# DISCUSSION CLASS WEEK 7

## CS 141 F20

Dynamic Programming

Graphs Basics

# INTRO TO **PALINDROME**

- A word, phrase, or sequence that reads the same backward as forward
  - madam, 1991,  11, 5, a, cc
  - All strings of length 1(one) is a palindrome

# FUNCTION

```
bool isPalindrome(string str)
    n = str.length
    i = 1, j = n
    while i < j
        if str[i] != str[j]
            return false
        i++, j--
    return true
```

- Time complexity??

# GENERALIZED FUNCTION

```
bool isPalindrome(string str, int start, int end)
  i = start, j = end
  while i < j
      if str[i] != str[j]
          return false
      i++, j--
  return true
```

- Time complexity??

# BUILDING PALINDROME

- **madam** is palindrome, then **ada** would be a palindrome too
- Opposite is true too.

- Adding same chars to both ends of a palindrome would still be a palindrome
- **P** is a palindrome, then **aPa** would be a palindrome too. (P is palindrome of length >=0)

# PROBLEM DEFINITION

- Given a string, find the length of the longest palindrome
  - madam = 5
  - babad = 3 (bab, aba)
  - dbabad = 3 (bab, aba)
  - cbbd = 2 (bb)
  - a = 1
  - ac = 1 (a, c)

# NAÏVE SOLUTION

- Iterate over starting and ending position and check if it's a palindrome

```
int longestPalindrome(String str)
    n = str.length, ans = 1
    for i = 1 to n
        for j = i+1 to n
            if (isPalindrome(str, i, j))
                ans = max(ans, j-i+1)
    return ans
```

- Time complexity??
- Can we make things better?

# A BETTER NAÏVE SOLUTION

- If we already found a palindrome of length *x*, we only would need to check for palindrome for length > *x*

```
int longestPalindrome(String str)
    n = str.length, ans = 1
    for i = 1 to n
        for j = i+ans to n
            if (isPalindrome(str, i, j))
                ans = max(ans, j-i+1)
    return ans
```

# BETTER IDEA??

- If we **know** all the palindrome of size x, we can easily check for palindromes of x+2 with O(1)
  - Know = Memorization
- Finding answers of bigger palindrome easier with smaller palindrome. Bottom-up approach

# FORMING DP

**Base Cases**

- All strings of length 1(one) is a palindrome
- Let's find all palindromes of length 2

**Recurrence**

- Let's find palindromes of length x using palindromes of length x-2

# MEMORIZATION

- How do we store the palindromes?
- 2d array signifying starting and ending points

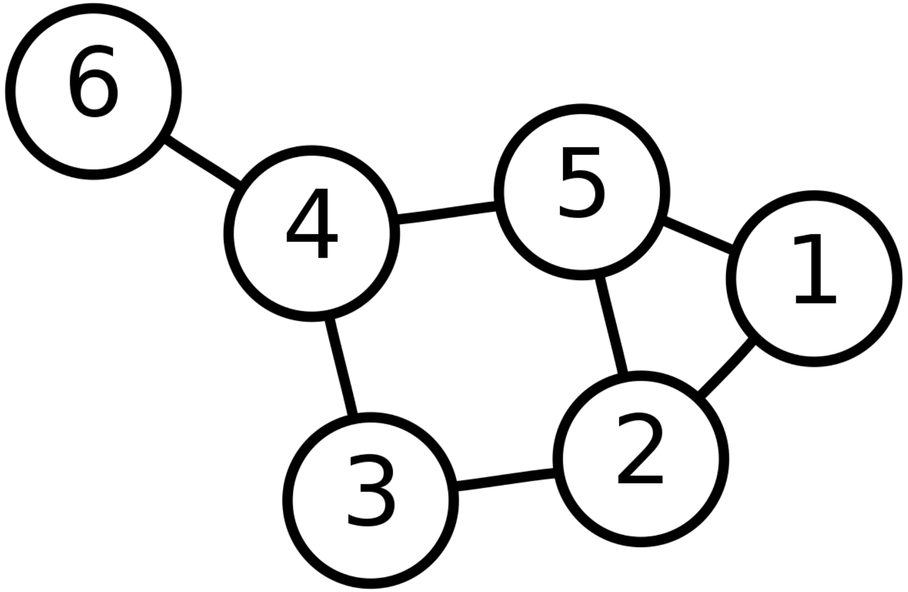
- `if (isPalindrome(str, i, j) == true) => mem[i][j] = 1`
- `mem[x][y] = mem[x+1][y-1] & str[x] == str[y]`

# DP ALGORITHM

```
int longestPalindrome(string str)
  n = str.length, ans = 1
  mem[n][n] = {0}//2d array initialized
  for i = 1 to n
      mem[i][i] =1
  for i = 2 to n
      if str[i]== str[i-1]
          mem[i-1][i] = 1
  for len = 3 to n
      for i = 1 to n
          j = i + len -1
          if (str[i] == str[j] && mem[i+1][j-1])
              mem[i][j] = 1
              ans = len
  return ans
```

# GRAPHS

- A graph G=(V,E) consists of vertices (V, |V|=n) and edges (E, |E|=m) that connect vertices together



