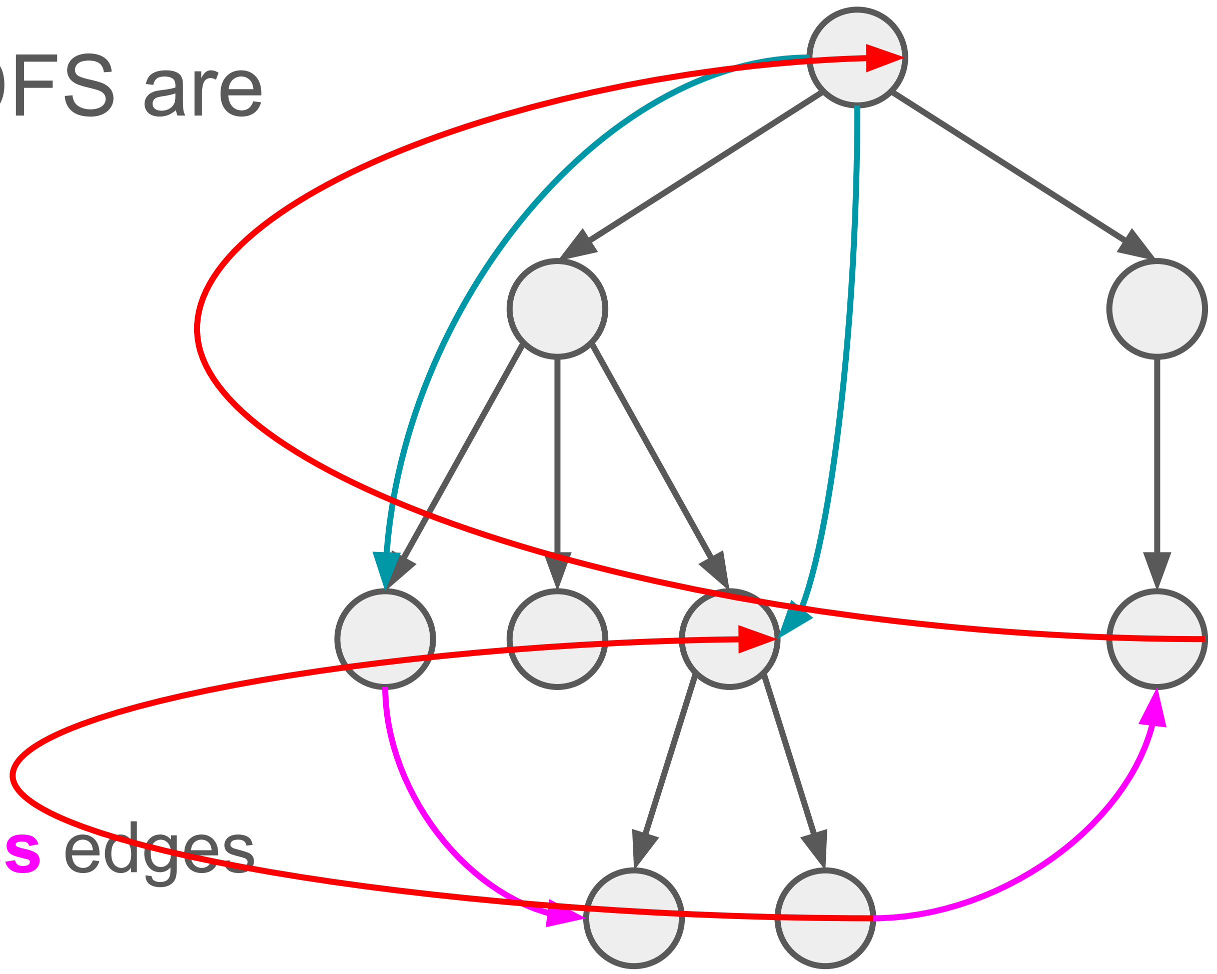
DFS Spanning Tree

[Credits: SG IOI Training Team]

The untraversed edges by DFS are classified into:

- **Forward edges**
- **Cross edges**
- **Back edges**

Note: You don't have to know the significance of **Forward** and **Cross** edges in CS2040C.



DFS Application — Cycle Detection

What you need to know in CS2040C:

- The only edges that can create cycles are **back edges.**
- **Back edges** always point from a vertex to one of its ancestors
- Therefore to detect cycles in graph:
    - Run DFS.
    - Check for **back edges**
        - If exists, then there is at least a cycle.
        - If not exists, then there are no cycles.

DFS Application — Vertex States

During DFS, we maintain 3 visitation states for each vertex:

1. Unvisited : Vertex is still “untouched” by DFS
   Visiting : Vertex is “touched” but not yet completed by DFS
   Visited : Vertex is “touched” and completed by DFS

At any level of the DFS recursion:

- Ancestors of current vertex will be in Visiting state
- “Touched” non-ancestors will be in Visited state

DFS Application—Back Edge Detection

Back edge always point from current vertex in DFS recursion to one of its ancestors.

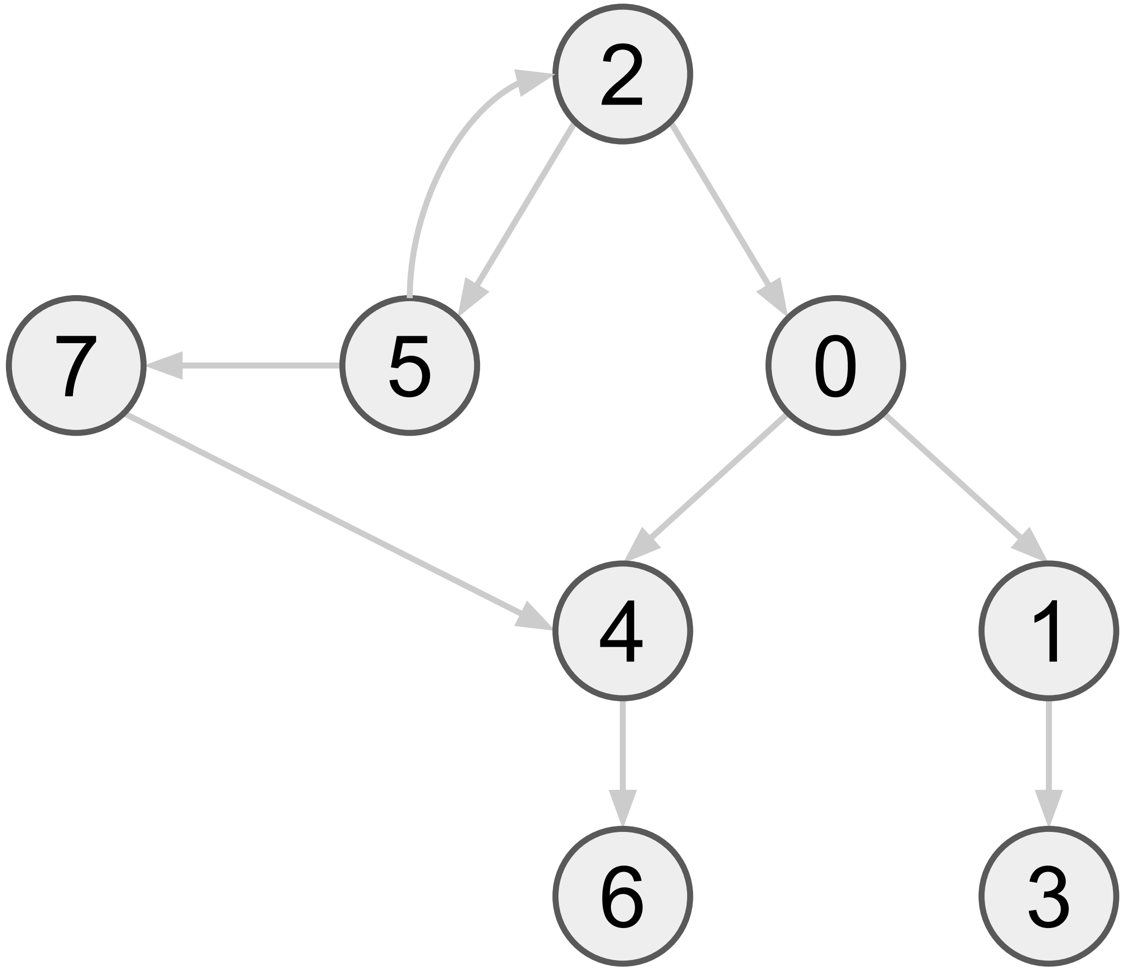
Therefore to detect **back edge**, it suffices to check if the edge go from Visiting state → Visiting state.

