CS 180 midterm revision

Celebrity Problem
Two Egg Problem
Stable Matching
Time Complexity
Graphs
Greedy Paradigm
Interval Scheduling

BFS	
Algorithm	queue (FIFO) data structure to push neighbors from the start into the queue, and once visiting, mark the node as visited. Then when we look at the first neighbor put all its neighbors in the queue and mark the neighbor as visited. So on and so forth.
Runtime	The runtime of looking at the neighbors and putting into the queue is $(n-1)$ at worst case, and doing that for n nodes giving us O(n(n-1))=O(n^2) Another way of saying this is sum of all degrees of each node. When adding an edge we always increase the degree by 2, so we see is $2E \rightarrow O(E)$ per component. Thus the complete analysis is $O(E+n)$ as this applies to both complete and incomplete graphs! E is the number of edges and could be n.
Notes	BFS checks for cycle in a graph

Code	`Unmark all vertices`
	`Choose an arbitrary starting vertex v`
	`mark v and push it onto a queue`
	`Queue Q`
	`While queue is not empty`
	`  Choose the first on the queue x`
	`  Mark x as visited`
	`  For each unmarked neighbor w`
	`  Add w to the end of the queue`

Bipartite (Two-Colorable) Graphs	
Algorithm	Imagine lining up your vertices into two vertical columns Left and Right. From this we can make our connections making sure that no edge exists in the vertical column itself. That is the edges only connect between Left and Right. Any simple path is bipartite. We can then use BFS to find out if a graph is bipartite. We do this by giving a vertex a color (blue or green) and if at any point we have a conflict in colors then it can't be bipartite
Runtime	Same as BFS
Notes	
DFS	
Algorithm	Go down one direction, then following the same direction go down until you hit a leaf node. Then backtrack and go down the second path from the node before current and continue down.

	stack (LIFO) data structure to push neighbors from the start into the stack and once visiting, we mark as visited.
Runtime	visit 2E thus O(2E) time steps with a more complete analysis of being runtime of $O(E + n)$.
Notes	DFS trees are not unique, can use DFS to find cycles in a graph. Recursion is helpful for DFS.

Code	`DFS(vertex v)`
	`Visit(v)`
	`For each child of v, w`
	`  DFS(w)`

Unweighted minimum path	
Algorithm	BFS. f we look at a vertex on level m then we know that that vertex is connected to at least one vertex at level m-1 because if it wasn't then we wouldn't have discovered it as a neighbor and then it would not be in the BFS Tree. This way we can traverse from level m all the way back to the start, with a distance of m.
Runtime	This still has the worst case runtime of traversal of a BFS – O(V + E).
Notes	Since it is unweighted, the number of edges that a path goes through matters
Topological sorting, DAG (Hamiltonian path in DAG) A topological sort of a directed graph is a linear ordering of its vertices such that for every directed edge, u comes before v	
Algorithm and *runtime for each step*	1. compute the in-degree of every vertex (number of edges that go into the vertex) and the out-degree of every vertex (number of edges that go out of the vertex) *This takes O(E) because for every edge we increment the in-degree and the out-degree of a vertex which takes 2 steps for all edges in the graph which is 2E.* 2. All the vertices that have an in-degree of 0 are known as sources. We can have more than 1 source in a DAG. Go through the list of vertices from before, and put the vertices with in-degree 0 and put it into a separate list called sources list. *We are going through the n vertices in the list then this is O(n).* 3. Pick one source and output it. This source will now have out-degrees. Because we have output this source, we can remove this source from both the lists, and all of its neighbors don't have an in-degree from this source anymore, so we can decrement the in-degree for these vertices. So we basically go through initial list, and find these vertices that are connected to the source and decrement their in-degree. And if they are 0, we put them into the source list. *Outputting the arbitrary source is O(1); it takes O(n-1) to modify the in-degrees as at worst we can have n-1 neighbors for 1 source, and modifying the source list is O(n)* 4. We then repeat step 2. *And we do the above steps n times. O(n^2)*
Runtime	Pessimistic time complexity: O(E+n(n-1))=O(E+n^2)= O(n^2) However, overall we only modify in-degrees E times, then we are not going to check the same edge again, O(E). Similarly, we modify the source at most O(n) times. Therefore, we have O(E+n)

