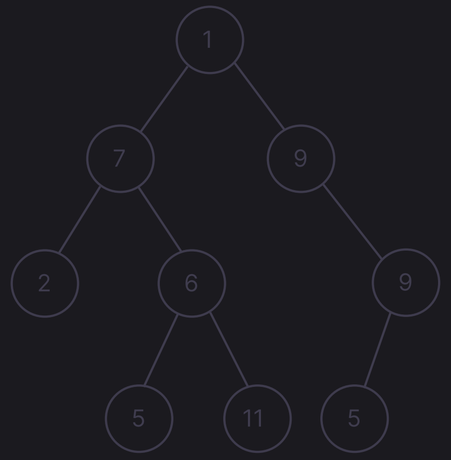
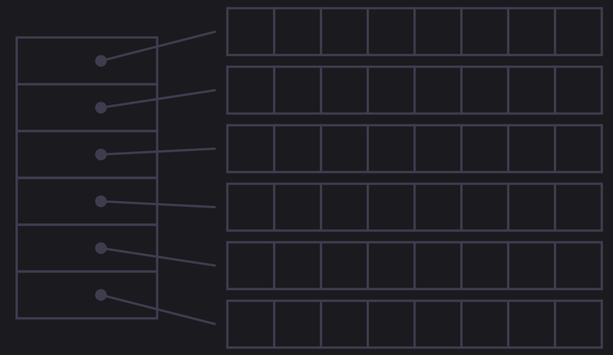
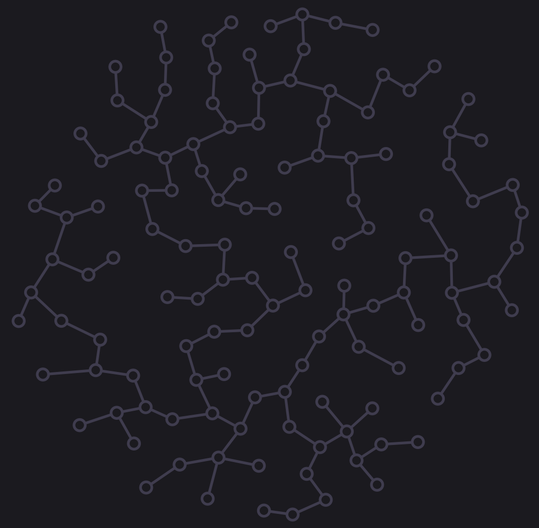


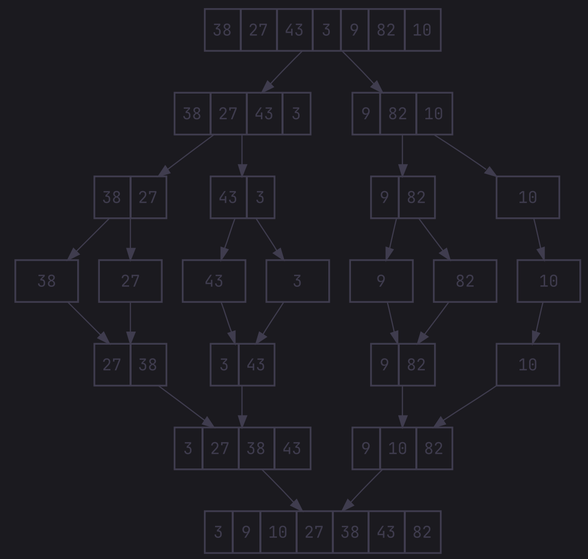


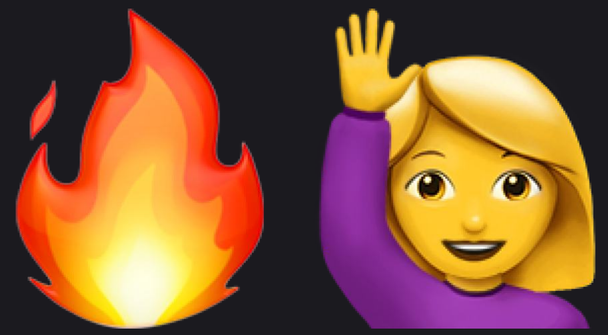
data structures & algorithms

Tutorial 8

(Week 9)







Burning questions from the previous tutorial?

This week’s lab

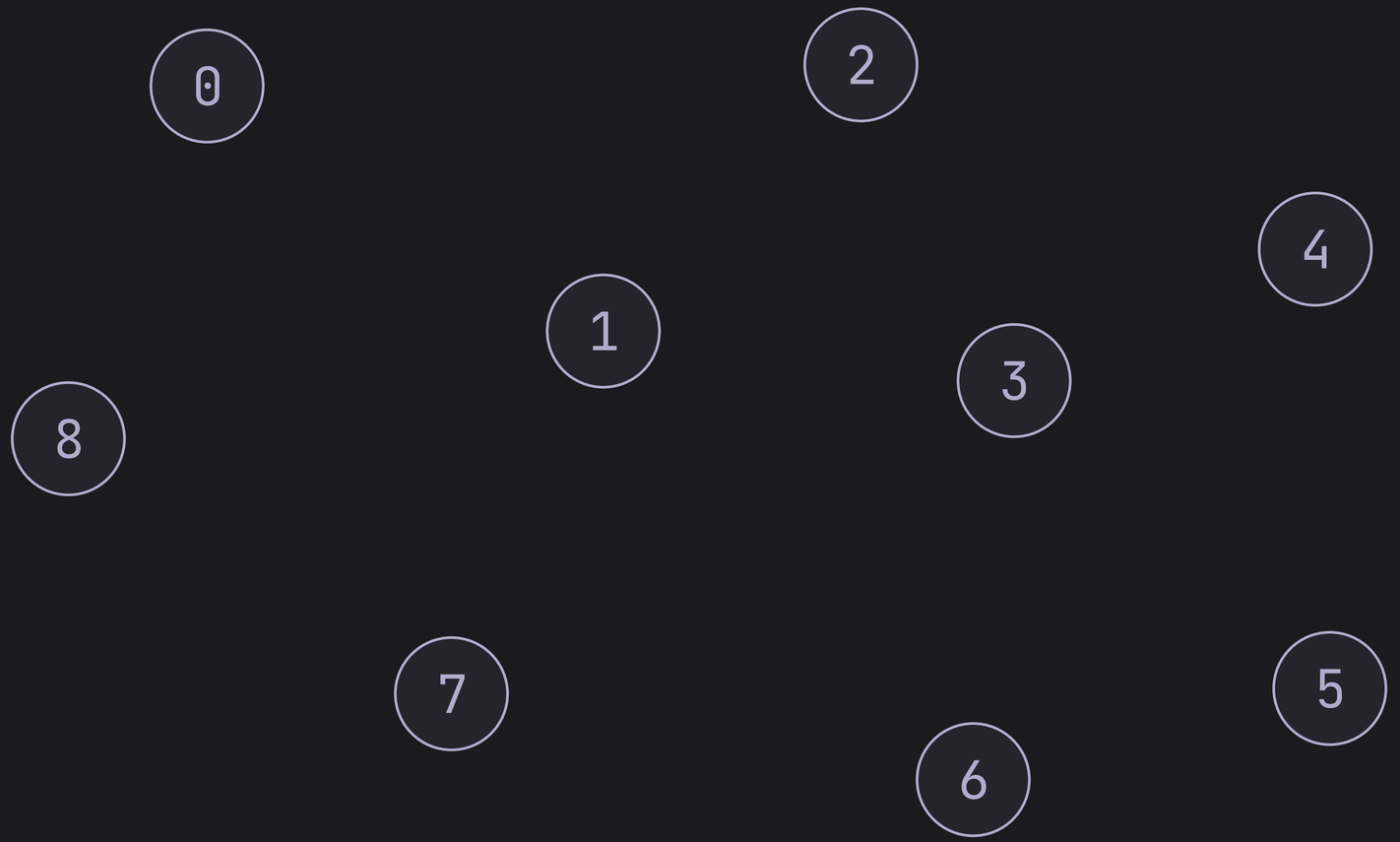
Minimum Knight Moves

This week we are learning about graphs and some very powerful graph algorithms

- Graphs
- Depth First Search (DFS)
- Breadth First Search (BFS)
- Connected Components
- Minimum Knight Moves

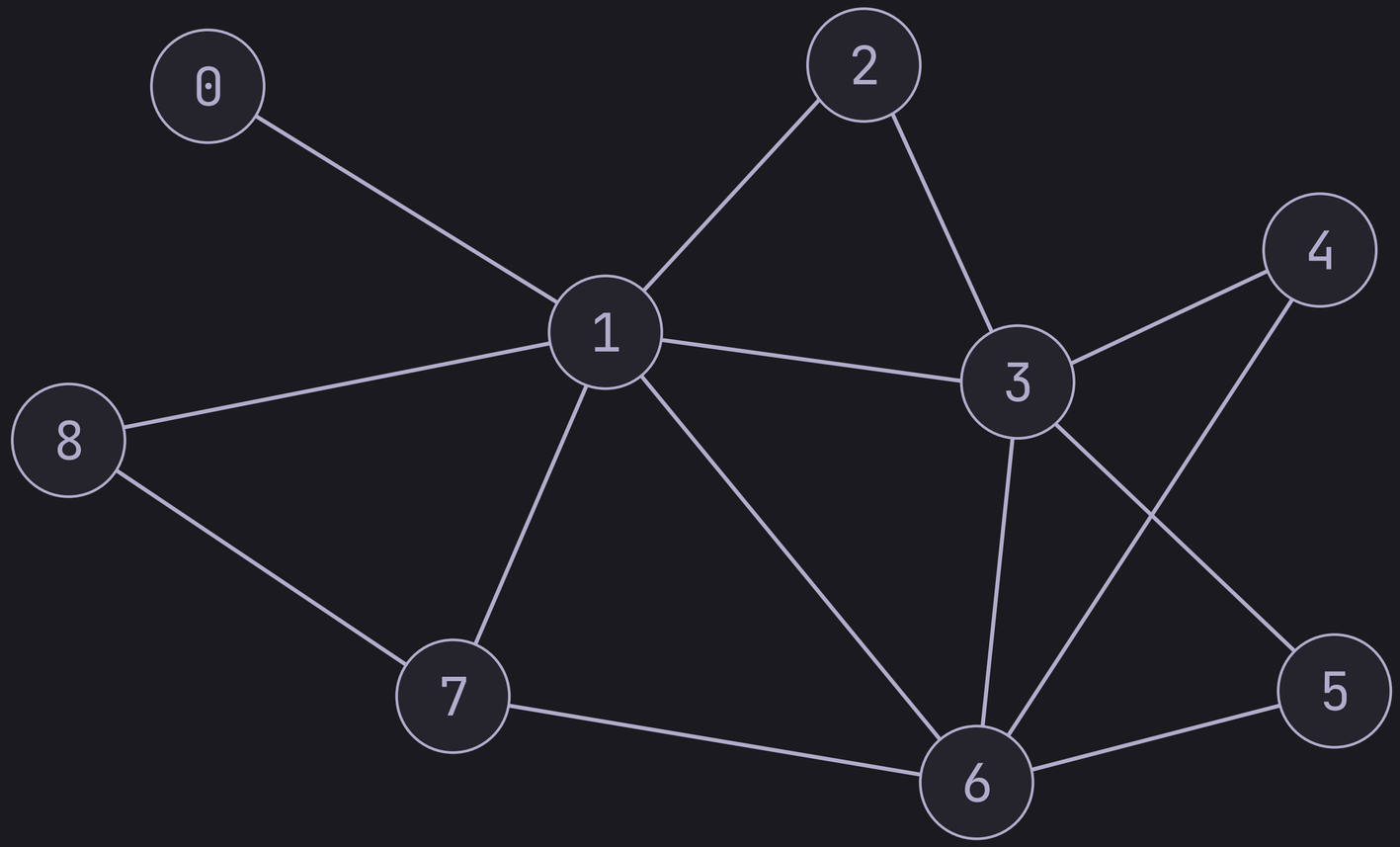
Graphs

Graphs



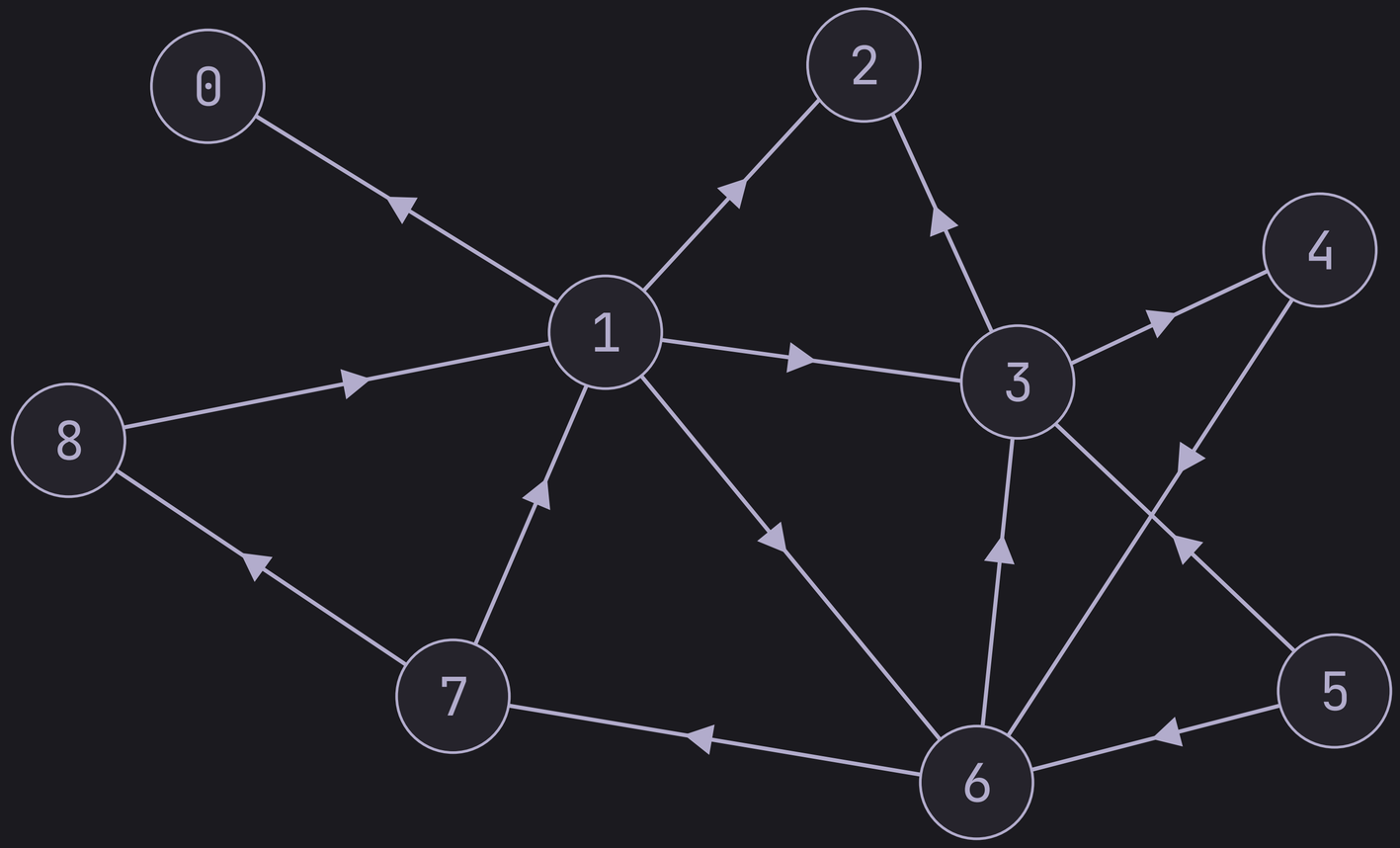
Suppose we have a collection of points that we call vertices

Graphs



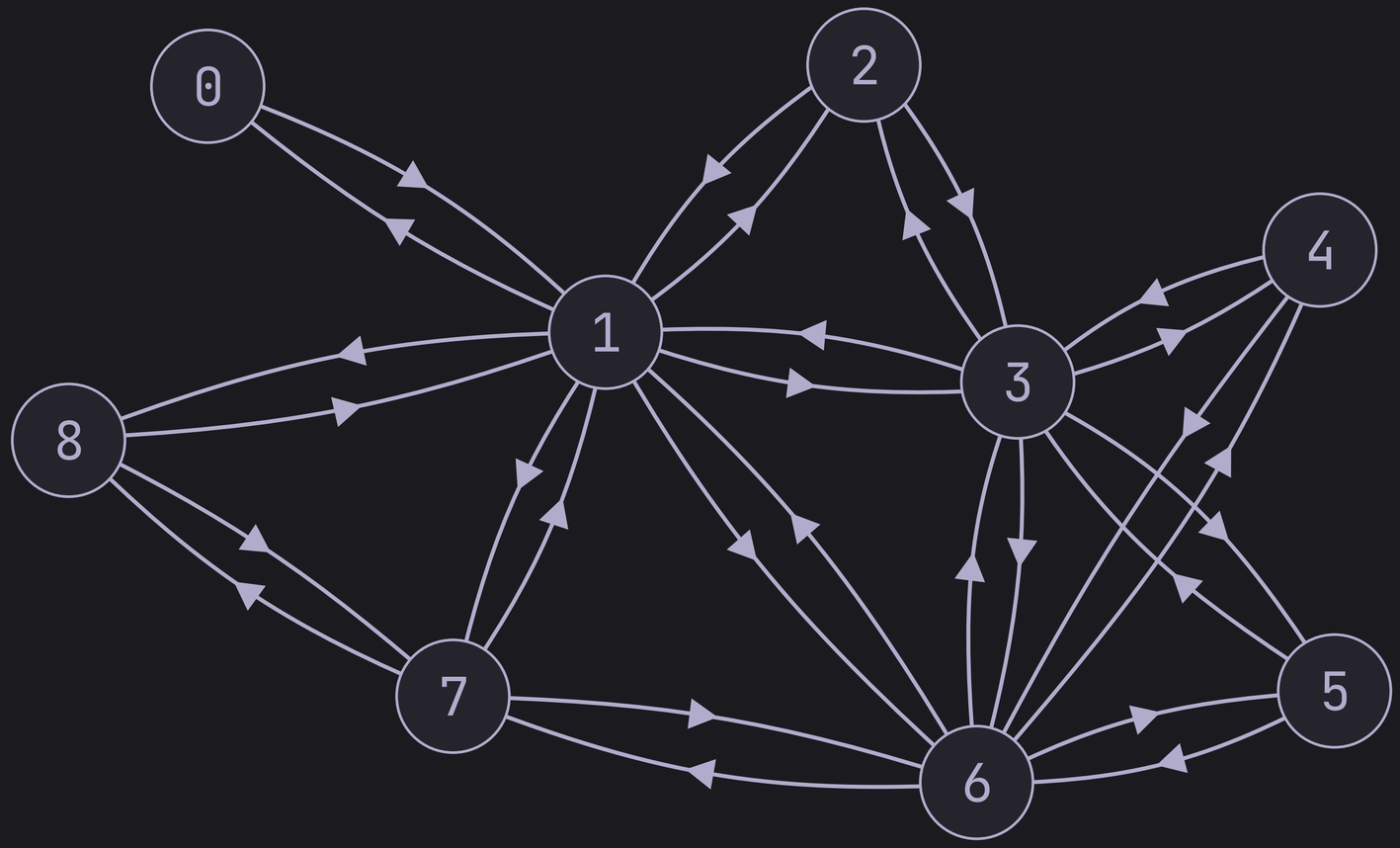
And we connect those vertices with edges

Graphs



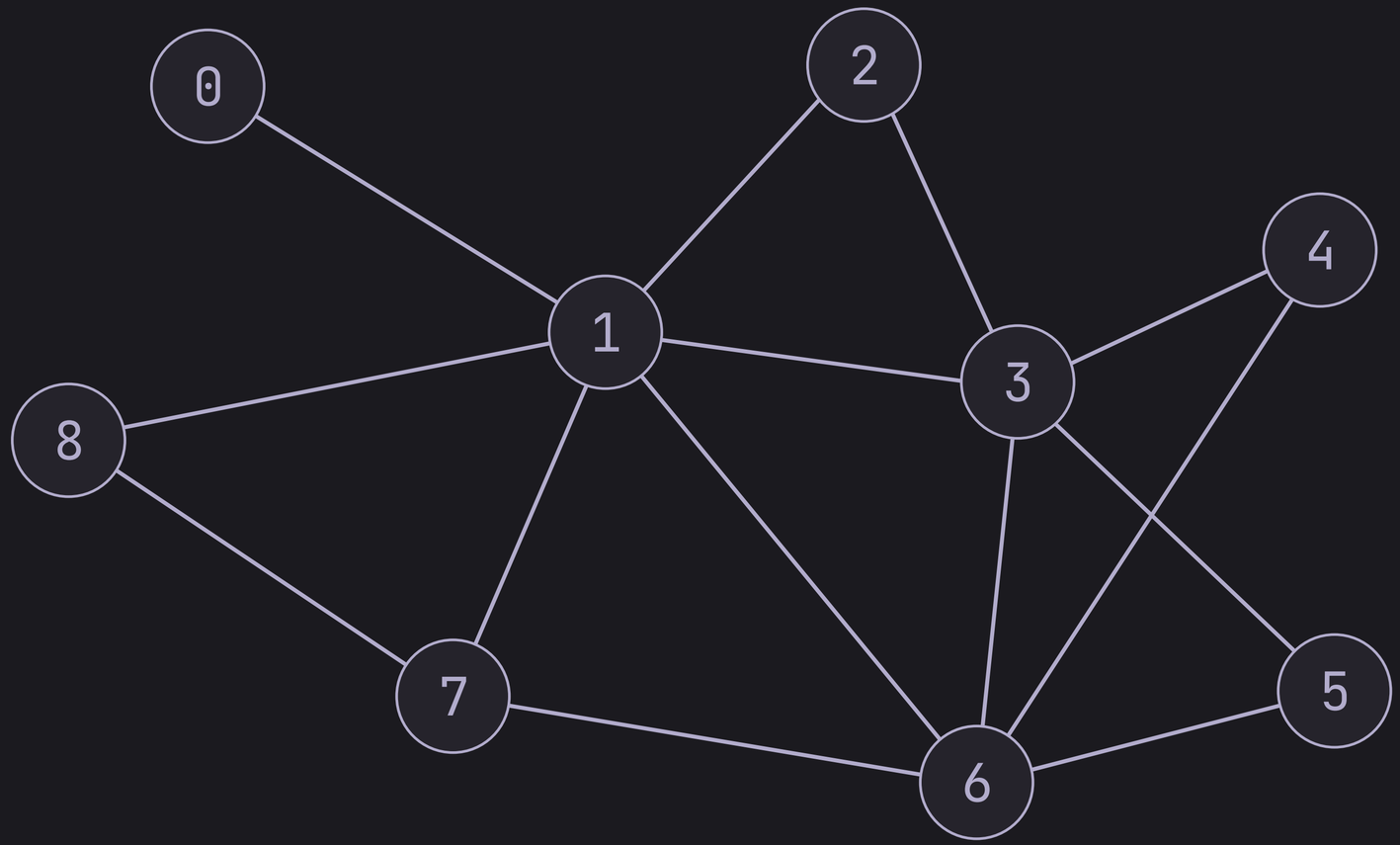
We can also have directed edges in a directed graph

Graphs

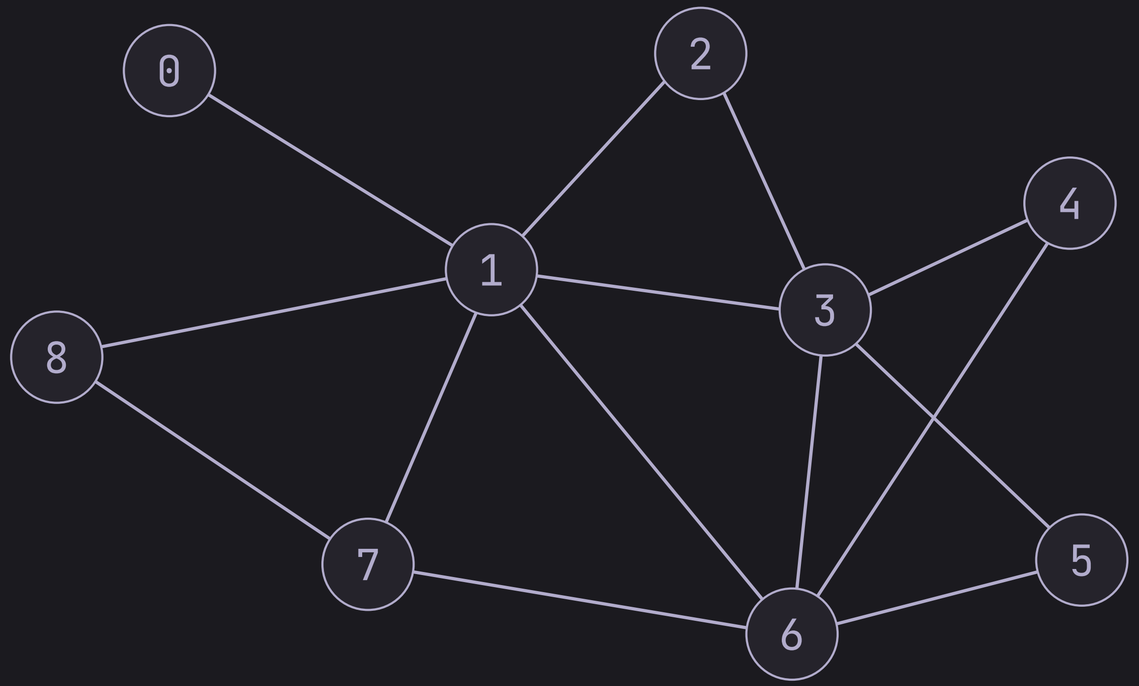


An undirected graph is really a directed graph in disguise

Graphs



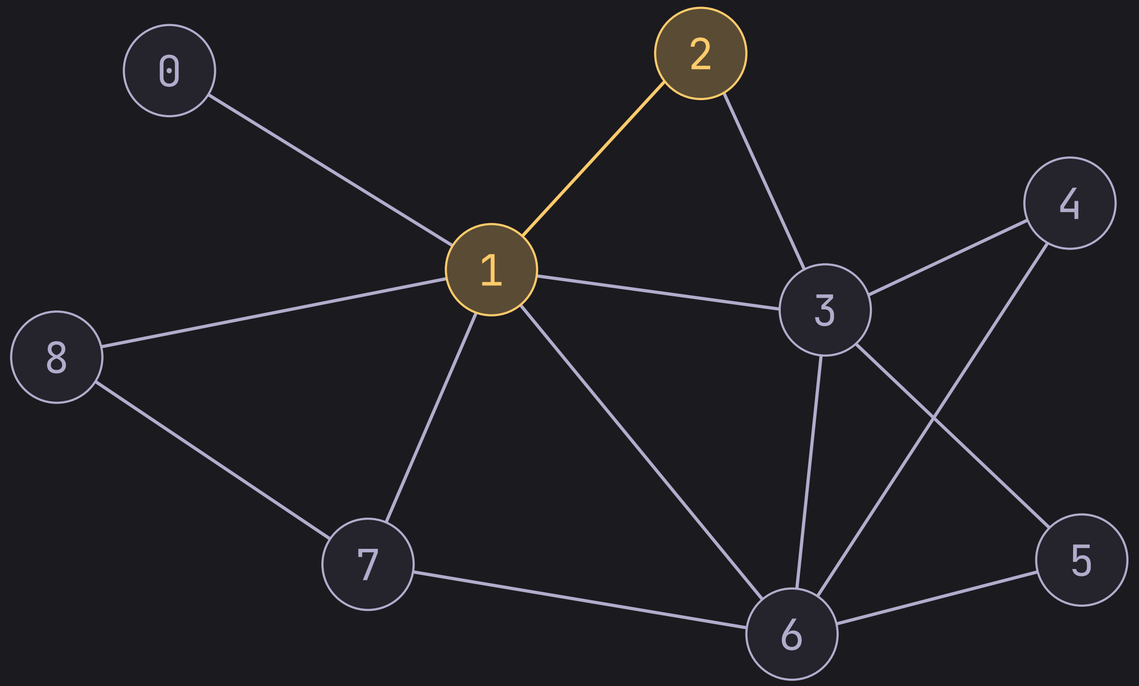
Graphs



from \ to	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0

One common way to represent a graph is an adjacency matrix

Graphs

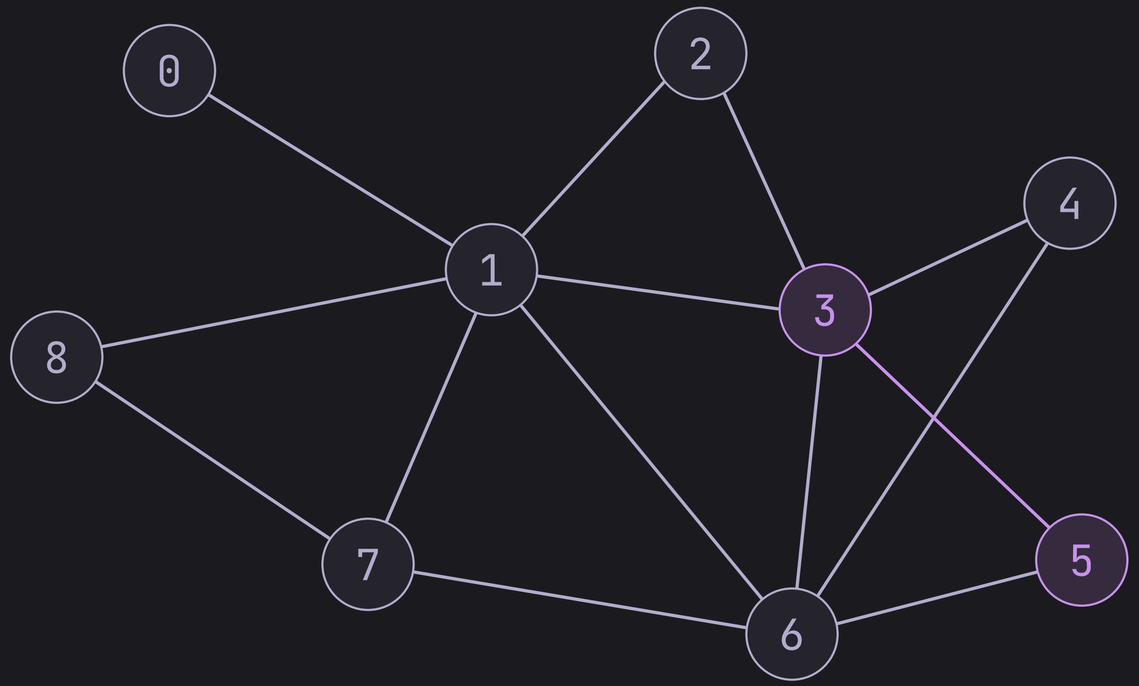


to

from	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	**1**	0	0	0	0	0	0
2	0	**1**	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0

If two vertices share an edge, then we put a 1 in the matrix

Graphs



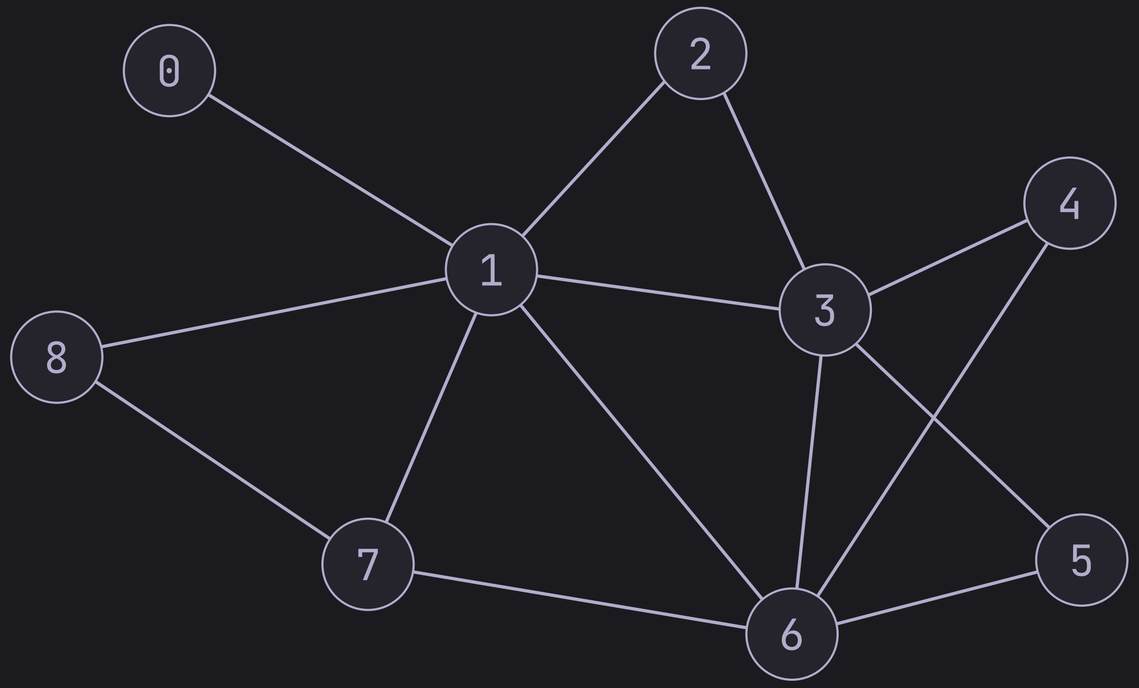
to

from

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0

If two vertices share an edge, then we put a 1 in the matrix

Graphs

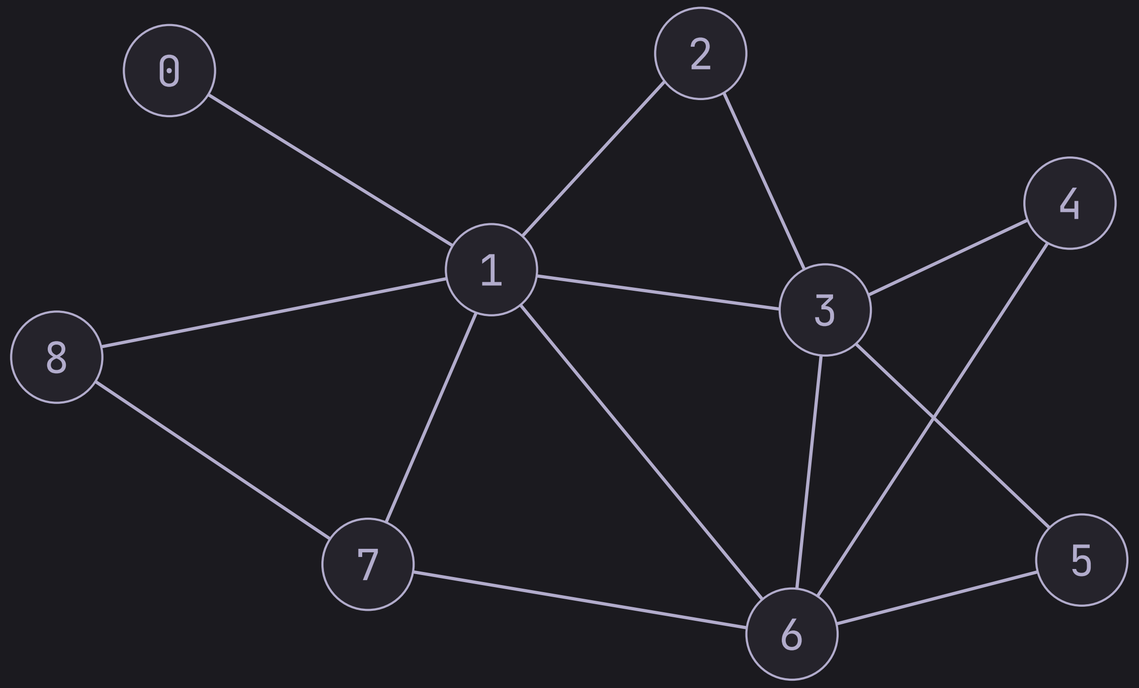


to

from	0	1	2	3	4	5	6	7	8
0	0	1	0	0	0	0	0	0	0
1	1	0	1	1	0	0	1	1	1
2	0	1	0	1	0	0	0	0	0
3	0	1	1	0	1	1	1	0	0
4	0	0	0	1	0	0	1	0	0
5	0	0	0	1	0	0	1	0	0
6	0	1	0	1	1	1	0	1	0
7	0	1	0	0	0	0	1	0	1
8	0	1	0	0	0	0	0	1	0