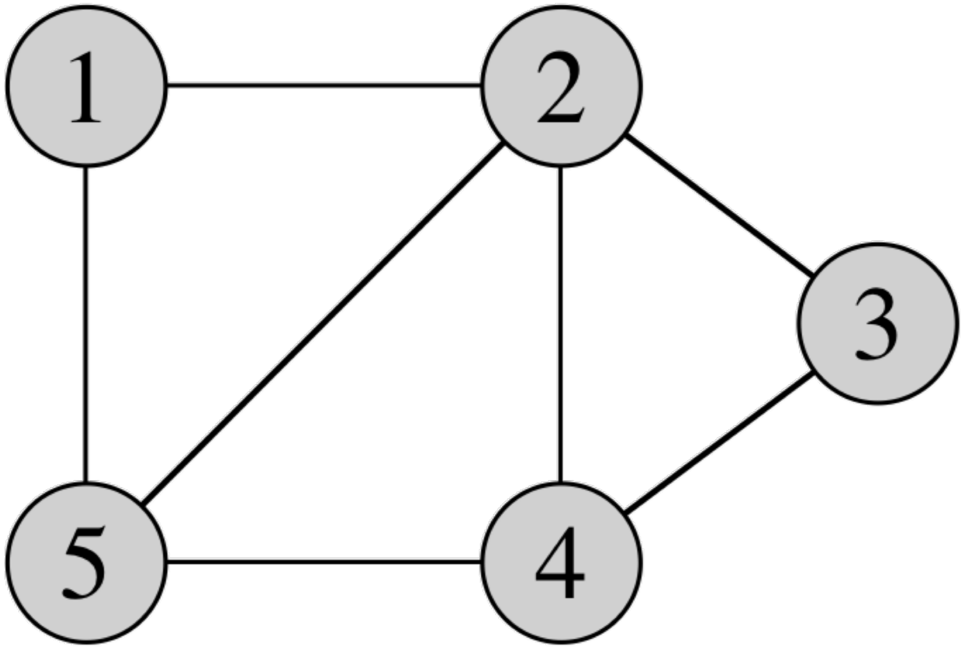

n= 6, m=7

# TYPES OF GRAPHS

- Directed and Undirected graphs
- Weighted and Unweighted graphs
- Connected graphs
- Bipartite graphs
- Acyclic graphs

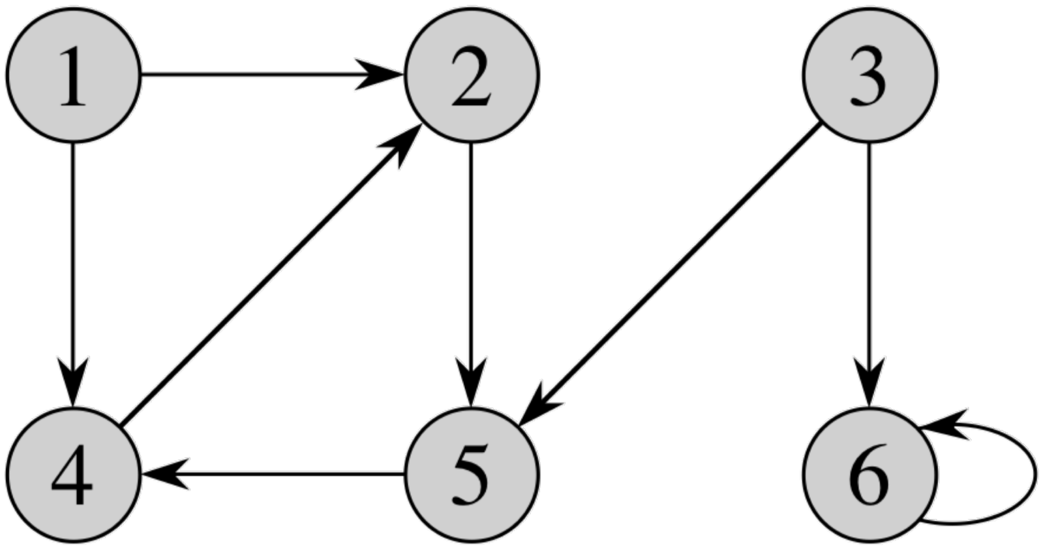
# UNDIRECTED GRAPHS

- No direction in edges
- An edge can be traversed in both ways
- E.g., Facebook friends, most roads, most flights



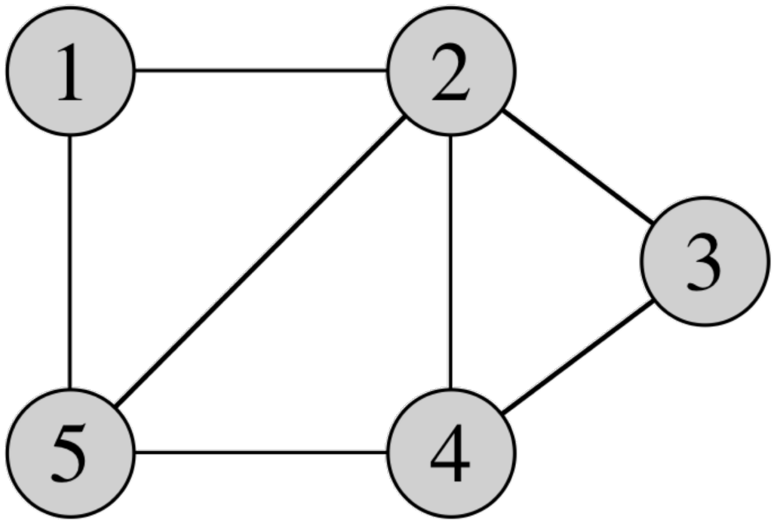
# DIRECTED GRAPH

- Direction on edges
- An edge can be traversed in one direction
- E.g., Twitter follows, code flow analysis

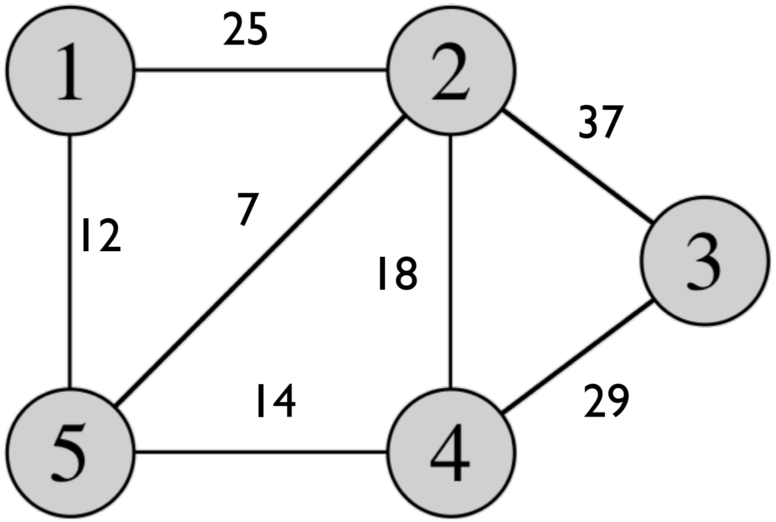


# WEIGHTED GRAPH

- Vertices and/or edges can be assigned weights
- Weights can be cost, capacity, etc.
- E.g., road network, computer network