Notes	A graph contains a cycle cannot be topologically sorted due to paradoxical ordering. The first vertex must have no incoming edges.
Partition DAG (for any two vertices with in a group there is no path between them)	
Algorithm	1. Take all the sources of the graph and put them into group i. Delete all the sources and change all the in-degrees (Topological sort). 2. Repeat step 1 with the new sources and put into group i + 1.
Runtime	The runtime is O(E+n). This works because by definition sources have no connections as in-degrees thus all sources are independent to each other. Assuming we have k nodes linked as a Linked List, then we will get k groups.
Notes	
Shortest path in DAG/Dijkstra algorithm	

Algorithm	`Let distance of start vertex from start vertex = 0`
	`Let distance of all other vertices from the start = infinity`
	`For each unvisited vertex, choose the vertex v with the smallest known distance from the start vertex until there is no more unvisited vertex`
	`  for the each its unvisited neighbors w`
	`  calculate distance from the start vertex`
	`  if the calculated distance < the known distance`
	`  update the shortest distance`
	`  update the previous vertex with the current vertex`
	`  endif`
	`  mark the current vertex w as visited`
	`  endfor`
	`endfor`

Proof	By induction: Base case |S|=1, d(s) = 0 $P_{optimal}$ cannot be shorter than P_d because it is already at least as long as Pd by the time it left S
Runtime	O(V^2) Graph represented using adjacent list can be reduced to O(E log V) with the help of binary heap.
Notes	The algorithm does not always find the shortest path if some edges have negative values
Prim's algorithm/MST	

Algorithm	Given G=(V, E), we partition it into two subsets. MST theorem states that for any partition of edges, the minimum edge from left to right must be in the minimum spanning tree.
	
	`Start with set S having no vertices, and set P have no edges`
	`Pick an arbitrary node v in G, put it into the set S`
	`For all neighbor vertices w of vertices in set S` (O(E))
	`  Find w which has the smallest v-w weight such no cycle is created` (O(logn))
	`  Put w into S, put v-w into P`
	`  Repeat the process until all vertices are visited OR until P contains n-1 edges such that all vertices are visited`

	(ensure that one of the end points of e touches S and the other does not, and this is the only difference with Kruskal's algorithm)
Runtime	By using **minimum binary heaps**, time complexity is O(m*logn) For a minimum binary heap: Extraction: O(logn) Contain: O(1) Add: O(logn) decrease: O(logn)
Notes	Prim's algorithm finds a minimal spanning tree in a **connected, weighted** graph G In Prim's, you always keep a connected component, starting with a single arbitrary vertex. You look at all edges from the current component to other vertices and find the smallest among them. You then add the neighboring vertex to the component, increasing its size by 1. In N-1 steps, every vertex would be merged to the current one if we have a connected graph. If disconnected, there will be no spanning tree at all
Kruskal's algorithm/MST	

Algorithm	`Start with S having no edges`
	`Add an edge of minimum weight not contained in S, to S, such that S does not contain a cycle`
	note that it is allow to choose an edge that is not connected to any edge in S
	Keep doing this until S contains n-1 edges