DFS Application — Cycle Detection

Why do we need Visiting state?

In other words why can't **back edges** go from Visited → Visited?

To find out, we shall trace out DFS on the graph starting from vertex 2.



Click here for the graph in VisuAlgo

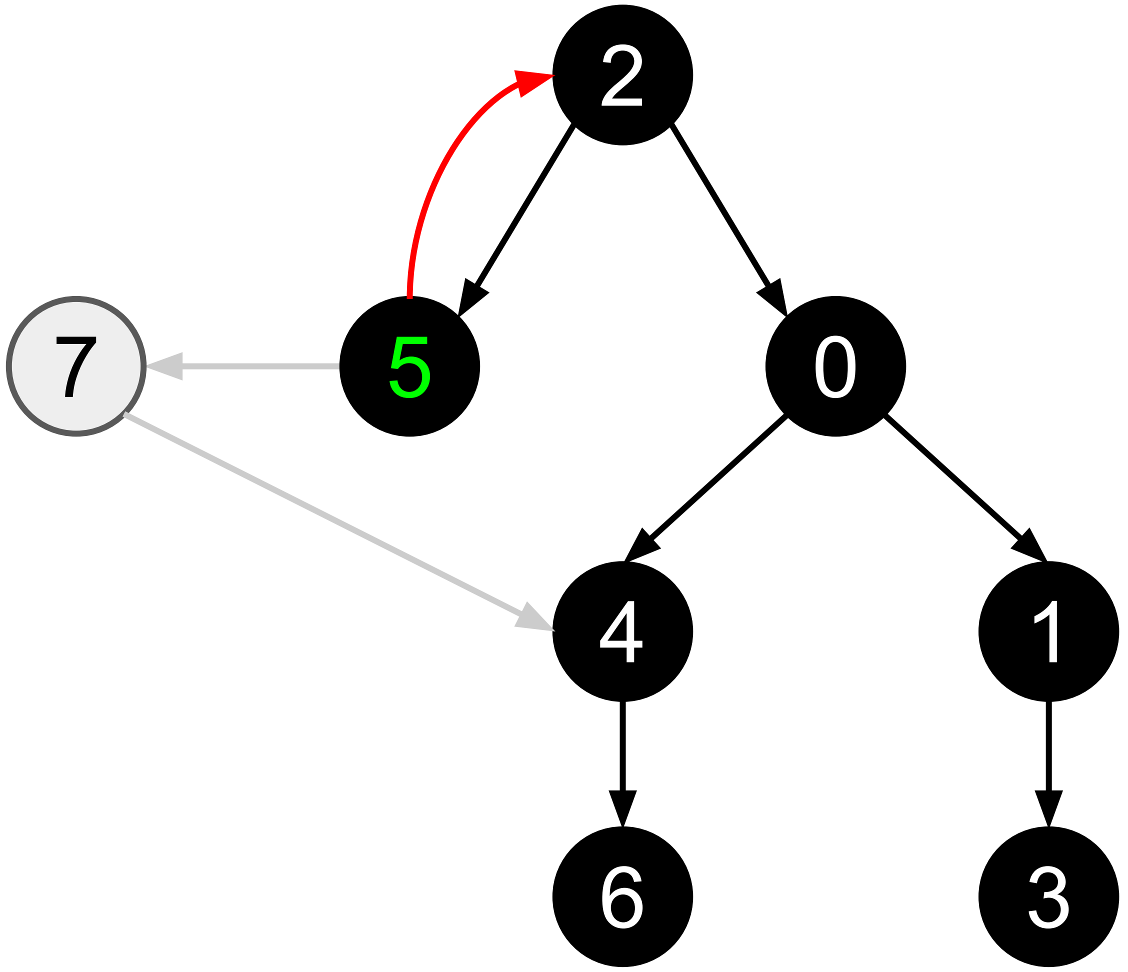
DFS Application — Cycle Detection

Without Visiting state:

Vertex 5: Current Vertex

Vertex 2: Visited

Since 2 is Visited state, we conclude that 5→2 is a **back edge**, which is indeed the case. So far so good...



DFS Application—Cycle Detection

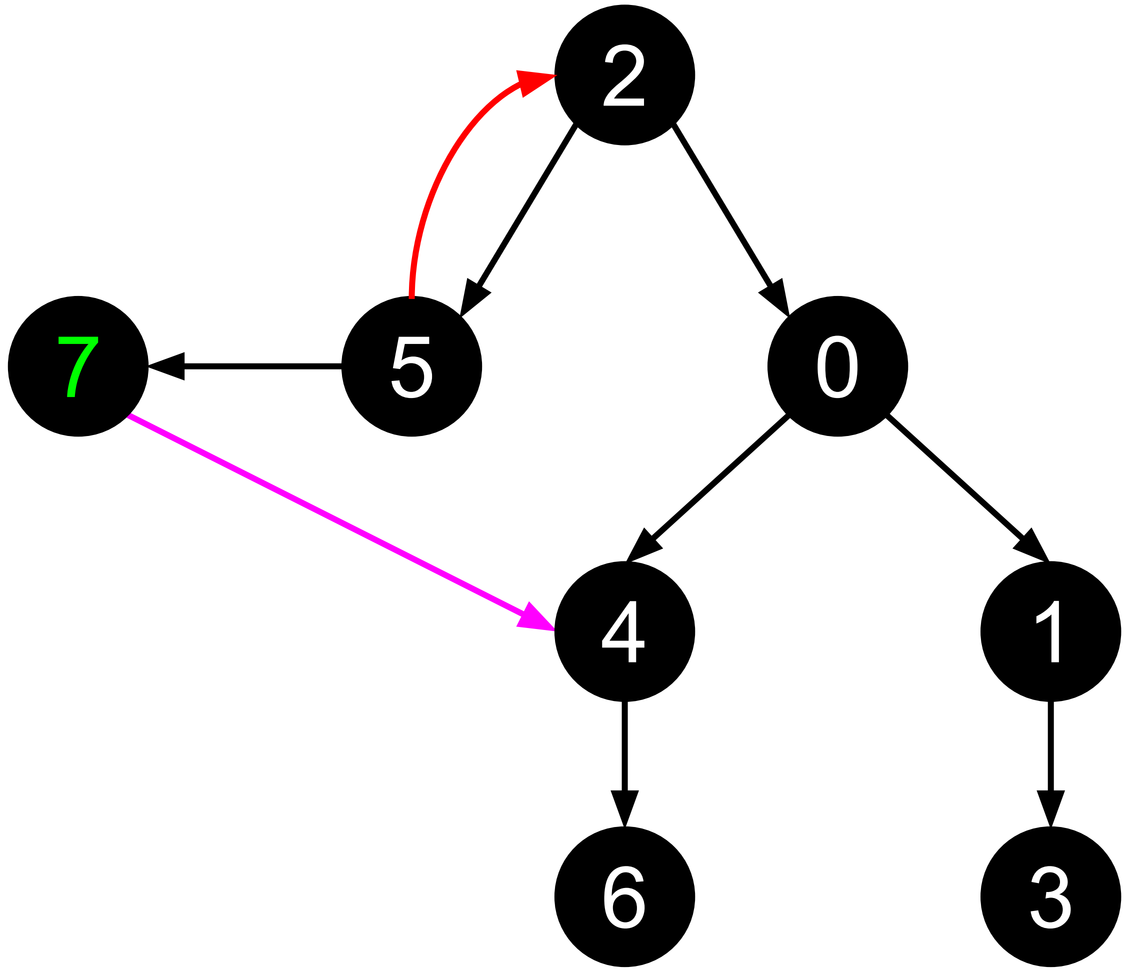
Without Visiting state:

Vertex 7: Current Vertex

Vertex 4: Visited

Since 4 is Visited state, we conclude that 7→4 is a **back edge**.