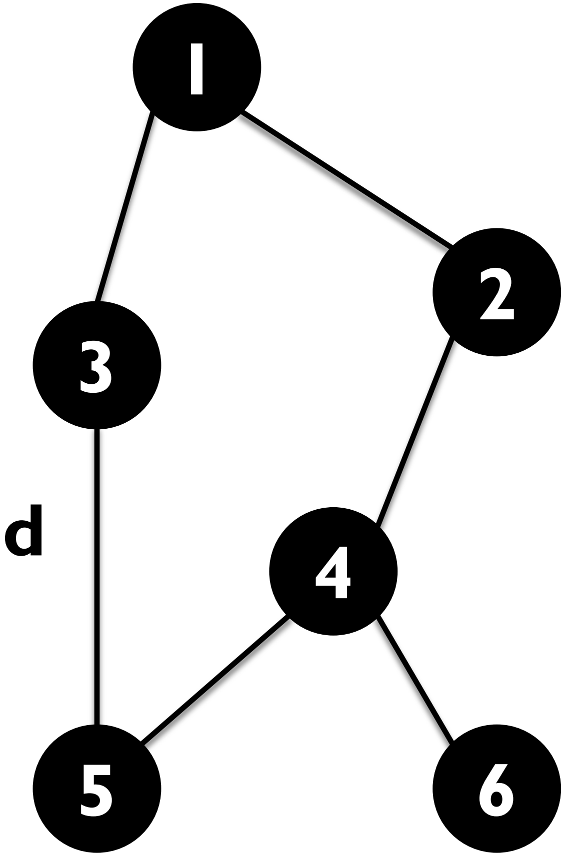

**Unweighted Graph**



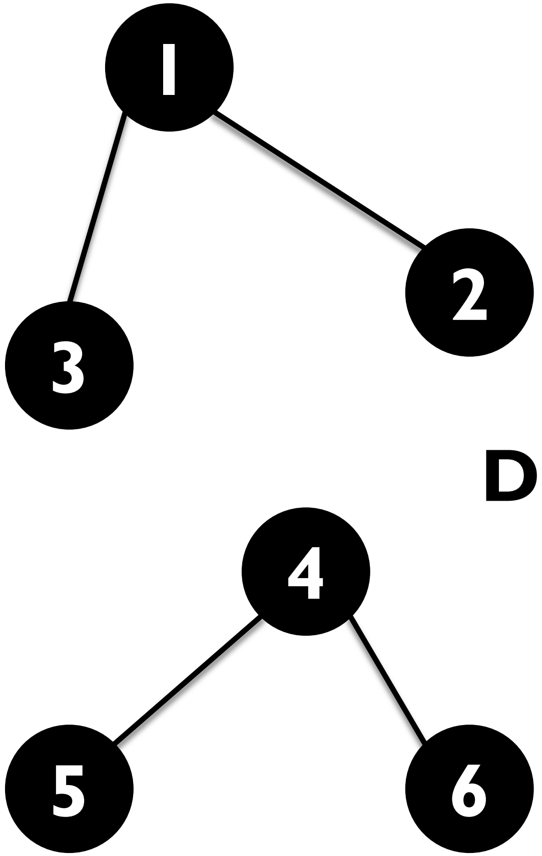

**Weighted Graph**

# CONNECTED GRAPHS

- For simplicity, most graph algorithms assume the graph is connected.  Otherwise, we can run connectivity first, and work on each component.




**Connected Graph**

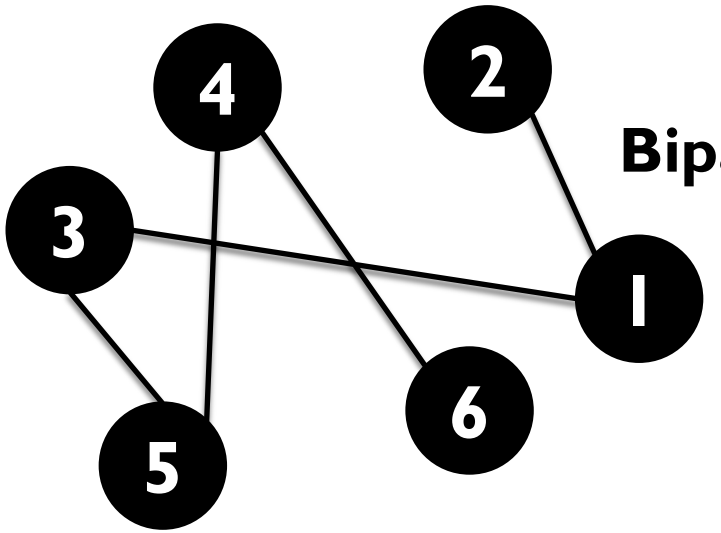



**Disconnected Graph**

# BIPARTITE GRAPH

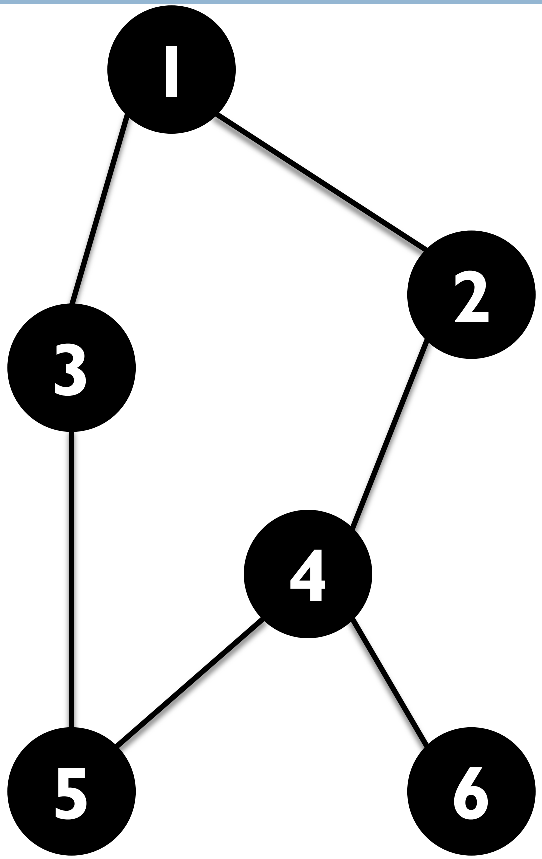
- A graph where the vertices can be partitioned into two subsets:
  - No edges within a subset and all the edges are between two subsets

