Graph Representation
Adjacency matrix or adjacency list per vertex
Dense: $E \approx V^2$
Sparse: $E \approx V$

Searching
Depth first: from any node, investigate the whole sub-tree before others
Breath first: from any node we visit all adjacent nodes in turn before going deeper

Topological Sort (linearize)
Output vertices in an order such that no edge from any vertex v to a vertex that was output before v
Resolves dependencies
Solve in O(V) using depth first search: output vertices in reverse order of finishing time, colour to avoid cycles
Proof by considering colour

Minimal Spanning Tree
A smallest-weighted subset of the edges that connects all nodes in the graph
Must be a tree if cycles have weights >= 0

Generic Algorithm
Grow a subgraph A by iteratively adding a safe edge of the full graph (an edge that ensure the subgraph remains a subset of some MST)
A *cut* (S, V–S) is a partition of nodes into two sets.
Given a subgraph A of G, a cut of G *respects* A iff no edge of A goes across cut
Theorem: given a graph G and a subgraph A that is a subset of the MST of G, for any cut that respects A the lightest edge of G that goes across the cut is safe for A (proof by considering that there must be some other edge going across the edge other than the lightest one, show that edge must weigh the same as the lightest or a contradiction occurs)

Prims Algorithm
Force A to be a tree at each stage: add the shortest edge that joins a new vertex to the tree
Implement with priority queue, priority = distance from connected subset, O(heapify+V*(extract min)+E*(change key))

Kruskal's Algorithm
Allow A to be a forest during execution: add the shortest edge that does not add a cycle to A
Implement with disjoint set, gives O(V*(make set)+(sort E edges)+E*((find set)+union)

Single Source Shortest Path
δ(s,v) = shortest possible path from s to v
d[v] = working shortest path from s to v

Generic Algorithm
Initially d[v] = ∞ for all v except s, since d[s] = 0
At end, d[v] = δ(s,v)
Relaxation: given an edge (u, v) of weight w set d[v] = min(d[v], d[u] + w)

Bellman-Ford Algorithm
Works even in the presence of –VE edge weights, reports –VE cycles
Iteratively relax all edges V times: O(VE). Add O(E) post-processing phase: if a relaxation occurs here then a –VE cycle is present
Proof by considering that a shortest path with more than V edges in it must contain a cycle, and after i iterations d[u] is the length of shortest path with at most i edges

Dijkstra's Algorithm
Cannot deal with –VE edge weights or report cycles
Maintain a set S of vertices to which shortest paths have been discovered, at each stage add the vertex $u \notin S$ with smallest d[u] until V = S
Implement with a priority queue, u priority = d[u].
Gives O(V*(extract min)+E*(change key))

All-Pairs Short Path Matrix Method

Either run single source algorithm on all pairs or:
Let $L^{(m)}$ by the matrix of shortest paths that contain no more than m edges, then W (adjacency) = $L^{(1)}$

$$L_{i,j}^{(n+1)} = \min(L_{i,j}^{n}, \min_{k=1 \to N}(L_{i,k}^{n} + w_{k,j}))$$

$$L_{i,j}^{(n+1)} = \min_{k=1 \to N}(L_{i,k}^{n} + w_{k,j})$$

This must converge to L after N-1 steps, due to cycling, so:

$$L^{N-1} = L^{N} = \ldots = L^{\infty}$$

Can map to matrix notation:

$$L^{n+1} = L^{n}.W$$

Now we can repeatedly "square" L to discover the first L^m where m >= n-1: $O(V^3 \lg V)$

Floyd-Warshall

Implementation not lectured, has costs of $O(V^3)$

Johnson's Algorithm

Extension of Dijkstra's algorithm that allows for negative weight edges
Adds a new node s with zero weight edges from it to all other nodes, and runs Bellman-Ford to check for –VE cycles and find h(v) = δ(s,v). Change weights such that: w'(u,v) = w(u,v) + h(u) – h(v) (now any path through the graph will be weighted by the same amount that gets rid of –VE edges since w'(u, v) + w'(v, w) = w(u, v) + h(u) + w(v, w) – h(w) and clearly h(u) + w(u,v) >= h(v)).
Then run Dijkstra for each node in the graph to find paths, hence has O(V^2*(extract min)+V*E*(change key))

Maximum Flow

Determine the maximum flow between a source and sink (taking weights as capacities)

Generic Algorithm

Flow network is a graph with a source, sink, *capacity function* c : (u, v) □ N_0 and *flow* f: (u,v) □ Z is such that f(u,v) <= c(u,v), f(u,v) = -f(v,u) and flow in = flow out for all vertex
Residual capacity c_f(u,v) = c(u,v) – f(u,v) is the extra amount of flow that can be pushed through that edge
Residual network is the graph obtained by taking edges with $c_f > 0$ and labelling them with it (it is itself a flow). An *augmenting path* is a path from source to sink in this network, and the *residual capacity* is the maximum amount of flow we can push through it
Compute the residual network, find an augmenting path and push the residual capacity through this path and repeat while possible (note that this may not converge quickly or at all in general)

Edmonds-Karp Algorithm

Choose the augmenting path with the smallest number of edges: $O(VE^2)$

Bipartite Graph Matching

Matching is a collection of edges such that each vertex is included in at most one of the selected edges
Maximal matching is one such that if any edge not in it is added, it stops being matching
Maximum matching is one that contains the largest possible number of edges (poss. many)
We can solve by recasting it as a maximum flow problem: add a edge with unit capacity from a source to everything in one part of the graph and to a sink to everything in the other part: since flows must be integers the result of the previous algo. will be a maximum matching!
Alternating path: path whose edges are alternately matched and unmatched
Augmenting path: alternating path which starts and ends on unmatched vertices
Matching is maximum iff there is no augmenting path in it

Segment Intersection

Given segments p_1p_2 and p_3p_4 determine whether they intersect at any point
Cross product: sign of result tells you what side of the vector the other occurs on

	Check p_1 and p_2 against p_3p_4: give up if they don't lie on opposite sides. Check p_3 and p_4 against p_1p_2: if they lie on different sides then we have an intersection! If a cross product is 0 at any point then must do a check to see if the collinear point lies on segment
Polar Coordinate Sort	Can improve the simple minded angle comparison by replacing it with cross product! Can eliminate some points of convex hulls early by checking if they are within the polygon defined by the points with {min, max} {x, y} coordinate

Graham's Scan	Let p_0 = point with lexically lowest (y, x) value, sort other points by polar angle with p_0:
	1. S.push(p_0)
	2. S.push(p_1)
	3. S.push(p_2)
	4. For i = 3 to M:
	    a. While (angle between S.nextToTop, S.top and p_i makes a non-left turn) do S.pop
	    b. S.push(p_i)
	Note that this takes care of the boundary case where two consecutive points are collinear
	Dominated by sort, so O(VlgV)

Jarvis's March	Start from the bottommost leftmost point and choose the point with the minimum polar angle w.r.t. the current point as our next current point until we reach one with topmost y. Then do the same from the bottommost rightmost point and maximum angle, and fix up flat tops / flat bottoms. Has O(Vh) where h is the number of vertices in the convex hull