$\boxed{12 \quad 4}$ $\boxed{1 \quad 9}$ $\boxed{11 \quad 6}$ $\boxed{18 \quad 3}$

min $\boxed{12}$ move down list and compare each one, replacing as we find new min. Takes n-1 time

max same as min. n-1 time. How do we perform better than $2n$?

Seems like we can't beat this, right? but this isn't true

Group in groups of 2, smaller is compared with min, larger is compared with larger.

Perform $\frac{n}{2}$ comparisons to find min and max in each group.

Then every single one is compared against something; n

Hence a better answer is $\frac{3}{2}n$

Lecture 5 Graphs 1/19/18

Graphs are defined by a set of vertices V and a set of edges E.

A graph can be represented by an adjacency matrix

$$\begin{array}{c c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \left[\begin{array}{ccc} & & \\ \square & & \\ & & \end{array}\right] \end{array}$$

It's an nxn matrix where n = # edges. a 1 at 1,2 means there is an edge there. Otherwise it is 0.

In an undirected graph this adjacency matrix is symmetric.

Another representation is by linked list:

$1 \longrightarrow \boxed{2} \longrightarrow \boxed{5}$

$2 \longrightarrow \boxed{1}$

3.
⋮

- In the adjacency matrix there will always be n^2 entries, whereas the linked list is dependent on the number of edges.

However to find if i and j are directly connected then we just need to look it up on the matrix. We have to search the linked list though.

Path - A sequence of edges that takes you from one vertex to another

Cycle - A path where the beginning vertex and end vertex are the same.

Connected - If I can go from any vertex to any other vertex the graph is connected.

Max # of edges : $\binom{n}{2}$ → dense.
Min # of edges (still connected): $n-1$ → sparse

BFS (Breadth First Search)
Look at things that are close first before moving onto farther elements.

As you visit vertices, you look at all neighbors and start placing them in a first in, first out data structure if they have not yet been explored (or marking them somehow prior to insertion).

We visit every edge twice, and hence BFS runs in $O(|E|)$ time. This works only for connected graphs. For general graphs, we must consider disconnects, hence the order is $O(|E|+|V|)$.

<u>BFS Tree</u> - Tree created by the "exploratory" edges of a BFS. The starting point of the BFS will be the root. The paths in the BFS tree to the root will be the shortest paths.

DFS (Depth First Search)

Follow a path down to its completion when searching.

Once again, the edges that find a vertex for the first time with this algorithm form a DFS tree.

Edges that let us see a vertex for a second time are called <u>backward</u> edges. If there are backward edges then there is a cycle in the graph.

This algorithm also runs in $O(|E|+|V|)$, also written as $O(e+n)$

<u>Majority Problem</u>

Consider an election with candidates. We have an array showing who each vote was for.

$\boxed{2\ 3\ 2\ 1\ 2\ 3\ 2\ 1}_n$

A winner has a <u>majority</u> $>\frac{n}{2}$

We see that there is at most 1 majority.

We sort the array. A candidate that has a majority must appear in $n/2$. Thus, we take this candidate and count its number of appearances. If it is $> n/2$ then we found our candidate, otherwise there is no candidate with a majority.

This algorithm takes n time to check and nlogn to sort, so overall it is O(nlogn).

How can we solve this problem in O(n)?

Pick 2 different elements. Remove them.
Keep doing so until there is only 1 candidate left. ← If no left at end, then no majority candidate
This candidate is the only majority possibility: check it.

A problem: how do we pick different elements? How do we remove them?

We have a candidate and a count:
Start from left to right:

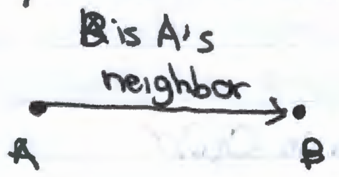
- If no candidate, make new candidate
- If candidate matches, add 1 to count
- If candidate doesn't match and count is 1, wipe both at
- If candidate doesn't match and count >1, decrement count.

At the end, the candidate that remains (if any) is the potential majority candidate.

Lecture 6

When we have a directed graph, we can only travel along edges in, one direction.

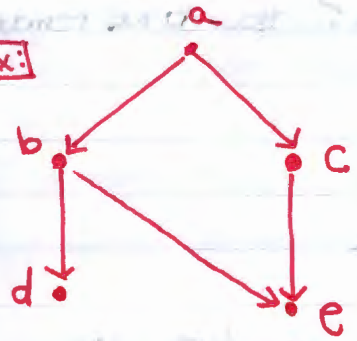
The algorithms for BFS and DFS are the same, with the simple change that a "neighbor" is a vertex that the curent vertex connects to in this way:



DAG - directed acyclic graph

Sorted graphs - we use topographical sort, where the directed relationship is maintained in the sorting

ex:



a c b e d

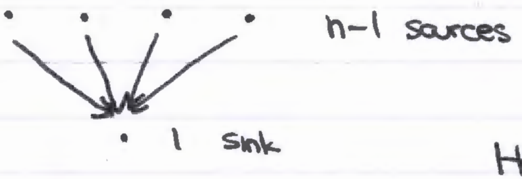
#of edges going into a vertex = in degree
#of edges going out of a vertex = out degree
undirected graphs just have <u>degrees</u>

if vertex has:

- indegree = 0, it is a <u>source</u>
- outdegree = 0, it's a <u>sink</u>

A topological sort is only possible on a DAG.

What is the highest possible of topo sorts for a connected DAG?



(n-1)! possible outputs!

Have all vertices init value of out and in to

Arbitrarily add edges and incr the appropriate values for appropriate vertices

↓

Algorithm for Topo Sort

Calc in degrees and outdegrees $O(n^2)$ $O(e)$

Generate list of sources $O(n)$

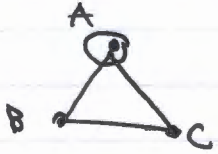
Output a source $O(1) \longrightarrow O(n)$

update source list $O(n) \longrightarrow O(n^2)$ $O(e)$

NP Complete Problems - for now, just know that they are very difficult

A graph is called bipartite if I can take the vertices of the graph and partition it into two groups and the edges of the graph only go between groups.

Is every graph bipartite? No. Think of odd cycles.



If A is in group 1, then B and C must both be in 2, but cannot because of the edge between them

How do we come up with an algorithm that will produce a bipartite graph?

BFS!