Filling this out we get the whole adjacency matrix

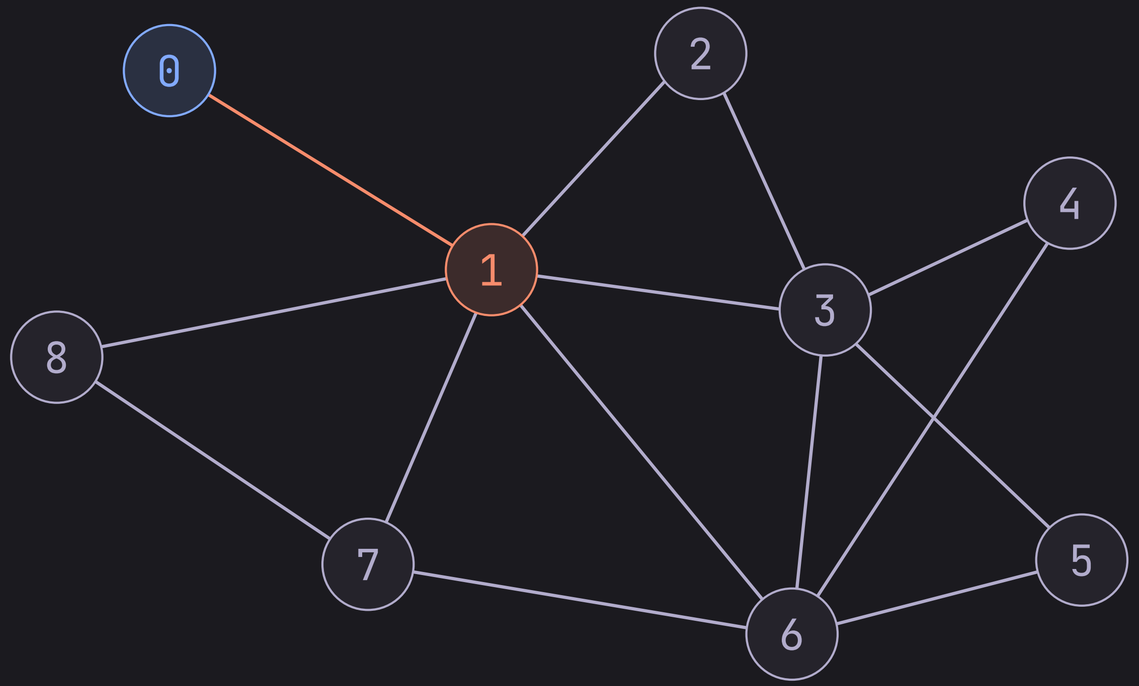
Graphs




```
0: { }
1: { }
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

Another common way to represent a graph is an adjacency list

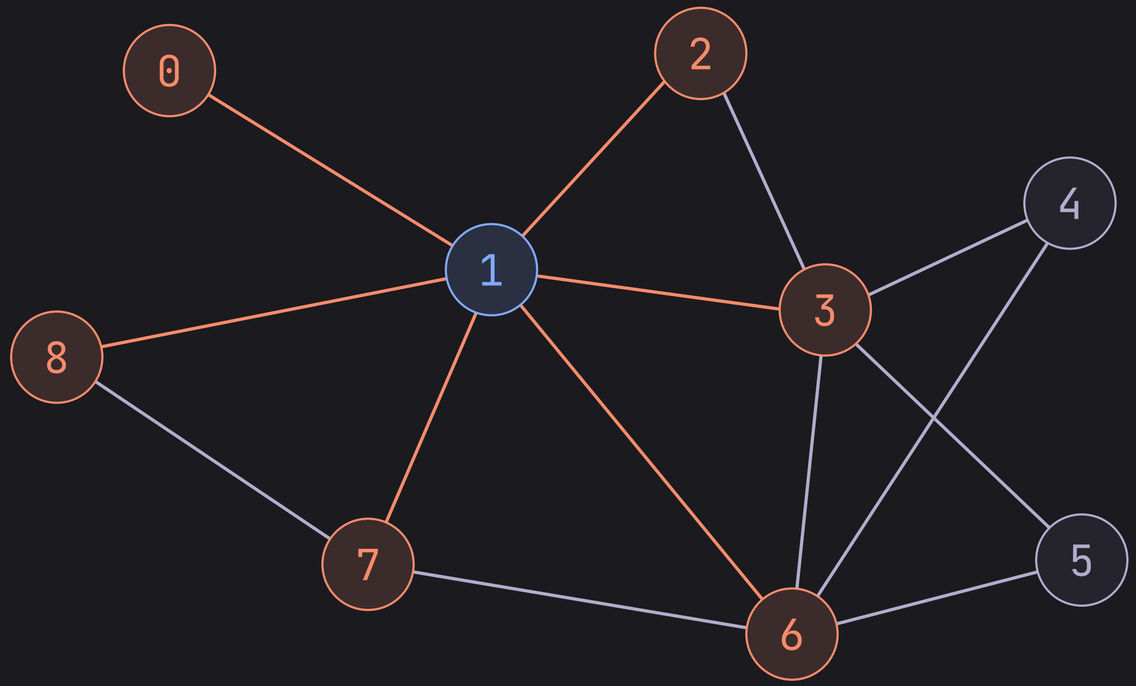
Graphs




```
0: {1}
1: { }
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

For every vertex we store a keep a list of its neighbours

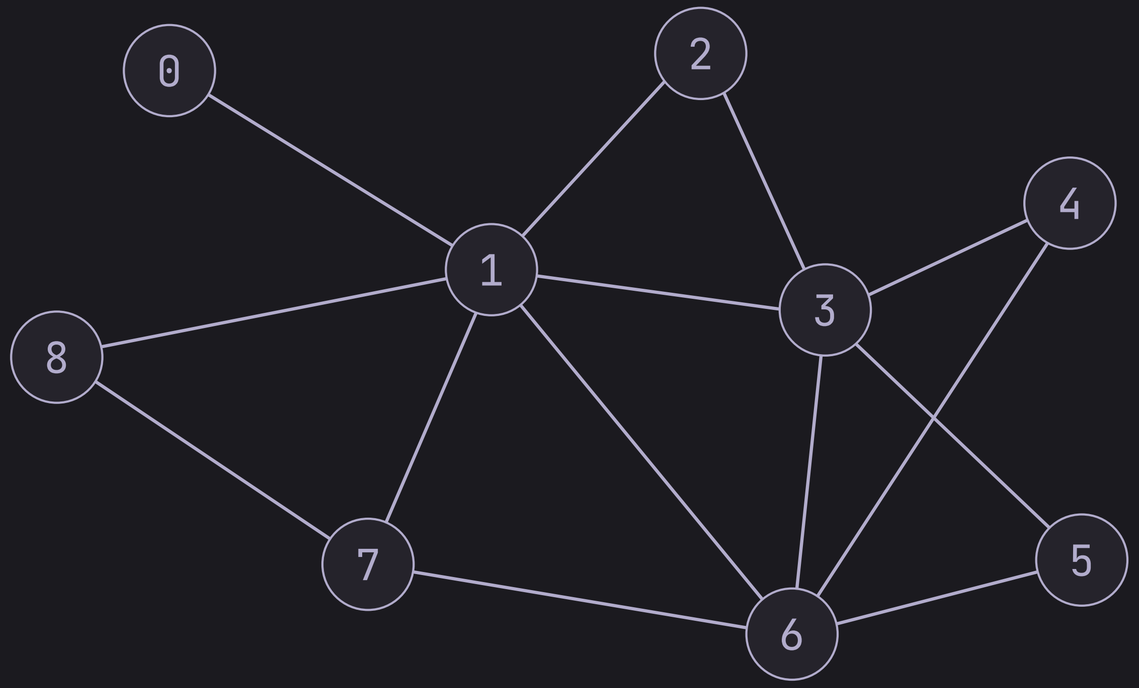
Graphs




```
0: {1}
1: {0, 2, 3, 6, 7, 8}
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

For every vertex we store a keep a list of its neighbours

Graphs




```
0: {1}
1: {0, 2, 3, 6, 7, 8}
2: {1, 3}
3: {1, 2, 4, 5, 6}
4: {3, 6}
5: {3, 6}
6: {1, 3, 5, 7}
7: {1, 6, 8}
8: {1, 7}
```

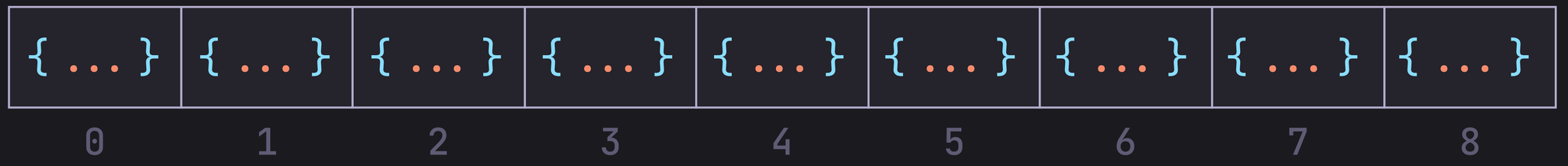
This is what the whole list looks like

Graphs

```
0: {1}
1: {0, 2, 3, 6, 7, 8}
2: {1, 3}
3: {1, 2, 4, 5, 6}
4: {3, 6}
5: {3, 6}
6: {1, 3, 5, 7}
7: {1, 6, 8}
8: {1, 7}
```

```
std::vector<std::unordered_set<int>>
```

Graphs




```
std::vector<std::unordered_set<int>>
```

An adjacency list is space efficient and easy to work with

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Inserting an edge
```

```
0: { }
1: { }
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Inserting an edge
std::set<int> neighbours1 = adjacencyList[1];
neighbours1.insert(0);
```

```
0: { }
1: {0}
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Inserting an edge

adjacencyList[1].insert(0);
```

```
0: { }
1: {0}
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Inserting an edge
adjacencyList[1].insert(0);
adjacencyList[0].insert(1);
```

```
0: {1}
1: {0}
2: { }
3: { }
4: { }
5: { }
6: { }
7: { }
8: { }
```

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Checking if there is an edge

adjacencyList[3].contains(5);
```

```
0: {1}
1: {0, 2, 3, 6, 7, 8}
2: {1, 3}
3: {1, 2, 4, 5, 6}
4: {3, 6}
5: {3, 6}
6: {1, 3, 5, 7}
7: {1, 6, 8}
8: {1, 7}
```

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Checking if there is an edge

adjacencyList[3].contains(5);
```

```
0: {1}
1: {0, 2, 3, 6, 7, 8}
2: {1, 3}
3: {1, 2, 4, 5, 6}
4: {3, 6}
5: {3, 6}
6: {1, 3, 5, 7}
7: {1, 6, 8}
8: {1, 7}
```

Graphs

```
std::vector<std::unordered_set<int>> adjacencyList(9);


// Checking if there is an edge

adjacencyList[3].contains(5);
```

```
0: {1}
1: {0, 2, 3, 6, 7, 8}
2: {1, 3}
3: {1, 2, 4, 5, 6}
4: {3, 6}
5: {3, 6}
6: {1, 3, 5, 7}
7: {1, 6, 8}
8: {1, 7}
```

Graph Search Algorithms

There are two really famous algorithms

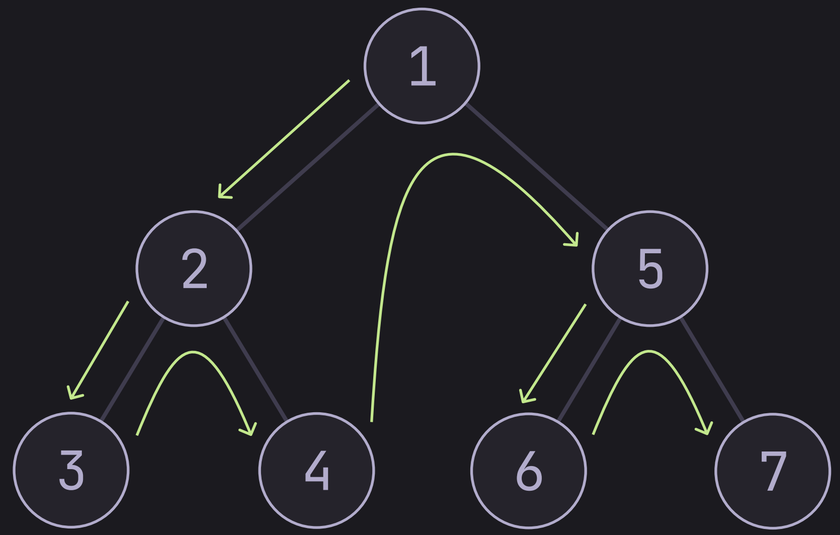
Depth First Search (DFS)

Breadth First Search (BFS)

And the good thing is they only differ very slightly in terms of coding them

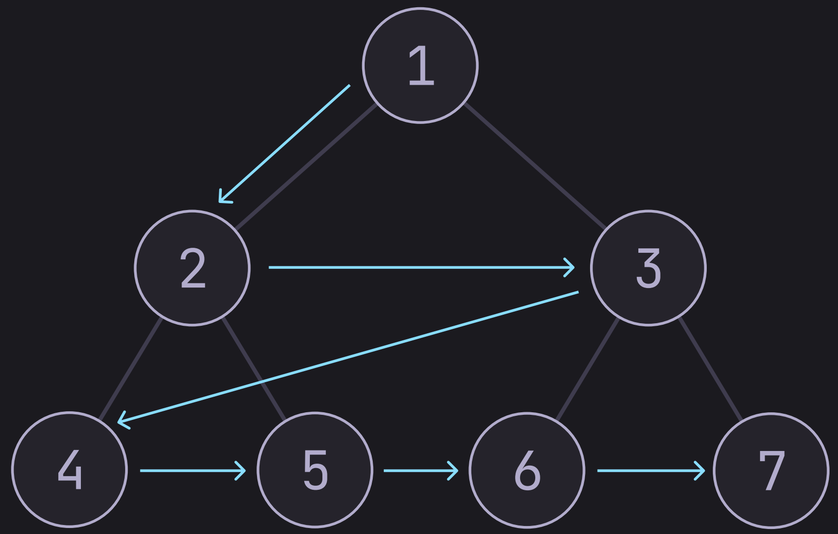
Graph Search Algorithms

Graph Search Algorithms



DFS

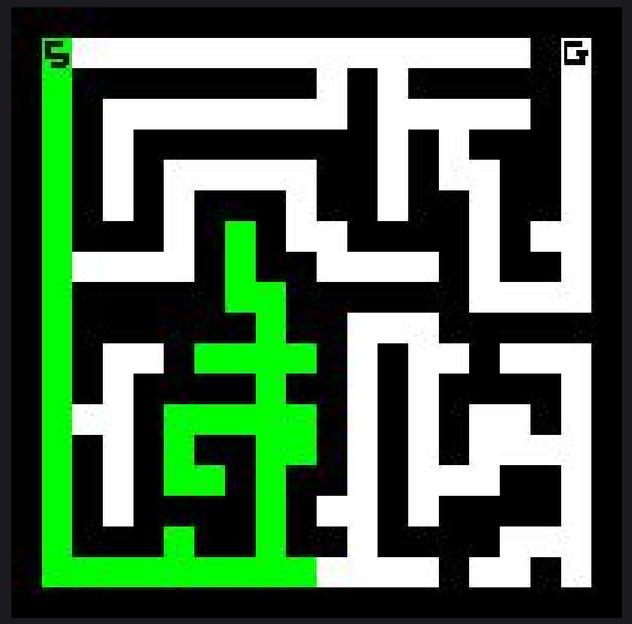
Drill down all the way down



BFS

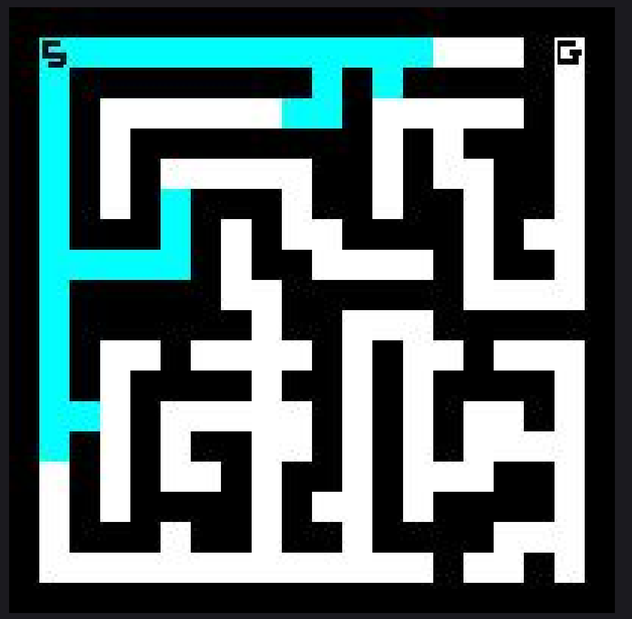
Explore Layer by layer

Graph Search Algorithms



DFS

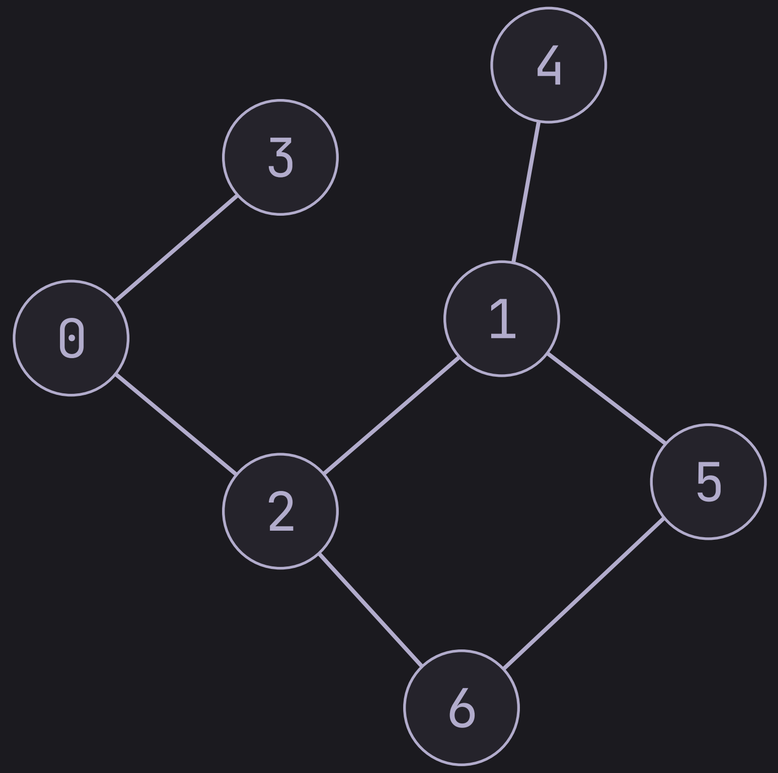
Like how a human
would search



BFS

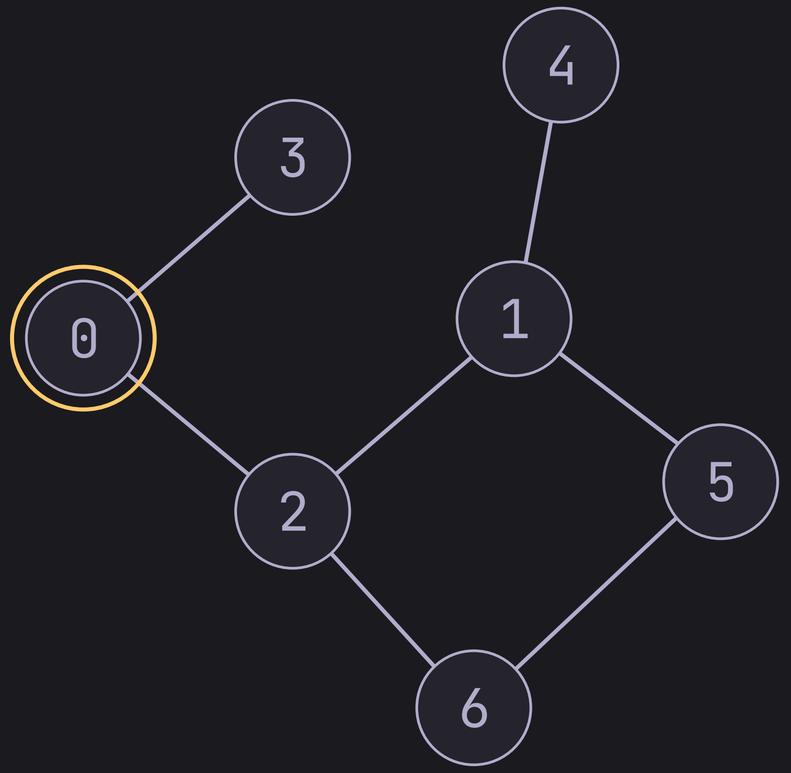
Like how a water
would flow

Depth First Search

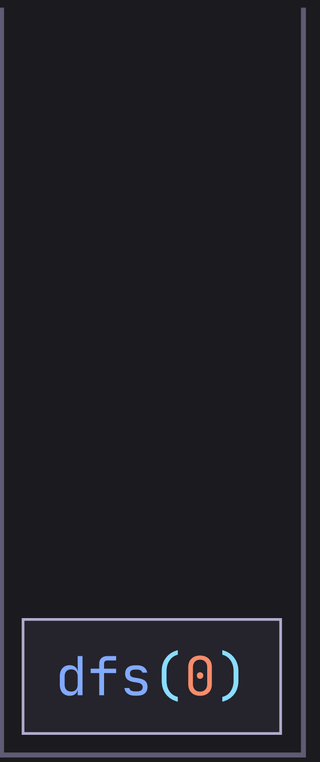


Suppose we have this graph and we want to traverse it

Depth First Search

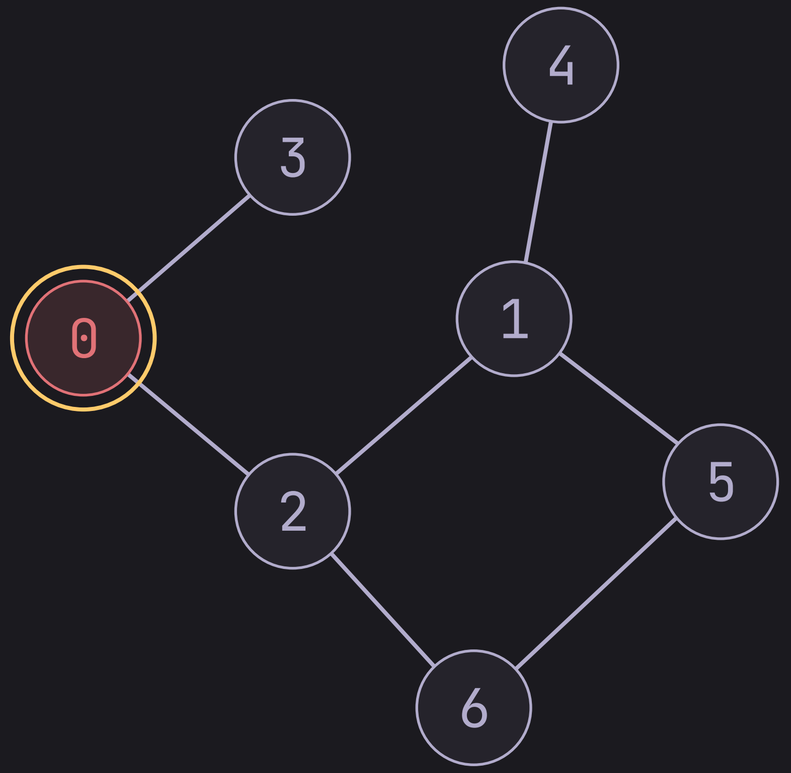



```
⟶ void dfs(int v, std::vector<bool>& visited) {
      visited[v] = true;
      std::cout << v << " ";
      for (auto u : vec[v]) {
        if (!visited[u]) {
          dfs(u, visited);
        }
      }
    }
```

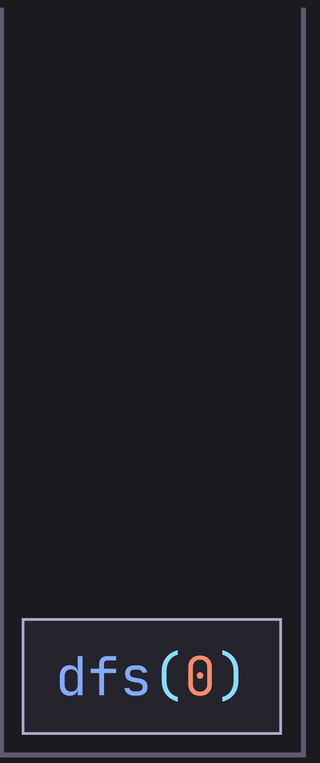


Call Stack

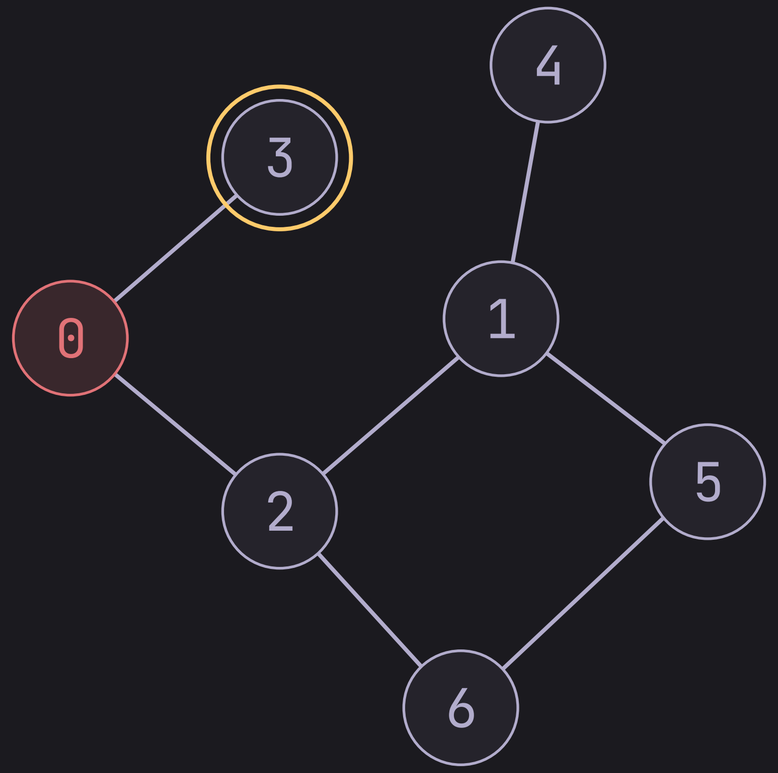
Depth First Search



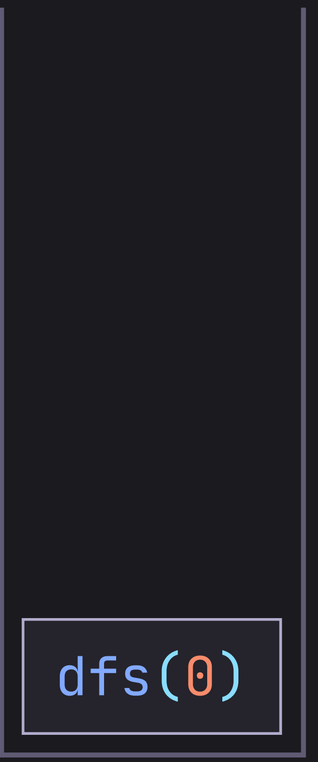

```
void dfs(int v, std::vector<bool>& visited) {
⟶  visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```



Depth First Search

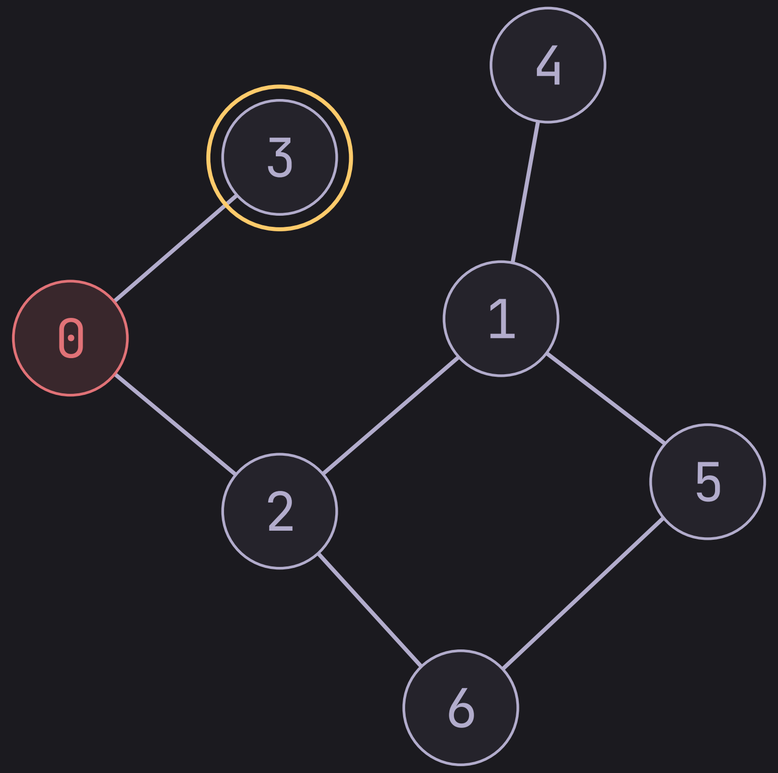



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
    ⟶ dfs(u, visited);
      }
    }
  }
```

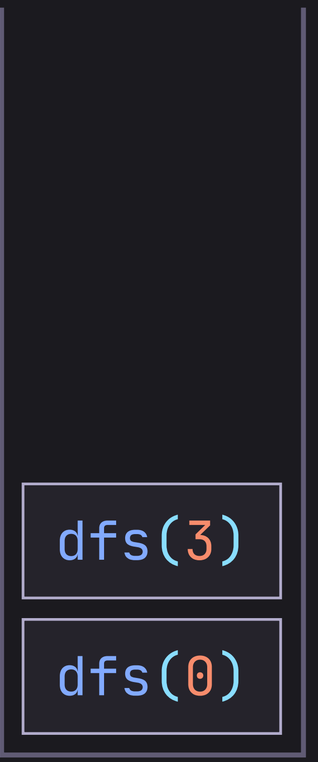


Call Stack

Depth First Search

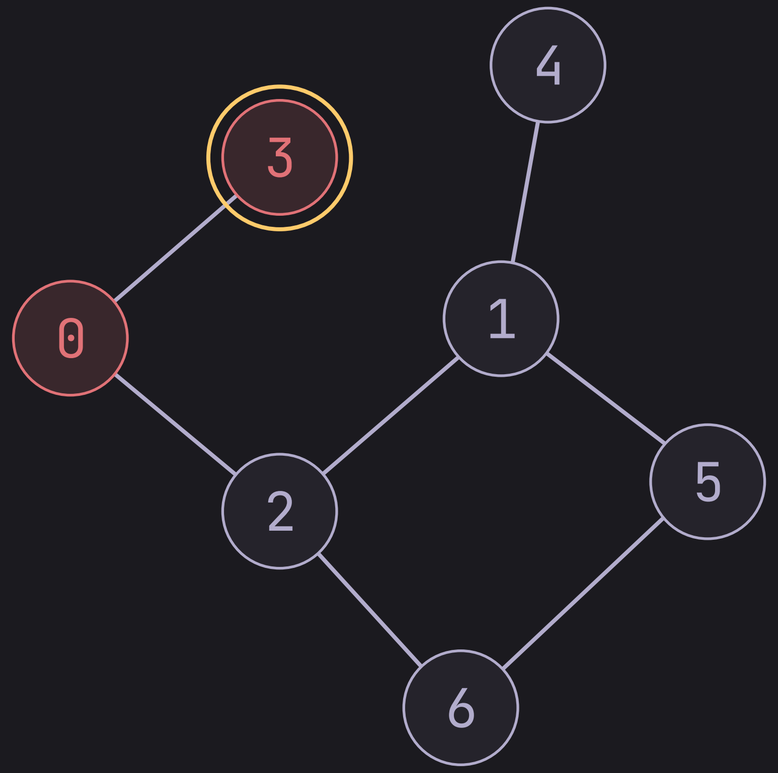



```
⟶ void dfs(int v, std::vector<bool>& visited) {
      visited[v] = true;
      std::cout << v << " ";
      for (auto u : vec[v]) {
        if (!visited[u]) {
          dfs(u, visited);
        }
      }
    }
```