This is wrong! It's a **cross edge**!

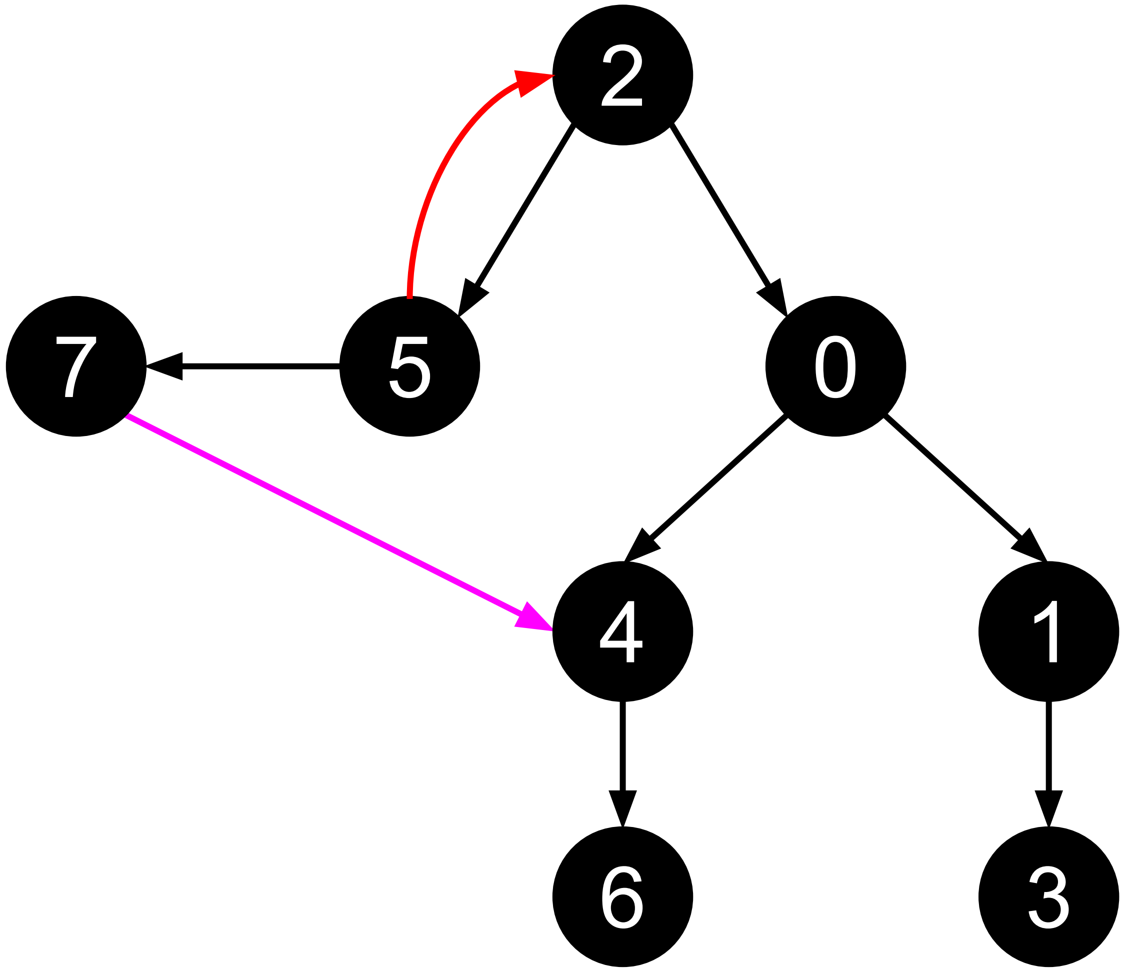


DFS Application — Cycle Detection

Without Visiting state:

Thus it is evident that we cannot differentiate between **back edges** and **cross edges** with just 2 states.

Next we shall see what happens if we introduce the Visiting state!



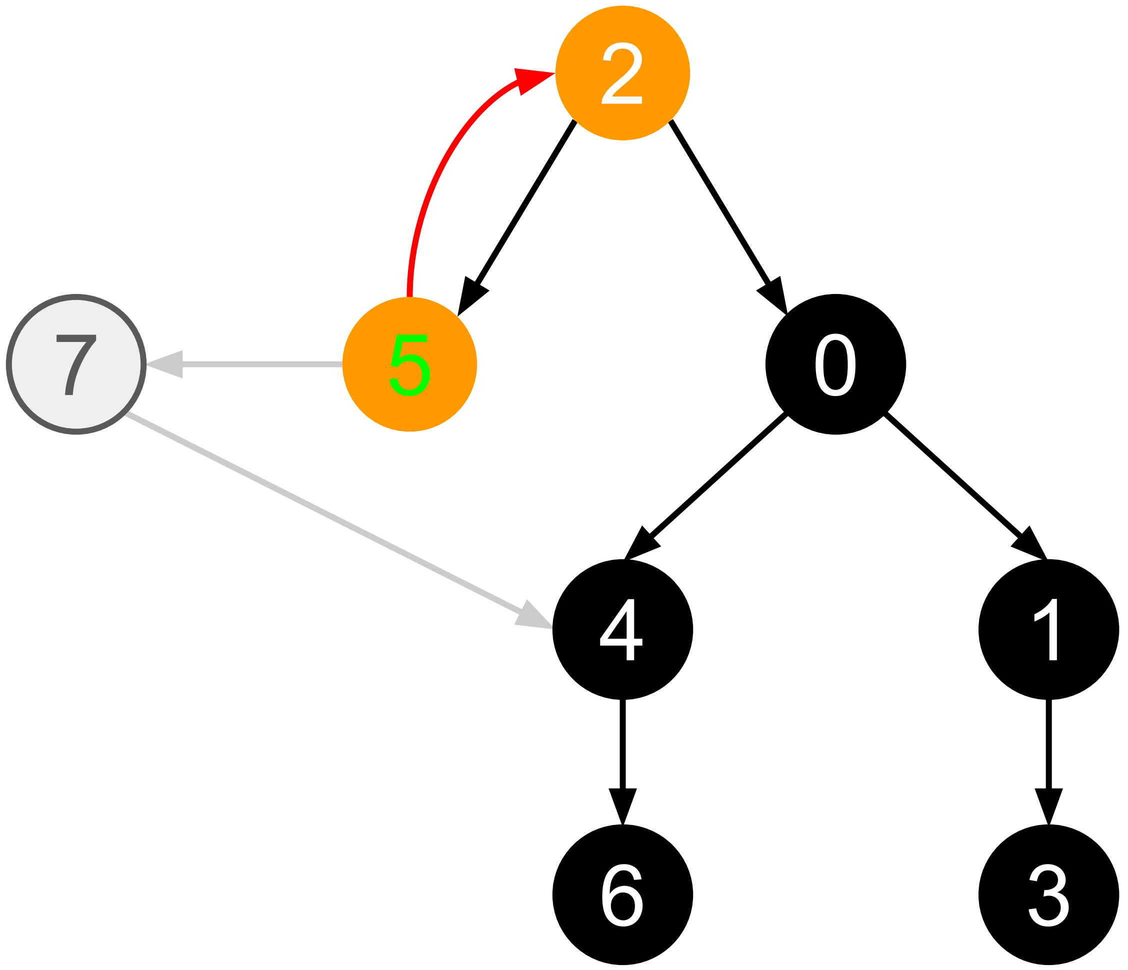
DFS Application — Cycle Detection

With Visiting state:

Vertex 5: Current Vertex

Vertex 2: Visiting

Since 2 is Visiting state, we conclude that 5→2 is a **back edge**, which is indeed the case!