- Usually, vertices in two subsets have different meanings
  - E.g., students and courses, courses and classrooms, jobs and applicants

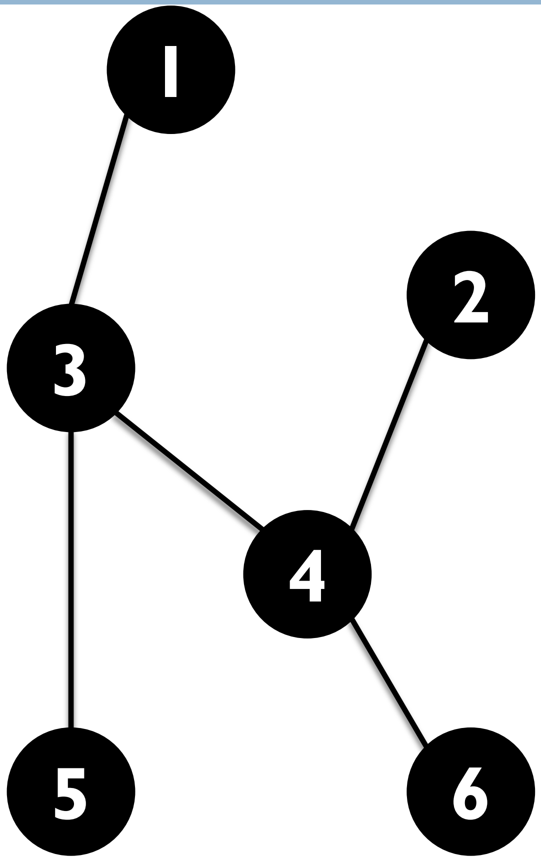



Bipartite Graph

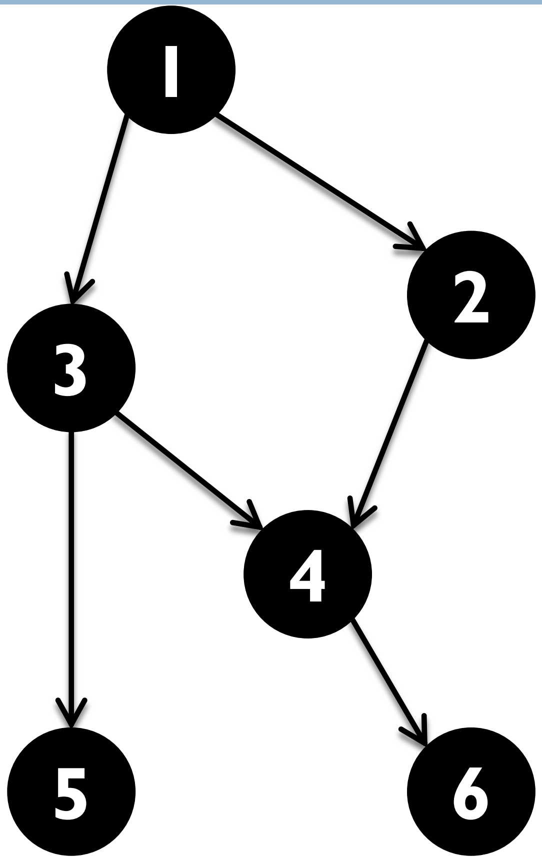
# CYCLIC GRAPH




**Cyclic Graph**




**Acyclic Graph**




**Directed Acyclic Graph (DAG)**

# GRAPH REPRESENTATIONS

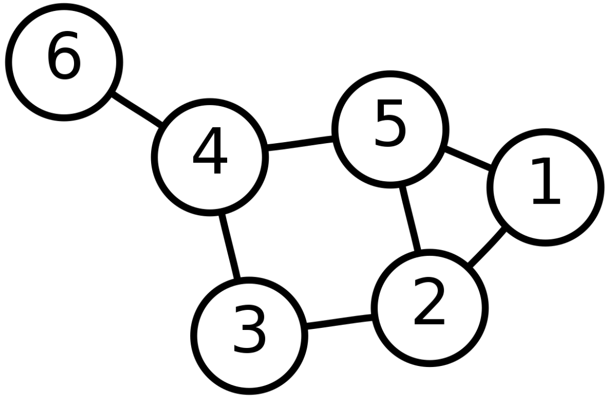
- Adjacency matrix:
  - Use a 2D matrix ADJ of size **n x n**
  - If there is an edge between vertices a and b, ADJ[a][b]= $w_{a,b}$ (for unweighted graphs ADJ[a][b]=1)
  - If there is not an edge between a and b, ADJ[a][b]=0
  - Takes too much space (O($n^2$))
- Adjacency list:
  - Create n singly linked lists, whose root nodes correspond to vertices in the graph
  - Each linked list holds all neighboring vertices of the vertex represented by the root node

# COMPRESSED SPARSE ROW

- Adjacency matrix stores unnecessary information: too sparse!!
- We only need to know when there is an edge, we can infer the other case from non-existence of an edge
- Idea:
  - Only store the existing edge information in an array Edges
  - Keep an extra array, Offset, which indicates which location to look in the Edges array for searching a specific vertex's neighbors

| Vertex IDs | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Offset | 0 | 2 | 5 | 7 | 10 | 13 |

| Edges | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 5 | 1 | 5 | 3 | 2 | 4 | 3 | 5 | 6 | 1 | 2 | 4 | 4 |