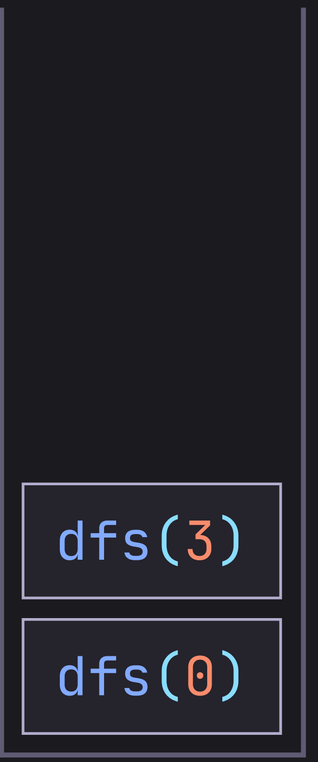


Call Stack

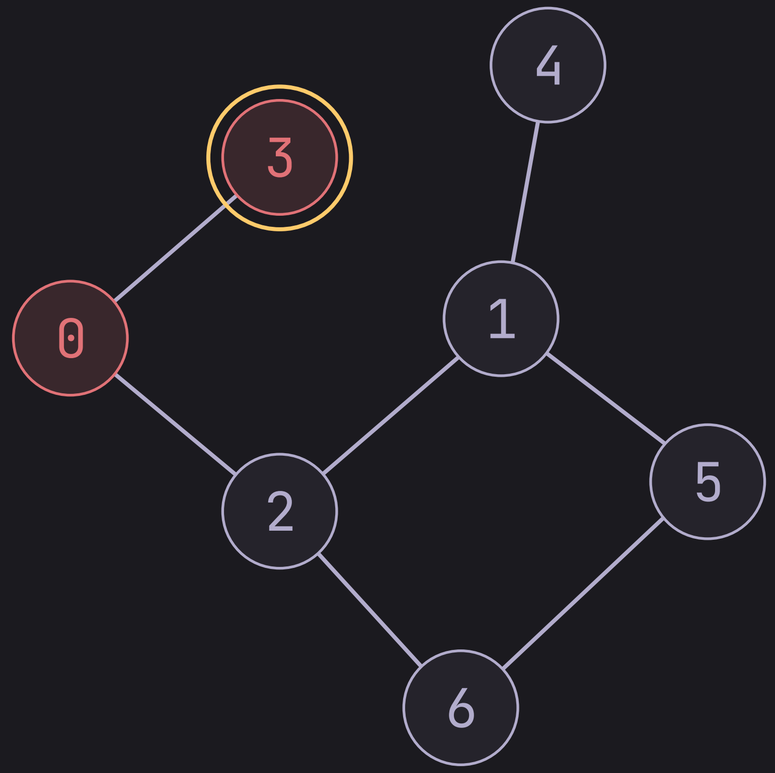
Depth First Search



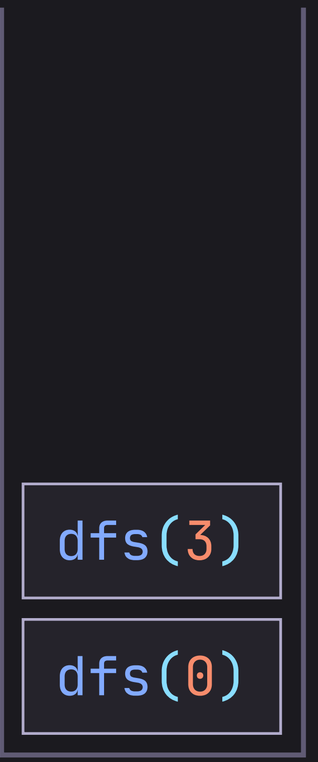

```
void dfs(int v, std::vector<bool>& visited) {
⟶  visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```



Depth First Search

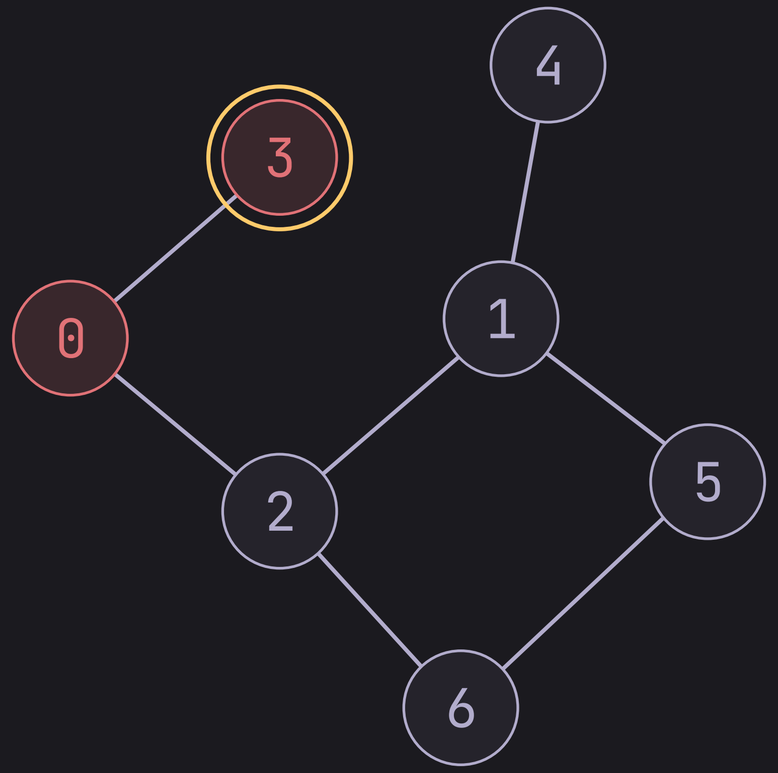



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
→   for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```

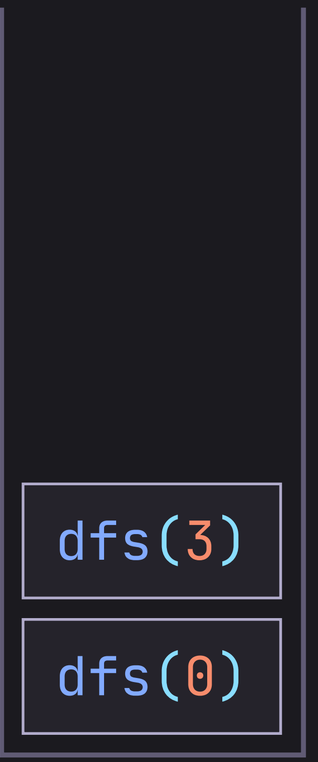


Call Stack

Depth First Search

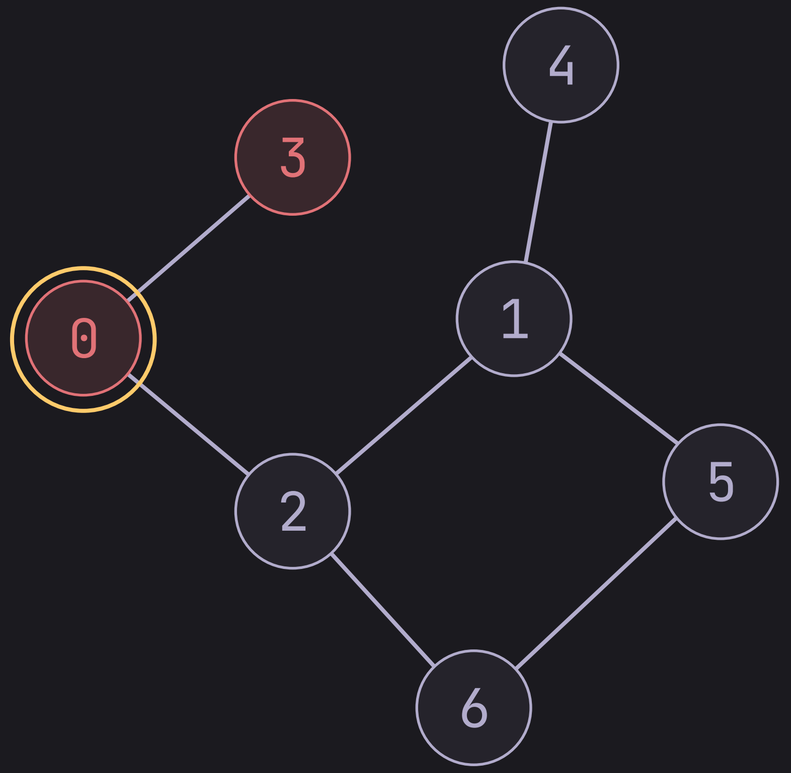



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
→ }
```

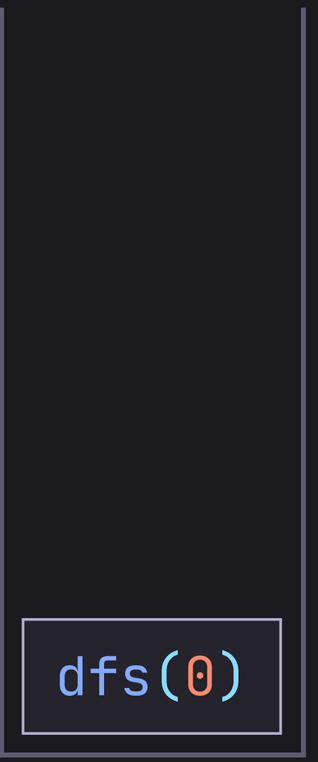


Call Stack

Depth First Search

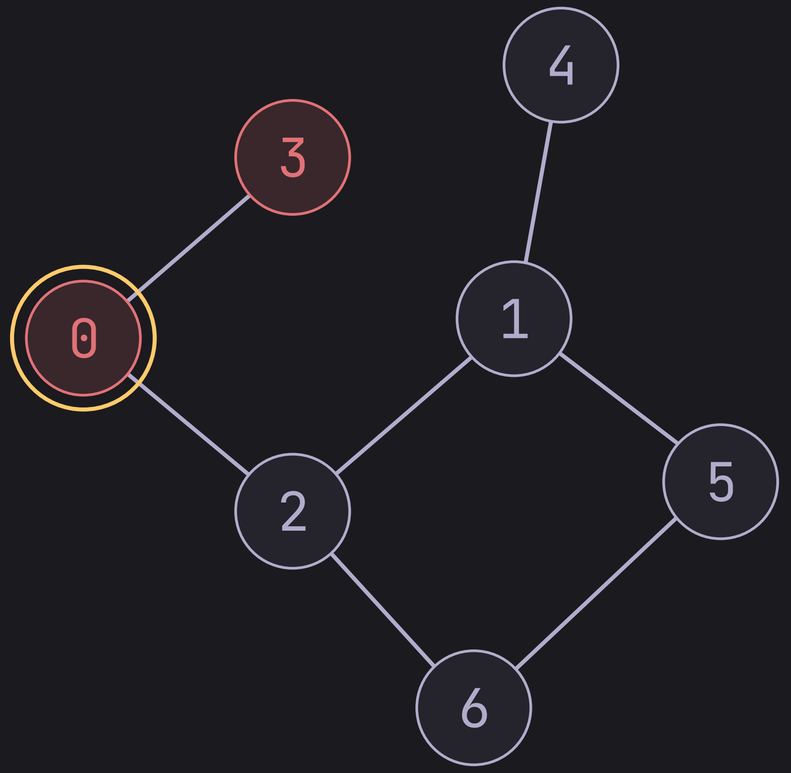



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
    ⟶ dfs(u, visited);
      }
    }
  }
```

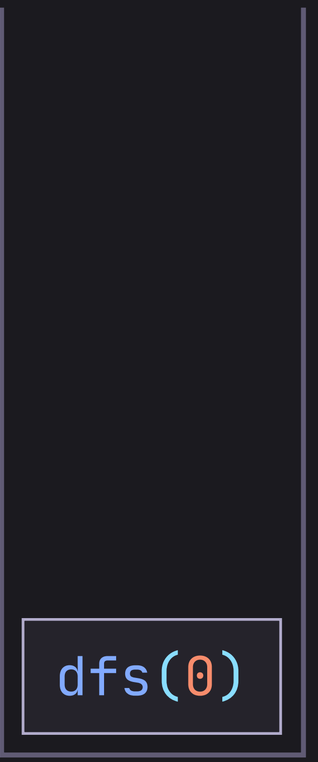


Call Stack

Depth First Search

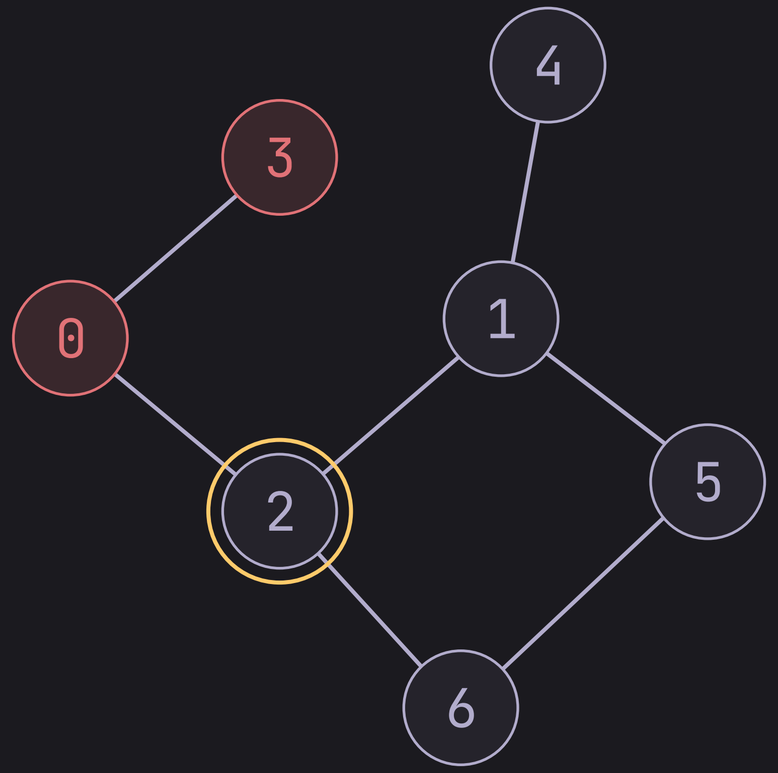



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
⟶  for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```

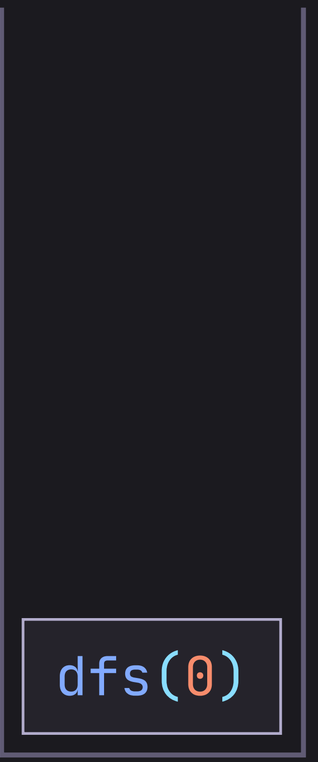


Call Stack

Depth First Search

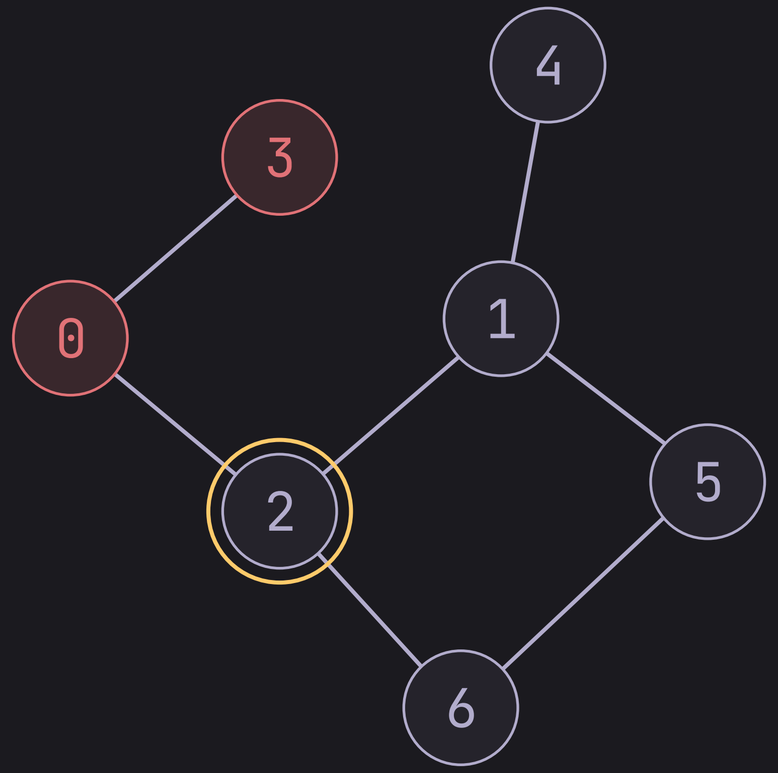



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
    ⟶ dfs(u, visited);
      }
    }
  }
```



Call Stack

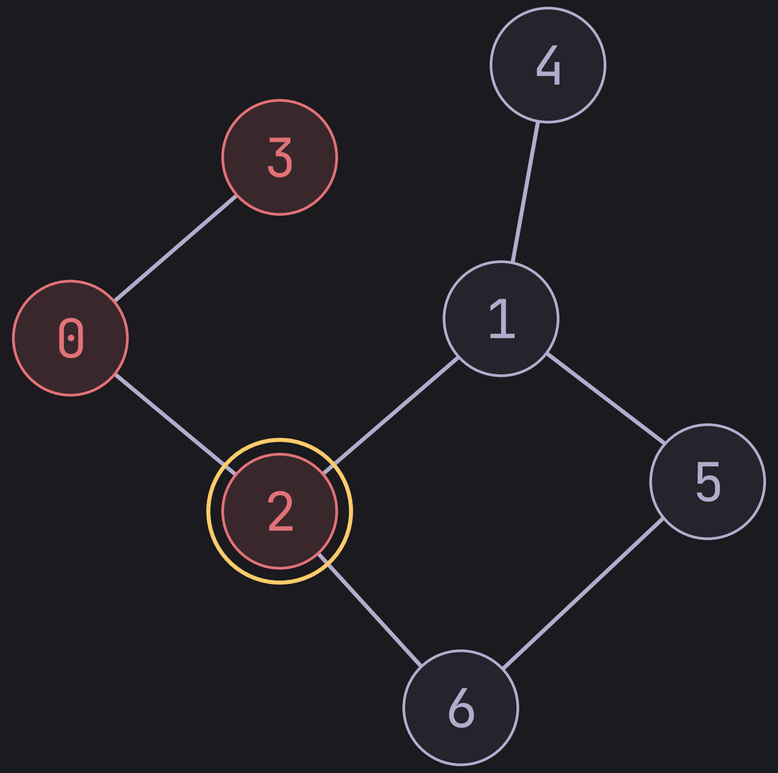
Depth First Search




```
⟶ void dfs(int v, std::vector<bool>& visited) {
      visited[v] = true;
      std::cout << v << " ";
      for (auto u : vec[v]) {
        if (!visited[u]) {
          dfs(u, visited);
        }
      }
    }
```

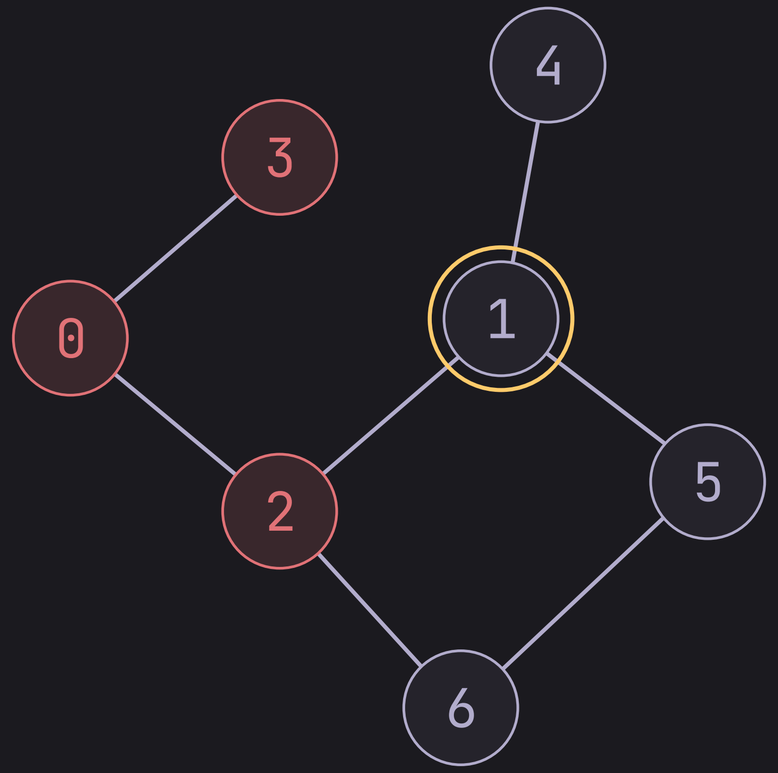
Call Stack

Depth First Search

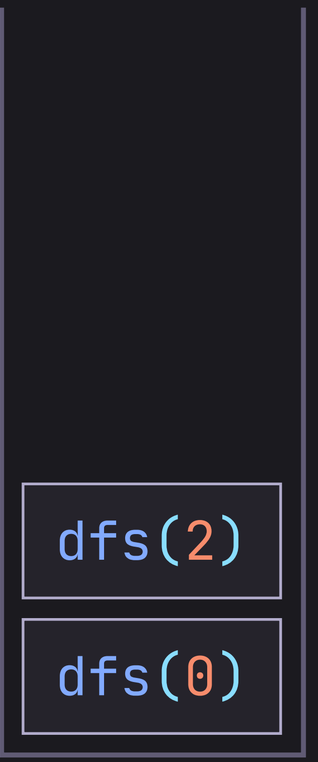



```
void dfs(int v, std::vector<bool>& visited) {
⟶  visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```

Depth First Search

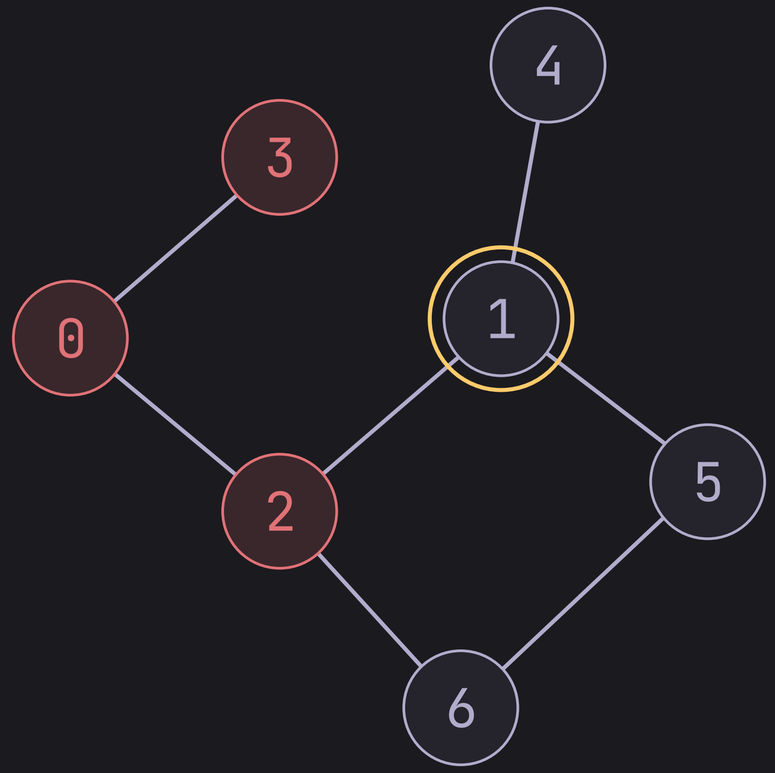



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
    ⟶ dfs(u, visited);
      }
    }
  }
```

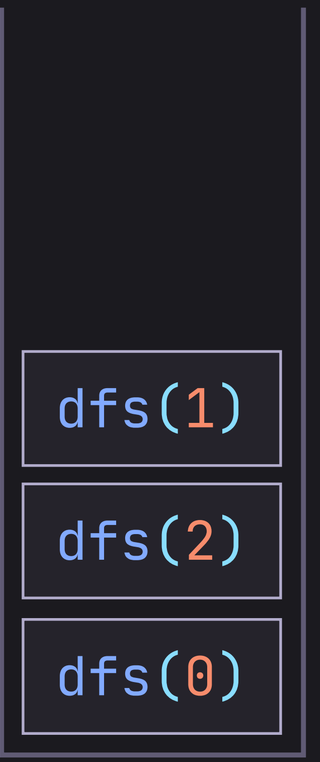


Call Stack

Depth First Search

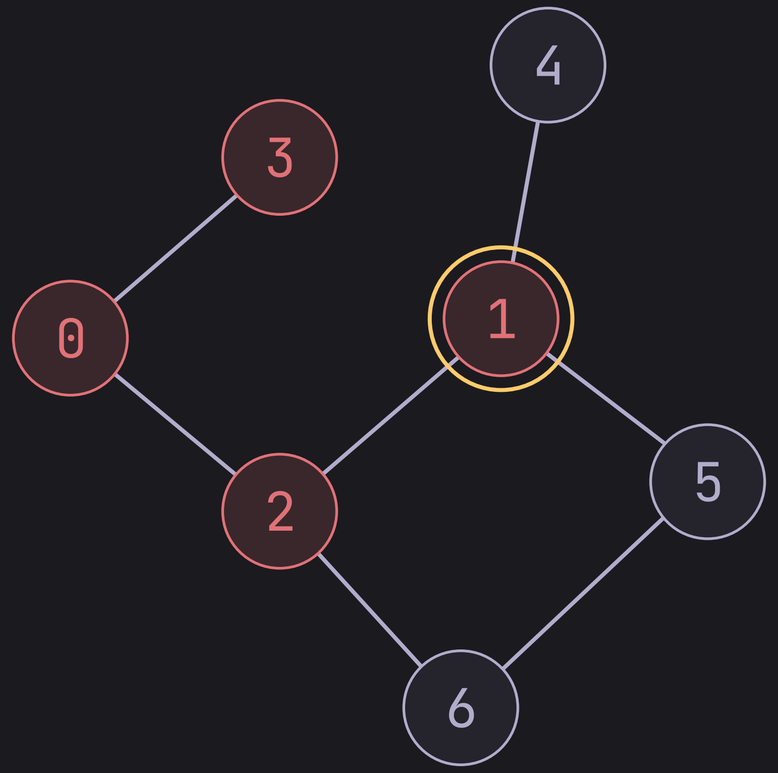



```
⟶ void dfs(int v, std::vector<bool>& visited) {
      visited[v] = true;
      std::cout << v << " ";
      for (auto u : vec[v]) {
        if (!visited[u]) {
          dfs(u, visited);
        }
      }
    }
```

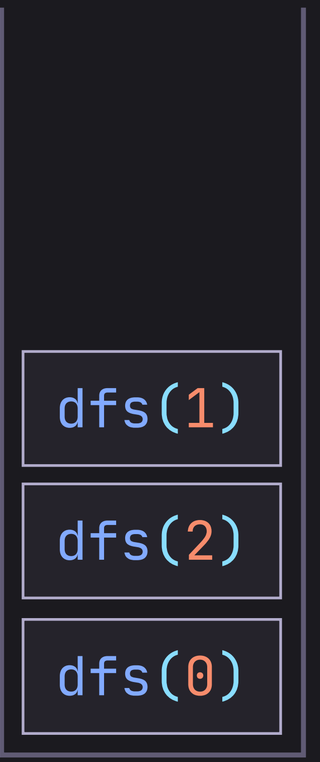


Call Stack

Depth First Search

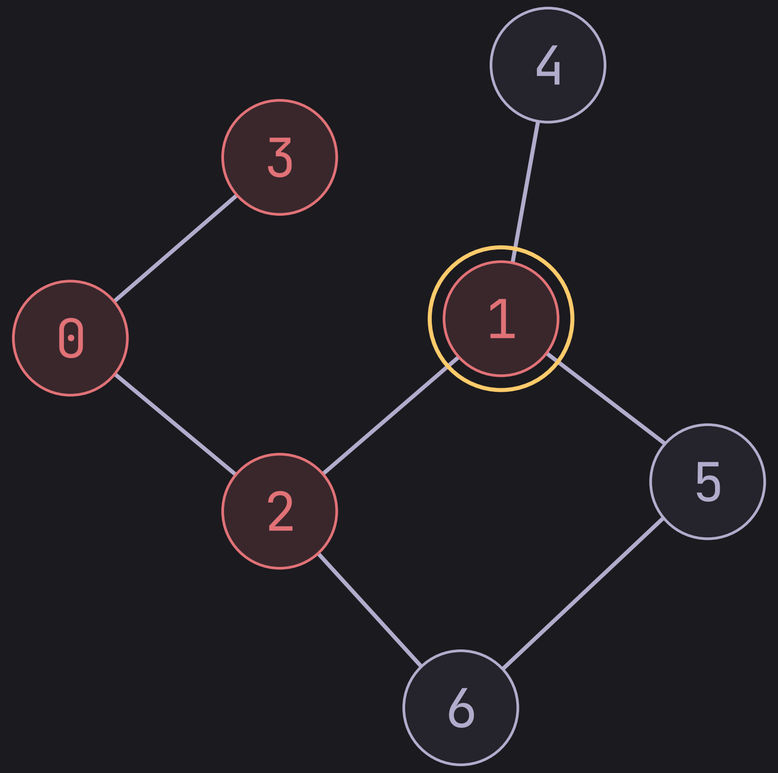



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```

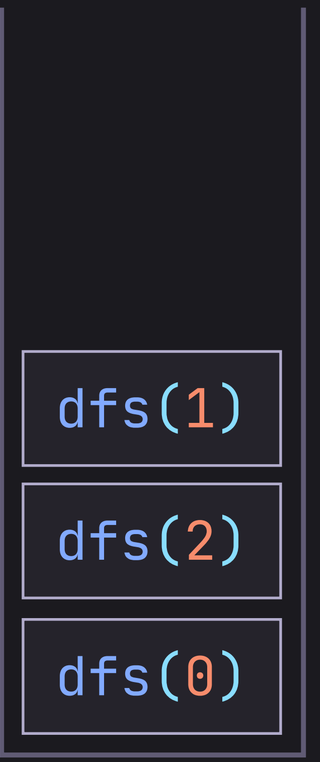


Call Stack

Depth First Search

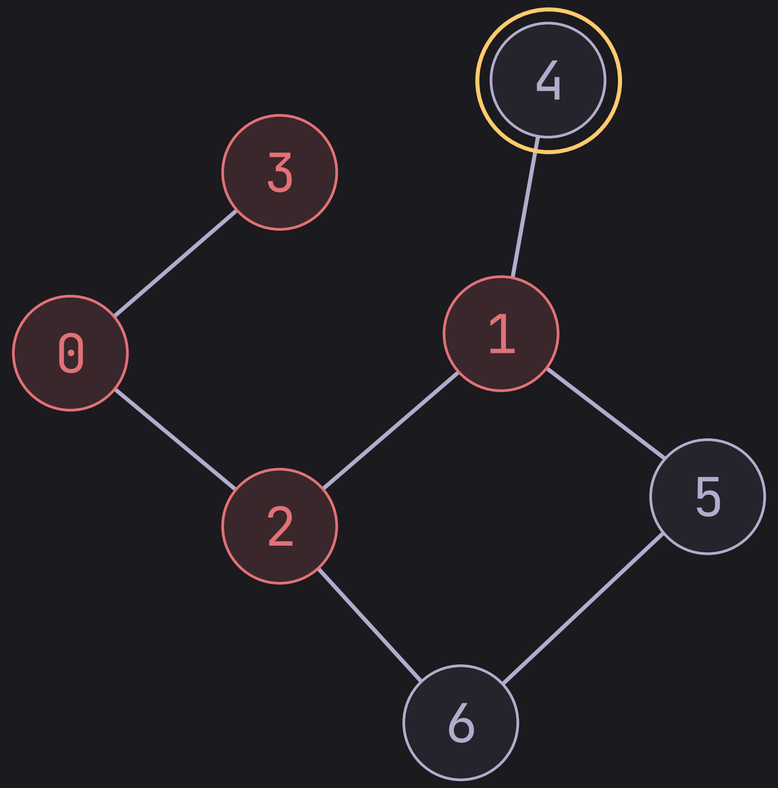



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
    ⟶ dfs(u, visited);
      }
    }
  }
```

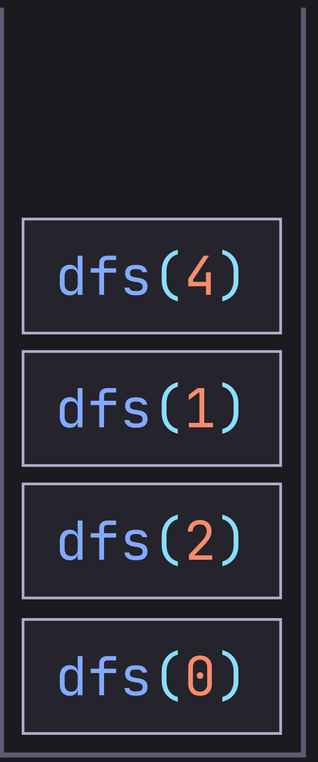


Call Stack

Depth First Search

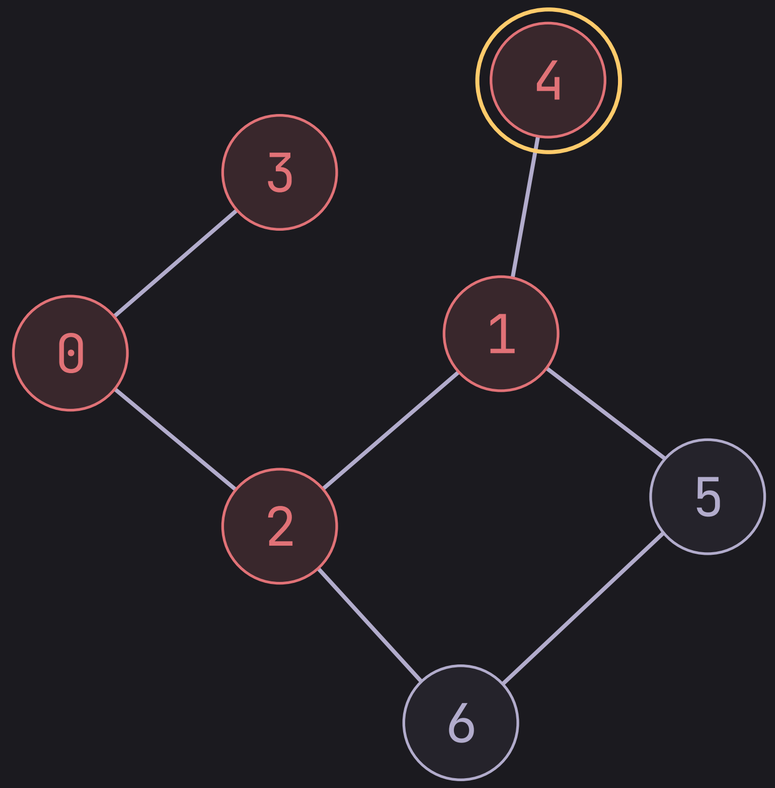



```
⟶ void dfs(int v, std::vector<bool>& visited) {
      visited[v] = true;
      std::cout << v << " ";
      for (auto u : vec[v]) {
        if (!visited[u]) {
          dfs(u, visited);
        }
      }
    }
```



Call Stack

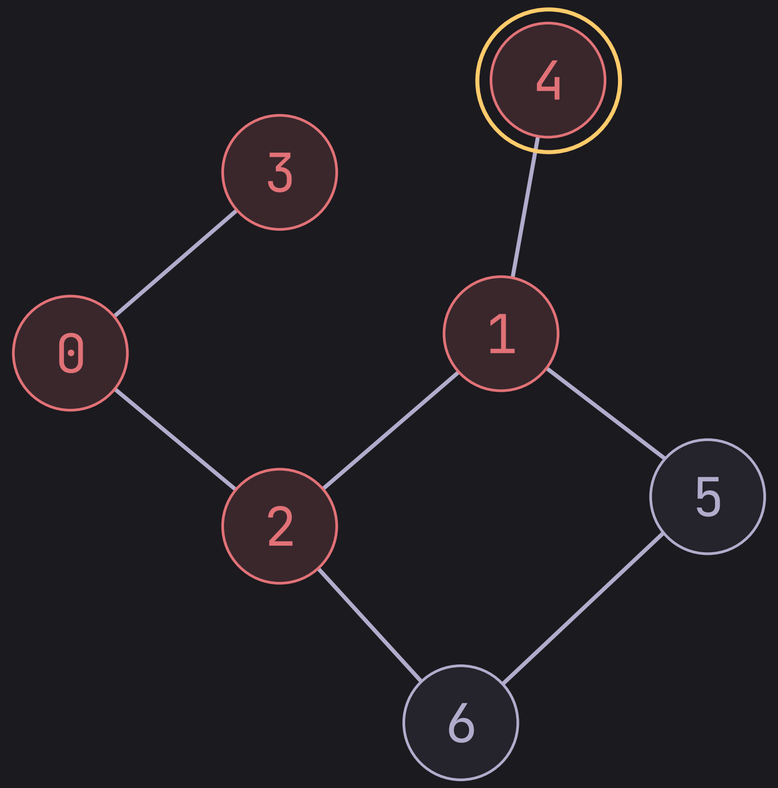
Depth First Search



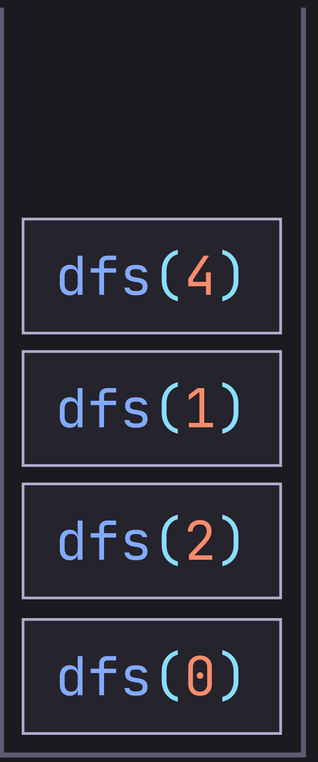

```
void dfs(int v, std::vector<bool>& visited) {
⟶  visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```

Call Stack

Depth First Search

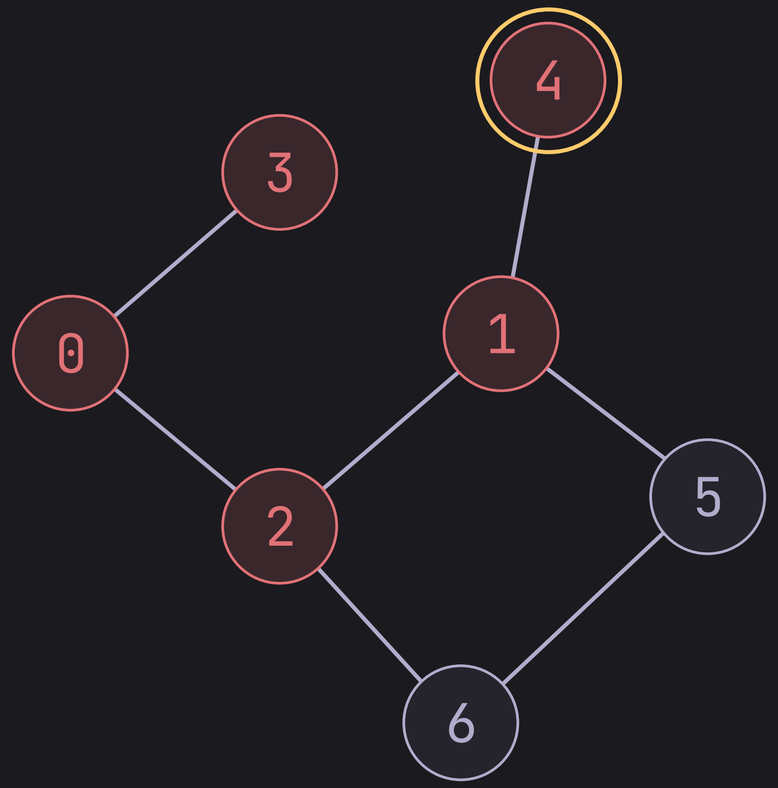



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
→   for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```