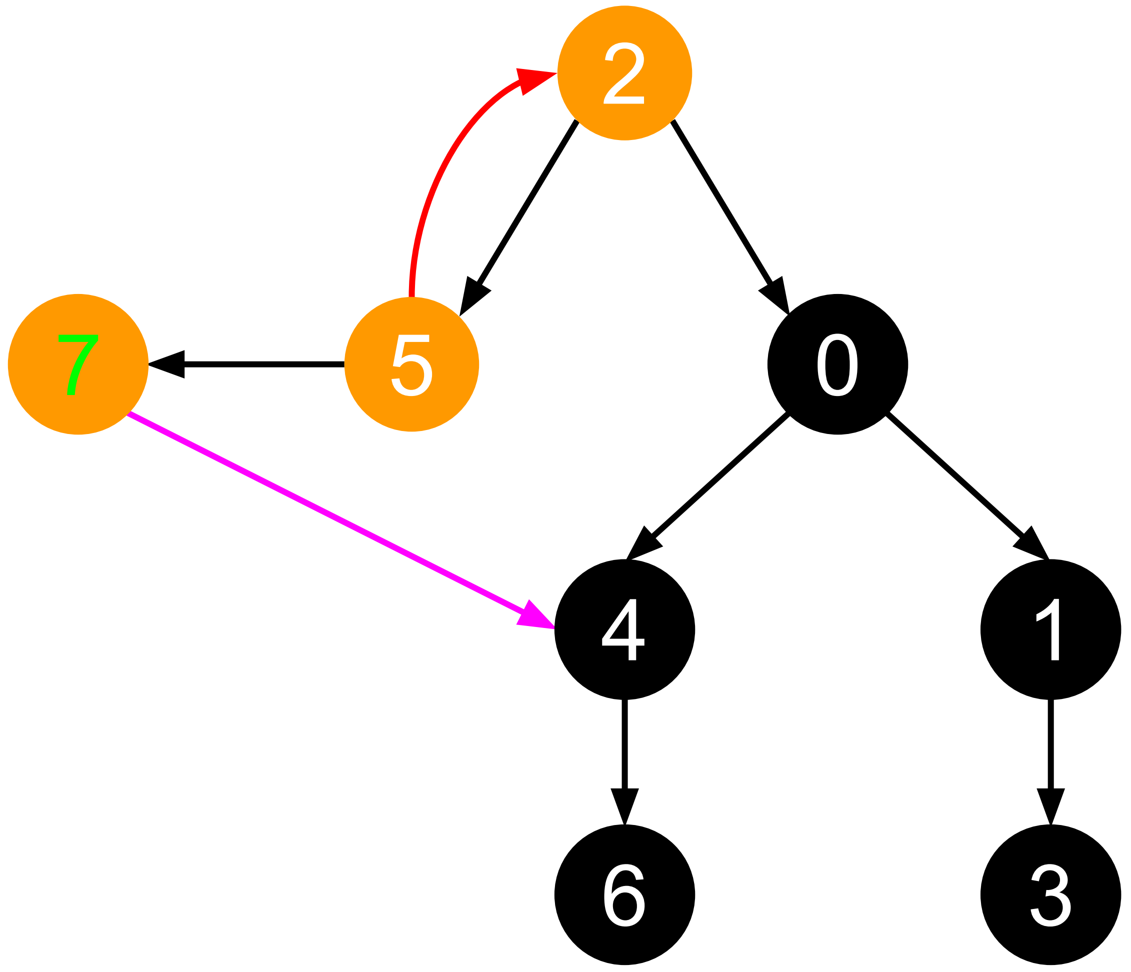
DFS Application — Cycle Detection

With Visiting state:

Vertex 7: Current Vertex

Vertex 4: Visited

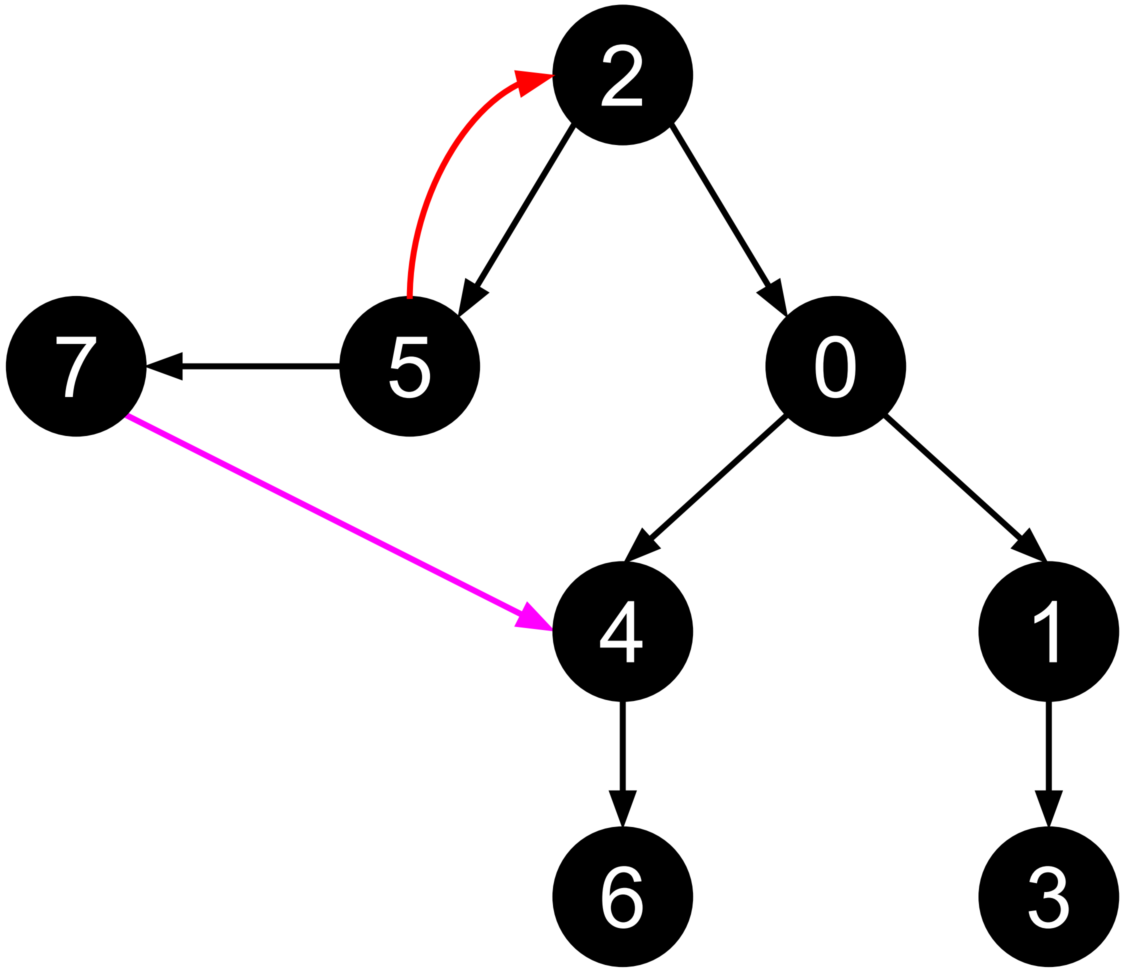
Since 4 is Visited state, we conclude that 7→4 is a **cross edge**, which is indeed the case! We did not make the same mistake as before.



DFS Application—Cycle Detection

With Visiting state:

Thus we need both Visiting and Visited states in order to differentiate between **back edges** and **cross edges**!



DFS Application—Cycle Detection

The following is one way to implement the modified DFS subroutine for cycle detection in a graph:

```
void detect_cycle(int v_id) {
    if (is_visited(nb_id)) return;
    mark_visiting(v_id);             // Mark visiting
    for (auto &nb_id : AL[v_id]) {
        if (is_visiting(nb_id))  // If backedge
            has_cycle = true;
        else
            detect_cycle(nb_id);
    }
    mark_visited(v_id);              // Mark visited
}
```

DFS Application — Cycle Detection

Our main procedure might look something like this:

```
...
bool has_cycle = false;
/* Iterate over each vertex v_id in graph */
for (int i = 0; i < AL.size(); ++i) {
    detect_cycle(i);  // Detect on every vertex v_id
}
/* Do something about the detection */
if (has_cycle)
    cout << "This graph has cycle(s)!" << endl;
...
```

Test Yourself

Do we need Visiting state for undirected graph?