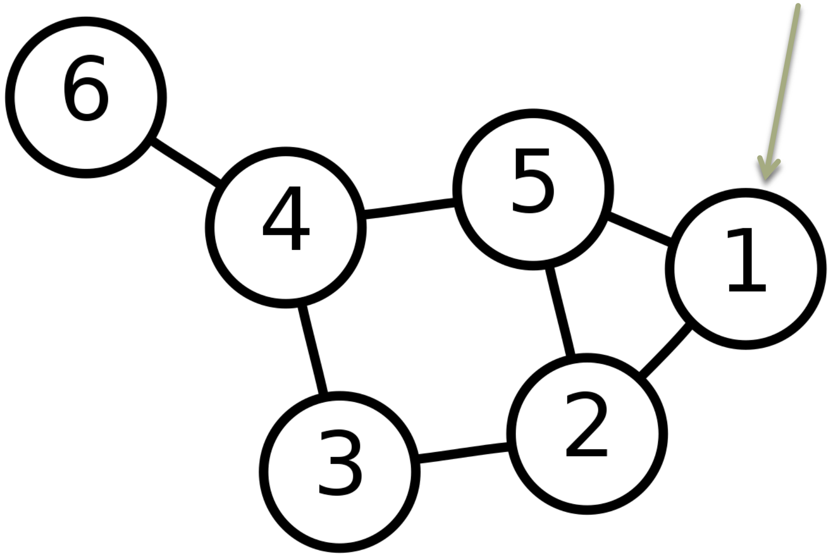



Space: O(n+m)

# GRAPH TRAVERSALS: BREADTH FIRST SEARCH (BFS)

| I | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |

- Start from I
- Mark I as visited
- Next, add 2 and 5 to the queue (visit them next)




| I | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| I | 0 | 0 | 0 | 0 | 0 |

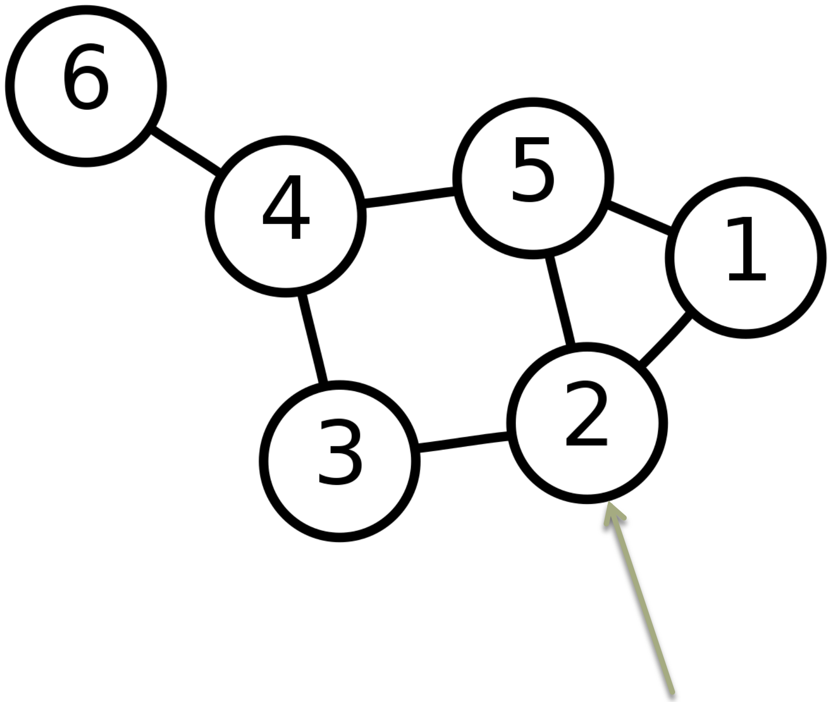
| 2 | 5 | | | | |
|---|---|---|---|---|---|