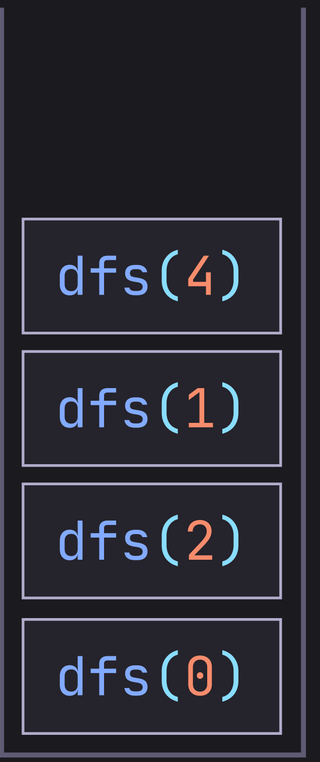


Call Stack

Depth First Search

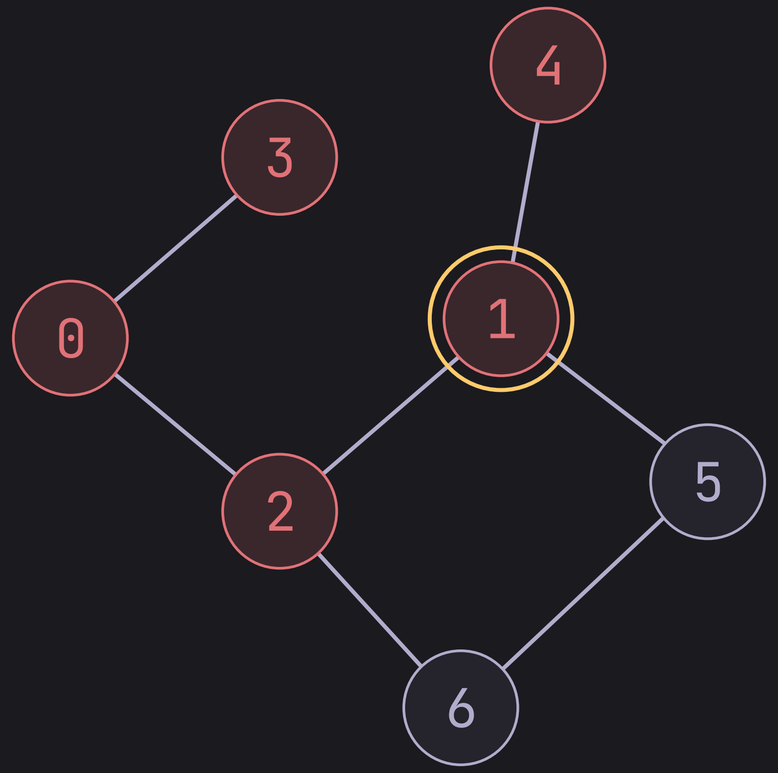



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
}
```

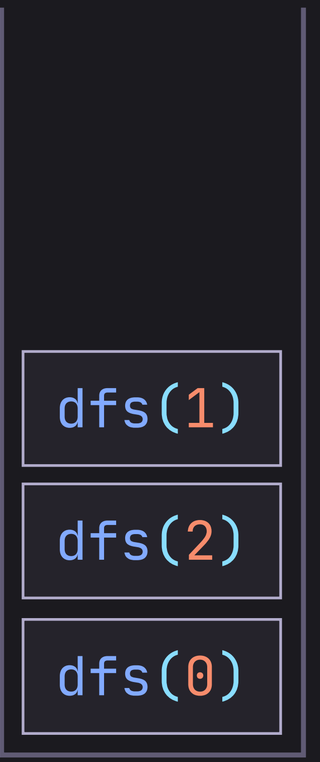


Call Stack

Depth First Search

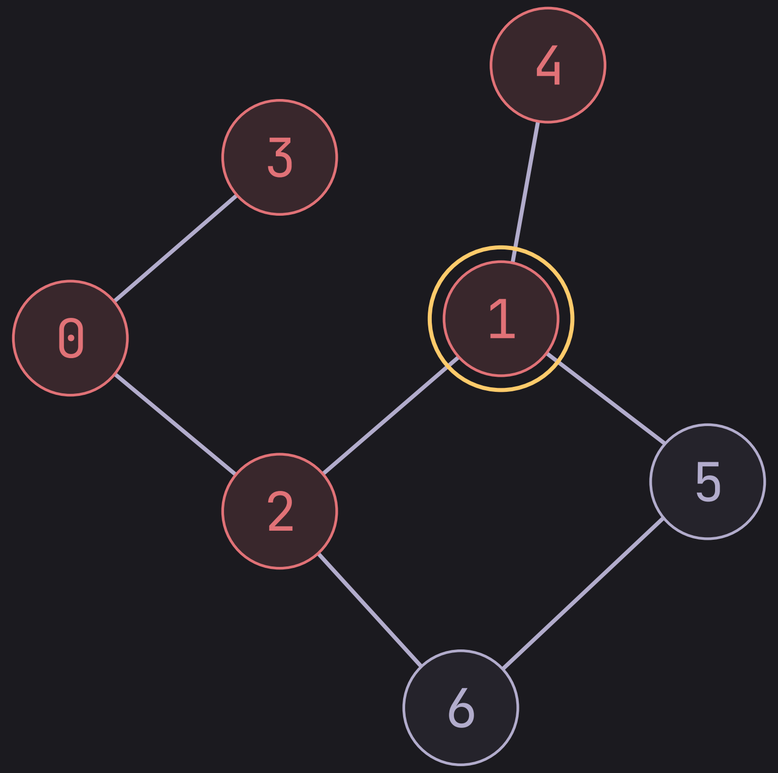



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
    ⟶ dfs(u, visited);
      }
    }
  }
```

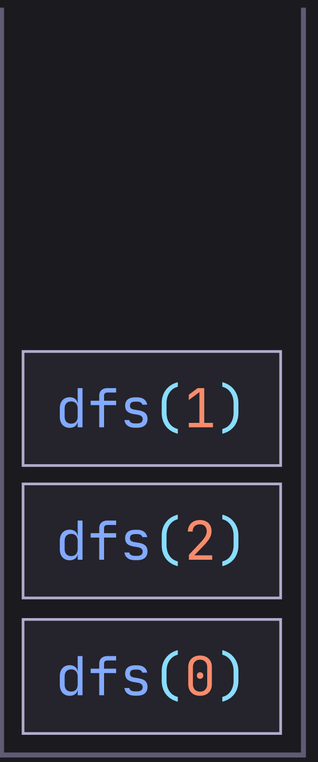


Call Stack

Depth First Search

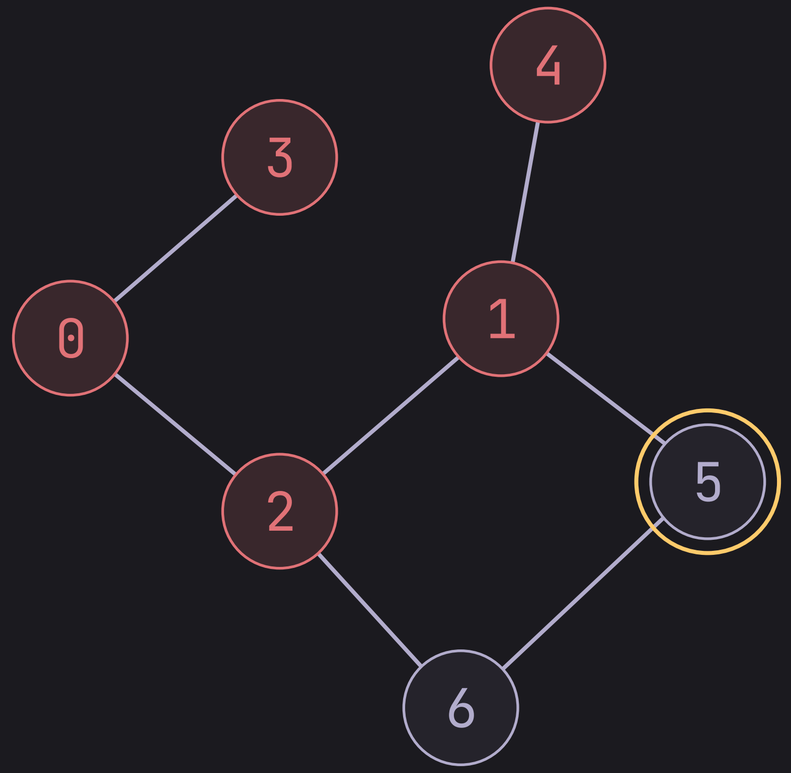



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
→   for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```

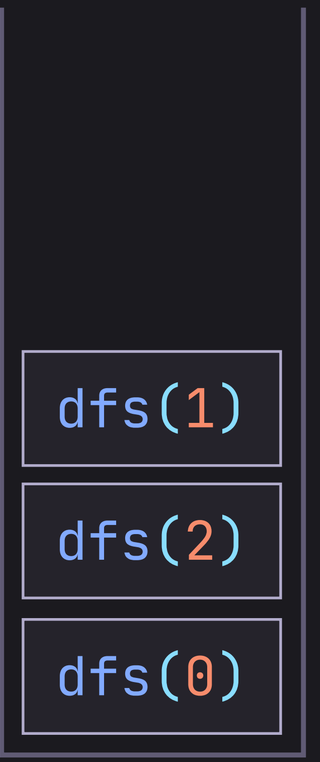


Call Stack

Depth First Search

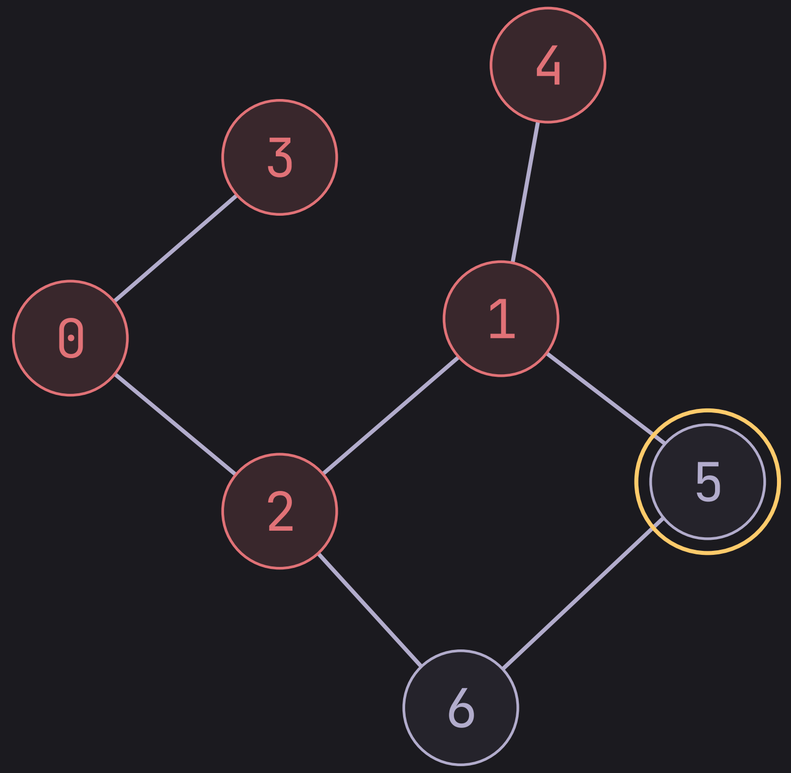



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
    ⟶ dfs(u, visited);
      }
    }
  }
```

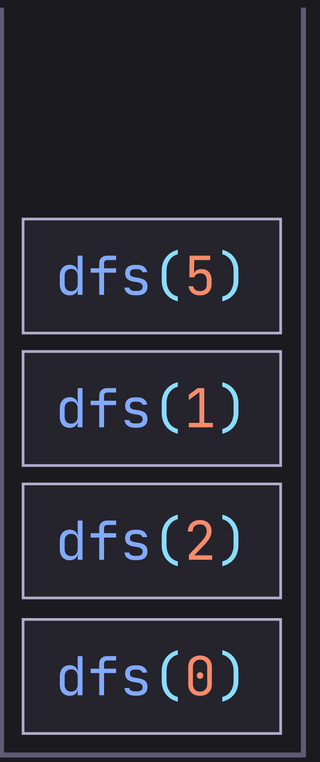


Call Stack

Depth First Search

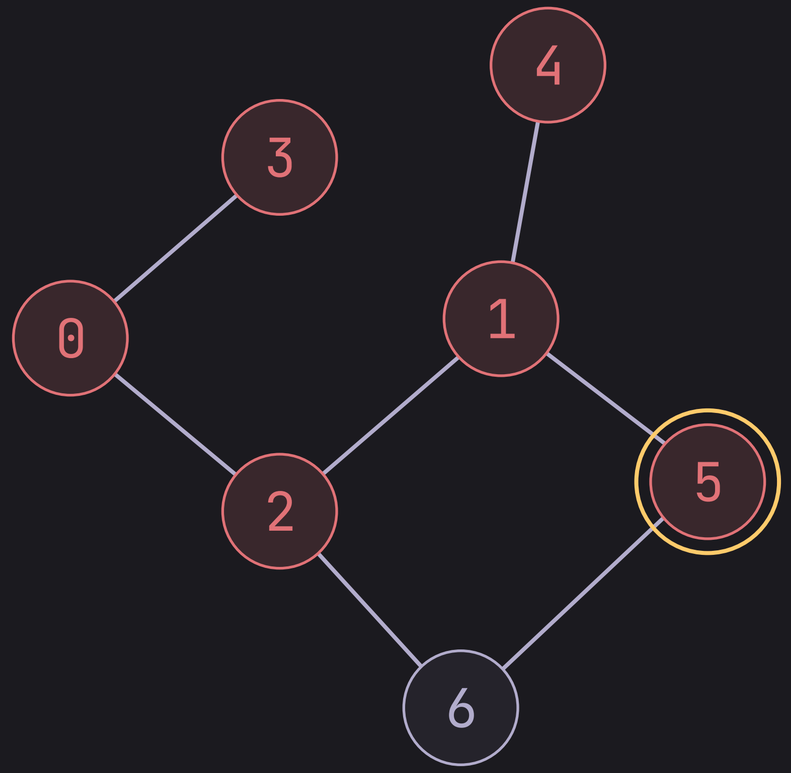



```
⟶ void dfs(int v, std::vector<bool>& visited) {
      visited[v] = true;
      std::cout << v << " ";
      for (auto u : vec[v]) {
        if (!visited[u]) {
          dfs(u, visited);
        }
      }
    }
```

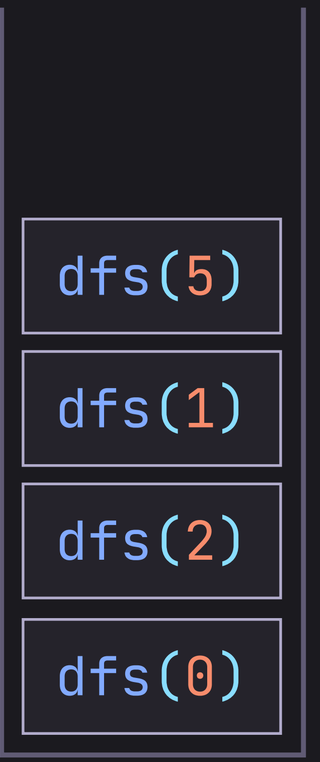


Call Stack

Depth First Search

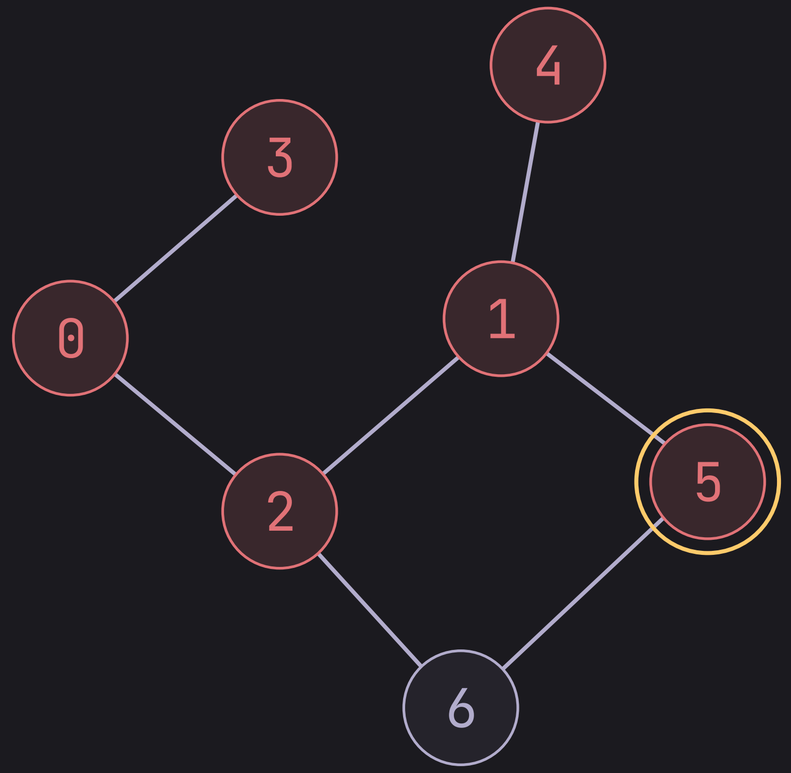



```
void dfs(int v, std::vector<bool>& visited) {
⟶  visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```

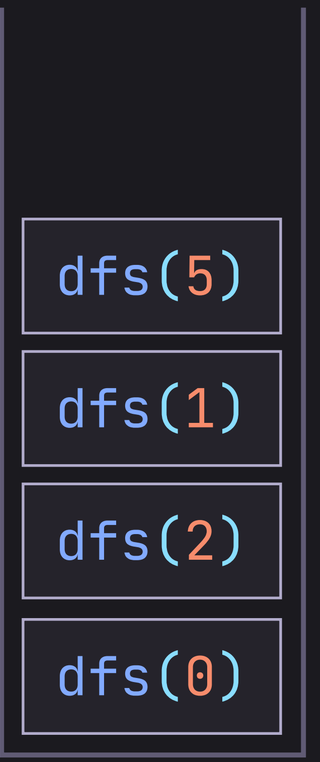


Call Stack

Depth First Search

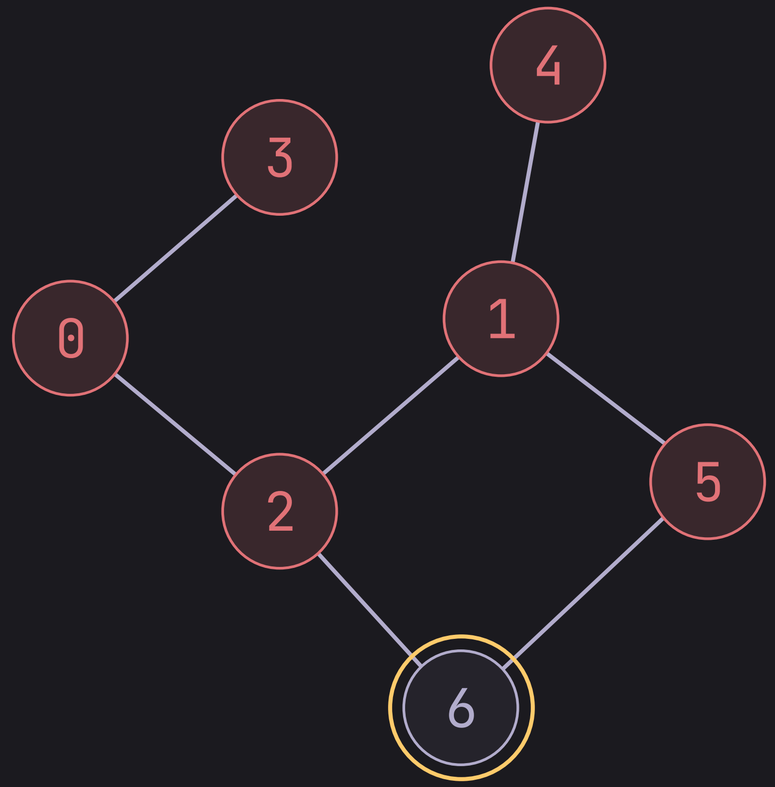



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
→   for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```

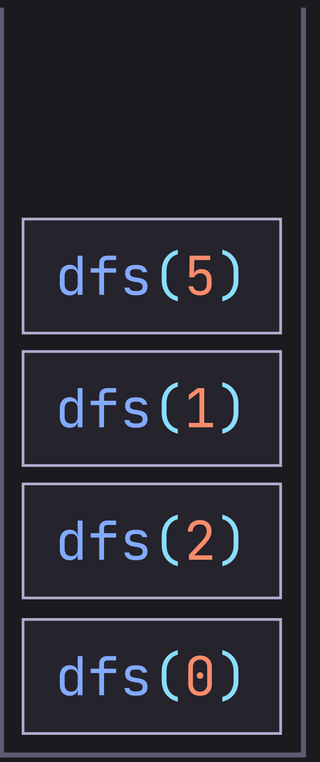


Call Stack

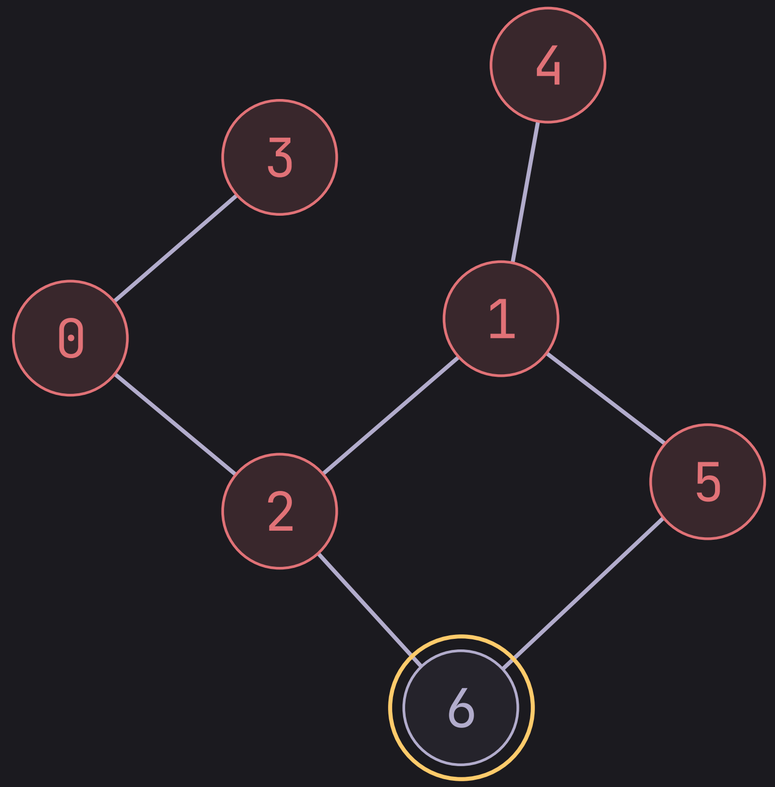
Depth First Search



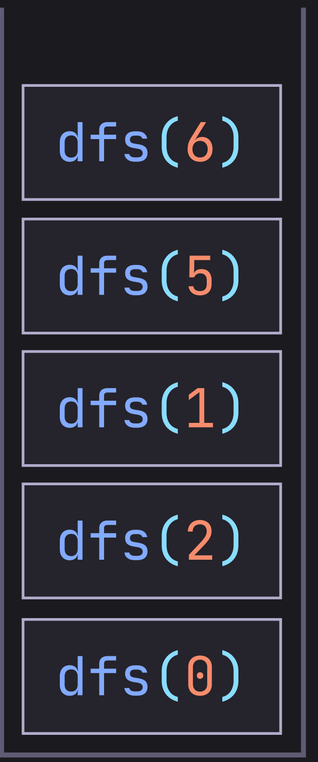

```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
    ⟶ dfs(u, visited);
      }
    }
  }
```



Depth First Search

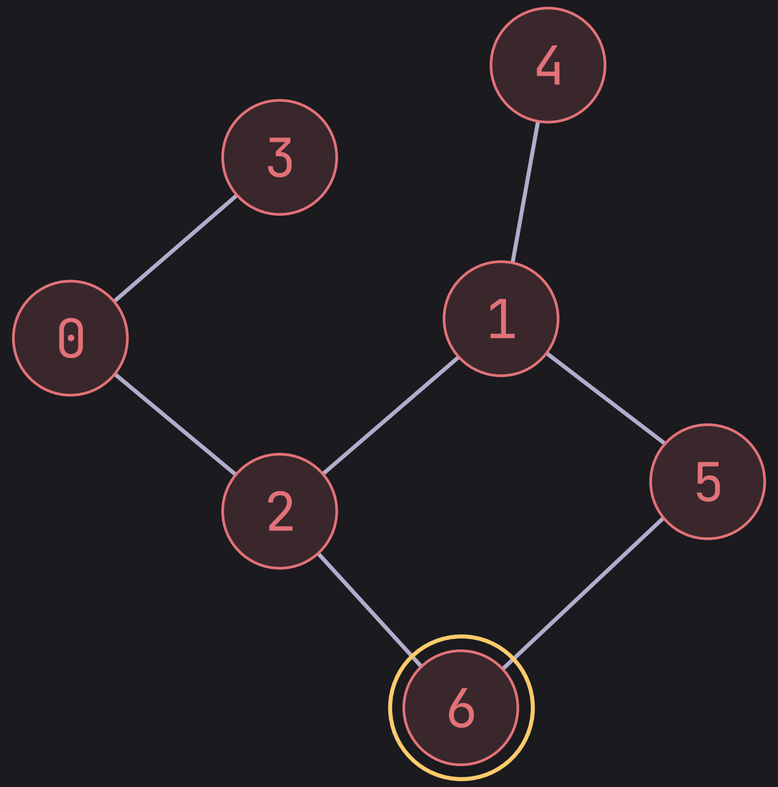



```
⟶ void dfs(int v, std::vector<bool>& visited) {
      visited[v] = true;
      std::cout << v << " ";
      for (auto u : vec[v]) {
        if (!visited[u]) {
          dfs(u, visited);
        }
      }
    }
```

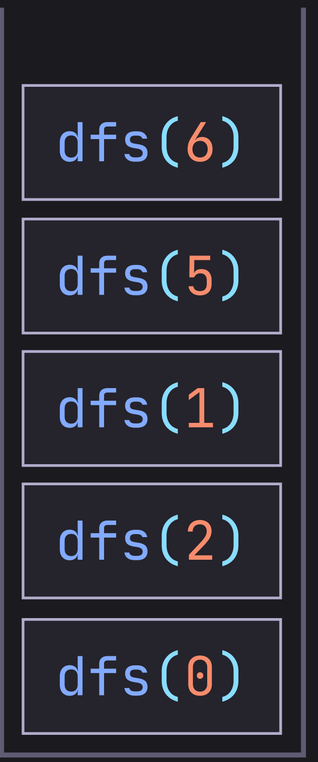


Call Stack

Depth First Search

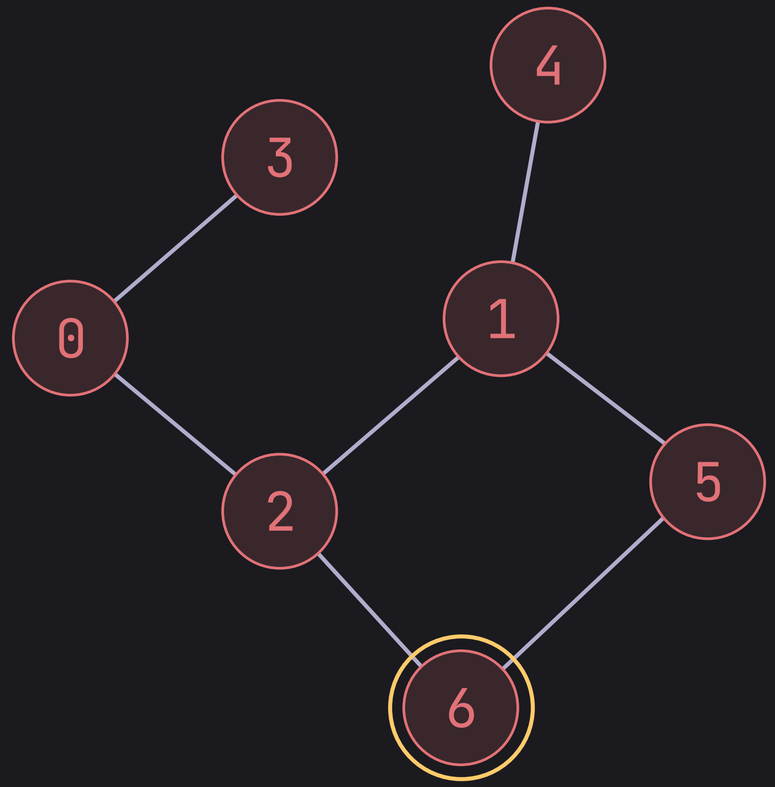



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```



Call Stack

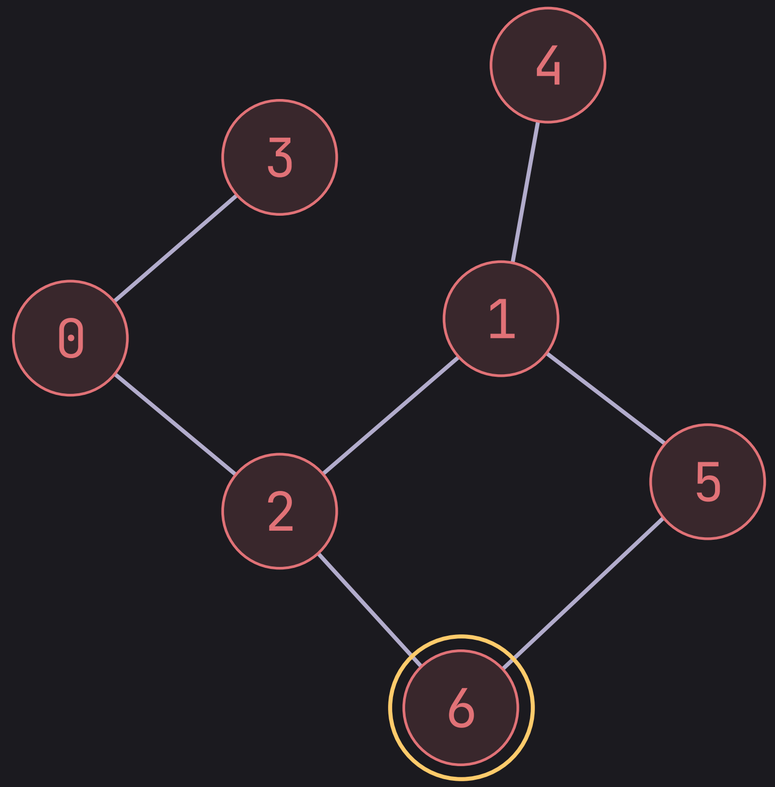
Depth First Search



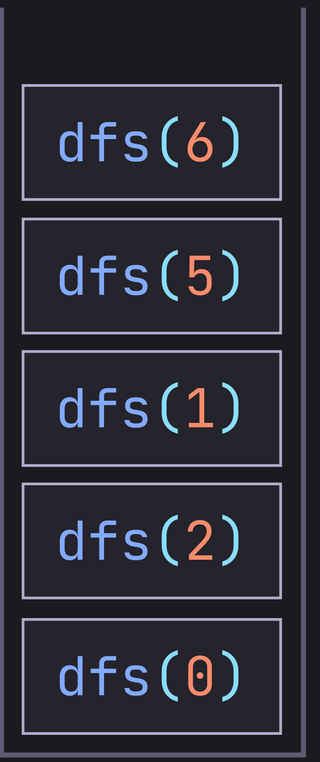

```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
→   for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```

Call Stack

Depth First Search

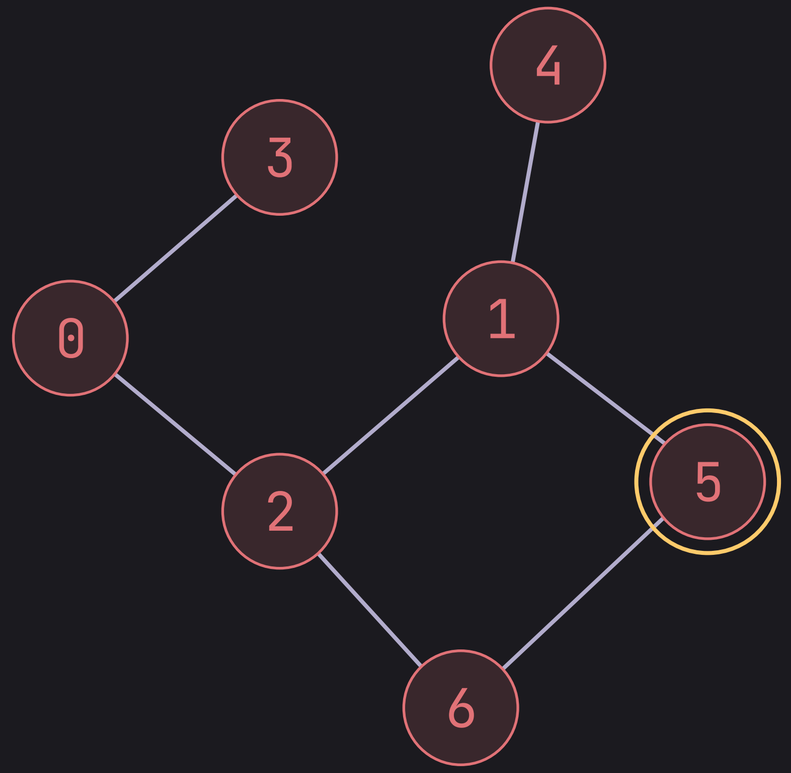



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
}
```