Why or why not?

Test Yourself

Do we need Visiting state for undirected graph?

Why or why not?

No. There are no cross edges in undirected graph.

Deeper Stuffs about Topological Sort

Recap: Topological sort

Topological sorted order on a **DAG** is:

- A **linear ordering** of all the DAG's vertices.
- Criteria:
    - Graph must be a DAG.
    - For every edge $u \to v$ in the graph, u must come before v in the topological sort!

Test yourself

Given the toposort [1, 2, 3, 4, 5]. Is it guaranteed that vertex 1 has an outbound edge to 2?

Test yourself

Given the toposort [1, 2, 3, 4, 5]. Is it guaranteed that vertex 1 has an outbound edge to 2?

Answer: No! The rule just say that if $u \rightarrow v$, then v must come after u in the toposort. However, **the converse is not true**! Knowing that v come after u in the toposort is insufficient for us to conclude that $u \rightarrow v$.

So in this case, all we can say is 1's neighbours are on its right in the toposort but we can't tell who they are if they exist.

Recap: Toposort implementation — DFS subroutine

Only 1 new line added to DFS!

```
void toposort(int u) {
    if (visited[u]) return;
    visited[u] = true;
    for (Integer v : AL[u]) {
        toposort(v);
    }
    tps.add(u); // Append to toposorted list
}
```

$\left.\vphantom{\begin{matrix}a\\b\\c\\d\\e\end{matrix}}\right\}$ Standard DFS code

Recap: Toposort implementation — Main routine

Our main procedure looks like this!

```
...
/* Iterate over each vertex v_id in graph */
for (int i = 0; i < AL.size(); ++i) {
    toposort(i);  // Call toposort on vertex v_id
}
/* Reverse toposorted list tps */
Collections.reverse(tps);
...
```

Assume vertices in graph numbered from 0 to V

Test yourself!

Why do we need to call toposort on every vertex?

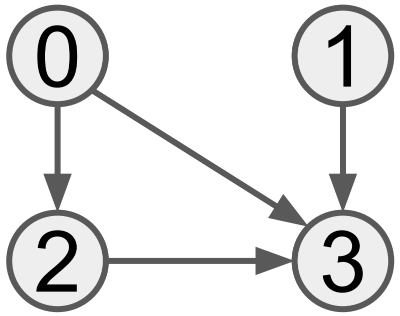
Is DFS suitable all the time? (large graphs?)

Test yourself!

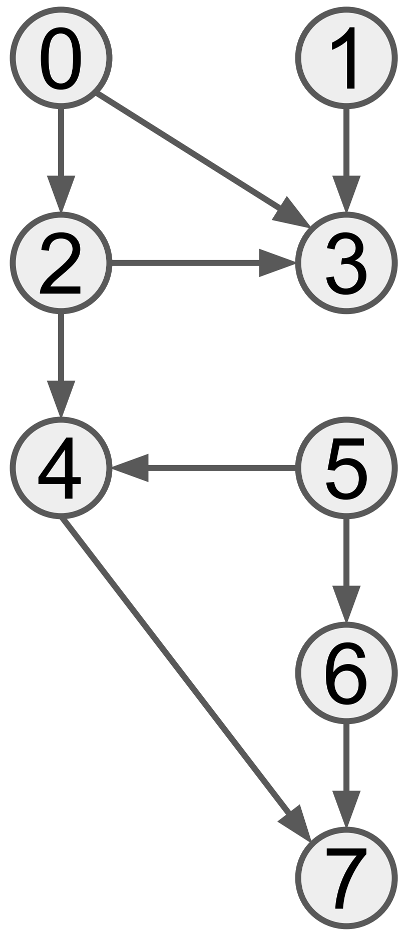
Why do we need to call toposort on every vertex?

Answer: Because a given vertex might not be able to reach every other vertex in the DAG! In the generalised case, want the toposort on the entire graph, not just on a single vertex.

E.g. Consider the graph on the right.


$$\begin{tikzcd} 0 \arrow[d] \arrow[dr] & 1 \arrow[d] \\ 2 \arrow[r] & 3 \end{tikzcd}$$

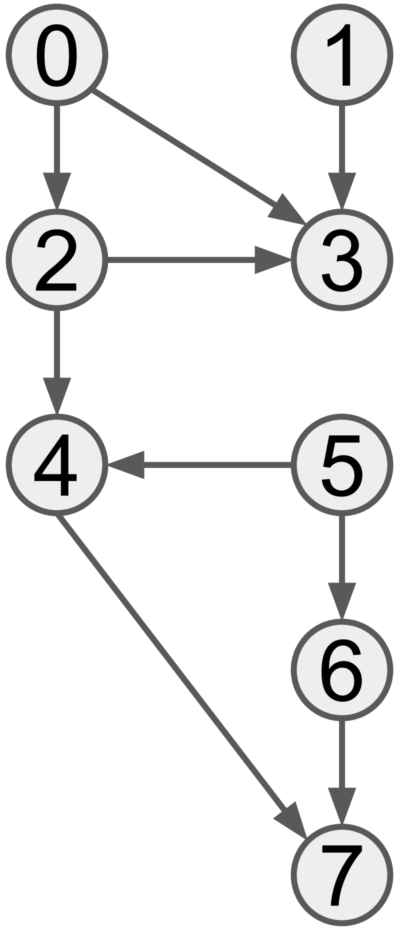
Recap: Topological sort V.S. DFS



DFS visitation order: 0, 2, 3, 4, 7, 1, 5, 6

Is this what we want?

Recap: Topological sort V.S. DFS



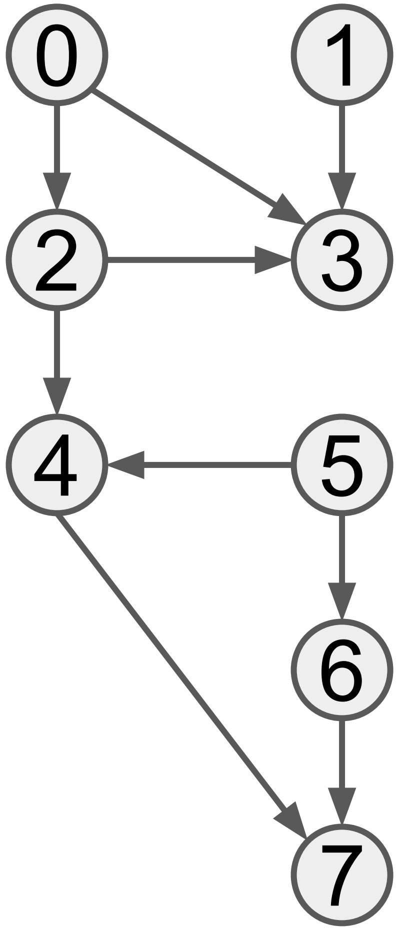
DFS visitation order: 0, 2, 3, 4, 7, 1, 5, 6

Is this what we want?

Answer: No! Because 1 should come before 3!

This proves that DFS visitation order **is not the same as** toposort order!