# GRAPH TRAVERSALS: BREADTH FIRST SEARCH (BFS)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 |

- Pick 2 (Pop 2 from queue)
- Mark 2 as visited
- Add 3 to the queue

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 |



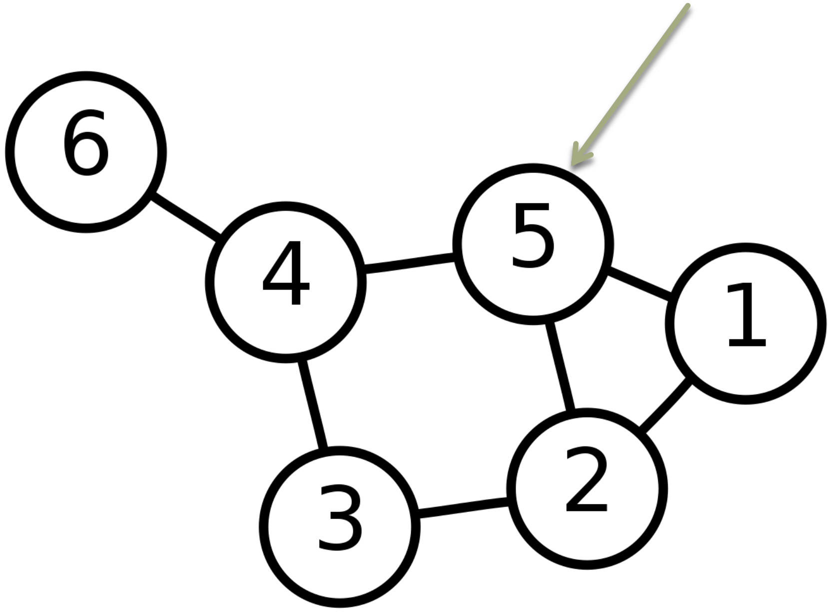

| | | | | | |
|---|---|---|---|---|---|
| 5 | 3 | | | | |

# GRAPH TRAVERSALS: BREADTH FIRST SEARCH (BFS)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 |

- Pick 5
- Mark 5 as visited
- Add 4 to the queue

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 | 0 |



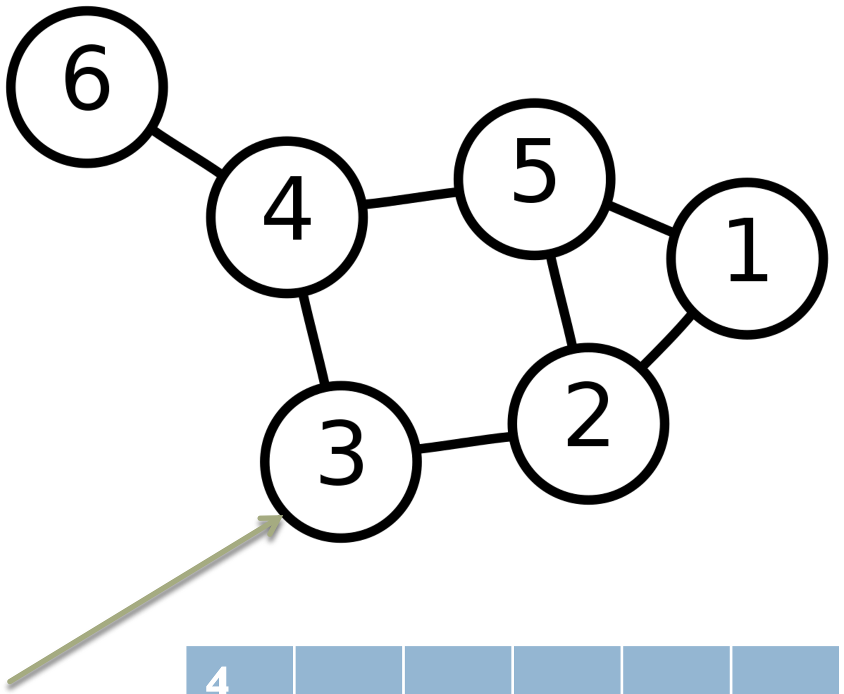

| 3 | 4 | | | | |
|---|---|---|---|---|---|

# GRAPH TRAVERSALS: BREADTH FIRST SEARCH (BFS)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 | 0 |

- Pick 3
- Mark 3 as visited

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 0 |



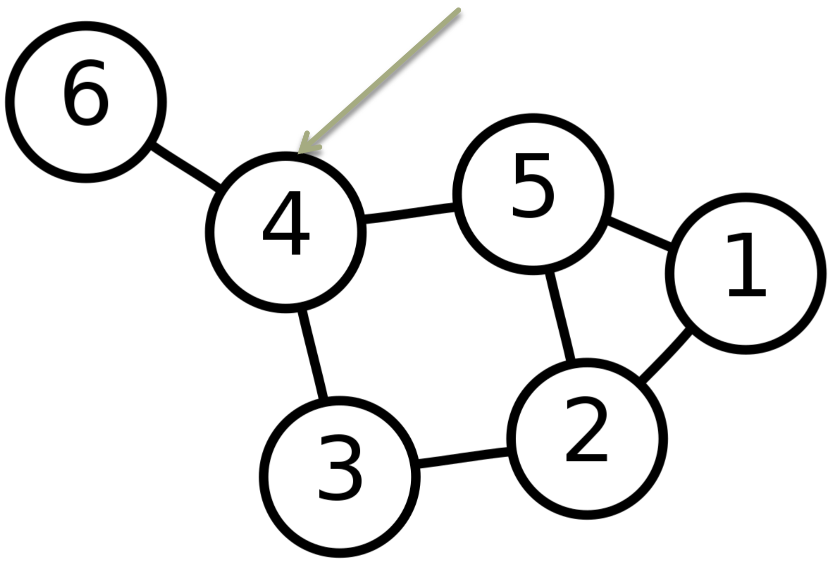

| | | | | | |
|---|---|---|---|---|---|
| 4 | | | | | |

# GRAPH TRAVERSALS: BREADTH FIRST SEARCH (BFS)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 0 |

- Pick 4
- Mark 4 as visited
- Add 6 to the queue

