Selection Problem

Randomized algorithm for kth smallest (Given an array A and k): Pick an arbitrary element from A, call it v. Create 3 lists: L, which contain elements less than v, E, which contain elements equal to v, and G, which contains elements greater than v. Now if k <= |L|, then the kth smallest is in L, so repeat the procedure on L. If k > |L| + |E|, then run the procedure on G and subtract |L| + |E| from k. Otherwise, k smallest is v.
Note: This algorithm works because, on average, after 2 split operations, we will reduce the array to 3/4th's of it's original size (i.e. we get one "good" random guess of v where v is between the 25th and 75th percentiles), a recurrence which is linear.

Graph Vocab

Degree: In an undirected graph, this is the number of edges incident (that touch) a given vertex **Forest**: a set of trees, **Tree**: An edge which belongs to a tree **Back Edge** An edge which goes from descendant to ancestor **Forward Edge** A NON tree edge which goes from ancestor to descendant **Cross Edge** An edge that satisfies none of the above criteria

DFS

Algorithm: for all vertices, initialize time to 0 and visited to false. If not visited, then DFS-VISIT(u): **DFS-VISIT(u)**: mark u as visited, set d(u) to time; set time to time + 1. For all adjacent vertices v of u if v is unvisited, then DFS-VISIT(v); set f(u) to time; set time to time + 1

BFS

Algortihm: Start at a vertex s, whose distance is 0 and enqueue it. For each vertex that isn't s, set it's distance to inf. While the queue is not empty, dequeue a vertex (call it d). For each of d's adjacent vertices, if it's distance is inf, then set its distance to dist(d) + 1, add it d to it's predecessor array, and enqueue it. Takes linear time (O(|V| + |E|)).

Top Sort

Algortihm: Run DFS to compute finishing times. If there's a back edge, then stop (cyclic graph cannot be top sorted). Otherwise, place the vertices on a horizontal line in decreasing order of finishing times. Then all of the edges between the listed vertices go from left to right.

Strongly Connected Components

Defined as a maximal subgraph where any vertex can be reached from any vertex.
Algorithm: Run DFS. Then, run DFS on the transpose of the graph (i.e. flip the directions) starting from the vertex with the maximal finishing time. The resulting forest is the set of strongly connected components.
The above works based on the property of a forefather vertex. A vertex v is said to be the forefather of a vertex u (denoted ϕ(u)) if and only if v can be reached from u AND v's DFS finishing time is maximal. It can be shown that ϕ(u) is an ancestor of u, which implies that we can reach u from ϕ(u). This means that the two vertices belong to the same SCC, so the algorithm follows: we check to see which vertices we can reach from an initial vertex and then which vertices we can reach the other way.

Kruskal's

Algorithm:
We start with a Graph defined by a set of edges E and vertices V. We will build the MST in a set A (initially empty).
for all v $\in$ V do Make-Set(v)
Sort edges in E by ascending weight

for all (u, v) ∈ E, in order of ascending weight do the following:
    if Find-Set(u) ≠ Find-Set(v) then set A to Union(u, v).

Details on Union Find

The idea of Union Find is to store the set of graph vertices as a set of disjoint linked lists where the head of each list is the name of the set. The set is initialized via Make-Set, which creates a 1 element linked list representing each vertex in the graph. It has two operations:
Find: Returns a pointer to the name of the set (i.e. the head of the linked list)
Union(u,v): Combines the two disjoint sets which contain the vertices u and v respectively. Note that the weighted union rule applies: we always append the smaller list onto the longer
The efficiency of the algorithm is $O(|E|log(|E|)$ as the time it takes to sort the edges dominates the union find operations (which takes no more than $O(|V|log|V|)$)

Prim's

The idea of Prim's algorithm is to keep adding the minimal weight edge that connects the tree to a vertex not in the tree.
Note: For a vertex v ∈ V - U, closest(v) is the closest neighbor of v in U.
Algorithm:
For a graph with vertices V and edges E, we build the MST in sets T and U, where T is the tree and is empty initially and where U is the set of vertices in the tree which starts off with an arbitrary vertex v_1.
for all $v \in V - U$ do closest(v) = v_1
while U ≠ V
    Set min = inf
    for all v ∈ V - U, do
        define a new vertex variable, next
        if weight(v, closest(v)) < min, then set min = weight(v, closest(v)) and set next to v
    Add next to U. Add the edge from next to closest(next) to T.
    for all v ∈ V - U, do
        if weight(v, closest(v)) > w(v, next) then set closest(v) = next;
The efficiency is $O(|V|^2)$ as the running time is dominated by the time it takes to update the closest neighbor of each vertex.

Dijsktra's algorithm

Algorithm (Given a graph G with Vertices V, Edges E, a weight function w, and a starting point s):
for all v ∈ V, do d(v) = ∞, π(v) = null
d(s) = 0
S = ∅
while V is NOT EMPTY:
    let u be an edge variable
    do u = Extract-Min Edge
    S = S ⋃ {U}
    for all v ∈ Adjacent(u) do
        if d(v) > d(u) + w(u,v), then d(v) = d(u) + w(u,v) and π(v) := u;

Time Complexity: O((|V| + |E|)log(|V|)) using a binary heap.