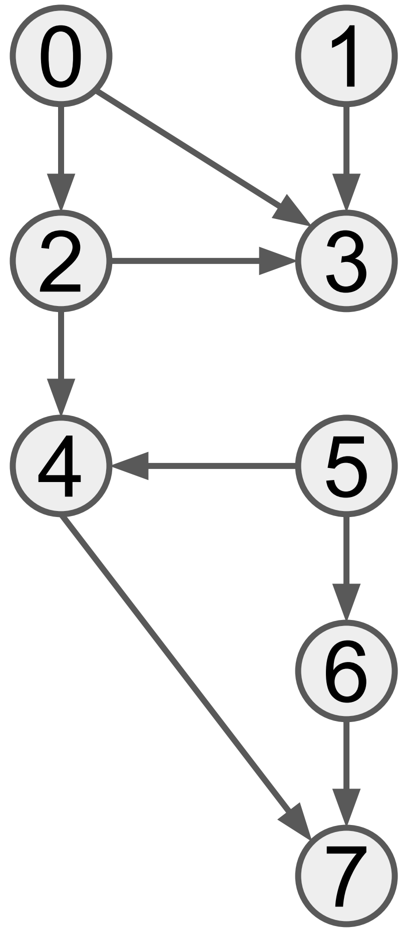
Recap: Topological sort V.S. DFS



DFS visitation order: 0, 2, 3, 4, 7, 1, 5, 6

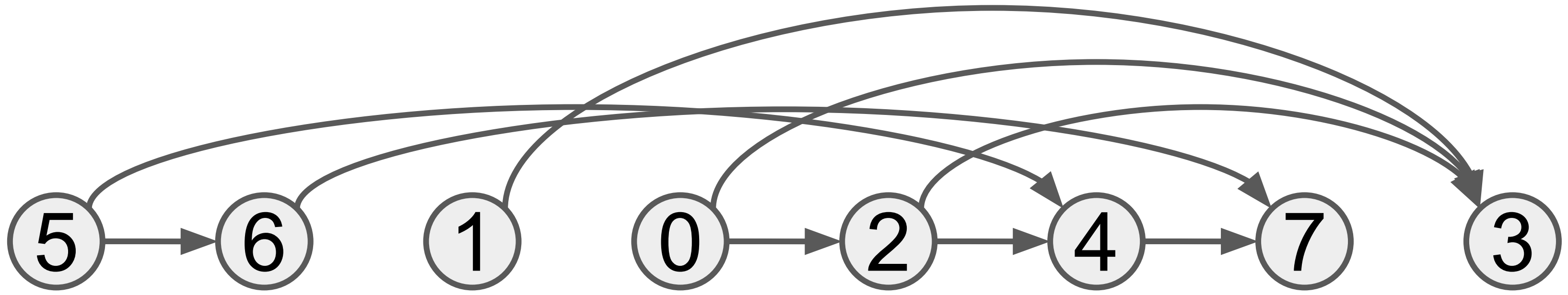
Toposort order: 5, 6, 1, 0, 2, 4, 7, 3

Recap: Topological sort properties



Toposort order: 5, 6, 1, 0, 2, 4, 7, 3

With this sequence, we can also redraw the graph in a linear fashion!



Recap: Topological sort properties

Corollary

Realise this also means that in the DAG, if vertex u **has a path** to vertex v, then v **must also be on the right** of u in the toposort list!

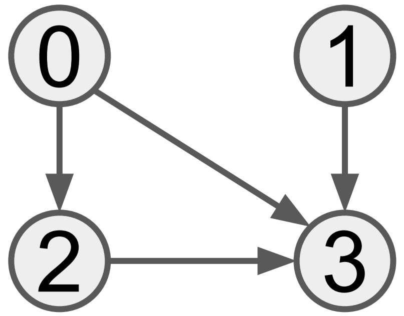
Question 2

- Please review https://visualgo.net/en/dfsbfs (either DFS or BFS version)
- The modified DFS or modified BFS (Kahn's) topological sort algorithm only gives one valid topological ordering
- How can we find **all possible valid topological orderings** for a given DAG?

Valid topological orderings

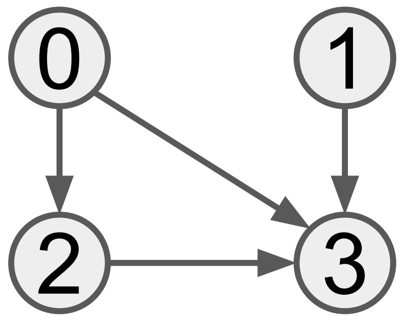
Recall: Toposort of a graph is not unique. Multiple possible sorted orders exist! However, a DAG must have **at least** one topological ordering.

What are all of the toposorts for this graph?



Valid topological orderings

Recall: Toposort of a graph is not unique. Multiple possible sorted orders exist! However, a DAG must have **at least** one topological ordering.


$$\begin{tikzcd} 0 \arrow[d] \arrow[dr] & 1 \arrow[d] \\ 2 \arrow[r] & 3 \end{tikzcd}$$

What are all of the toposorts for this graph?

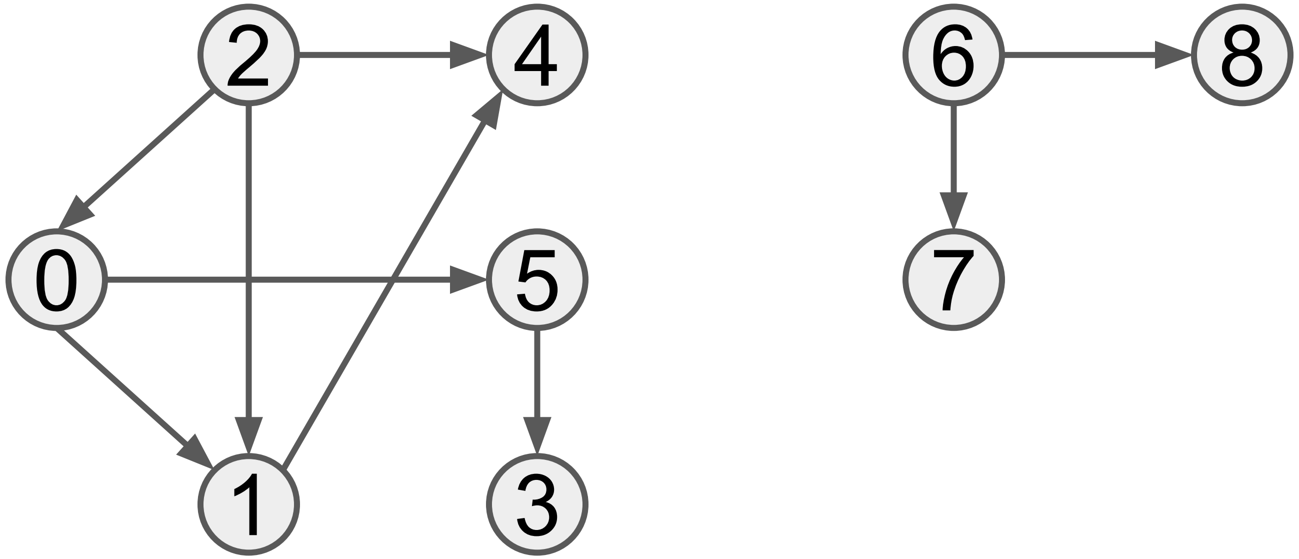
Answer:

- 1, 0, 2, 3
- 0, 1, 2, 3
- 0, 2, 1, 3

Realize that:

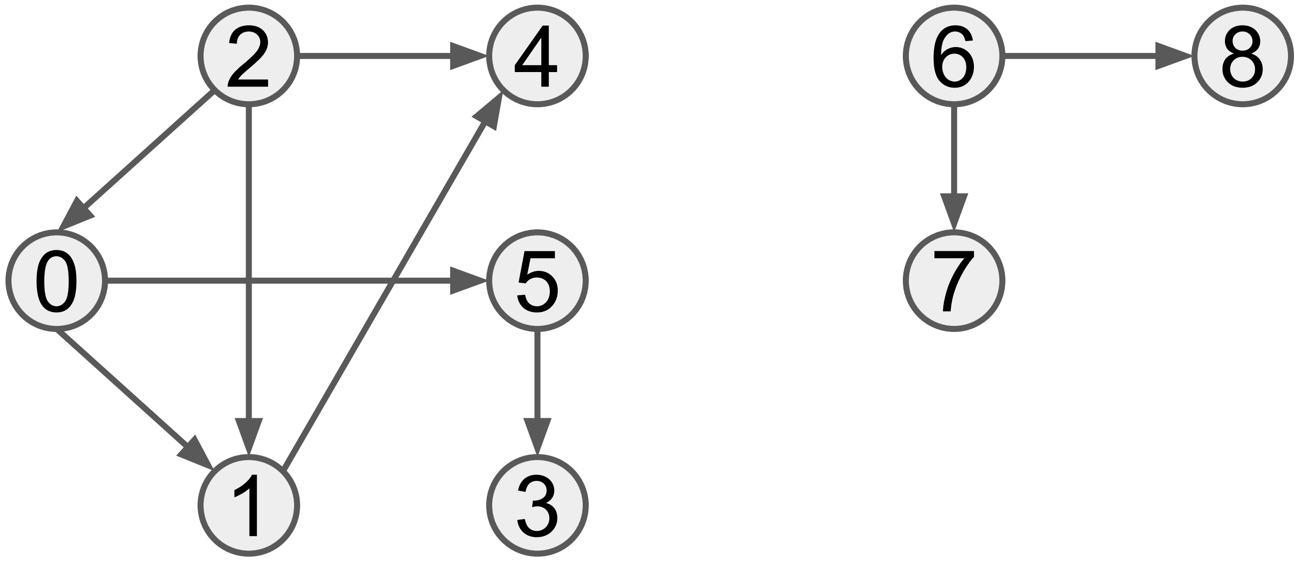
- Every other vertex point to 3 and therefore 3 **must be** the last in the toposort
- It doesn't matter how 1 is ordered relative to 0 or 2 in the toposort. Why?

Valid topological orderings



How many valid topological orderings does this graph have?

Valid topological orderings



How many valid topological orderings does this graph have?

Answer: 1008. Don’t try manually counting!

Valid topological orderings—Starting point

Question to ask as a starting point:

What kind of DAG has the

- Minimum
- Maximum

number of valid topological orderings?

Valid topological orderings—Starting point

What's the **minimum** number of valid topological ordering for a DAG?

Valid topological orderings—Starting point

What's the **minimum** number of valid topological ordering for a DAG?

Answer: 1. Because a DAG must have at least 1 valid topological ordering.

Valid topological orderings—Starting point

What kind of DAG has exactly 1 topological ordering?

Valid topological orderings—Starting point

What kind of DAG has exactly 1 topological ordering?

Answer: A linked list!

Valid topological orderings—Starting point

What’s the **maximum** number of valid topological orderings for a DAG?

Valid topological orderings—Starting point

What's the **maximum** number of valid topological orderings for a DAG?

Answer: A topological ordering is a permutation of V vertex numbers. So maximum number of toposorted orders is $V!$

Valid topological orderings—Starting point

How can we construct a graph such that every permutation is a valid toposort order?

Valid topological orderings—Starting point

How can we construct a graph such that every permutation is a valid toposort order?

Answer: A disconnected graph! i.e. a graph with V edgeless vertices. Any relative ordering of vertices is a valid toposort since there are no edges.

Self-read

Valid topological orderings—Approach

Back to the question: How can we find all possible valid topological orderings for a given DAG?

Worse case: $V!$ topological orderings.

Brute Force Checking: We can iterate through all $V!$ permutations and check each one if they are valid.

Self-read

Valid topological orderings—Validating permutation

How to check if a permutation is valid?

For every edge in the graph from $u \rightarrow v$, we make sure that u appears before v in the permutation.

Invalid otherwise.

Self-read

Valid topological orderings—Naive approach

For **each edge** $u \rightarrow v$ in the graph, how to check if u comes before v in the permutation?

Naive approach: Iterate the entire permutation and check if we encounter u before we encounter v.

- $O(V)$ time **per edge**
- $O(VE)$ time **over entire graph**

Self-read

Valid topological orderings—Naive approach

Running this naive approach across all $V!$ possible permutations therefore incurs a total time complexity of

$$O(V! \times (VE))$$

Recall the code we presented a few slides ago to calculate the 1008 possible toposorts of the example graph?

It uses exactly this naive approach!

Self-read

Valid topological orderings—Slightly better approach

A slightly better approach:

For each permutation, we first pay a $O(V)$ penalty to preprocess it by using a table to map each vertex to their position in the permutation.

Thereafter, for **each edge** $u \rightarrow v$, we just need to check in the permutation if the position of u comes before the position of v.

- $O(1)$ time **per edge**
- $O(V+E)$ time **over entire graph** (including preprocessing time)

Self-read

Valid topological orderings — Slightly better approach

Example:

Say for instance we have the following permutation to validate:

1, 5, 6, 2, 7, 4, 3

So we first build the table on the right.

Then to validate 5→7 for instance, we check if 5 comes before 7 in the permutation by verifying 1 < 4.

Vertex	Permutation position
1	0
2	3
3	6
4	5
5	1
6	2
7	4

Self-read

Valid topological orderings—Slightly better approach

Running this slightly better approach across all $V!$ possible permutations therefore incurs a total time complexity of

$$O(V! \times (V + E))$$

Self-read

Valid topological orderings — Summary

Time complexities of various solutions to the problem:

- Naive approach (demo code): $O(V! \times VE)$
- Slightly better approach: $O(V! \times (V+E))$
- Backtracking: $O(V! \times V)$
- Backtracking with bitmask: $O(V!)$ (Beyond CS2040C)

Question 3

The modified BFS (Kahn's) topological sort algorithm is actually quite interesting! Read more about the details on Wikipedia.

On VisuAlgo, we used a `Queue` for the underlying data structure in the modified BFS. What if we replaced it with

- A `Stack`?
- A `PriorityQueue` or `TreeSet`?
- A `HashSet`?

Toposort implementation — Kahn’s algorithm

Pseudocode (modified from VisuAlgo)

```
void kahns_algo() {
  Q = queue(); // Initialize empty queue
  for (each vertex v in graph) { // Pre-populate Q
    if (v has no incoming edge) Q.push(v);
  }
  while (!Q.empty()) {
    u = Q.poll(); // Dequeue from Q
    tps.push_back(u);       // Append to toposort list
    for (each neighbor v of u) {
      remove_edge(u,v);     // Remove edge u→v from graph
      if (v has no incoming edge) Q.push(v);
    }
  }
}
```

Toposort implementation — Kahn’s algorithm

Although trickier to implement than modified DFS, Kahn’s algorithm is actually much more intuitive and its steps can be easily understood as follows.

So long as there are vertices in the graph:

- Greedily select the vertices with no *incoming* edges.
- Add them to toposort list.
- Remove these vertices and their edges from the graph.

Kahn’s algorithm: Implement using Stack

If we replace the `Queue` with a `Stack`:

- We will also get another valid topological ordering.
- Topological ordering in LIFO instead of FIFO order in case of ties (more than one vertices with 0 in-degree).

Kahn’s algorithm: Implement using Priority Queue

If we replace the `Queue` with a (max/min) `PriorityQueue` or `Set`:

- We will also get another valid topological ordering.
- Topological ordering in (max/min) priority order in case of ties (more than one vertices with 0 in-degree).

There are a few topological sort problems that require this modification!

Kahn's algorithm: Implement using Hash Table

If we replace the `Queue` with a `HashSet`:

- We will also get another valid topological ordering.
- Topological ordering in unspecified order in case of ties (more than one vertices with 0 in-degree).

Wow Kahn's algorithm works even when the vertices to be removed are not ordered in any way!

Kahn’s algorithm: Observations

- Basically Kahn's algorithm is quite versatile as it just need the data structure to be a set S for containing vertices
- Realize that at any one point in time, S always contains vertices with 0 in-degree (i.e. no incoming edges). That’s actually its sole purpose.
- We saw that S can be FIFO ordered, LIFO ordered, fully ordered, or not ordered at all! The order doesn’t seem to matter!

Kahn’s algorithm: Observations

Why doesn’t it matter how S orders its vertices?

- The order of vertices in S just captures the sequence in which the 0 in-degree vertices are to be removed and appended to toposort list
- Realize that 0 in-degree vertices do not point to each other and therefore their relative ordering in the toposort **does not matter**!