Runtime	Kruskal's algorithm sorts edges instead of vertices, so we can use merge sort to do this in O(E*logE)
Notes	Kruskal's algorithm finds a minimal spanning tree in a **connected, weighted** graph G In Kruskal's, you do not keep one connected component but a forest. At each stage, you look at the globally smallest edge that does not create a cycle in the current forest. Such an edge has to necessarily merge two trees in the current forest into one. Since you start with N single-vertex trees, in N-1 steps, they would all have merged into one if the graph was connected.
Both Prim's and Kruskal's algorithm work if some edges have negative value. the safe edge added to the subset of MST is always a least weight edge for Prim's and kruskal's algorithm.	
Clustering problem	
If you are given a data set and you want to group them in k clusters. The objects in the same cluster should be "close" to each other while the objects in separate clusters should be "far" away from each other.	
Algorithm	Use Kruskal's algorithm Add edges between pairs of point in order of increasing distance d(P_i, P_j). If P_i and P_j are already in the same set, do not add it. Stop when we obtain k connected components
Runtime	Same as kruskal's algorithm
Notes	Proof: We know we have k clusters because we stop when we get k clusters. By contradiction we assume there are clusters c1', c2' and c3' that are optimal which are different to our clusters c1, c2 and c3. This would mean that there is some cr (arbitrary cluster from c1, c2 and c3) that has some of cx' and cy'

	(arbitrary clusters from c1', c2' and c3'). Look at Vi from cx' and Vj from cy' which were both included in cr. Look at the last node Vx that belongs to cx' and the last node Vy that belongs to cy'. Basically if we take a cluster and break into separate clusters, then we will have a minimum distance that we will have already processed internally between the two.
Divide and Conquer	
Algorithm	
Runtime	Merge sort: T(n) = T(n/2)+T(n/2)+cn $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + cn$ $T(n) = 2T\left(\frac{n}{2}\right) + cn$ $\rightarrow 2\left[2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right] + cn$ $\rightarrow 2^2T\left(\frac{n}{2^2}\right) + 2cn$ $\rightarrow 2^3T\left(\frac{n}{2^3}\right) + 3cn$ ... $\rightarrow 2^iT\left(\frac{n}{2^i}\right) + icn$ $Let\ i = logn$ $T(n) = 2^{logn}T\left(\frac{n}{2^{logn}}\right) = logn \bullet cn$ $\rightarrow O(nlogn)$
	Dynamic Programming

Algorithm	Bellman-Ford algorithm:
	Initialize distance from of all vertices to be infinity, except the distance to the source to be 0.
	For (int i = 1; i <= V-1; i++)
	{
	for (int j = 0; j < E; j++)
	{
	int u = graph->edge[j].src;
	int v = graph->edge[j].dest;
	int weight = graph->edge[j].weight;
	if (dist[u] != infinity && dist[u] + weight < dist[v])
	dist[v] = dist[u] + weight;
	}
	}

Runtime	For any vertex k, it takes O(V) to get the minimum paths through neighbors of k. So O(V*E), in worst case, E=V^2
	ST flow problem, max-flow, min-cut
Algorithm	Find a path from S to T by BFS and increase f(e) by one Create an augmented path for each edge that has f(e) != 0, initialize or update c'(e) = f(e) If there is a path that goes through an augmented edge, decrement the flow of the original edge and decrement the capacity of the augmented path Repeat the above process until there are no more augmented paths from S to T

Runtime	The number of augmented edges is at most E and thus increase the overall number of edges to 2E. The runtime is the same as the efficiency of the path*number of flows because for every flow, it takes O(E) to find a path by BFS. Therefore, O(2E*f)= O(E*f) where f is the max-flow. This is not polynomial because f is not part of the input parameter. It is pseudi-polynomial like the knapsack problem.
Related problems	Create a bi-partite/ multi-partite graph and create a super node S as source of the first column, create a super node T as sink of the last column. The capacities of the edges in-between depend on the capacity of the vertices (eg. phone stations can connect to n phones: c(station-T)=n; each phone can connect to one phone station: c(phone-station)=1)
Min-cut	F is a max-flow in network G (if have not cleaned up the augmented graph, the actual flow of any edge is f(e)-f'(e) to eliminate all augmented edges.) The residual network G' can be calculated by r(e) = c(e)-f(e) and contains no augmented path from S to T ➔ edges are saturated F = c(S, T) for some cut (S, t) of G ➔ the max-flow equals to the capacity of the cut
Proof	For any cut that separates the network G into two parts, there is no way to send more flows than the sum of capacity on the cut, note it is possible to have parallel paths on the cut: max-flow <= min(cut capacity); To prove that OPT(max-flow) = min(cut capacity) we have the following equivalent statements: 1. f is the max-flow in G 2. the augmented network has no augmented path in a max-flow graph 3. f id equal to capacity of some cut of ST, f=c(S, T) To prove 3➔1: max-flow <= min(cut capacity) To prove 1➔2: by contradiction, if there was an augmented path, I would increase the max-flow by one To prove 2➔3: For a max-flow graph, we remove all saturated edges (i.e.: c(e)=f(e)) Every edge that connects to S directly or indirectly goes to the group S, every other edges gO to the group T, note it is possible for group T to contain many separated edges Then we can see that: cut=flow
Improve-ment of runtime	Looking back at the runtime of the Ford-Fulkerson algorithm we recall it being $O(f*E)$. But we can observe the following. Instead of continuously sending flow of 1 through each path, but rather find the bottleneck in the path and sending that much flow – same as saying, send the minimum of the capacities in the path, this would make the algorithm run in $O(E)$. However, in many cases, it is possible to continuously find a path with very low capacity, which would mean repeated