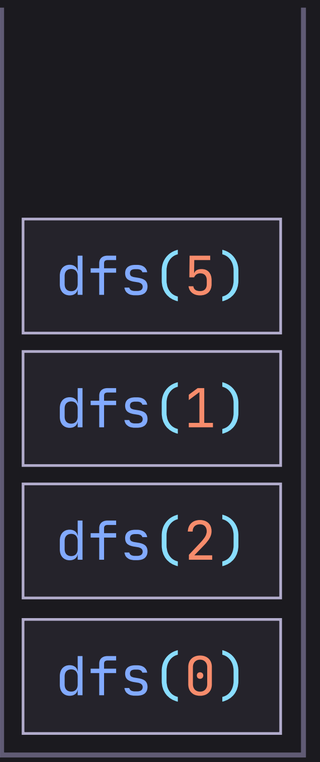


Call Stack

Depth First Search

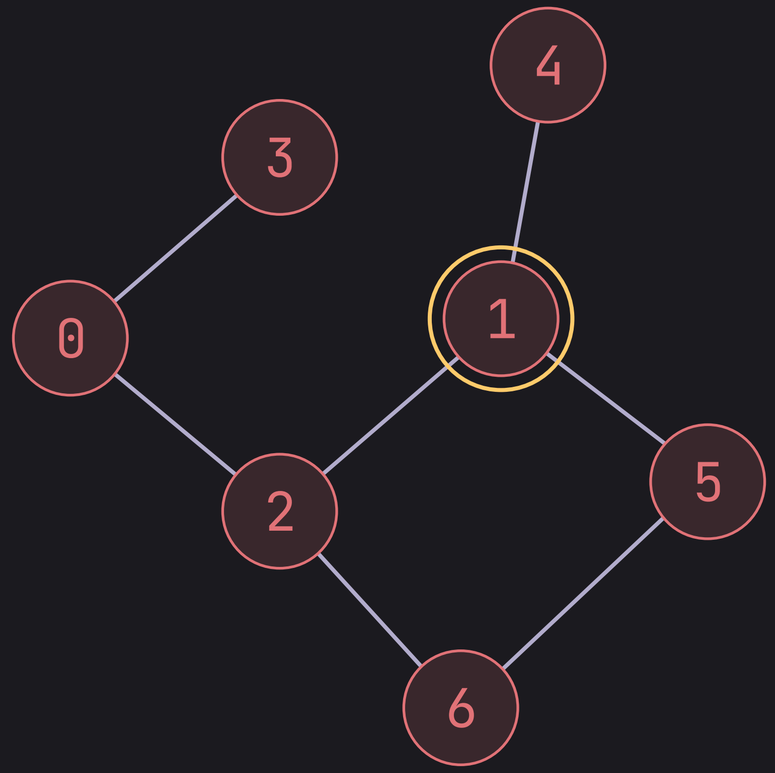



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
    ⟶ dfs(u, visited);
      }
    }
  }
```

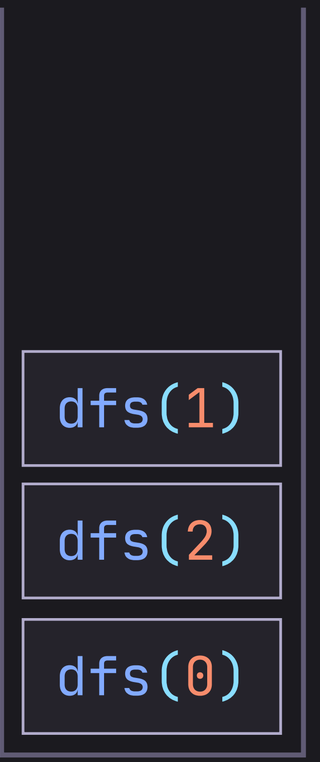


Call Stack

Depth First Search

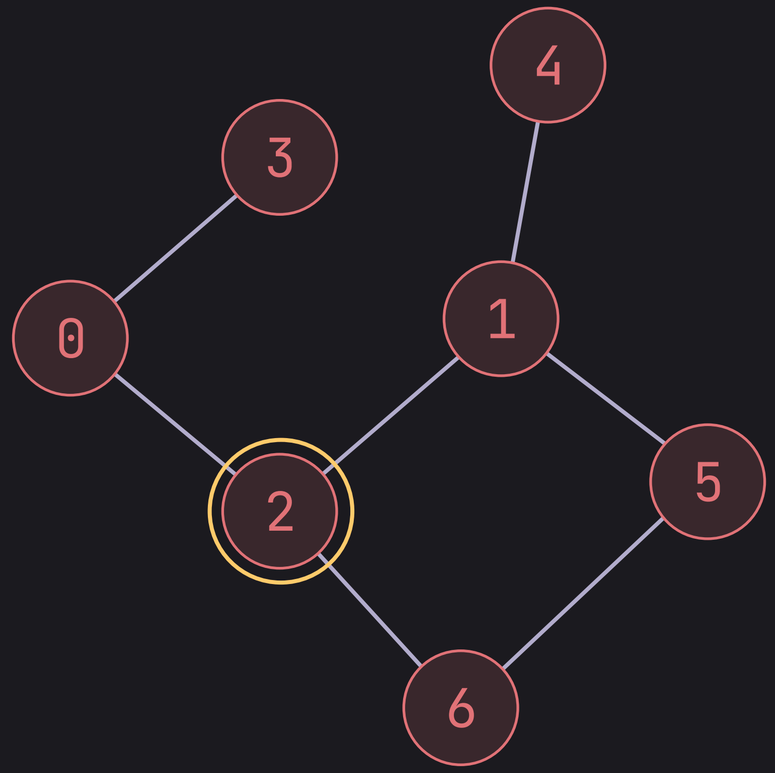



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```

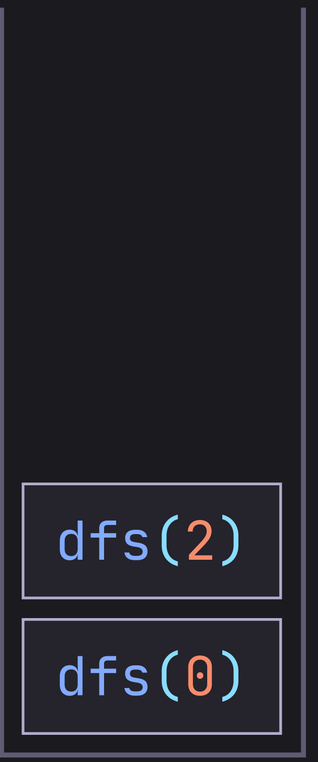


Call Stack

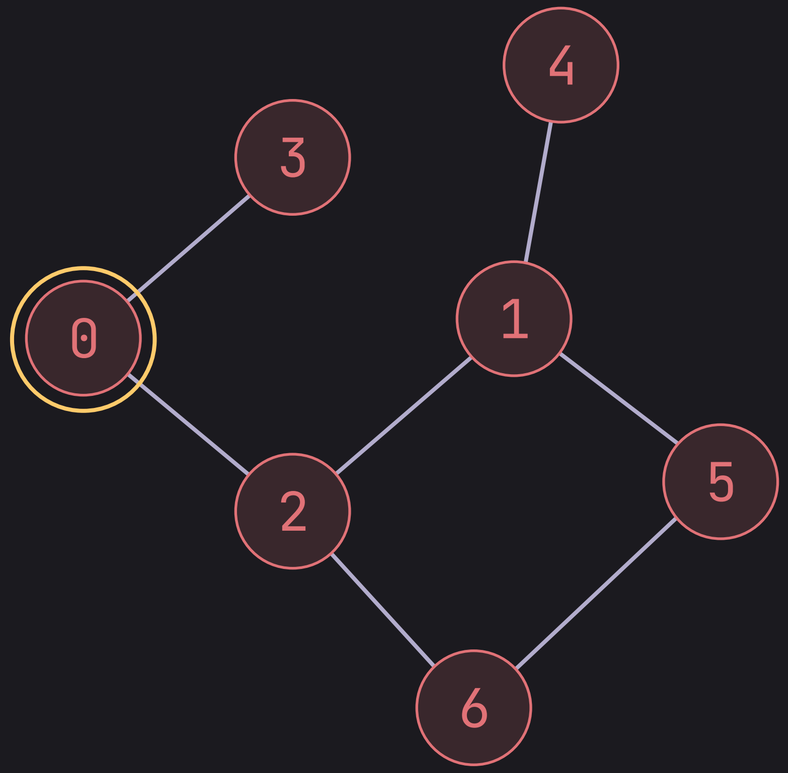
Depth First Search



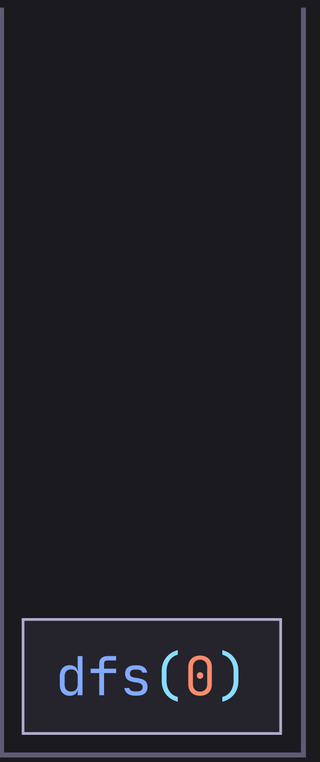

```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```



Depth First Search

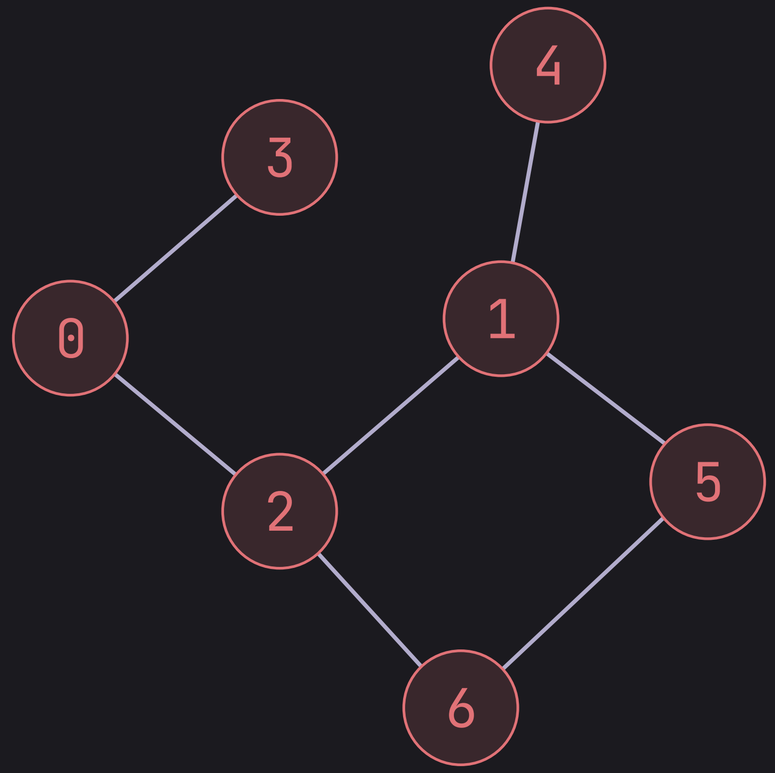



```
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```

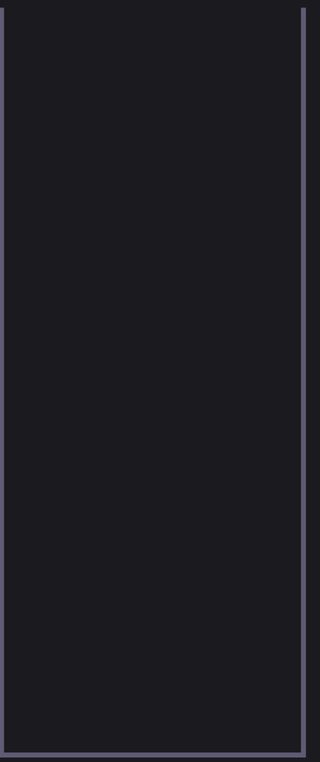


Call Stack

Depth First Search

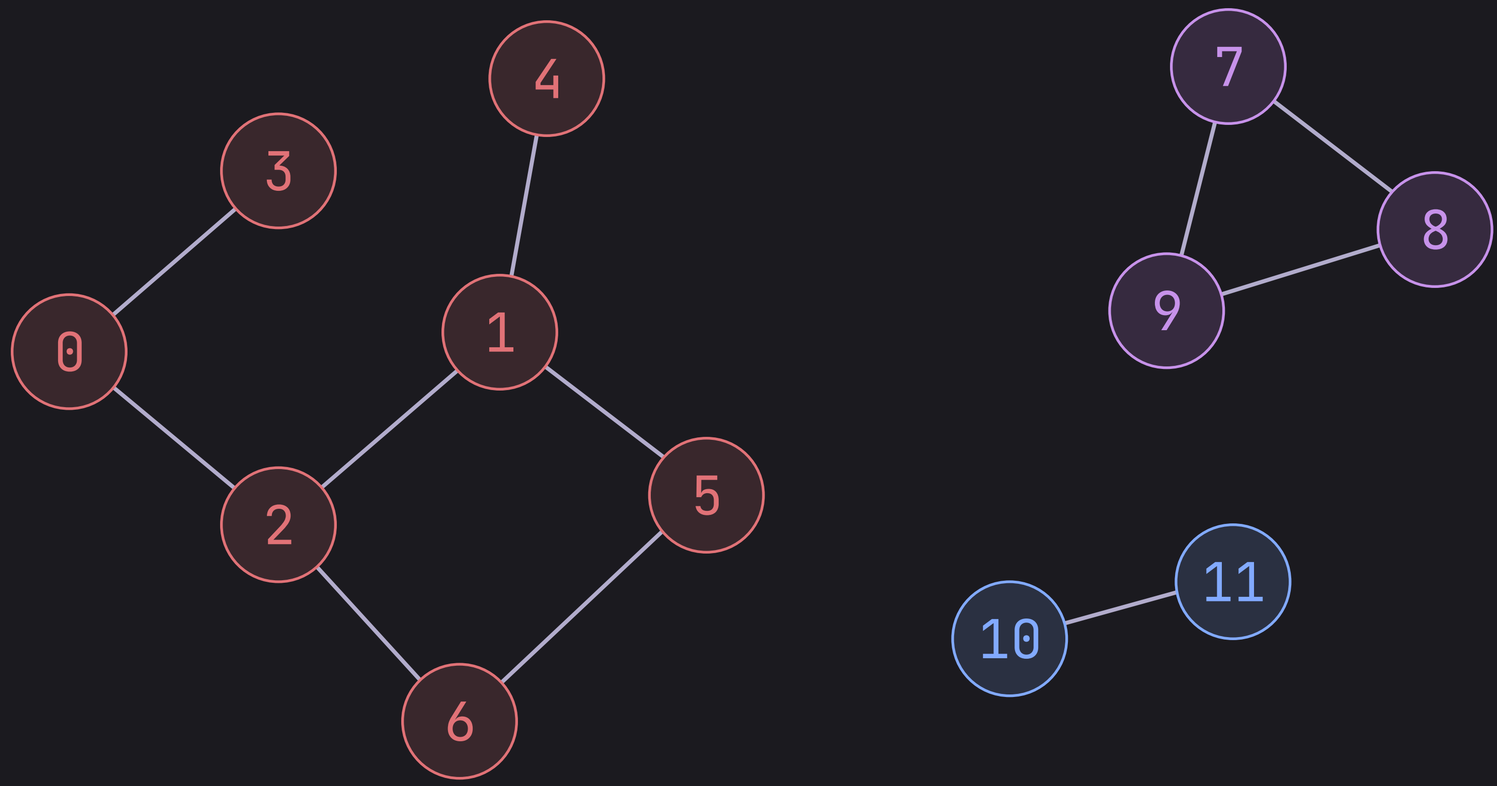



```cpp
void dfs(int v, std::vector<bool>& visited) {
    visited[v] = true;
    std::cout << v << " ";
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited);
      }
    }
  }
```

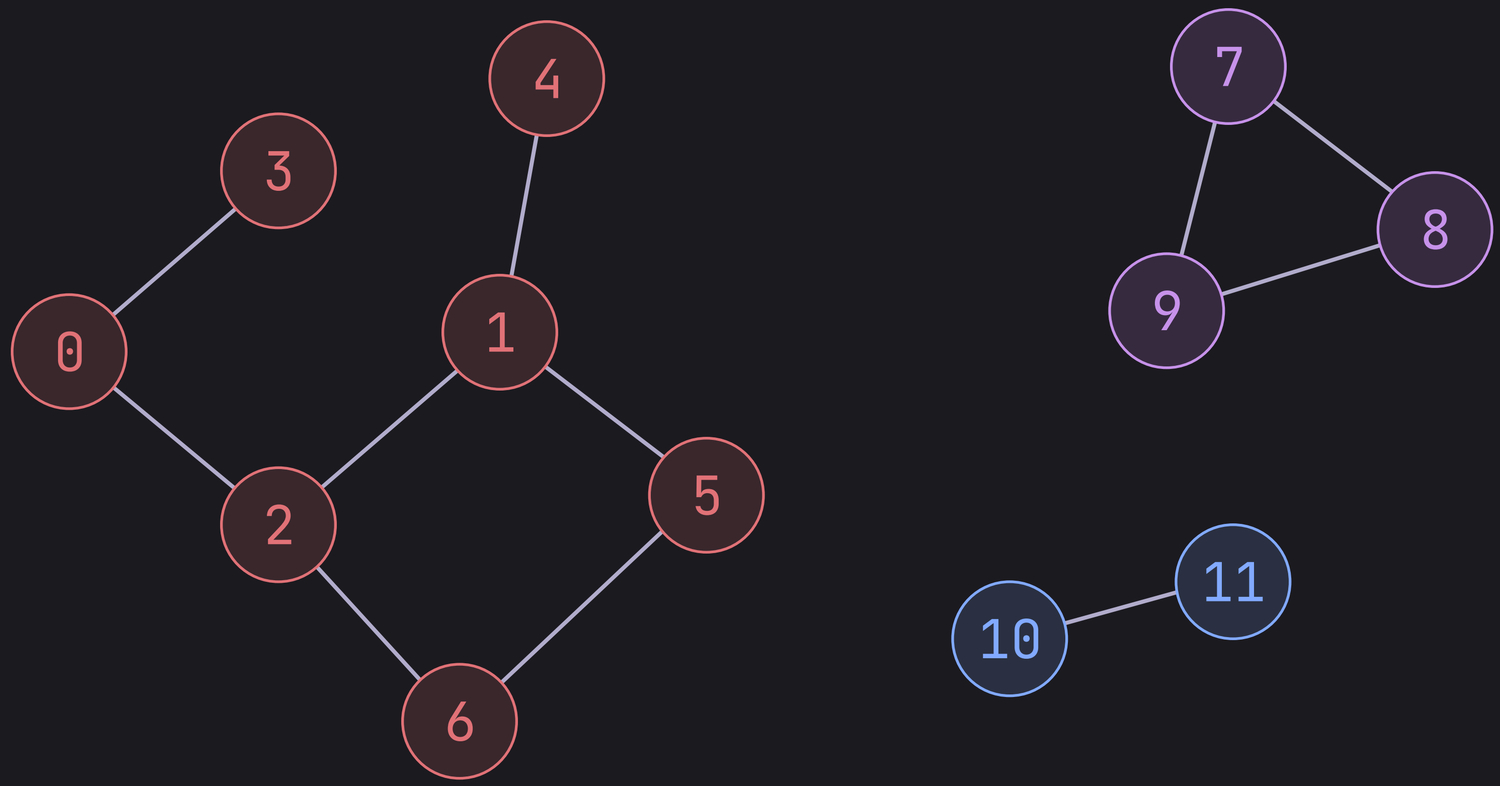


Call Stack

Connected Components



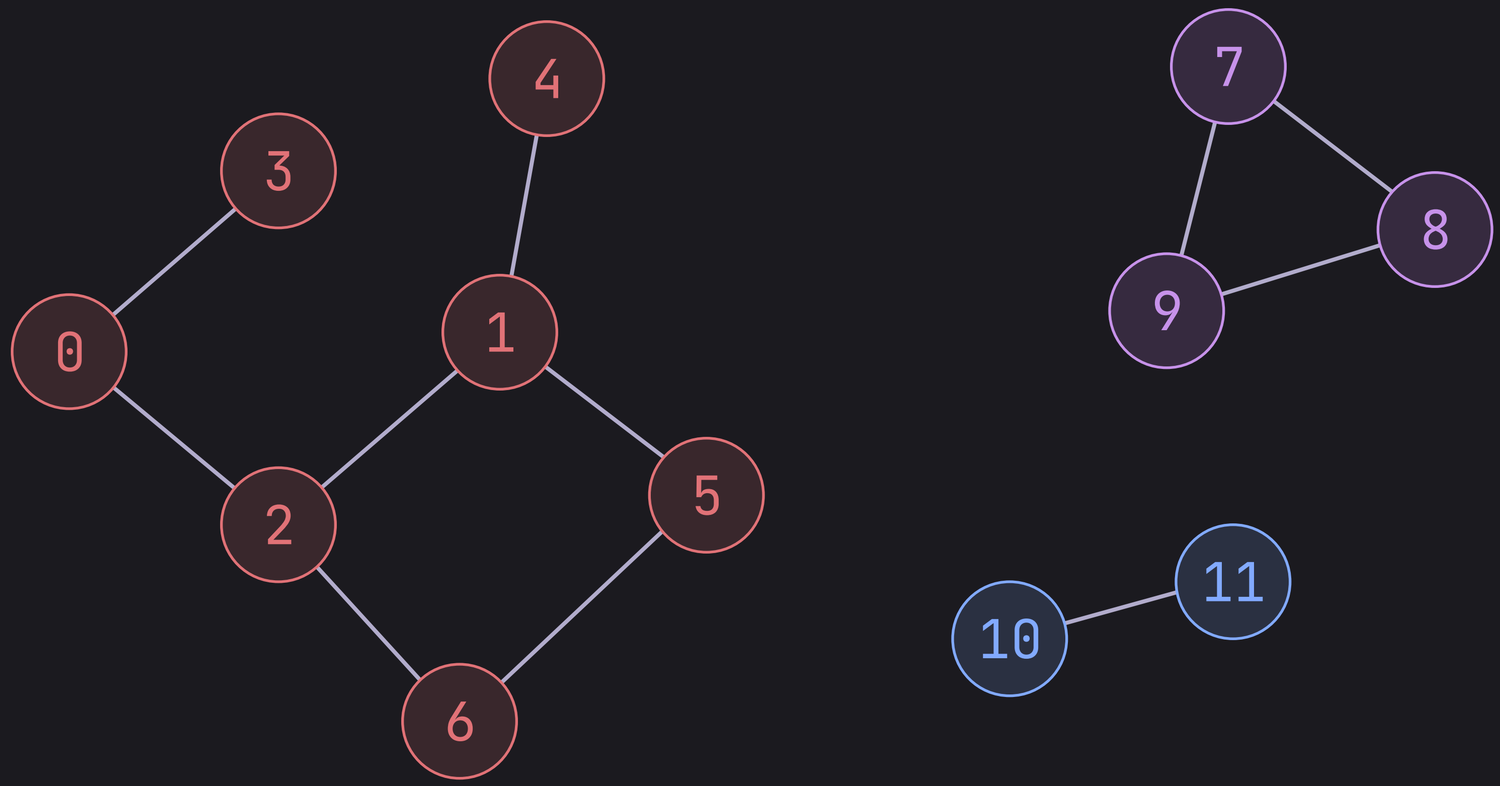
Connected Components



How do we efficiently find all the connected components?

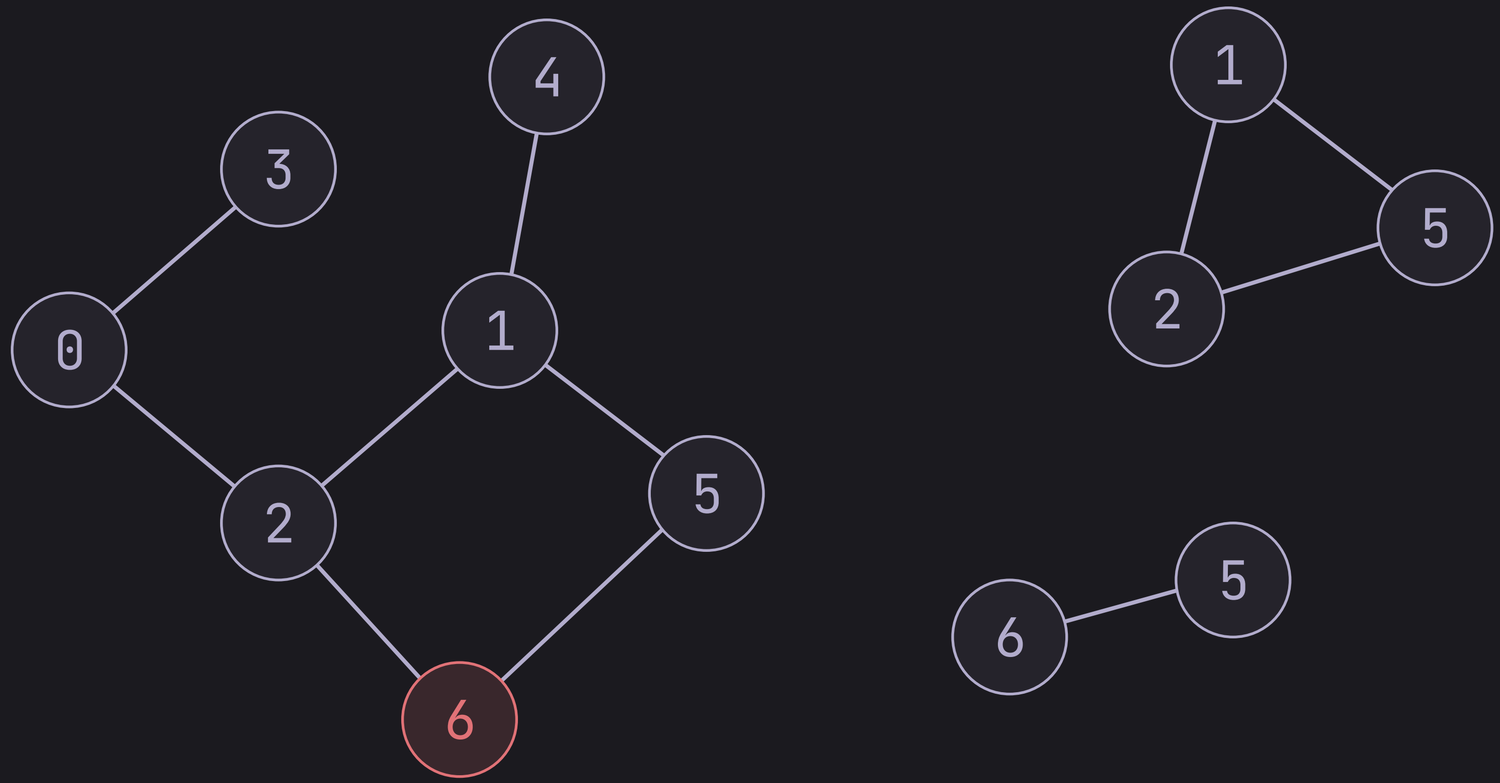
```
std::vector<std::set<int>> components
```

Connected Components



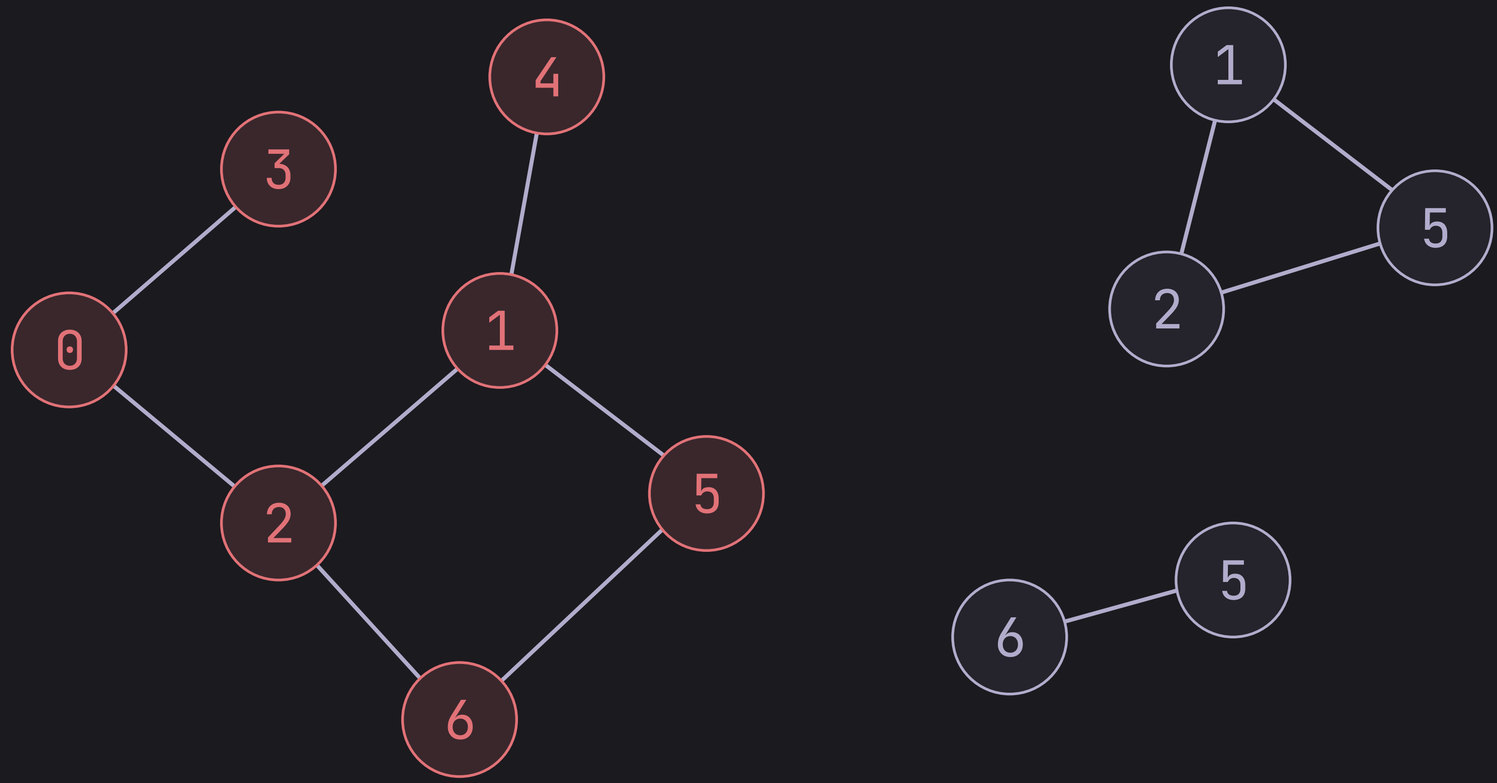
[{0, 1, 2, 3, 4, 5, 6}, {7, 8, 9}, {10, 11}]

Connected Components



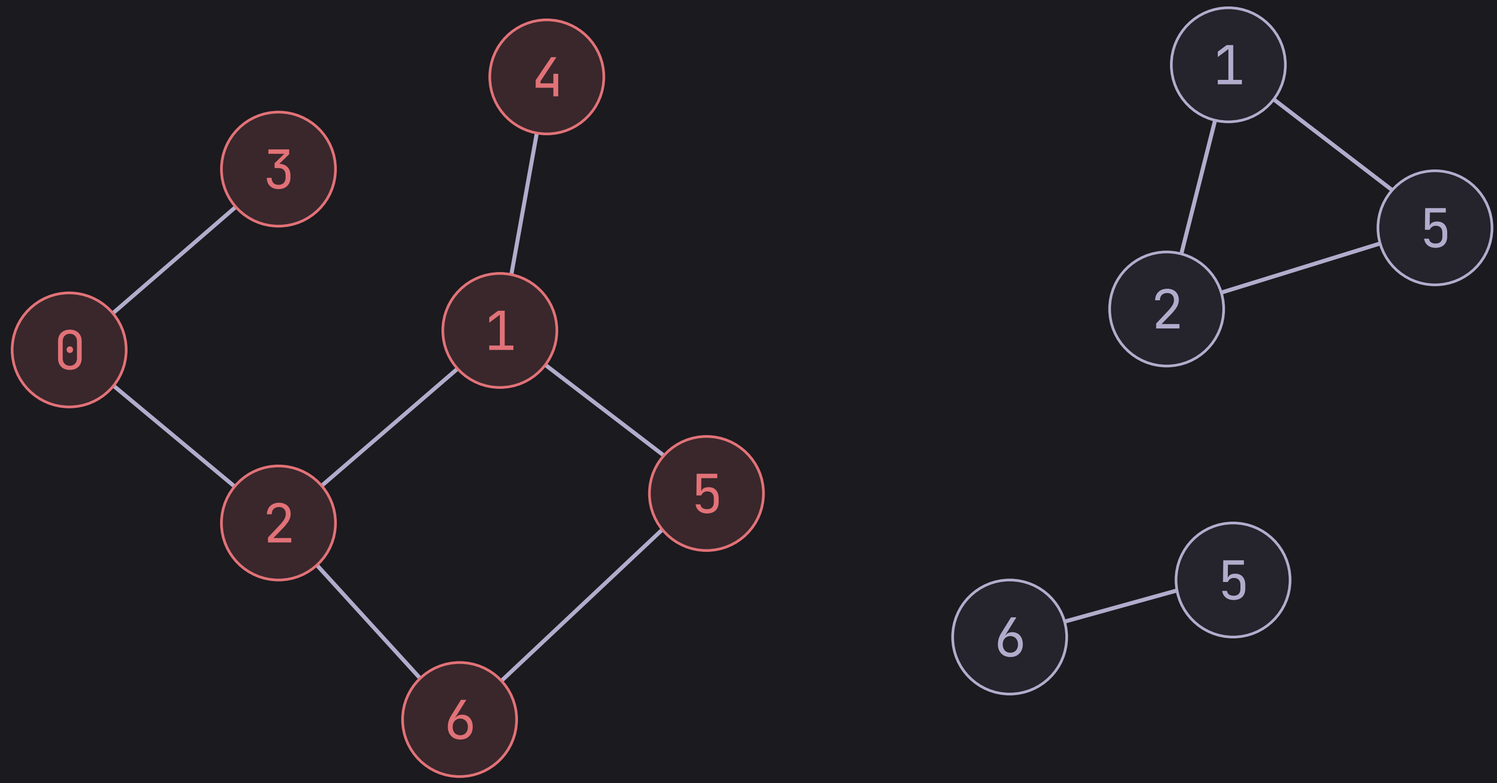
First we pick any vertex from our graph

Connected Components



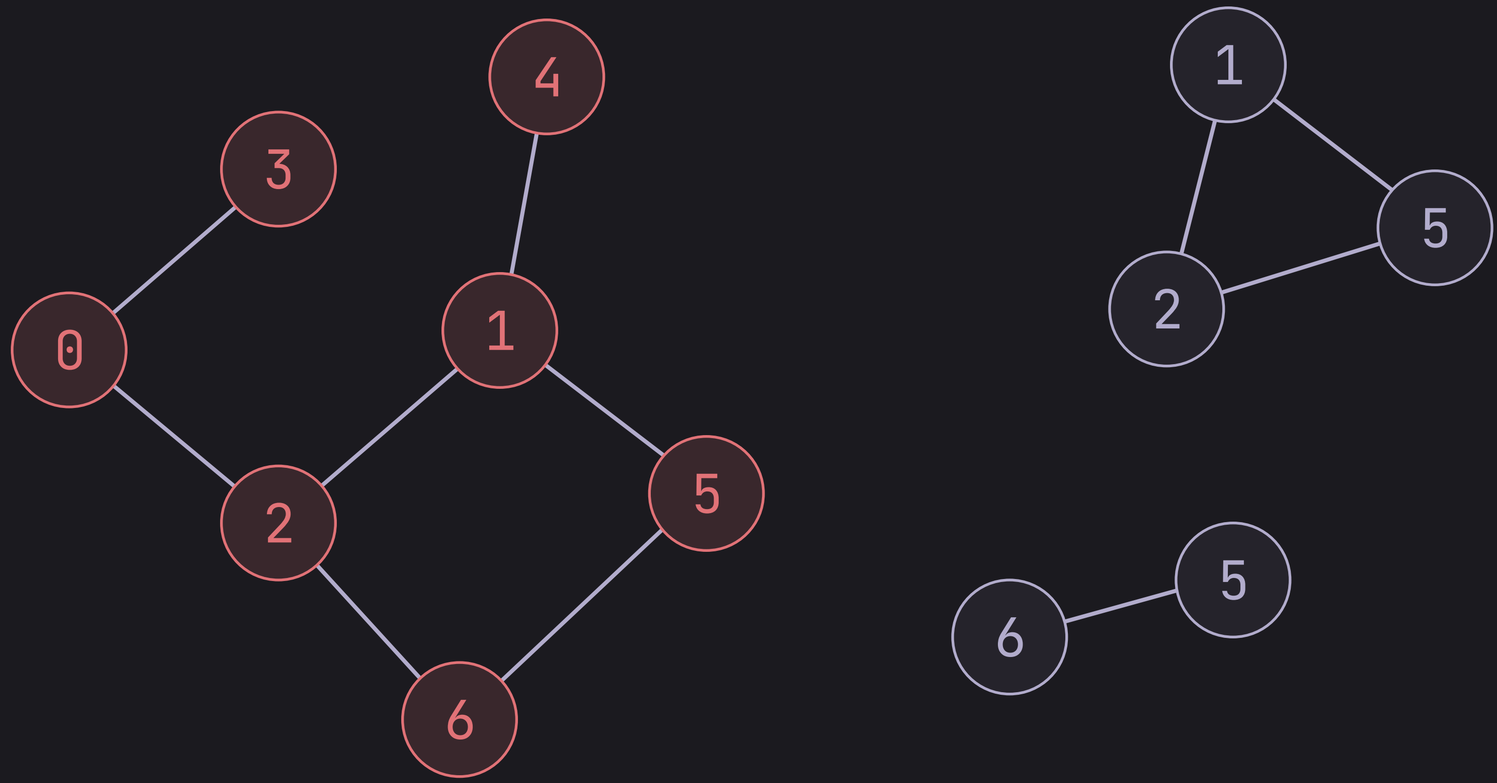
Then we perform a DFS, starting at that node