Toposort Sort—Protip

When coming up with a toposort by hand, it’s a lot simpler to use Kahn’s algorithm. You may want to do this for VisuAlgo quizzes.

To carry out Kahn’s algorithm, just follow these steps:

1. In any order, find vertices with no *incoming edges*
2. As soon as you find one:
    a. Append it to toposort list.
    b. Remove it and all its edges.
3. Repeat steps 1 & 2 until no more vertices are left in the graph

Question 4: /jetpack

Self-read

DFS/BFS Applications

Self-read

Example Problems

You have already seen several examples of DFS/BFS applications from the questions discussed in Week 10

1. https://nus.kattis.com/problems/countingstars
   - Easy flood-filling/finding CCs
2. https://nus.kattis.com/problems/reachableroads
   - Easy DFS/BFS application; finding CCs too
3. https://nus.kattis.com/problems/runningmom
   - Back edge/cycle detection problem

Self-read

Example Problem 1: Counting Stars

You are given a m by n grid of characters representing the night sky.

Each cell is either

- `#`: black pixels
- `-`: white pixels

White pixels that are adjacent vertically or horizontally are part of the same star.

You are to count the number of stars.

Self-read

Counting Stars — Approach

Whenever we encounter a star, we count it and then remove it from the picture.

For instance, black out all the `-` cells belonging to the star we wish to remove by replacing them with `#`.

Self-read

Counting Stars — Example

Let’s start off with this grid.

We iterate through row-wise followed by column-wise to find the first `-` cell.

```
##########
##--##-##
#--#######
#---#--##
###---###
##########
###--##-#
#####-####
##########
```

Counting Stars — Example

Encountered first star.

Black out all its -

```
#########
##--##-##
#--######
#---#--##
###---###
#########
###--##-#
####-####
#########
```

```
#########
######-##
#########
#########
#########
#########
###--##-#
####-####
#########
```

Counting Stars — Example

Encountered second star.

Black out all its -

```
#########
######-##
#########
#########
#########
#########
###--##-#
####-####
#########
```

```
#########
#########
#########
#########
#########
#########
###--##-#
####-####
#########
```

Counting Stars — Example

Encountered third star.

Black out all its -

```
#########
#########
#########
#########
#########
#########
###--##-#
####-####
#########
```

```
#########
#########
#########
#########
#########
#########
#######-#
#########
#########
```

Self-read

Counting Stars — Example

Encountered fourth star.

Black out all its -

```
#########
#########
#########
#########
#########
#########
#######-#
#########
#########
```

```
#########
#########
#########
#########
#########
#########
#########
#########
#########
```

Self-read

Counting Stars — Example

In total, there are 4 stars.

As a graph problem:

- Task: count the number of CC
- Cells → vertices
- Edges → between adjacent cells that are `-`
- Flood filling → DFS

```
##########
##########
##########
##########
##########
##########
##########
##########
##########
```

Self-read

Example Problem 2: UVa 469

URL: https://uva.onlinejudge.org/external/4/469.pdf

You are given a N by M grid of cells.

Each cell is either `L` or `W` representing *land* and *water* respectively.

Answer K queries of the following:

- Given a `W` cell, how many `W` cells are connected to it.

Self-read

UVa 469—Example

Diagonal cells are considered adjacent

Red: 12

Purple: 1

Blue: 4

Orange : 1

```
LLLLLLLLL
LLWWLLWLL
LWWLLLLLL
LWWWLWWLL
LLLWWWLLL
LLLLLLLLL
LLLWWLLWL
LLWLWLLLL
LLLLLLLLL
```

Self-read

UVa 469—Approach

If we group all connected `W` cells as 1 water body.

We need to obtain the <u>size of the water body</u> each `W` cell is in.

```
LLLLLLLLL
LLWWLLWLL
LWWLLLLLL
LWWWLWWLL
LLLWWWLLL
LLLLLLLLL
LLLWWLLWL
LLWLWLLLL
LLLLLLLLL
```

Self-read

UVa 469 — Graph Modelling

- Vertex: Cells of the grid
- Edge: 8 radial directions
  - Draw an edge between 2 adjacent `W` cells
- Number of vertices in the connected component: Size of the water body

```
LLLLLLLLL
LLWWLLWLL
LWWLLLLLL
LWWWLWWLL
LLLWWWLLL
LLLLLLLLL
LLLWWLLWL
LLWLWLLLL
LLLLLLLLL
```

Self-read

UVa 469 — Implementation

What algorithm can we use to solve this?

a) Depth First Search
b) Breadth First Search
c) Either one

```
LLLLLLLLL
LLWWLLWLL
LWWLLLLLL
LWWWLWWLL
LLLWWWLLL
LLLLLLLLL
LLLWWLLWL
LLWLWLLLL
LLLLLLLLL
```

Questions?
Attendance
Break

PS5 Debrief

https://github.com/stevenhalim/cpbook-code/blob/master/ch4/sssp/bfs.java https://visualgo.net/training?diff=Medium&n=5&tl=5&module=dfsbfs,sssp

PE preparation

Input / Output

- How to input an entire line.
  - And how to input space separated variables from an inputted line
- How to input strings
- How to output space separated variables on a single line

Implementation/Debugging Tips

- *‘Binary Search’* your code (if Runtime Error)
  - Terminate it after running half of your code.
  - Commenting out suspected problematic parts.
- Replace the segment with ‘non-optimized’ version, see if it results in the same output
- Compile regularly
- **D**on’t **R**epeat **Y**ourself (DRY principle)

Sit-in/Practical Exam Tips

- **Plan** what you want to code
  - Don’t dive into the code immediately
- Partition the task into subtasks
  - Handle them one by one
  - Modular Programming

```
/* Deduplicate the array using unordered_set */
/* Sort the array */
/* Find maximum of … */
```

Sit-in/Practical Exam Tips

- Clarify when in doubt
- Be suspicious of any *weird* limits
    - Remember *long long*? *long double*?
- More **practice** → code faster
    - Range based for-loops
    - STL Data Structures

Sit-in/Practical Exam Tips

When stuck (first half):

1. **Don’t panic**
2. Rethink the problem from another angle
   a. Each vertex can become more vertices?
   b. Restrict direction of edge? Flip direction?
   c. Not a graph question?
3. Read the questions again carefully (Anything wrong/missed?)
4. Data structure**s** are your friend :)

Sit-in/Practical Exam Tips

When stuck (second half):

1. **Don't panic (that much)**
2. Damage control
    a. "Fastest-to-code" implementation
    b. Handle **general case** first, abandon corner cases
    c. Try small cases
    d. ~~Make the code "look" similar to what you think it is **:X** (aka try to scam the marker...)~~

Past PE Discussion: /teque

- Problem: Implement queue that allows insertion in the middle.
- High level idea:
  - Keep two deque, with roughly equal sizes.
  - If one of them gets bigger, transfer an element from bigger to smaller.
  - On index query, check if it is in the first deque or second deque, and print accordingly.

Past PE Discussion: /nicknames

- First two subtasks: Use a DAT of 26 entries, and record the frequency.
- Full: Use a hashmap to save the frequency of all prefixes.

Past PE Discussion: /ads

- Subtask 1: Print the input.
- Subtask 1+2+3: Scan up/down/left/right to get boundary of image and erase accordingly.
- Note that images cannot touch, even at corners, so we can DFS/BFS with implicit graphs to find the boundaries of the most nested image the illegal symbol is found in.

Past PE Discussion: /arraysmoothening

- Subtask 1+2+3+4: Just use a HashMap to count.
- Full:
  - Always remove an element with maximum frequency.
  - Have a priority queue for the counts to remove K elements efficiently.

Past PE Discussion: /magicsequence

- Just implement Radix sort, but needs to be efficient.
- Subtask 1+2: Just generate and use library sort().
- Subtask 3: Counting sort.
- Full: Need to implement efficient Radix sort.
    - Radix sort with base 65536 passes.

Past PE Discussion: /teque

Thank You!

Official Feedback: https://blue.nus.edu.sg