Topics for this week

- Graphs
  - a generalization of trees (another useful datastructure)
  - adjacency lists representation
  - depth-first-search traversal (DFS)
  - path reconstruction
  - time complexity of DFS

- Python
  - scope of variables
  - lists vs non-lists as parameters
Let’s go back to our old problem:
- How do we search for a path in a maze?
  - How do we represent the problem?
Problem: Return of the MAZE

Let’s go back to our old problem:

- How do we search for a path in a maze?
- How do we represent the problem?

etc. O - nodes, if R rows and C columns then we have RxC nodes.

an edge: a connection between two nodes
Every **graph** contains **nodes** and **edges**, each edge connects two nodes (directed or undirected connection).

**Adjacency lists representation:**

- For every node we describe a list of its neighbors.
  - 0: [4]
  - 1: [2, 5]
  - 2: [1, 6, 3]
  - 3: [2, 7]
  - ...

- We will use a big list containing these as elements, i.e.,
  
  \[
  [ [4], [2, 5], [1, 6, 3], [2, 7], \ldots ]
  \]

  total RxC elements (adjacency lists) in the big list
Graphs: **Depth-first-search (DFS)**

How do we find all possible nodes that are reachable from the start? (I.e., how do we traverse the graph from the start?)

Pseudo code (the first attempt):

```python
def DFS(adjlists, startnode, endnode):
    for every neigh in adjlists[startnode]:
        call DFS(adjlists, neigh, endnode)
```

**Problem:** we keep going back and forth between nodes 0 and 4

**Solution:**
keep track of the nodes that have been already visited and do not go to a node that has been visited before.
Graphs: Depth-first-search (DFS)

How do we find all possible nodes that are reachable from the start? (I.e., how do we traverse the graph from the start?)

Pseudo code:

```python
def DFS(adjlists, startnode, endnode, visited):
    if startnode == endnode:
        print("done")
    for every neighbor of startnode:
        if neighbor has not been visited:
            mark neighbor as visited
            DFS(adjlists, neighbor, endnode, visited)
```


except the start that is `T`
Graphs: Depth-first-search (DFS)

How do we find all possible nodes that are reachable from the start? (I.e., how do we traverse the graph from the start?)

Some Python comments:
Graphs: Depth-first-search (DFS)

We are almost done 😊 The last thing: how do we reconstruct the path from the start to the finish?

Pseudo code:

```
def DFS(adjlists, startnode, endnode, visited):
    if startnode == endnode:
        parent = [0, 2, 6, 2, 0, 1, 10, 3, 4, ...]
    for every neighbor of startnode:
        if neighbor hasn't been visited:
            set the neighbor's parent to startnode
            neighbor as visited
            DFS(adjlists, neighbor, endnode, visited)
```

We start as a list of -1's except the start's position is itself
Graphs: Depth-first-search (DFS)

We are almost done 😊 The last thing: how do we reconstruct the path from the start to the finish?

Pseudo code:

def DFS(adjlists, startnode, endnode, visited):
    if startnode == endnode:
        return
    for every neighbor of startnode:
        if neighbor is -1:
            set the neighbor's parent to startnode
            DFS(adjlists, neighbor, endnode, visited)

def reconstruct_path(startnode, endnode, parent):
    print the end node
    tmpnode = endnode
    while tmpnode != startnode:
        tmpnode = parent of tmpnode
        print tmpnode

parent = [1, 2, 6, 2, 0, 1, 10, 3, 4, ...]
Graphs: Depth-first-search (DFS)

The last bits...

Time complexity:

number of recursive calls = same as number of nodes \( (n) \)

in our case: \#neighbors \leq 4

thus each recursive call takes:

\( \leq 4 \) iterations, \( 2 \) steps per iteration + recursive calls

\( \Rightarrow \) \( \leq 8n \) steps: \( O(n) \)

if each node has about \( n \) neighbors, then the time complexity is \( O(n \cdot n) = O(n^2) \)

Testing:

Usually we write:

\( O(n+m) \) where \( m = \#edges \)

bec. \( \#recursive \) calls = \( n \)

\# neighbors overall = \# edges