	iterations thus counteracting this improvement. The best way to deal with this is to send a flow of value that is the first power of 2 less than the maximum flow in the network. Find the remaining capacity, and repeat until every single edge is saturated. This is like a top down approach from a maximum capacity to the bottleneck value, thus giving us the final max flow. Thus, the algorithm takes $O(E^*\log C)$ where C is the maximum capacity edge in the network. Note that this is still pseudo-polynomial.
NP-complete, NP-hard	
NP-hard	Traveling salesman problem Max independent set, clique problem (clique problem in G is an independent set in the complement graph of G and vice versa)
NP-complete	Boolean satisfiability problem (SAT)
Transfor-mation	Y <=p X means Y is transformable to X in polynomial time 1. take input of Y and transform it to the input of X 2. solve X 3. take the output of X and transform it to an output that is recognizable by Y - if X can be solved in polynomial time, then Y can be solved in polynomial time This is an **Upper Bound Transformation**. (Proof by construction) - If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time (because if X was solvable in polynomial, then we would be able to solve Y, and so would have a contradiction). This is a **Lower Bound Transformation**. - if X cannot be solved in polynomial time, we **cannot** conclude about Y, there might be other ways that Y can be solved - it is possible that X is transformable from Y and Y is transformable from X , but not always holds. If we can prove both directions, then they are equivalent, and if one is NP-complete, the other one also NP-complete
	To show that a problem is NP-complete: 1. it takes polynomial time to verify the problem 2. reduction/transformation, i.e.: complement graph, addition of ST, set to vertex and element to edges
Max clique	Find the subgraphs of a graph such that for every node in the graph, it is connected to every other node (complete) Brutal force: find all subgraphs of a graph, test each one to see if complete, keep track of the largest
SAT problem	Reduce to max clique: K = number of clauses, each clause has x number of elements. For elements in the same clique, there is no edge For every element x, it is not connected to its conjugate. i.e.: x1=true, x1'=false, x1 and x1' are not connected This reduction is polynomial time. If the graph has a clique of size k, then SAT returns true.

	Proof: if x1=true, x2=true,… xn=true, then x1, x2,… , xn are connected if they are in different clauses, then they form a clique of size k.

Vertex cover	Vertex cover: set of vertices that covers all edges
	Vertex cover to clique
	Vertex cover graph is the complement graph of max clique:
	Graph G has clique of size k iff G'=conjugate(G) has vertex cover of size |V|-k
	For (u, v) in G, (u, v) is not in G'
	If (u, v) is in G,' then at least u or v does not belong to clique V' in G because every pair in V' is connected by and edge in G
	➔ at least one of u or v is in V-V'
	➔ edge(u, v) is covered by V-V'
	➔ continue, we get a size of |V|-k

Maximum independent set	MIS: no edges between each pair of vertices MIS to clique Having no edges in a set of vertices is the same as having all edges in the complement graph