$f(n) \leq cg(n) \rightarrow f = O(g)$
$\lim_{n\to\infty} f/g = 0 \rightarrow f = o(g)$

Master Theorem: $T(n) = aT(n/b) + cn^k$, $a \geq 1, b \geq 2$

$$T(n) = \begin{cases} O(n^{\log_b(a)}) & a > b^k \\ O(n^k \log n) & a = b^k \\ O(n^k) & a < b^k \end{cases}$$

If we think of depth first search as using an explicit stack, then the previsit number is assigned when the vertex is first placed on the stack, and the postvisit number is assigned when the vertex is removed from the stack. Note that this implies that the intervals $[preorder(u), postorder(u)] and [preorder(v), postorder(v)]$ are either disjoint, or one contains the other.

For any cycle in G(V,E), consider the vertex assigned the smallest postorder number. Then the edge leaving this vertex in the cycle must be a back edge by Claim 3.1, since it goes from a lower postorder number to a higher postorder number.
If DFS is started at a vertex v, then it will get stuck and restarted precisely when all vertices in the SCC of v, and in all the SCCs that are reachable from the SCC of v, are visited. Consequently, if DFS is started at a vertex of a sink SCC (a SCC that has no edges leaving it in the DAG of SCCs), then it will get stuck after it visits 3-8 precisely the vertices of this SCC.

The vertex with the highest postorder number in DFS (that is, the vertex where the DFS ends) belongs to a source SCC.

The vertex with the smallest postorder number in a DFS does not necessarily belong to a sink SCC

BFS runs, of course, in linear time $O(|E|)$, under the assumption that $|E| \geq |V|$. The reason is that BFS visits each edge exactly once, and does a constant amount of work per edge.

Dijkstra's algorithm runs in $O(|E| \cdot insert + |V| \cdot deletemin)$. Fibonacci heaps are the fastest and lead to a total runtime of $O(V \log V + E)$. Dijkstra's does not work on **negative edges**.

The **cut property** says that we can construct our tree greedily. Our greedy algorithms can simply take the minimum weight edge across two regions not yet connected. Eventually, if we keep acting in this greedy manner, we will arrive at the point where we have a minimum spanning tree.

One final property of Huffman Trees allows us to determine how to build the tree: the two symbols with the smallest frequencies are together at the lowest level of the tree. Otherwise, we could improve the encoding by swapping a more frequently used character at the lowest level up. This tells us how to construct the optimum tree greedily. Take the two symbols with the lowest frequency, delete them from the list of symbols, and replace them with a new meta-character; this new meta-character will lie directly above the two deleted symbols in the tree.