Connected Components



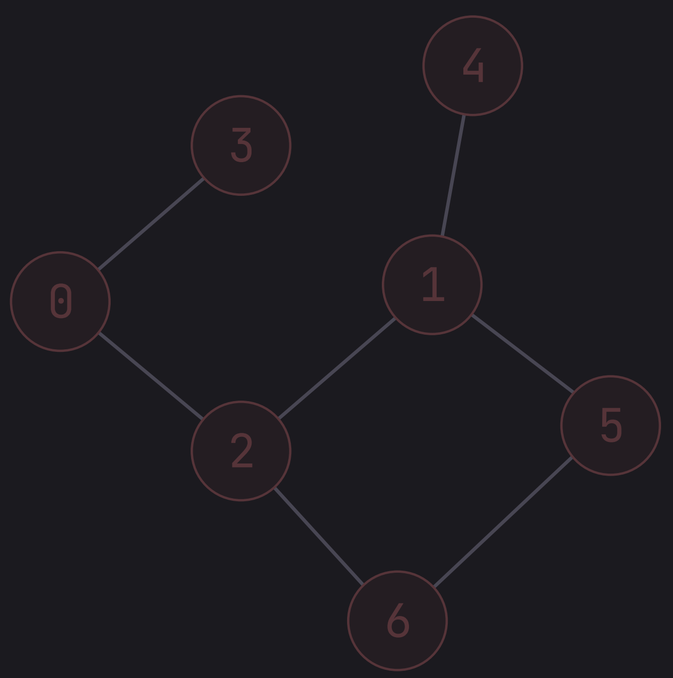
Instead of printing out the node, we add it to a set

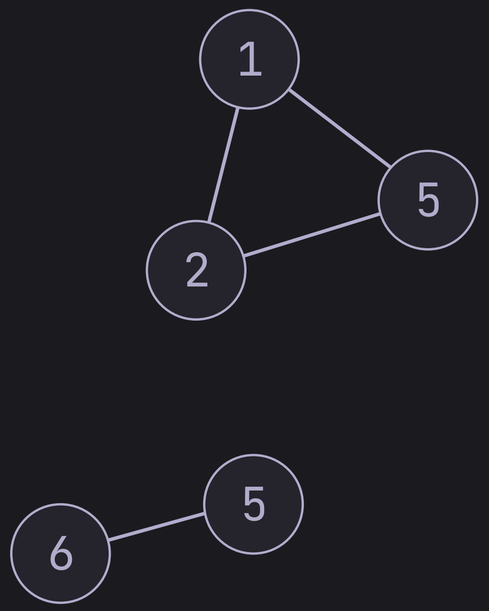
Connected Components



{0, 1, 2, 3, 4, 5, 6}

Connected Components

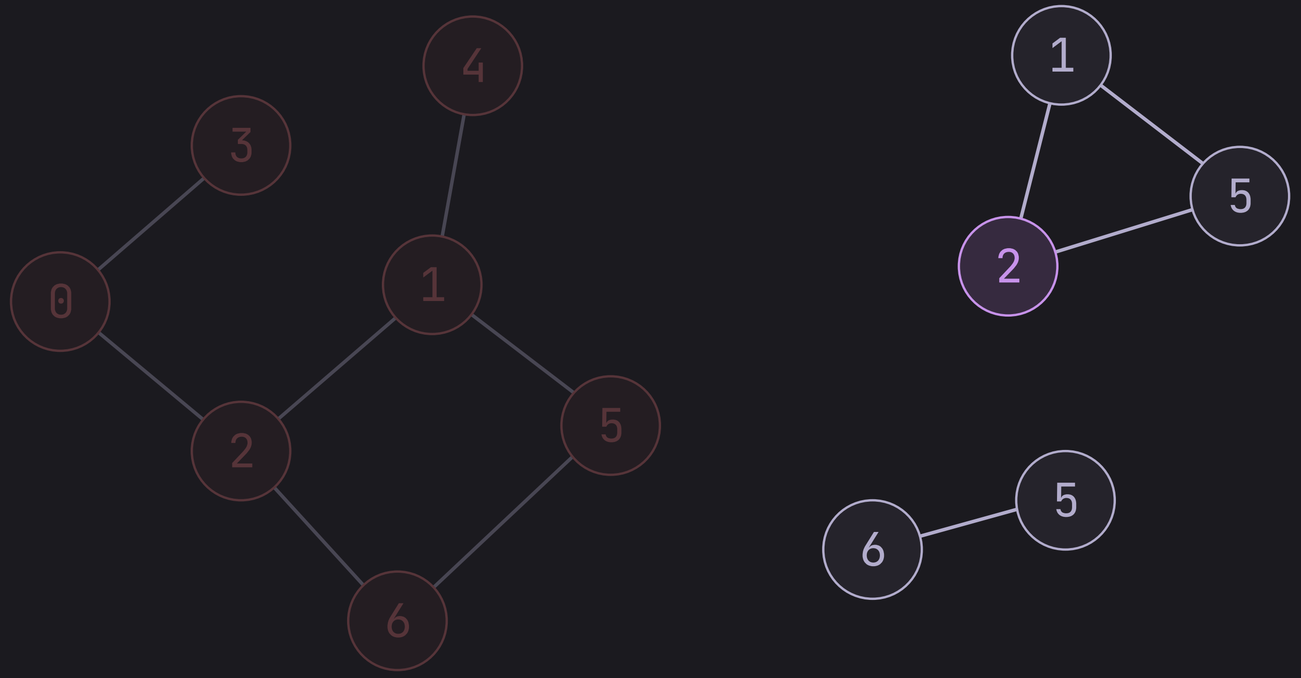




{0, 1, 2, 3, 4, 5, 6}

Now we pick another node that has not already been visited

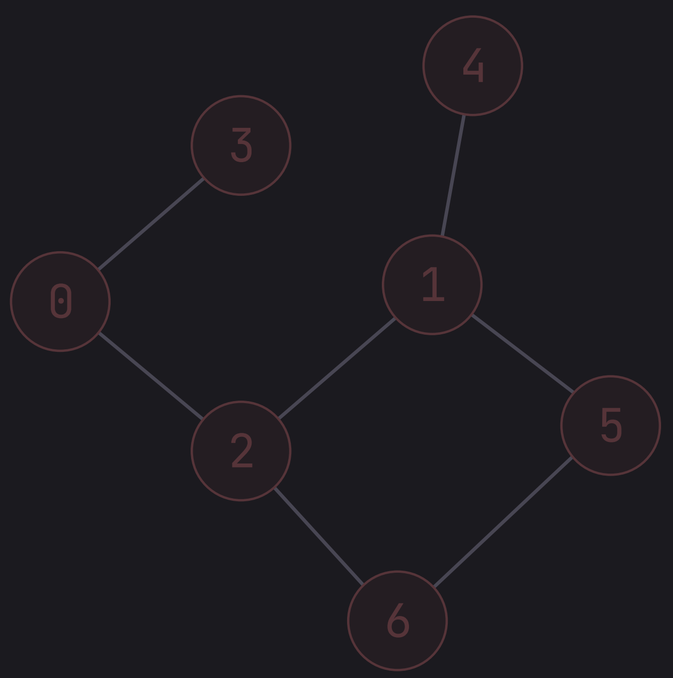
Connected Components

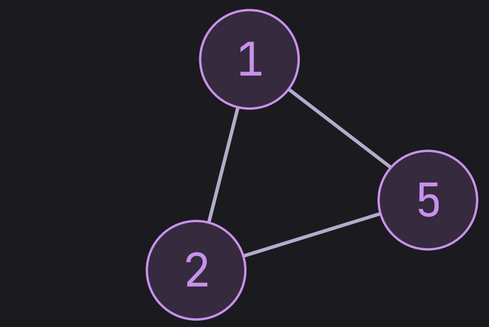


{0, 1, 2, 3, 4, 5, 6}

Then we perform a DFS, starting at that node

Connected Components

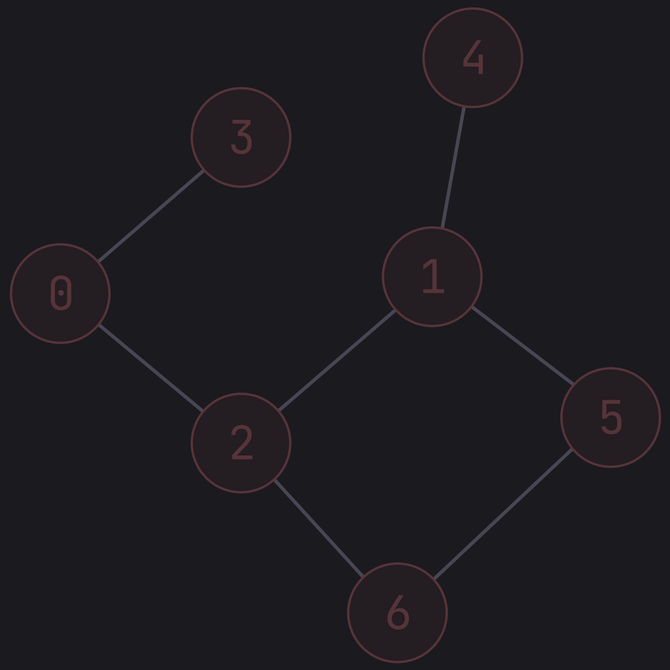


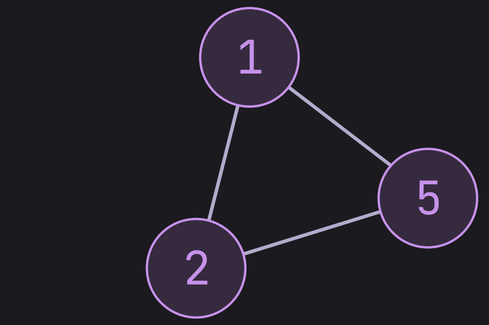


{0, 1, 2, 3, 4, 5, 6}

Instead of printing out the node, we add it to a set

Connected Components

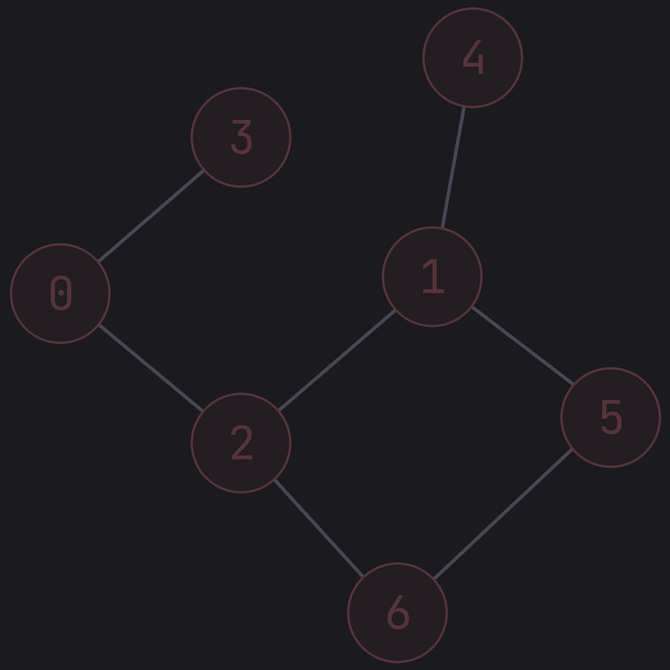


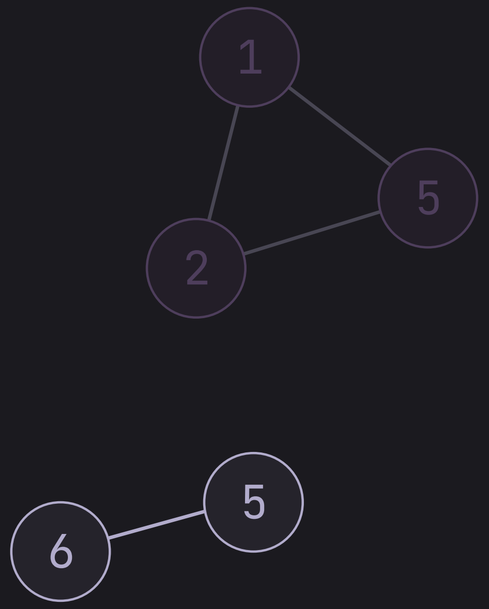


{0, 1, 2, 3, 4, 5, 6}

{7, 8, 9}

Connected Components



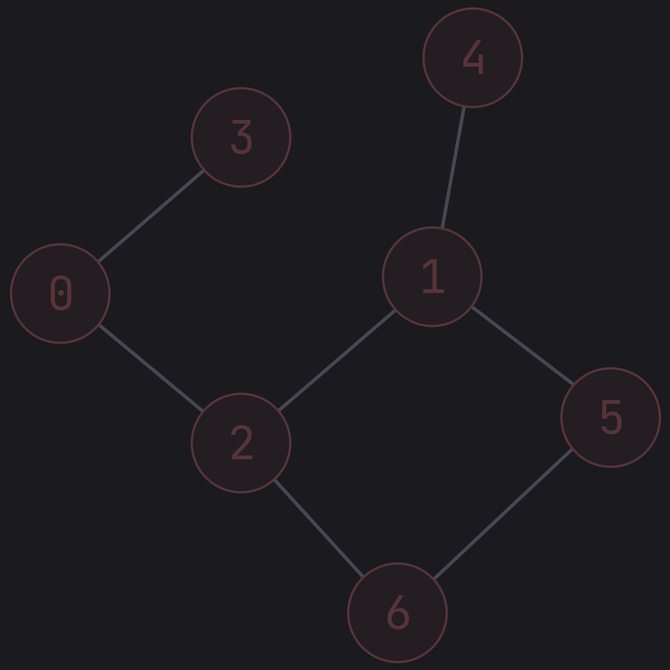


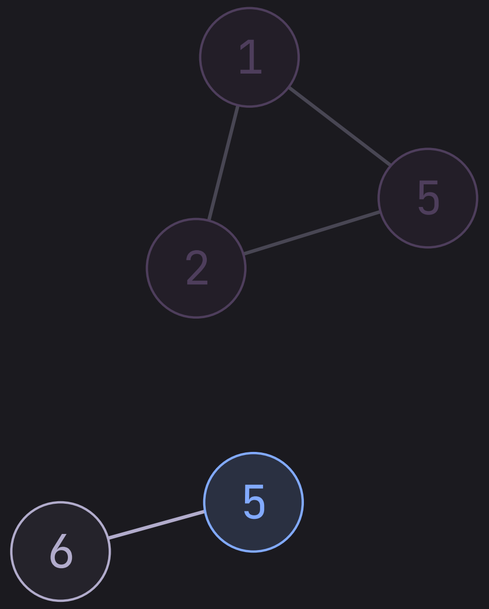
{0, 1, 2, 3, 4, 5, 6}

{7, 8, 9}

Now we pick another node that has not already been visited

Connected Components



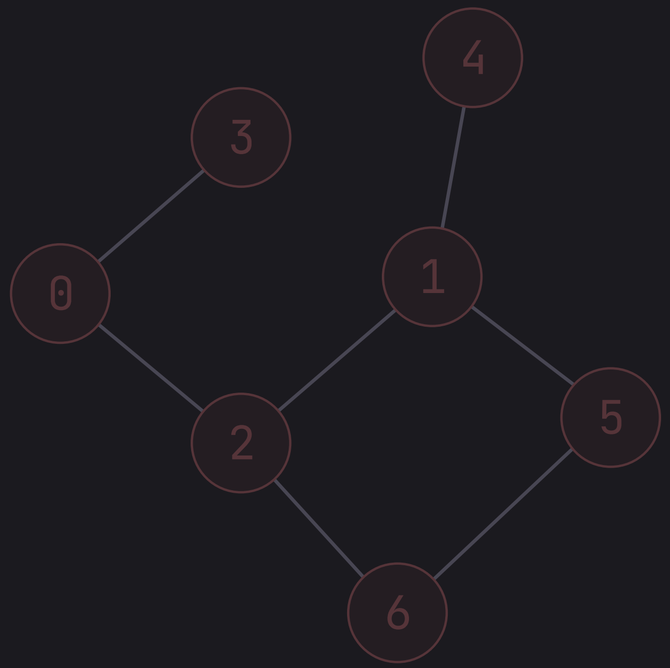


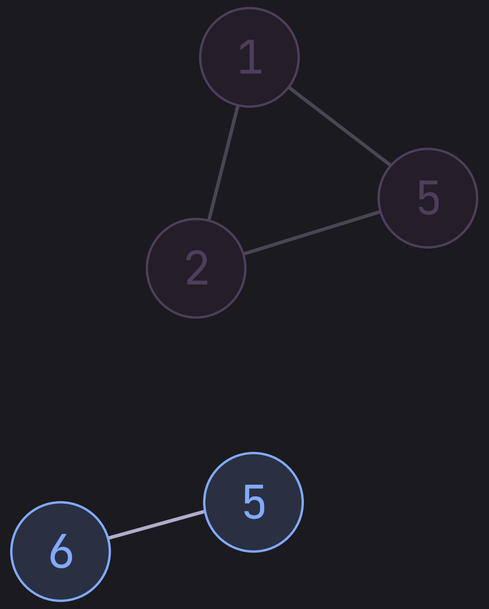
{0, 1, 2, 3, 4, 5, 6}

{7, 8, 9}

Then we perform a DFS, starting at that node

Connected Components



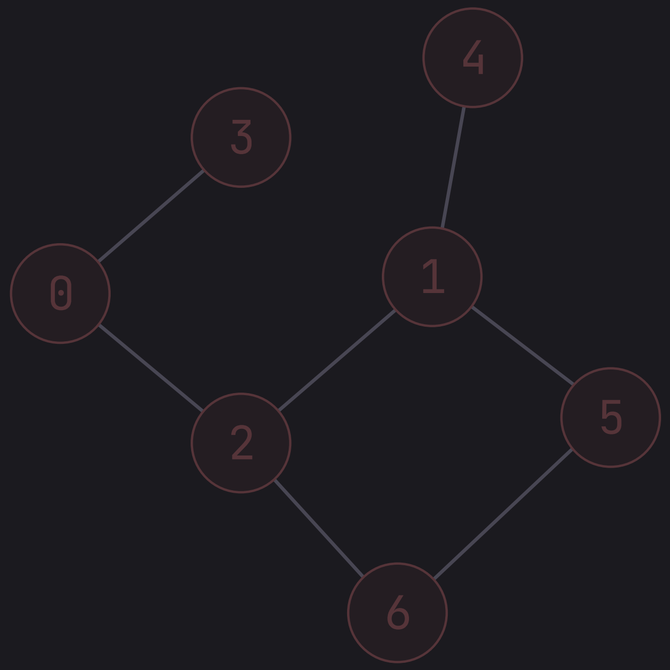


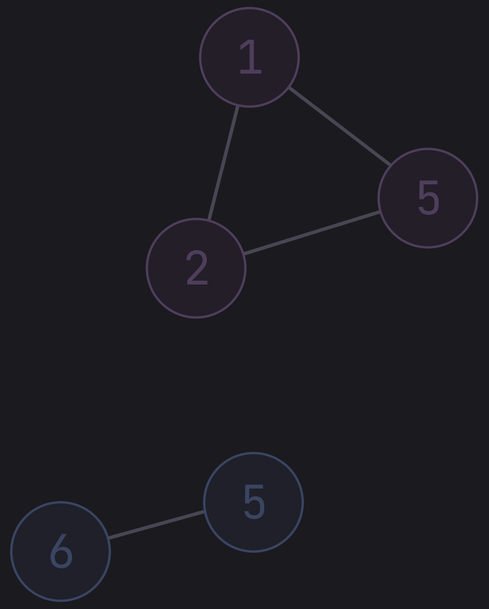
{0, 1, 2, 3, 4, 5, 6}

{7, 8, 9}

{10, 11}

Connected Components





{0, 1, 2, 3, 4, 5, 6}

{7, 8, 9}

{10, 11}

There are no more unvisited nodes, so we are done!

Connected Components

Lets give that a go on Ed!

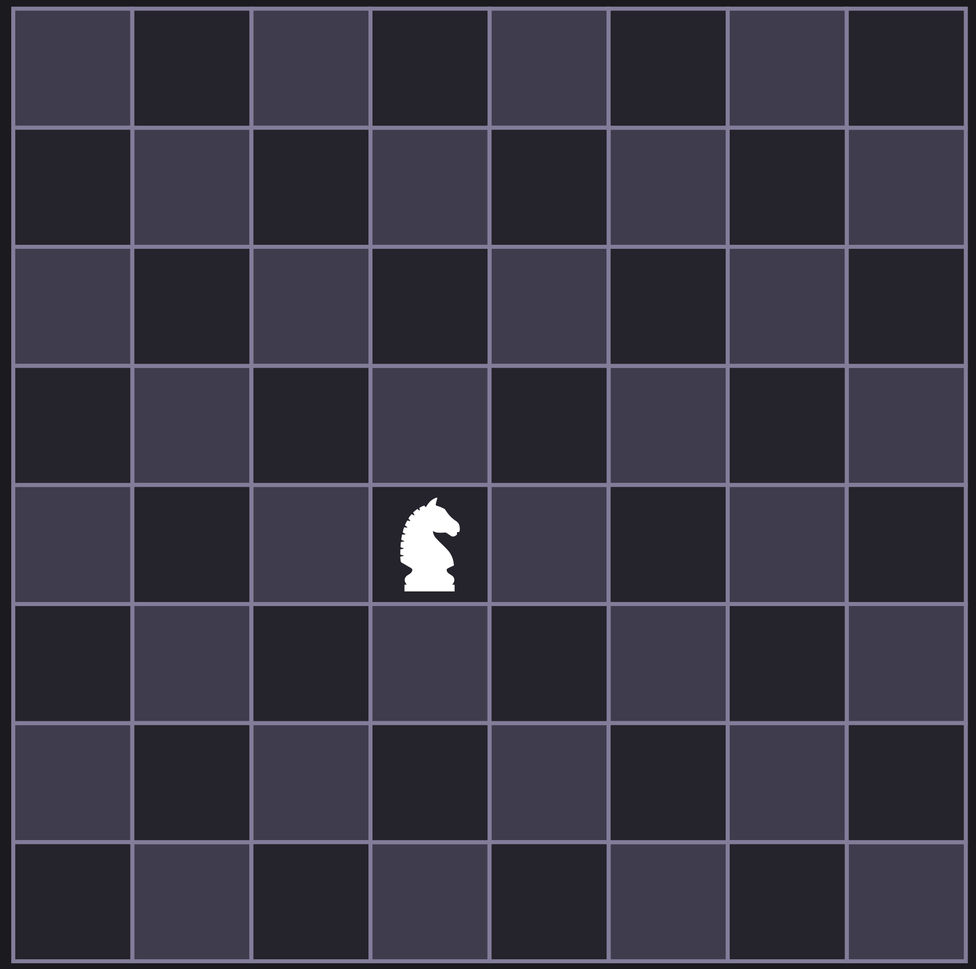
Connected Components