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 |



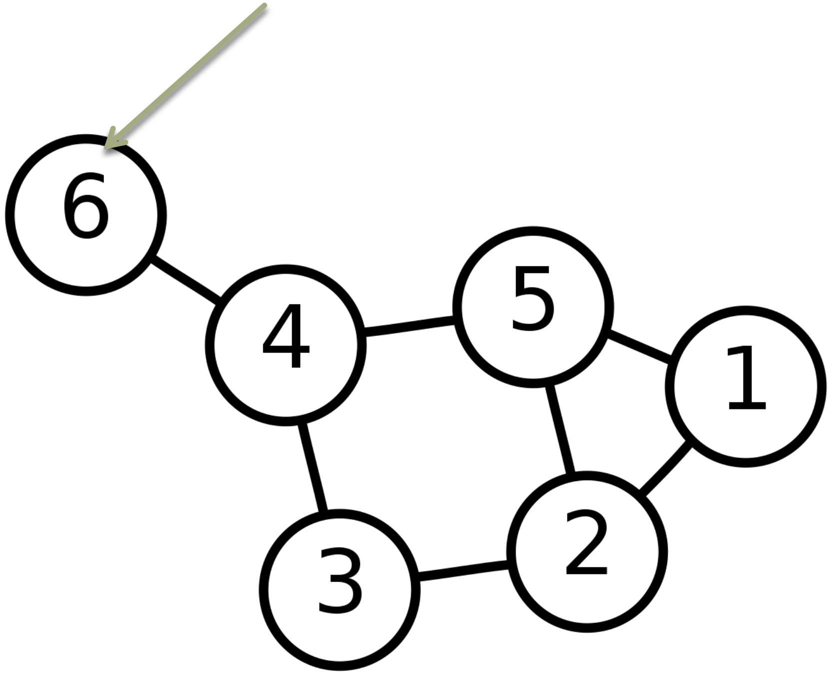

| | | | | | |
|---|---|---|---|---|---|
| 6 | | | | | |

# GRAPH TRAVERSALS: BREADTH FIRST SEARCH (BFS)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 |

- Pick 6
- Mark 6 as visited
- Queue is empty, we are done

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 |

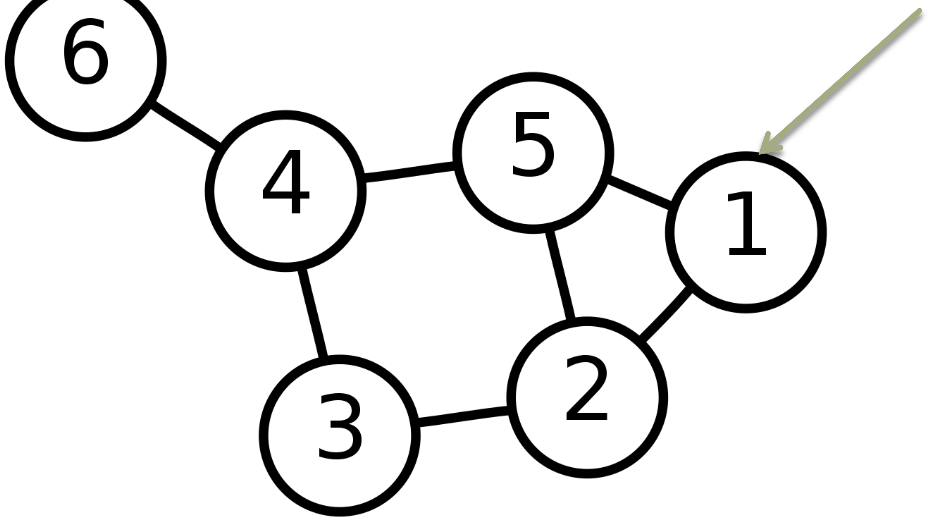



# GRAPH TRAVERSALS: DEPTH FIRST SEARCH (DFS)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |

- Start from 1
- Mark it as visited
- Put 5, 2 in stack

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 |



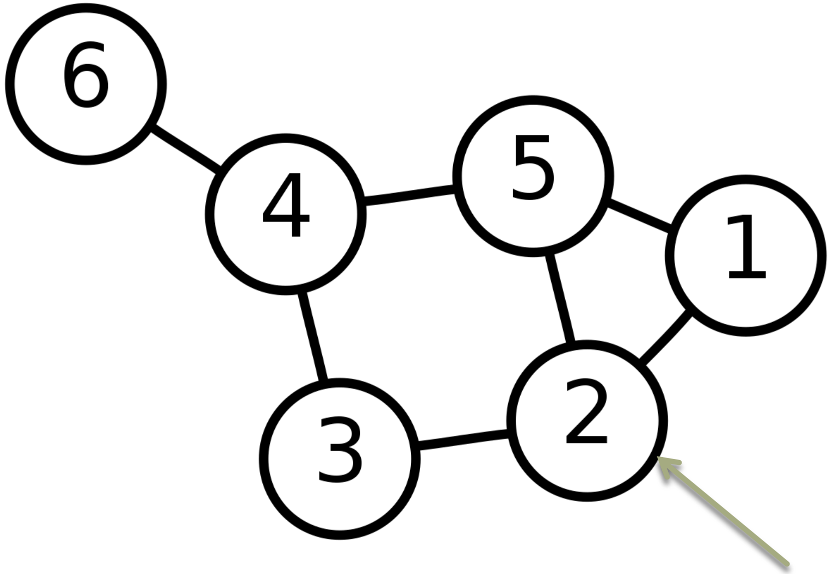

| 2 | 5 | | | | |
|---|---|---|---|---|---|

# GRAPH TRAVERSALS: DEPTH FIRST SEARCH (DFS)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 |

- Visit 2 (Pop 2 from stack)
- Mark it as visited
- Put 3 in stack

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 |

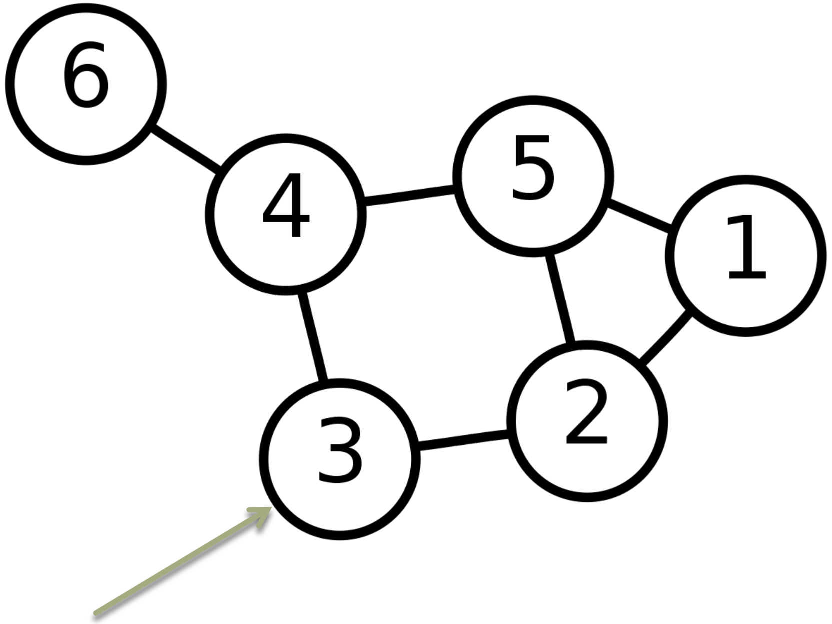



| 3 | 5 | | | | |
|---|---|---|---|---|---|

# GRAPH TRAVERSALS: DEPTH FIRST SEARCH (DFS)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 |

- Visit 3
- Mark it as visited
- Put 4 in stack



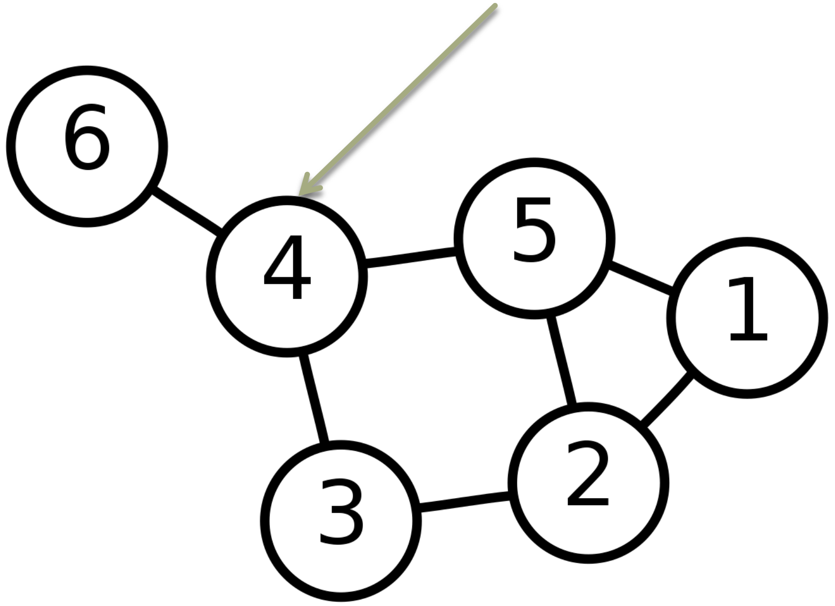

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 |

| 4 | 5 | | | | |
|---|---|---|---|---|---|

# GRAPH TRAVERSALS: DEPTH FIRST SEARCH (DFS)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 |

- Visit 4
- Mark it as visited
- Put 6 in stack




| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 |

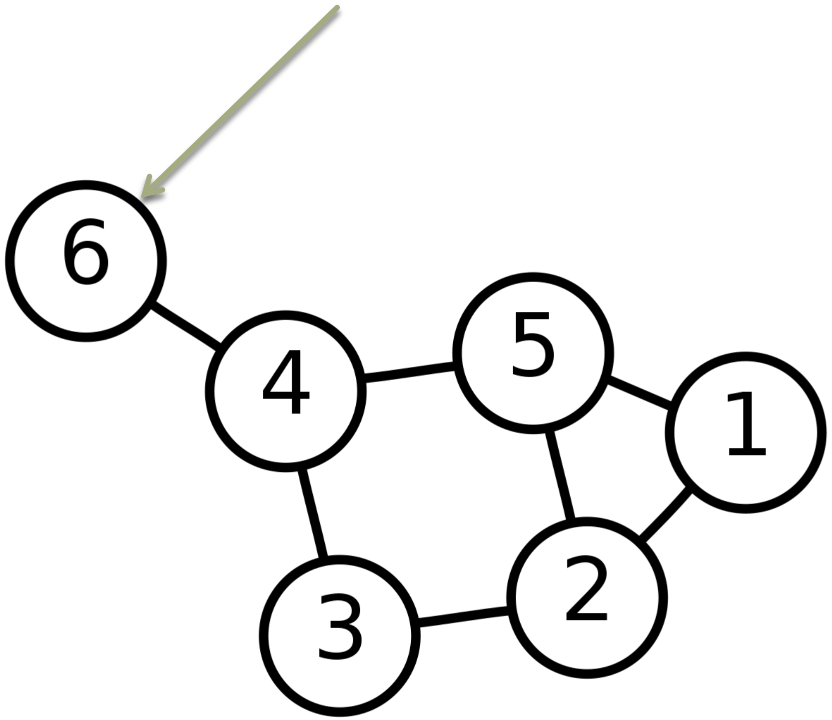
| | | | | | |
|---|---|---|---|---|---|
| 6 | 5 | | | | |

# GRAPH TRAVERSALS: DEPTH FIRST SEARCH (DFS)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 |

- Visit 6
- Mark it as visited

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 |



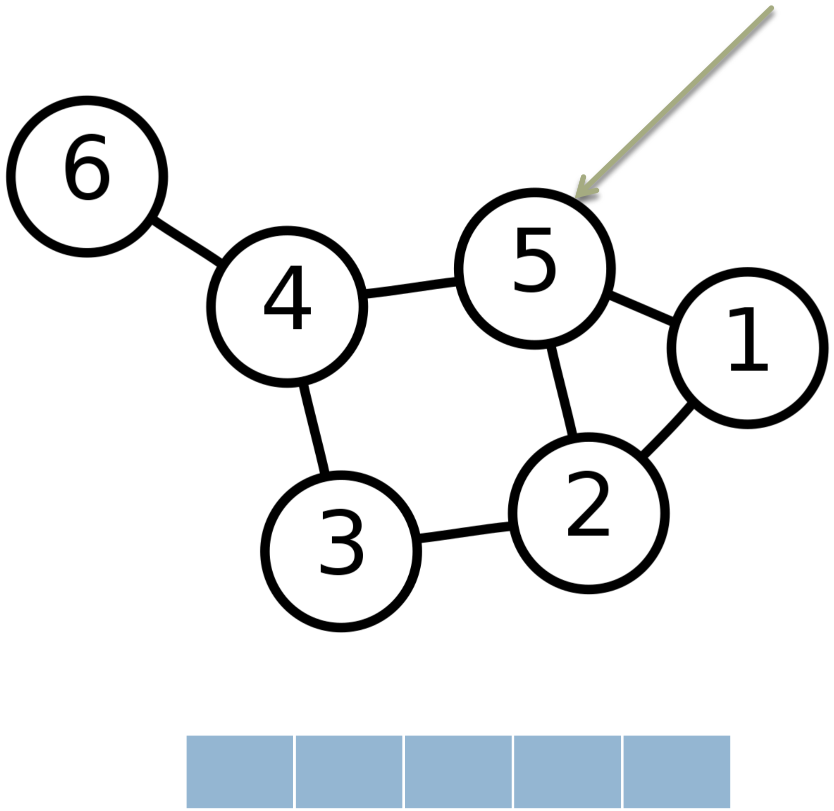

| 5 | | | | |
|---|---|---|---|---|

# GRAPH TRAVERSALS: DEPTH FIRST SEARCH (DFS)

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 |

- Visit 5
- Mark it as visited
- Stack is empty, we are done

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 |



# QUESTIONS??