```
void connectedComponents() {
  std::vector<bool> visited(numVertices);
  std::vector<std::set<int> > components {};
  for (int v = 0; v < numVertices; ++v) {
    if (!visited[v]) {
        std::set<int> component {}
        dfs(v, visited, component);
        components.push_back(component);
    }
  }
```

Connected Components

```
void dfs(int v, std::vector<bool>& visited, std::set<int>& component) {
    marked[v] = true;
    component.insert(v);
    for (auto u : vec[v]) {
      if (!visited[u]) {
        dfs(u, visited, component);
      }
    }
  }
```

Knight Moves

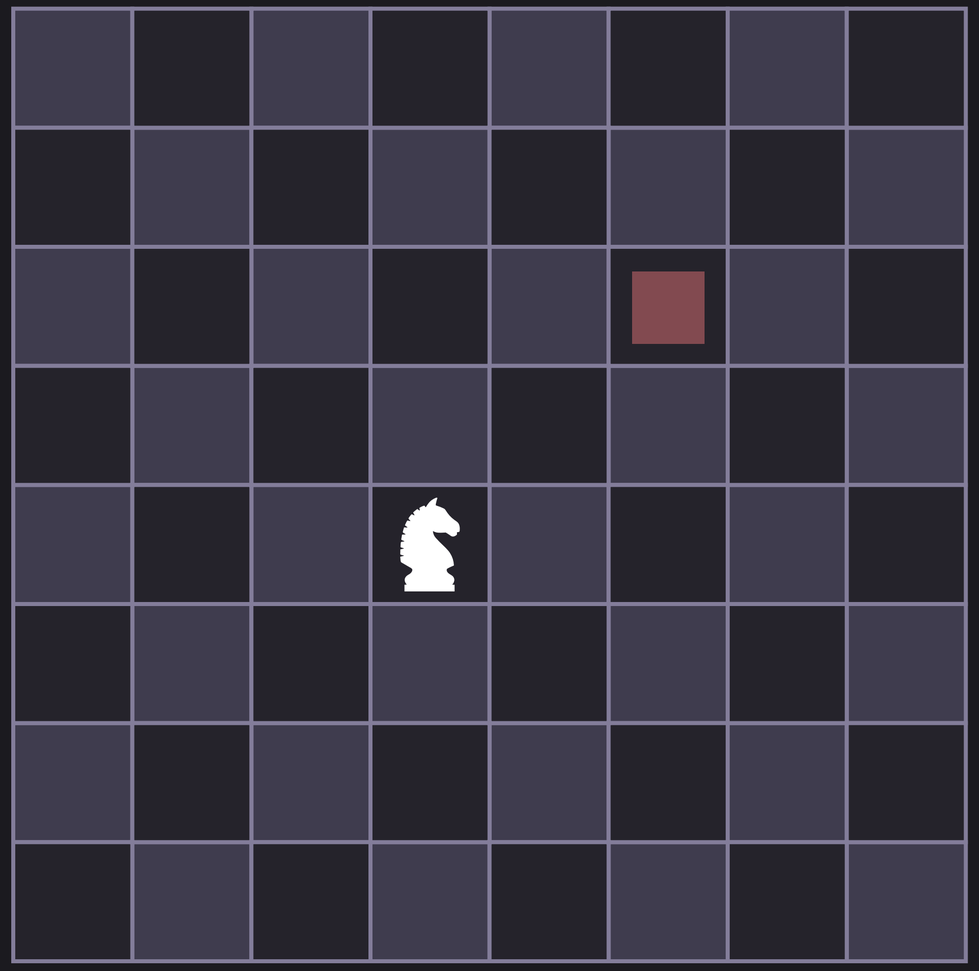


Imagine an infinite chessboard and a knight sitting on square (0, 0).

We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

Knight Moves

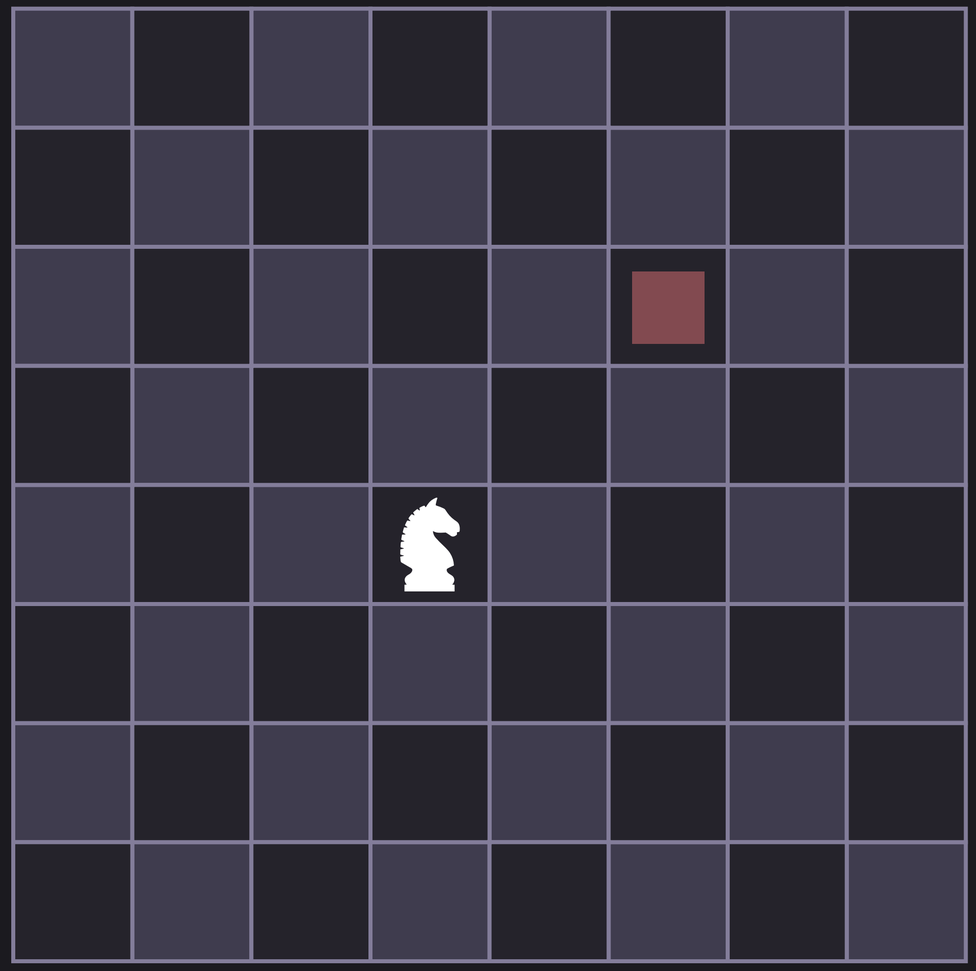


Imagine an infinite chessboard and a knight sitting on square (0, 0).

We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

Knight Moves

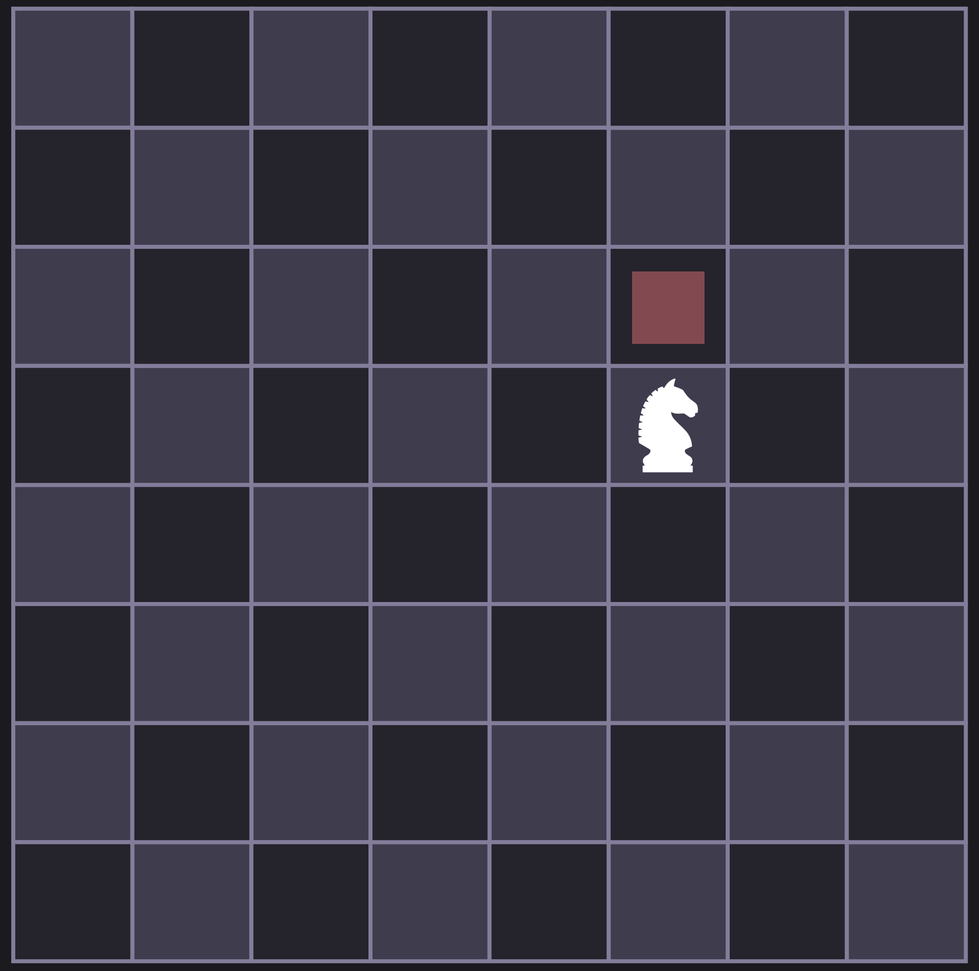


Imagine an infinite chessboard and a knight sitting on square (0, 0).

We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

Knight Moves

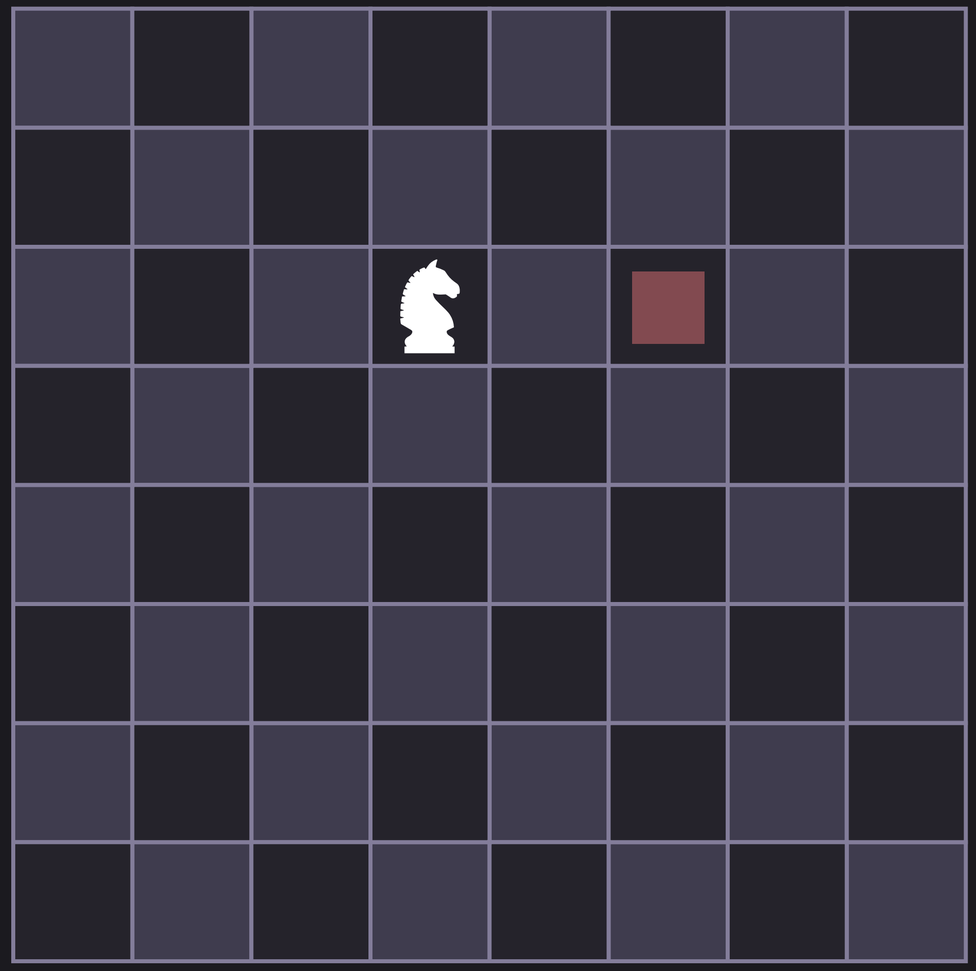


Imagine an infinite chessboard and a knight sitting on square (0, 0).

We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

Knight Moves

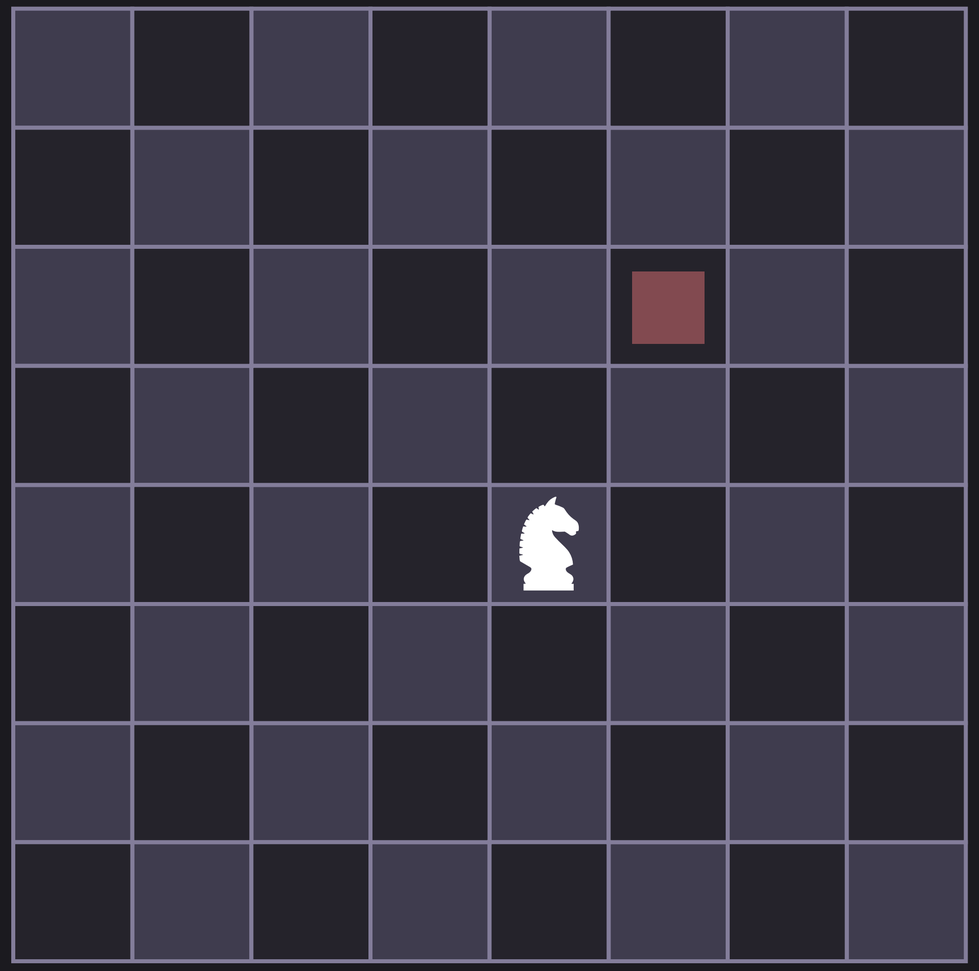


Imagine an infinite chessboard and a knight sitting on square (0, 0).

We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

Knight Moves

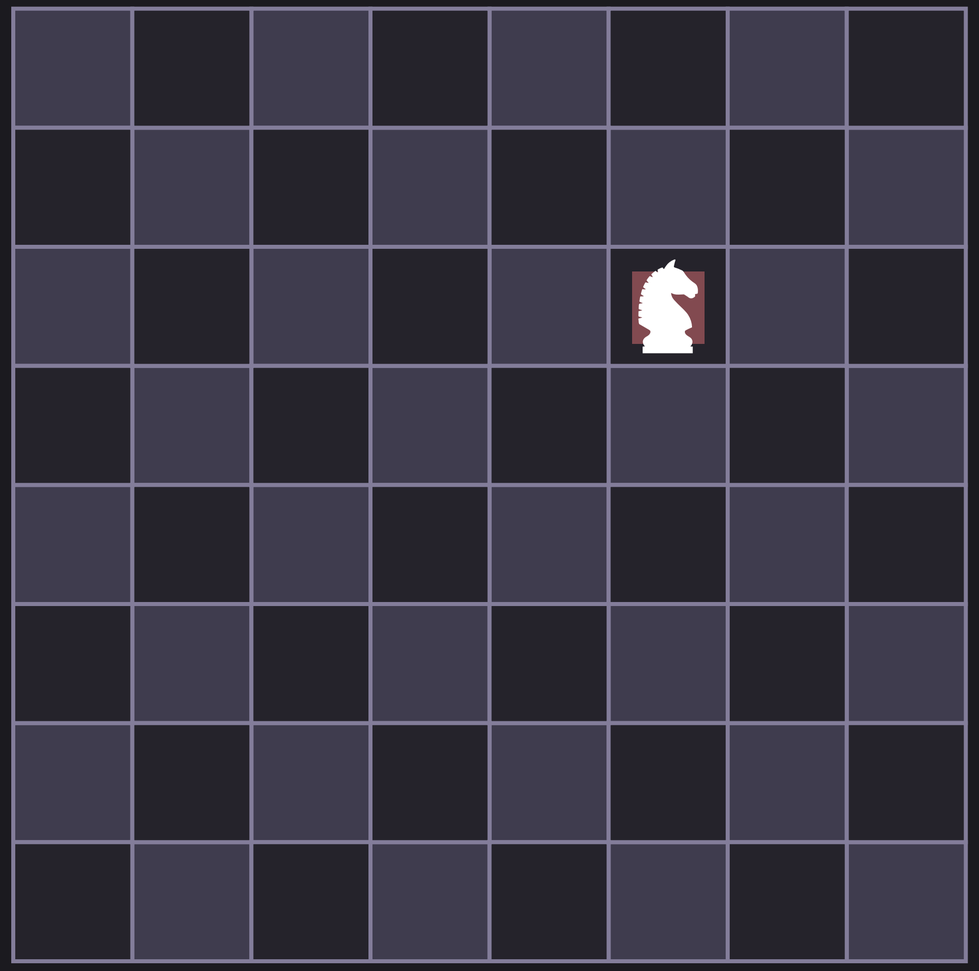


Imagine an infinite chessboard and a knight sitting on square (0, 0).

We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

Knight Moves

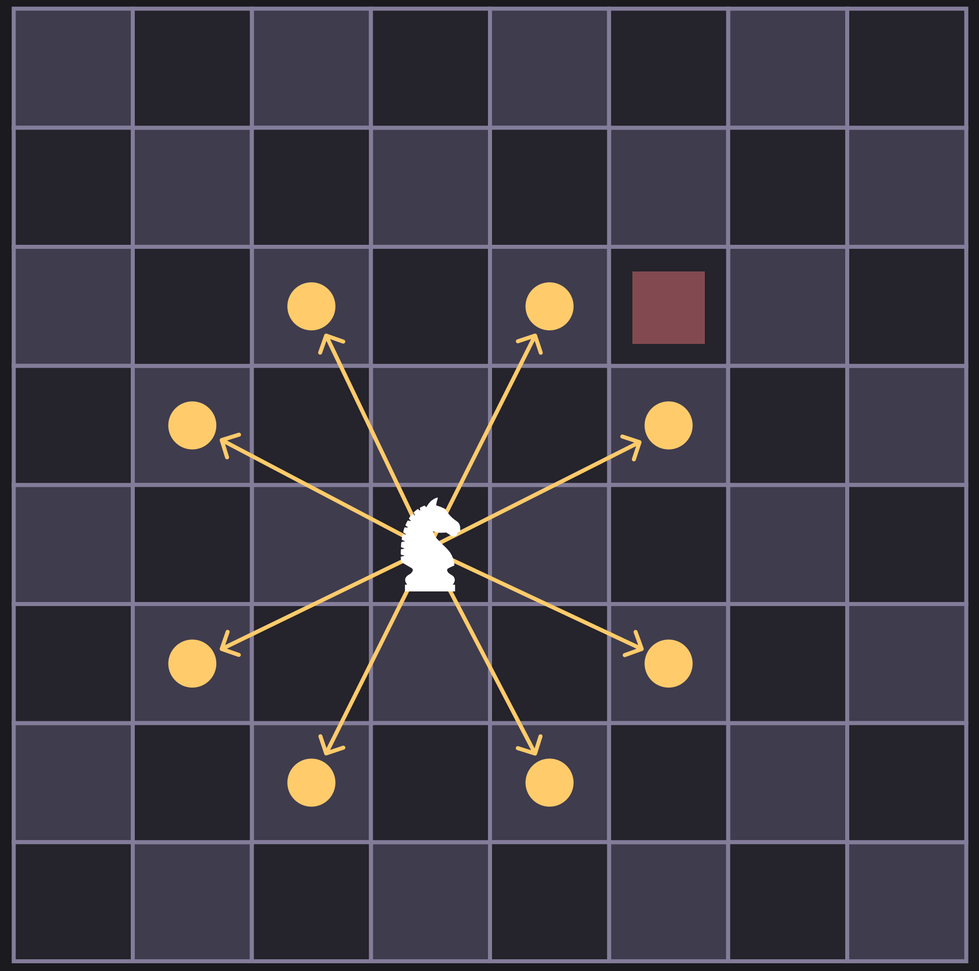


Imagine an infinite chessboard and a knight sitting on square (0, 0).

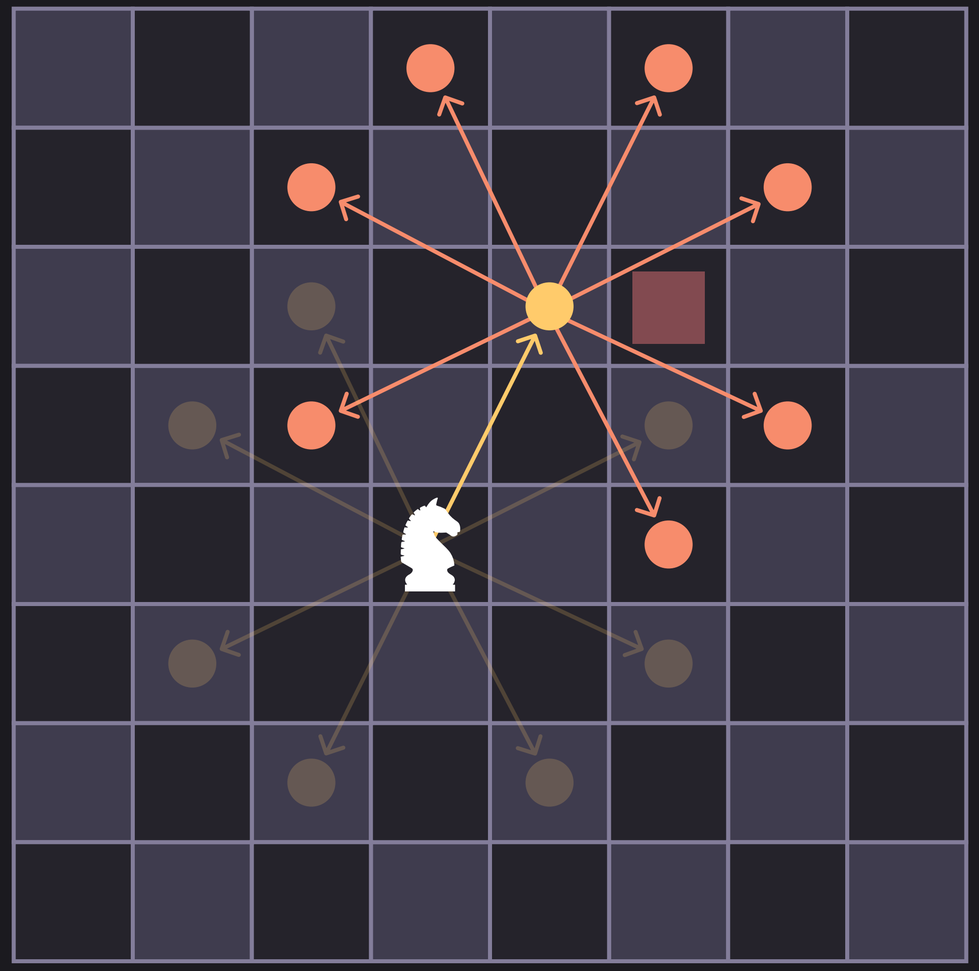
We are given a target square (x, y) on the chessboard.

The question is: what is the minimum number of moves the knight needs to take to travel from (0, 0) to (x, y), when the knight can move in the usual way it does in chess?

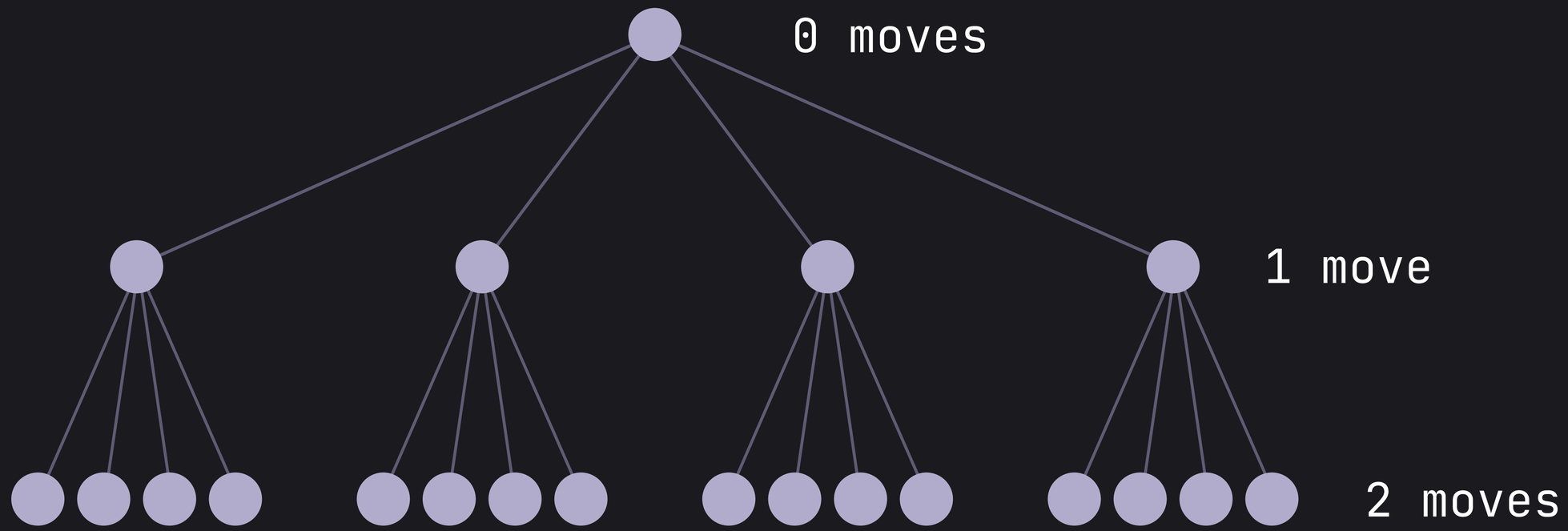
Knight Moves



Knight Moves

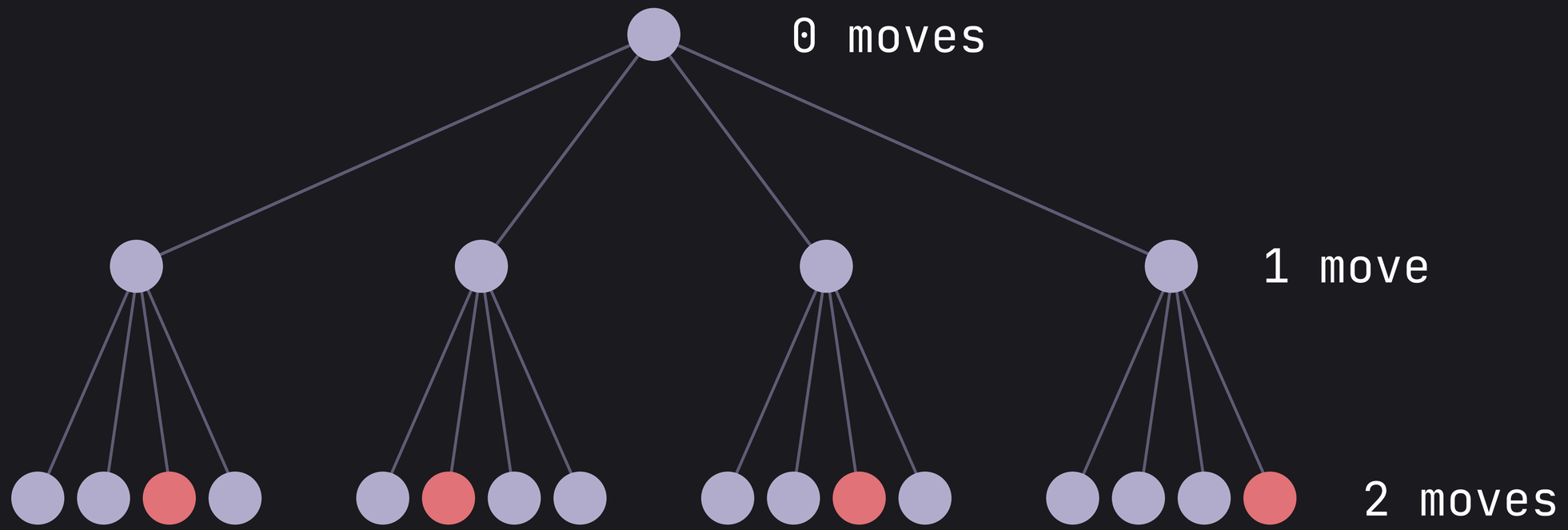


Knight Moves



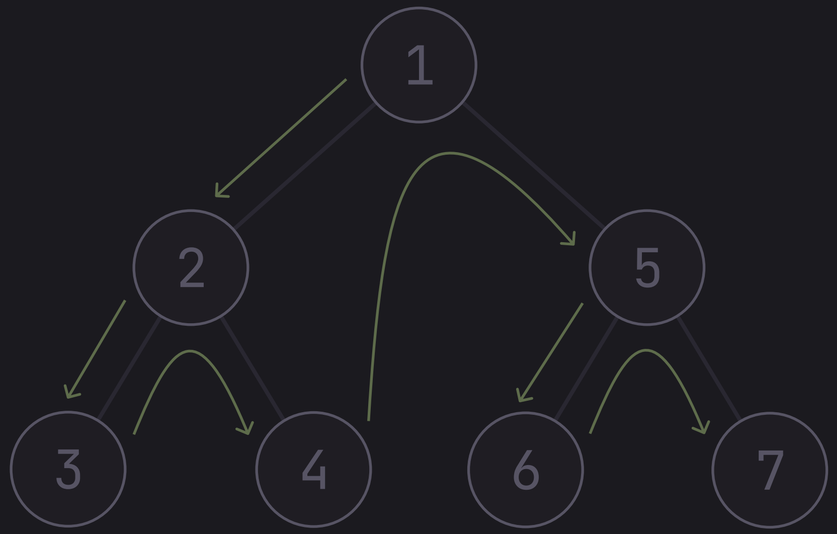
To find the shortest path we want to first explore all the vertices that are one move away, and then 2 moves away, and so forth

Knight Moves



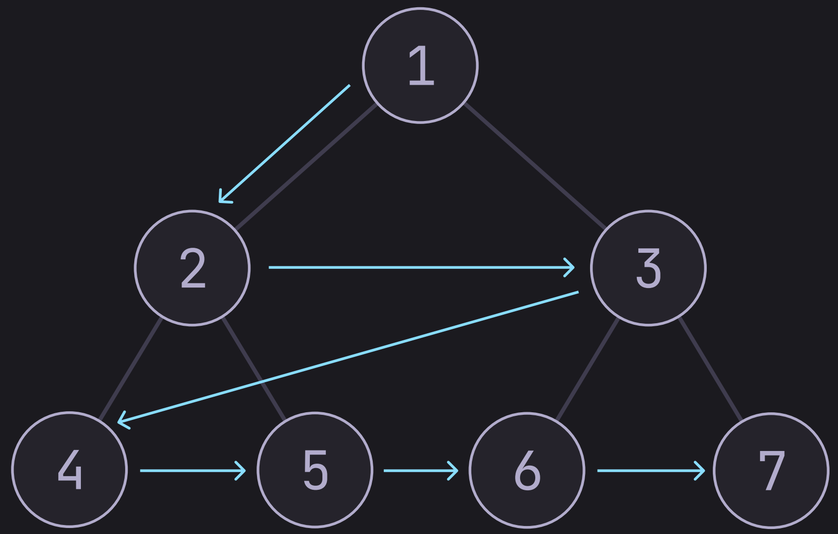
We also don’t want to go in circles so we can ignore vertices we have already seen

Graph Search Algorithms



DFS

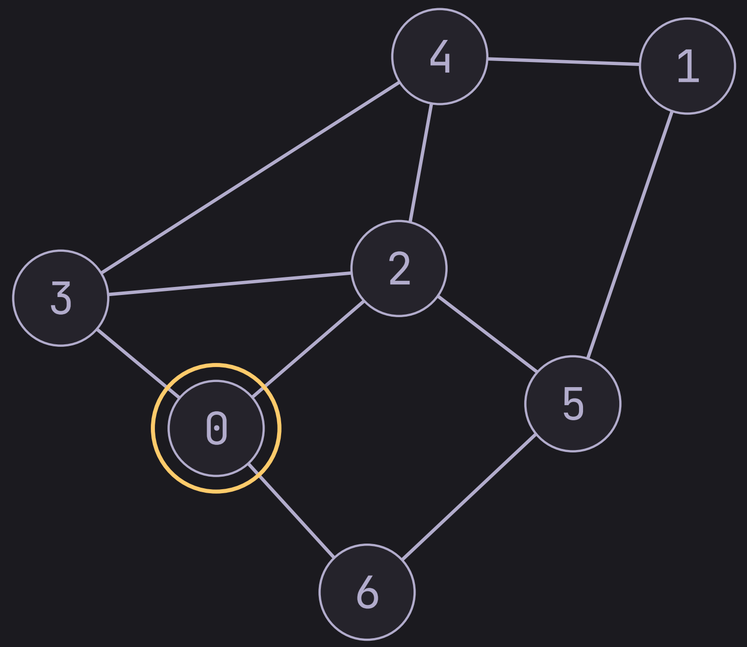
Drill down all the way down



BFS

Explore Layer by layer

Breadth First Search



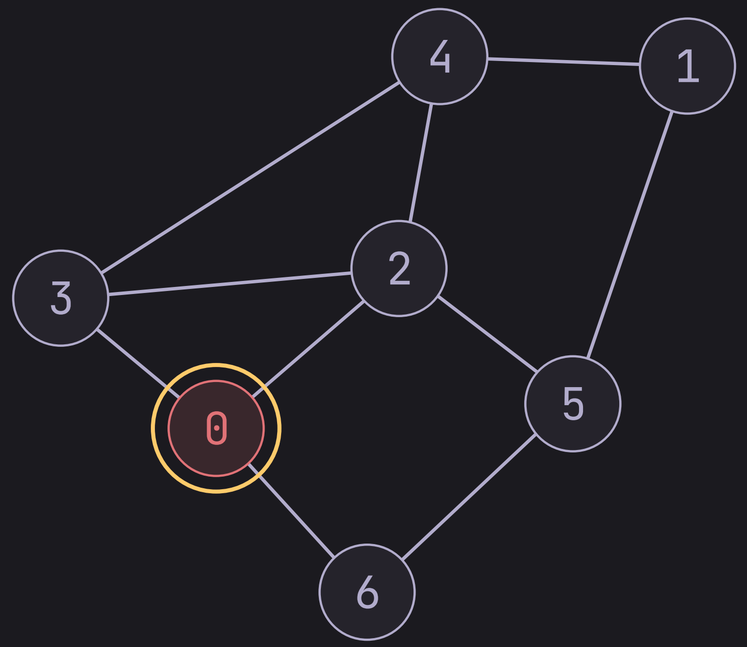
bfsQueue

```
⟶ void bfs(int start) {
    std::set<int> visited {}
    std::queue<int> bfsQueue;

    visited.insert(start);
    bfsQueue.push(start);

    while (!bfsQueue.empty()) {
      int v = bfsQueue.front();
      bfsQueue.pop();
      for (int u : adjacencyList[v]) {
        if (!visited.contains(u)) {
          visited.insert(u);
          bfsQueue.push(u);
        }
      }
    }
  }
```

Breadth First Search



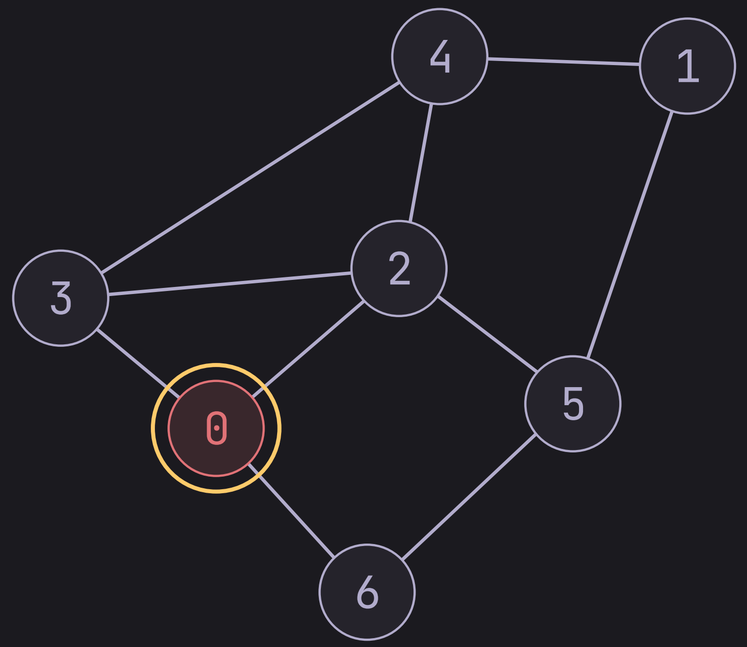
bfsQueue

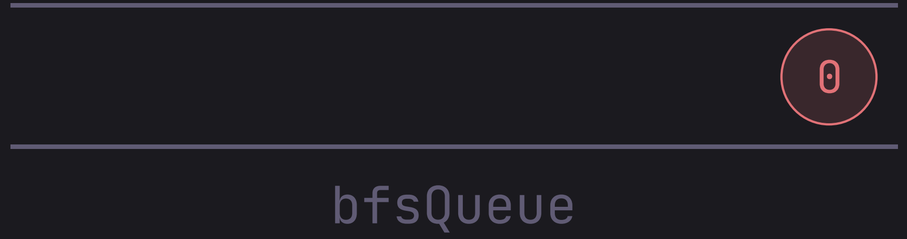
```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

→ visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



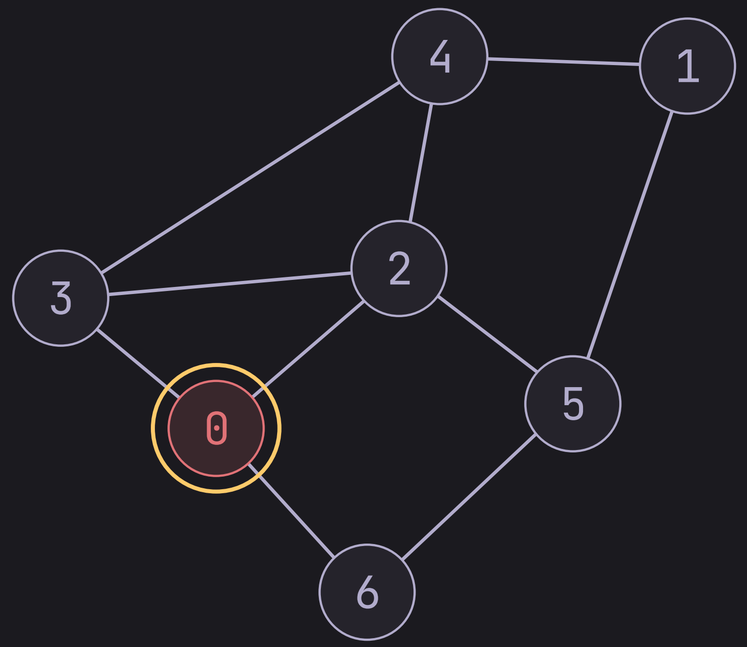



```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



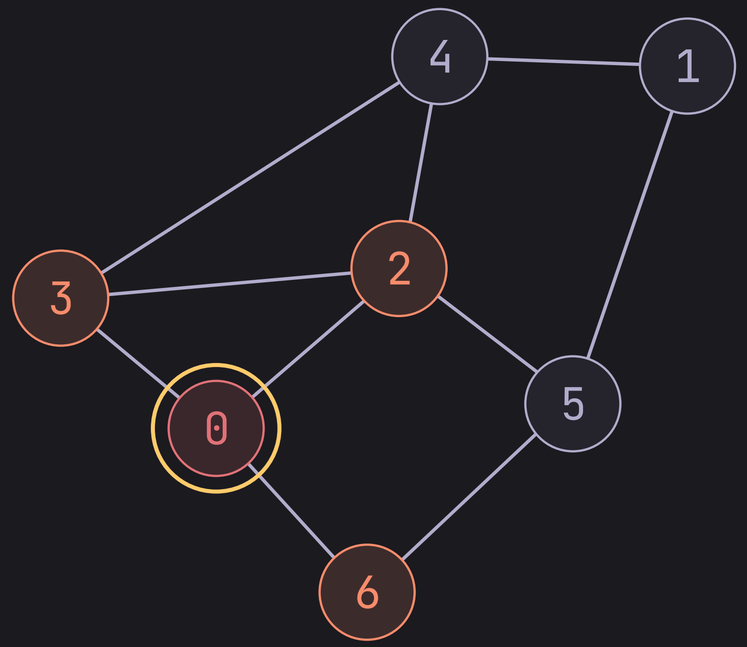
bfsQueue

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
⟶  bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



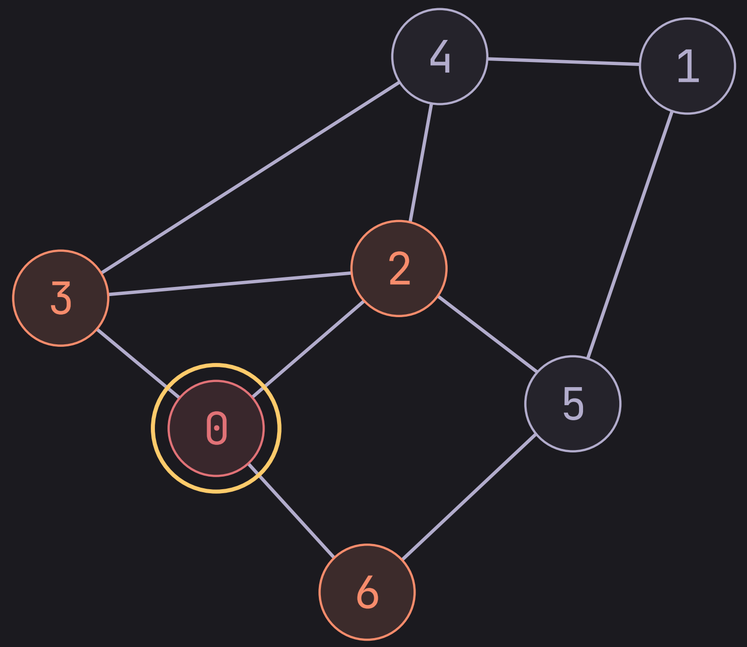
bfsQueue

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
    ⟶  visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search

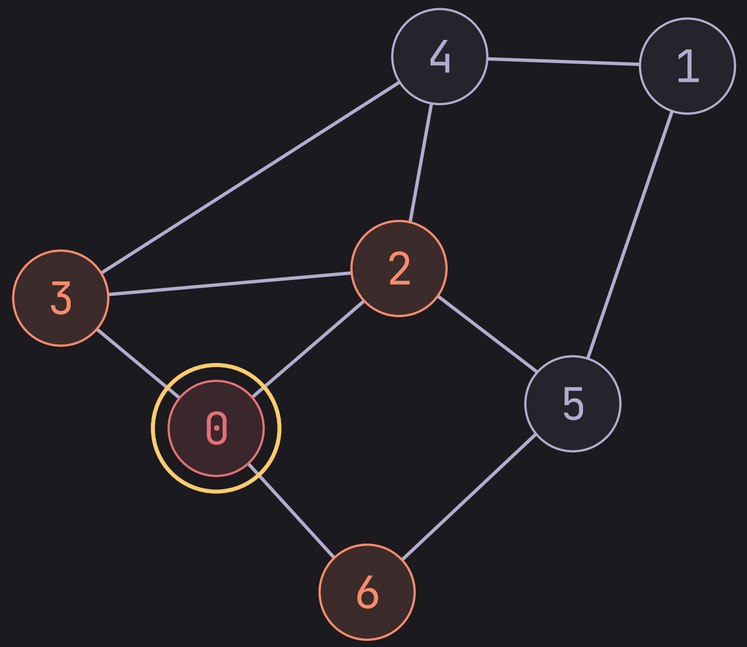



```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
      ⟶ bfsQueue.push(u);
        }
    }
  }
}
```

Breadth First Search

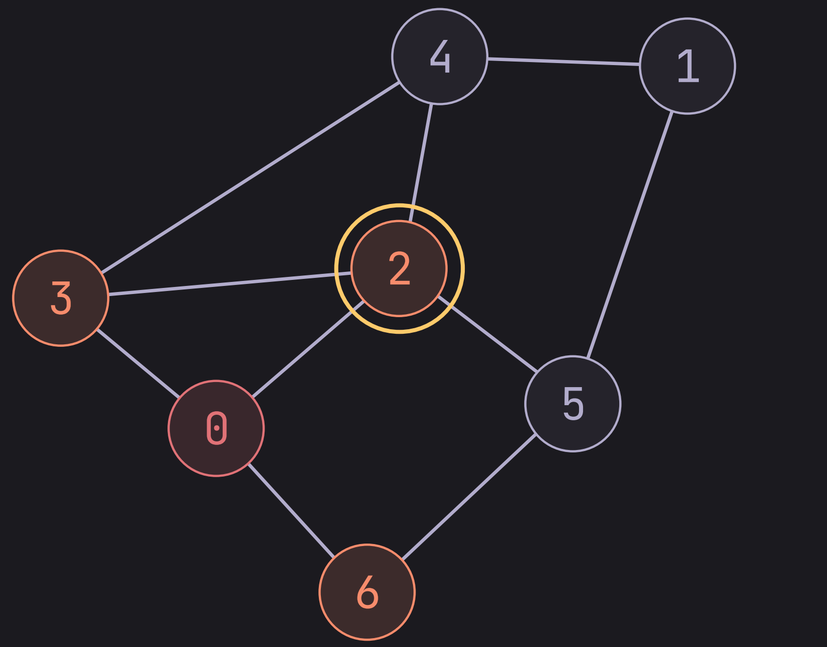


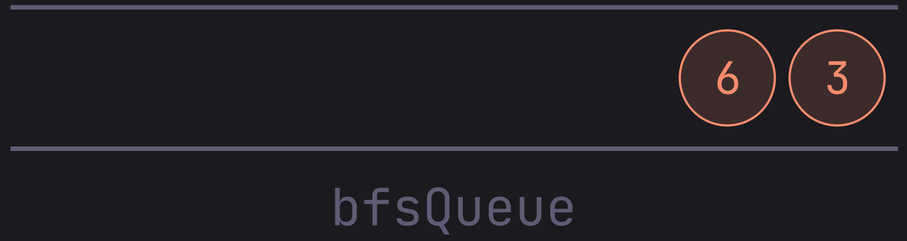

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



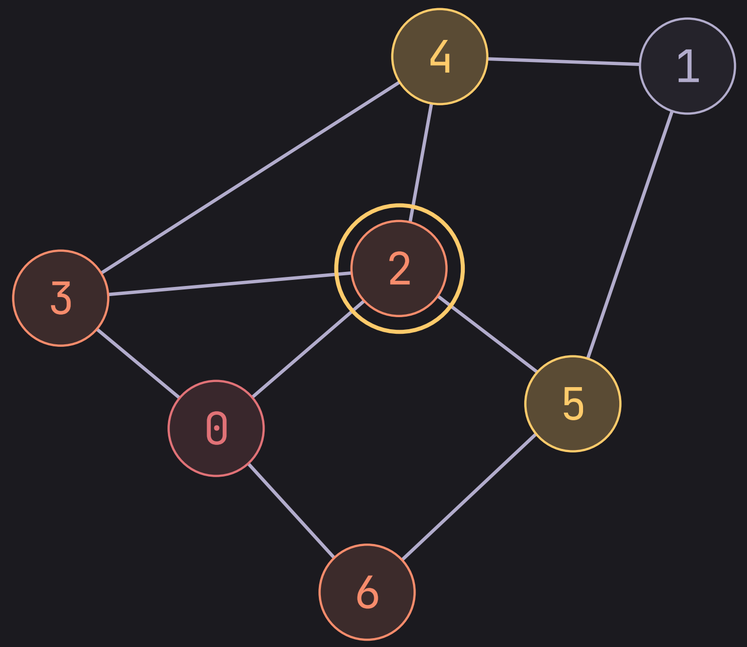


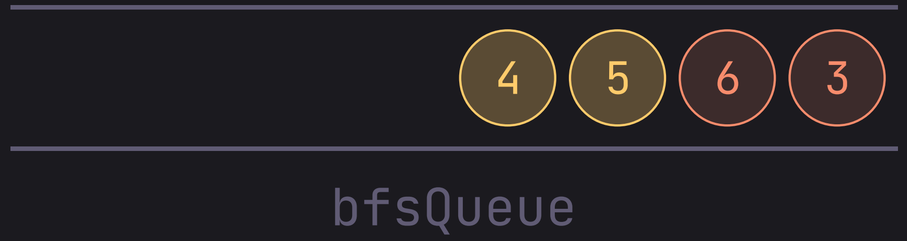

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
⟶  bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



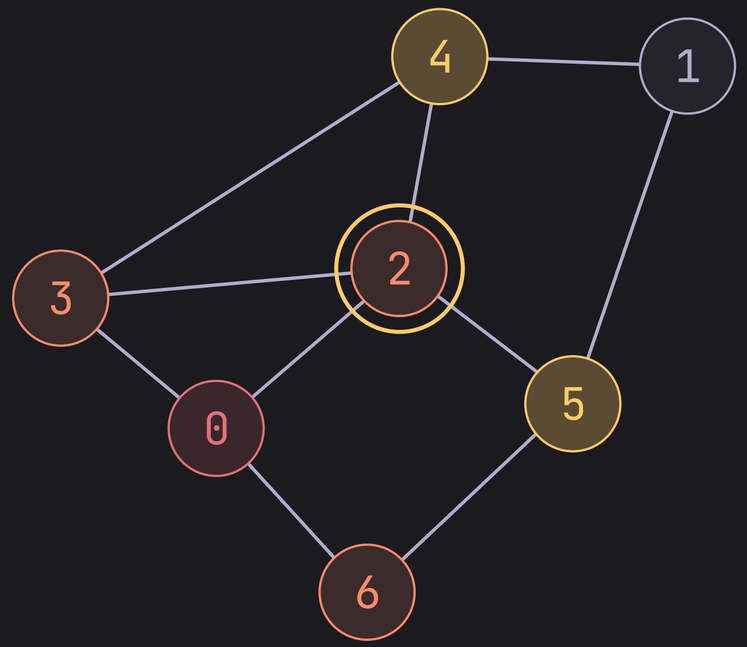


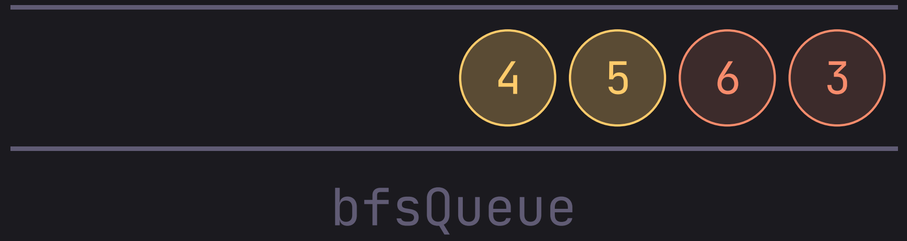

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
      ⟶ bfsQueue.push(u);
        }
    }
  }
}
```

Breadth First Search



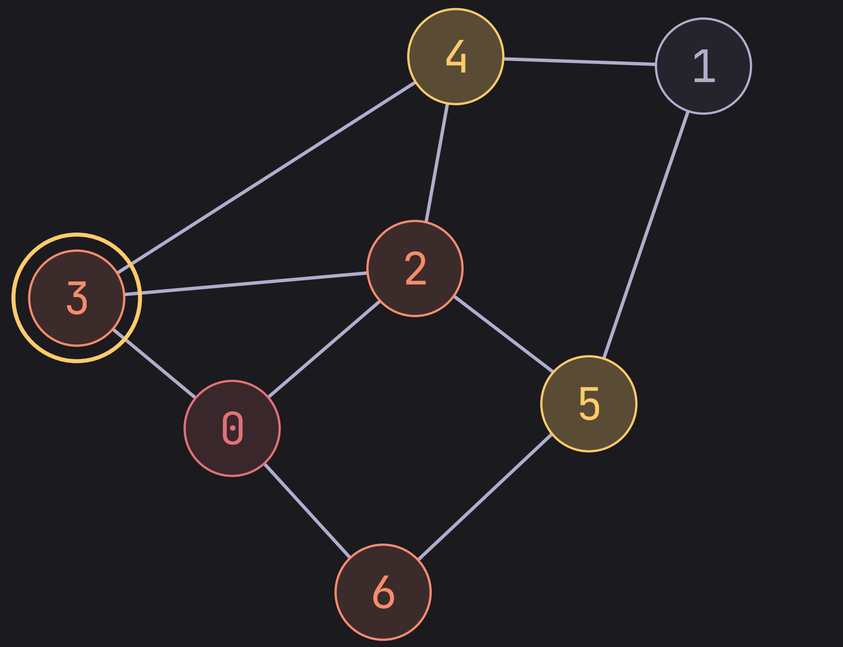



```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search

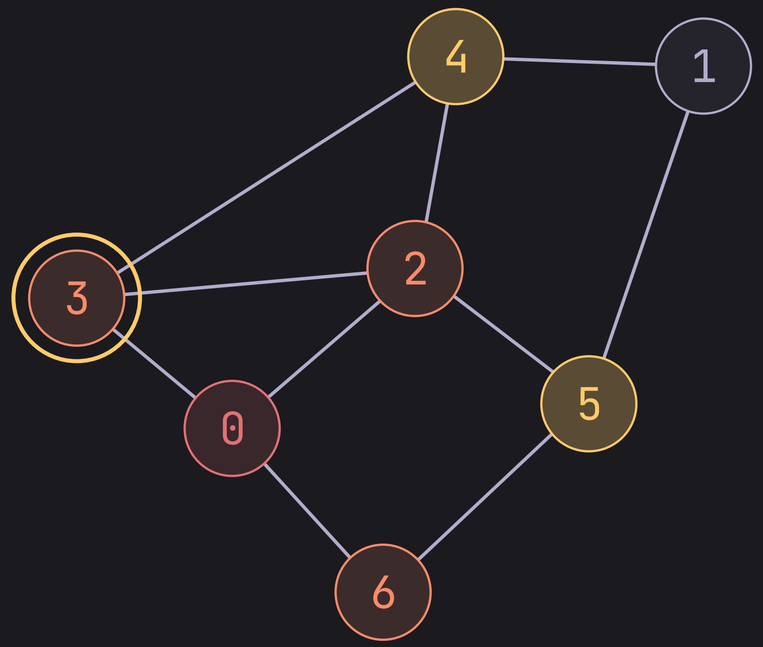



```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
→   for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search

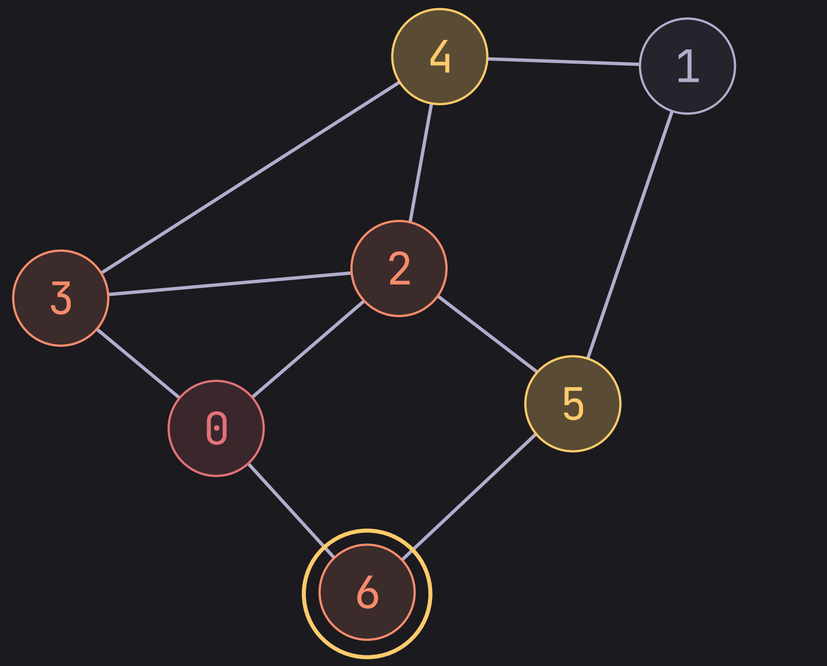


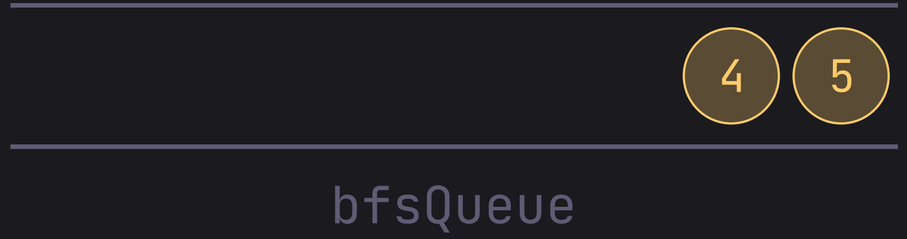

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



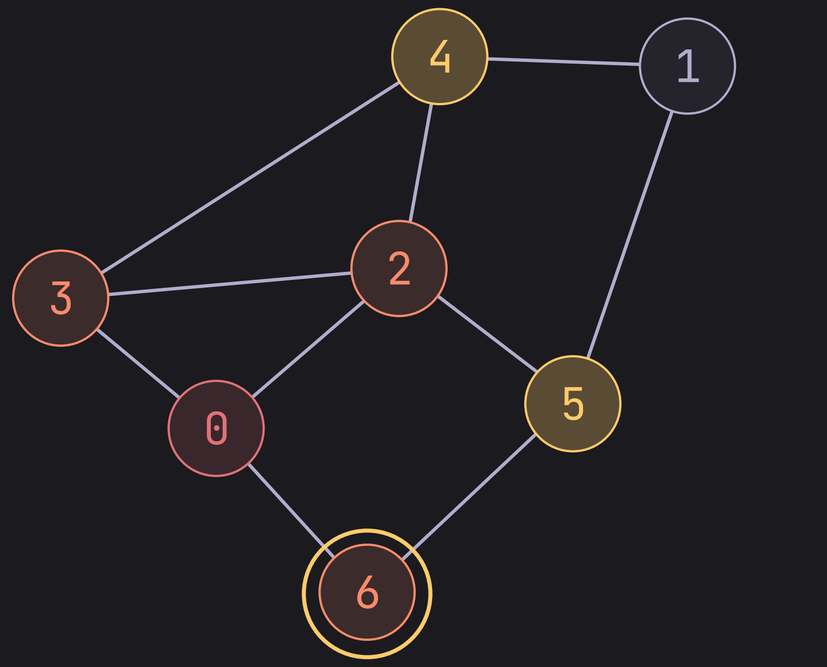


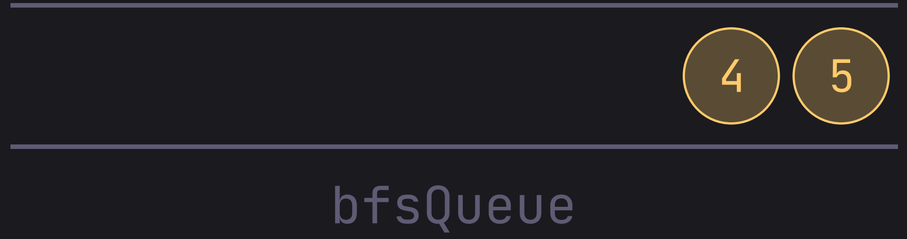

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
⟶  bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



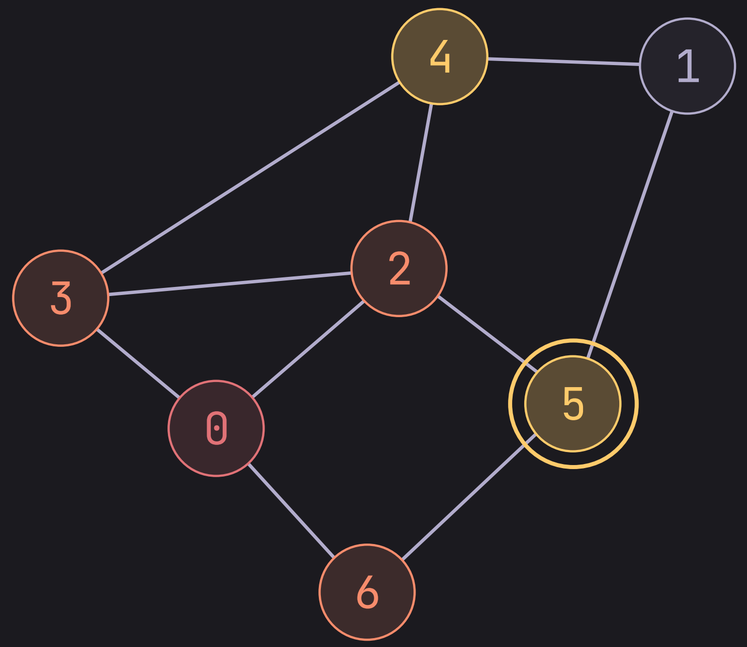


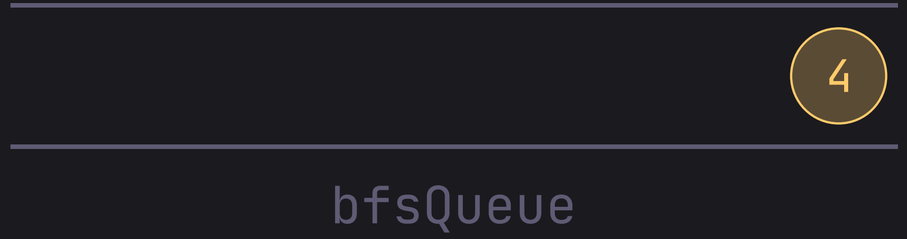

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



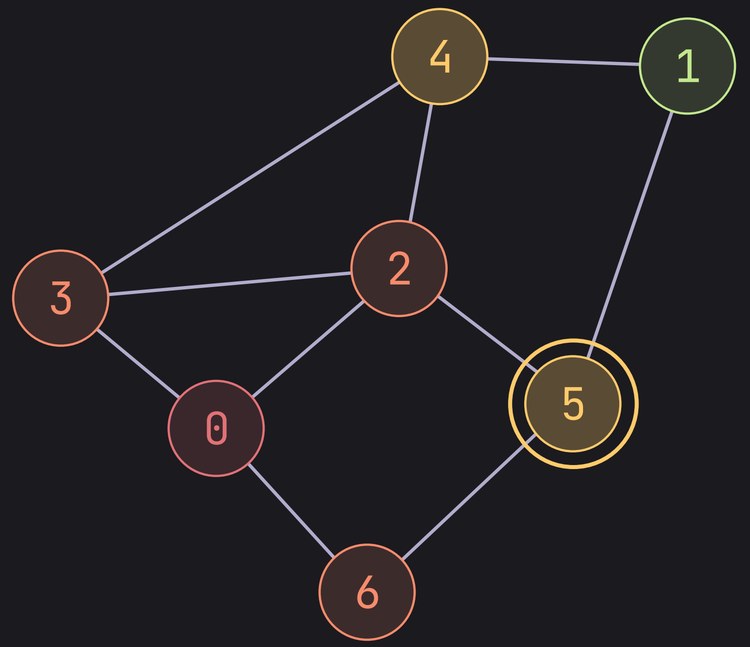


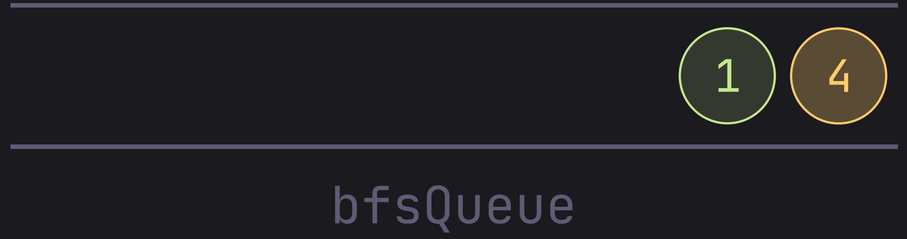

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
⟶  bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



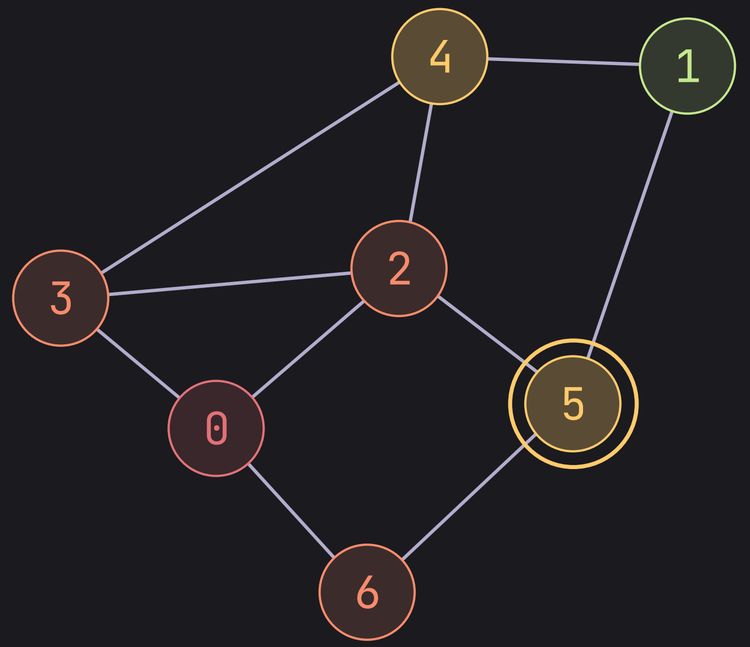


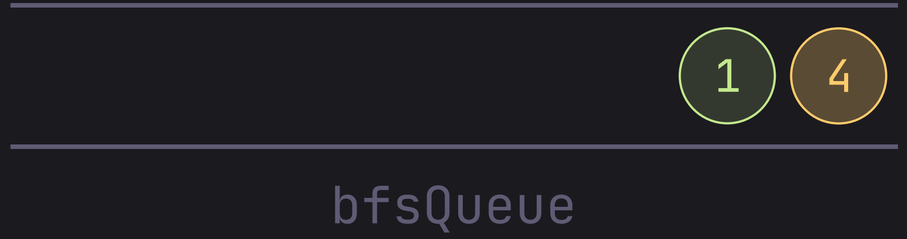

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
      ⟶ bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



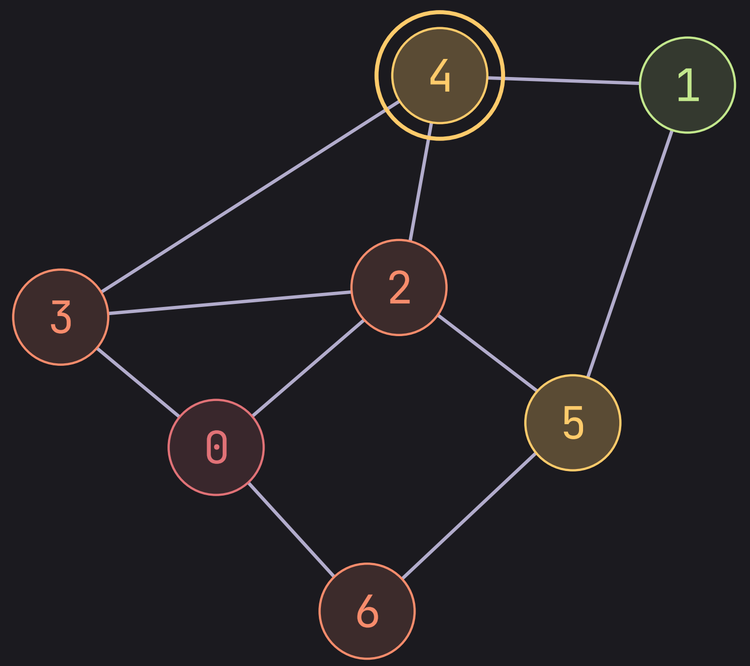


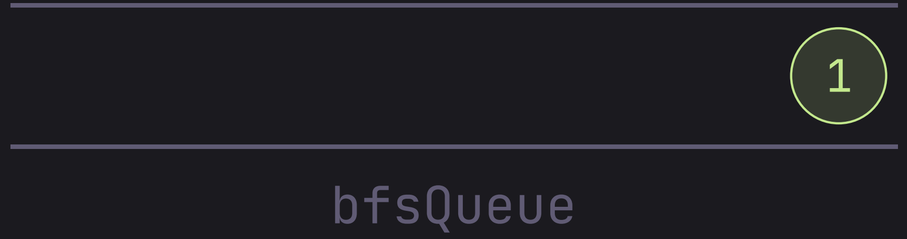

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



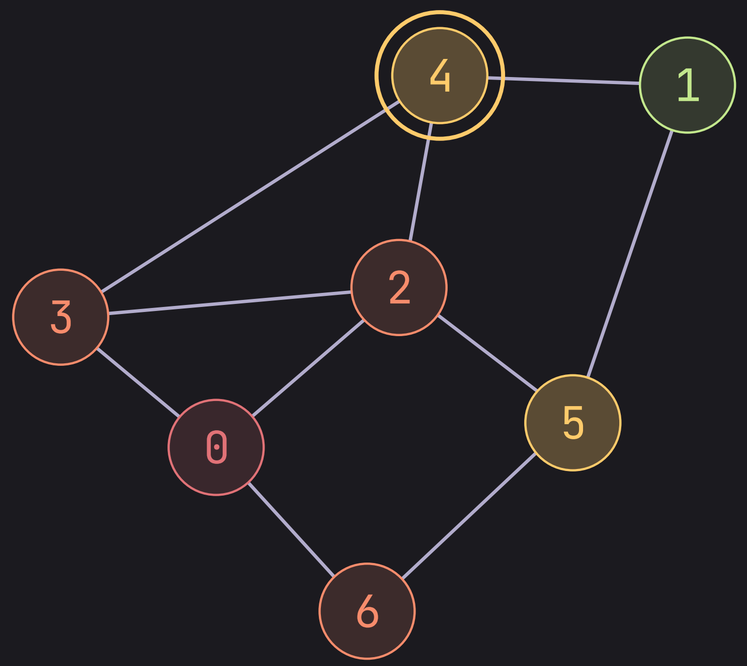


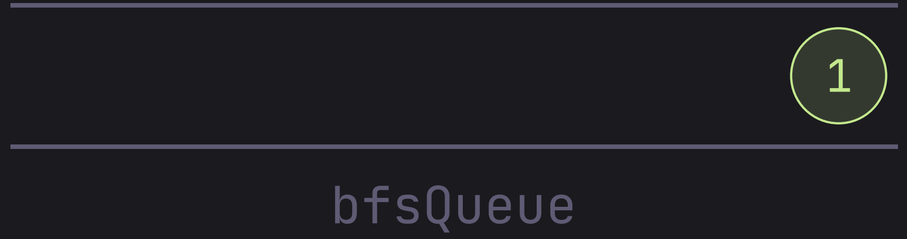

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
⟶  bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



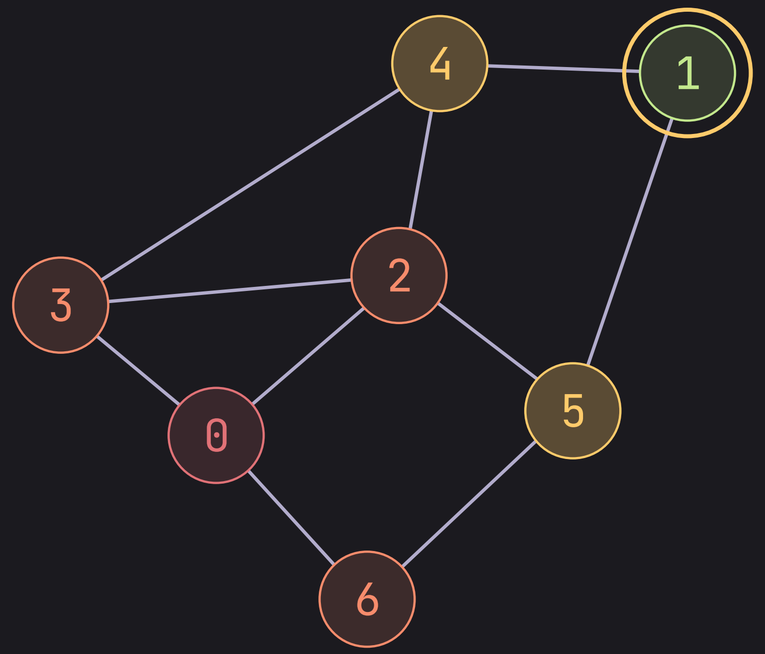



```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



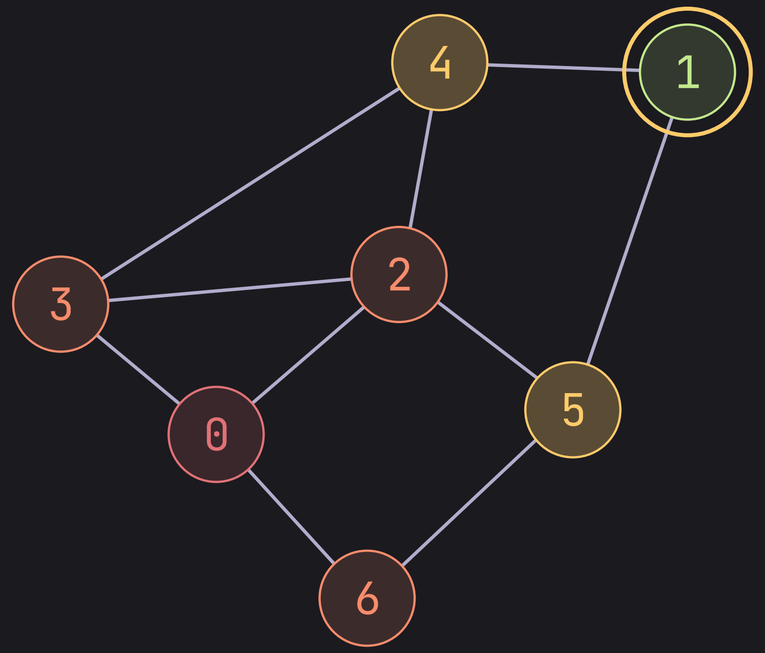
bfsQueue

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
⟶  bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



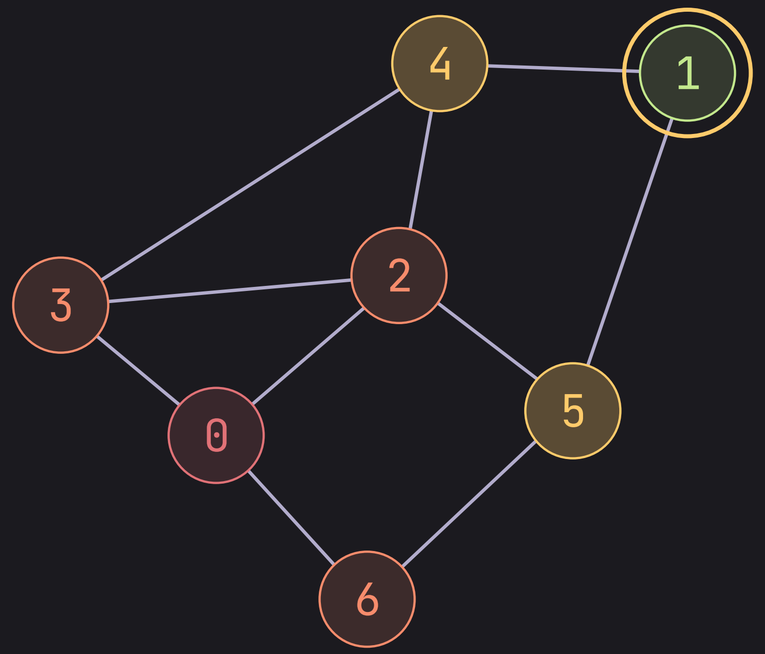
bfsQueue

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

→ while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

Breadth First Search



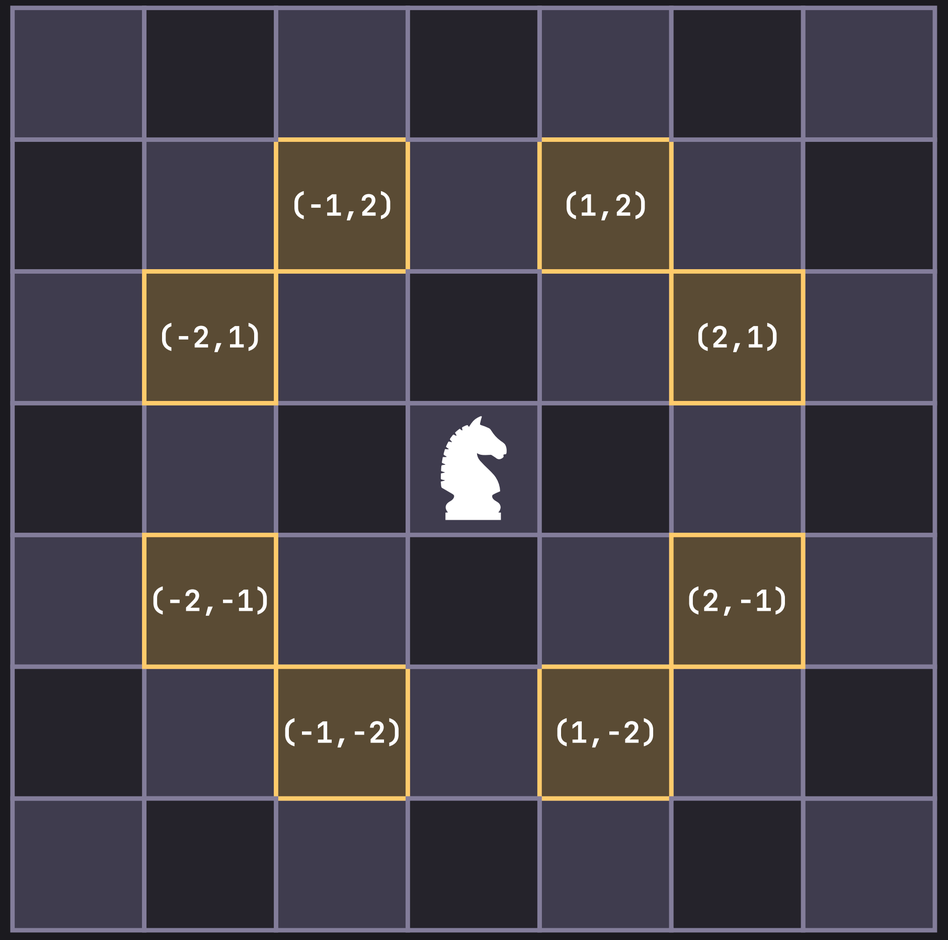
bfsQueue

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
→ }
```

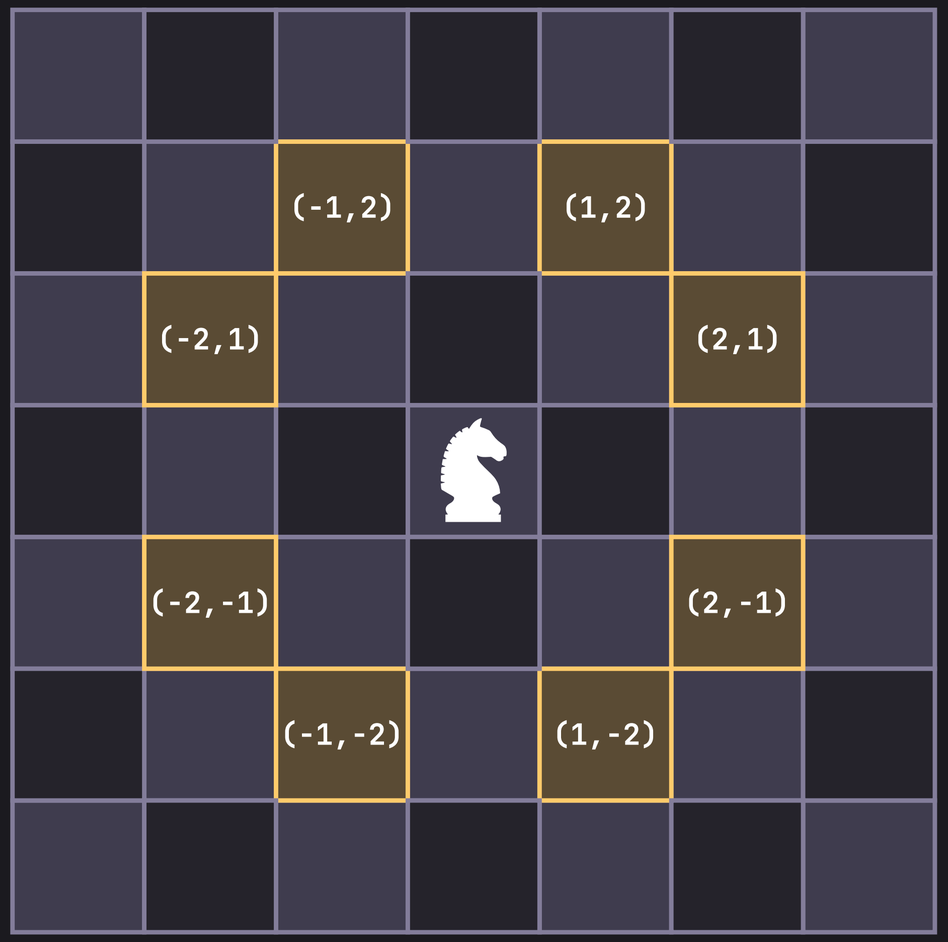
Night Moves



Now here is a **really** important idea!

We dont actually need to generate the whole graph. We just need to be able to calculate the neighbouring verticies

Night Moves



So if we are currently at point

$$(x,y)$$

Then the neighbouring points are

$$(x+2,y+1) \quad (x+1,y+2)$$

$$(x+2,y-1) \quad (x+1,y-2)$$

$$(x-2,y+1) \quad (x-1,y+2)$$

$$(x-2,y-1) \quad (x-1,y-2)$$

Night Moves

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : adjacencyList[v]) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

```
void bfs(int start) {
  std::set<int> visited {}
  std::queue<int> bfsQueue;

  visited.insert(start);
  bfsQueue.push(start);

  while (!bfsQueue.empty()) {
    int v = bfsQueue.front();
    bfsQueue.pop();
    for (int u : getNeighbours(v)) {
      if (!visited.contains(u)) {
        visited.insert(u);
        bfsQueue.push(u);
      }
    }
  }
}
```

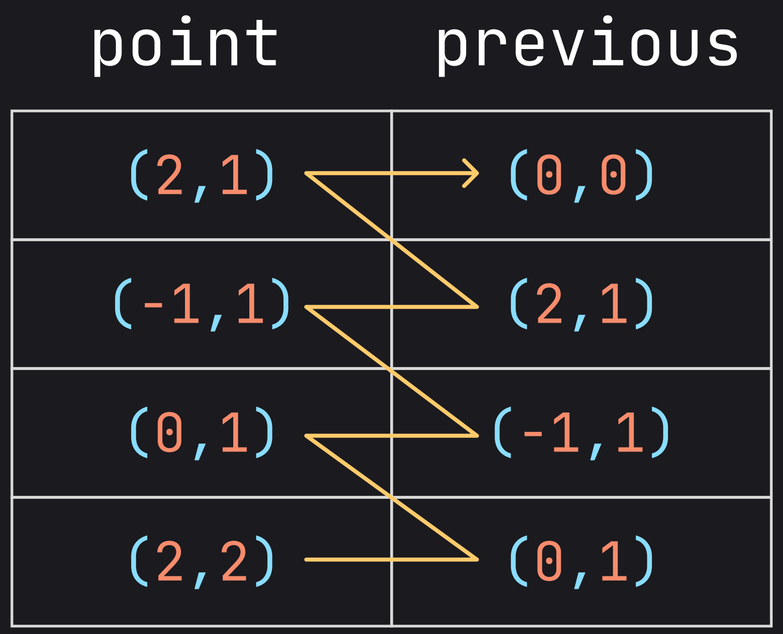
Night Moves

```
Knight::outputPath(const Point& dest,
                   const std::map<Point, Point>& prev)
```

point	previous
(2,1)	(0,0)
(-1,1)	(2,1)
(0,1)	(-1,1)
(2,2)	(0,1)

Night Moves

```
Knight::outputPath(const Point& dest,
                   const std::map<Point, Point>& prev)
```



Night Moves

```
Knight::outputPath(const Point& dest,
                   const std::map<Point, Point>& prev)
```

point	previous
(2,1)	(0,0)
(0,2)	(2,1)
(-2,1)	(0,2)
(-1,-1)	(-2,1)

prev[{-1,-1}] ⟶ {-2,1}

prev[{-2,1}] ⟶ {0,2}

prev[{0,2}] ⟶ {2,1}

prev[{2,1}] ⟶ {0,0}

Night Moves

Let’s try it on Ed