CS521 \ Notes for the Final Exam

Asymptotic Notations:

Notation	Definition	Limit
Big-O	$f(n) = O(g(n)) \Leftrightarrow \exists c, n_0 > 0 \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)$	$\lim_{n\to\infty} \frac{g(n)}{f(n)} > 0$
Small-o	$f(n) = o(g(n)) \Leftrightarrow \exists c, n_0 > 0 \forall n \geq n_0 : o \leq f(n) < c \cdot g(n)$	$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$
Big-Ω	$f(n) = \Omega(g(n)) \Leftrightarrow \exists c, n_0 > 0 \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)$	$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c \geq 0$ (non-negative constant)
Small-ω	$f(n) = \omega(g(n)) \Leftrightarrow \exists c, n_0 > 0 \forall n \geq n_0 : 0 \leq c \cdot g(n) < f(n)$	$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$
Big-Θ	$f(n) = \Theta(g(n)) \Leftrightarrow \exists c_1, c_2, n_0 \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ (some constant)

Notes:

- $f(n) = \Omega(g(n)) \Leftrightarrow g(n) = O(f(n))$
- $f(n) = \omega(g(n)) \Leftrightarrow g(n) = o(f(n))$
- $f(n) = \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \cap \Omega(g(n))$
- $f(n) = O(g(n)) \wedge g(n) = O(f(n)) \Rightarrow f(n) = \Theta(g(n))$
- $f(n) = \Omega(g(n)) \wedge g(n) = \Omega(f(n)) \Rightarrow f(n) = \Theta(g(n))$

Some hierarchies: ($>$ is "o", $=$ is "Θ")

$2^{2^{n+1}} > 2^{2^n} > (n+1)! > n! > e^n > n \cdot 2^n > 2^n > (3/2)^n > (\lg n)^{\lg n} = n^{\lg \lg n} > (\lg n)! > n^3 > n^2 = 4^{\lg n} > n \lg n = \lg(n!)$

$> n = 2^{\lg n} > \left(\sqrt{2}\right)^{\lg n} > 2^{\sqrt{2 \lg n}} > \lg^2 n > \ln n > \sqrt{\lg n} > \ln \ln n > 2^{\lg^* n} > \lg^* \lg n = \lg^* n > \lg \lg^* n > 1 = n^{1/\lg n}$

Divide and Conquer – recurrences:

Break a problem into sub-problems, solve sub problems and merge solutions.

Hanoi-tower: $T(n) = 2T(n-1) + 1 = O(2^n)$

Solving Recurrences:

1. Substitution method:

- Have a guess for $T(n)$
- Assume correctness for all $m < n$
- Substitute T-terms in the recurrence by the assumption, using constants, and derive upper / lower bounds (or both).

Example:

- $T(n) = aT\left(\frac{n}{b}\right) + f(n), \forall m < n : T(m) = O(m \lg m) \Rightarrow$
- To get an upper bound: $T(n) = a \cdot c \cdot \frac{n}{b} \lg \frac{n}{b} + f(n)$ – develop and derive an expression that is **exactly** $\leq c \cdot n \lg n$.

2. Recursion tree:

Build a tree from the levels in the substitution:

- Level 0: $f(n)$
- Level 1: a elements, each costs $f(n/b)$

…

- Level k: a^k elements, each costs $f(n/b^k)$
- Last level: $n^{\lg_b a}$ elements , each costs $\Theta(1)$

$T(n) = aT(n/b) + f(n) = a[aT(n/b^2) + f(n/b)] + f(n) = a^2T(n/b^2) + af(n/b) + f(n) = \cdots$

$= \underbrace{a^{\lg_b n}}_{=\Theta\left(n^{\lg_b a}\right)} \cdot \underbrace{T\left(n/b^{\lg_b n}\right)}_{=T(1)=\Theta(1)} + \sum_{k=0}^{\lg_b n - 1} a^k f(n/b^k) = [last\ row] + [all\ other\ rows]$

3. The Master Theorem:

For $a \geq 1, b > 1$, $f(n)$ function over non-negative integers and $T(n)$ the recurrence:

$T(n) = aT(n/b) + f(n)$

1. $\exists \epsilon > 0: f(n) = O\left(n^{\lg_b a - \epsilon}\right) \Rightarrow \boxed{T(n) = \Theta\left(n^{\lg_b a}\right)}$
2. $f(n) = \Theta\left(n^{\lg_b a}\right) \Rightarrow \boxed{T(n) = \Theta\left(n^{\lg_b a} \lg n\right)}$
3. $\exists n_0, \epsilon > 0, c < 1: f(n) = \Omega\left(n^{\lg_b a + \epsilon}\right)$ and $\forall n \geq n_0: af(n/b) \leq c \cdot f(n) \Rightarrow \boxed{T(n) = \Theta\left(f(n)\right)}$

<u>Notes for MT</u>:

The ϵ addition/subtraction denotes **polynomial** difference in running time between $f(n)$ and $n^{\lg_b a}$.

Cases (1) and (2) derive Θ for some relation between a, b.

<u>Simplified Master Theorem</u>:

For $T(1) = d, T(n) = aT(n/b) + cn$:

1. $a < b \Rightarrow T(n) = O(n)$
2. $a = b \Rightarrow T(n) = O(n \lg n)$
3. $a > b \Rightarrow T(n) = O\left(n^{\lg_b a}\right)$

Some recurrences:

- Binary search: $T(n) = T(n/2) + 1 = O(\lg n)$
- Linear search: $T(n) = T(n-1) + 1 = O(n)$
- $T(n) = 4T(n/3) + n \lg n \Rightarrow \Theta\left(n^{\lg_3 4}\right)$ (MT case 1)
- $T(n) = 3T(n/3) + n/\lg n \Rightarrow \Theta(n \lg \lg n)$ (tree)
- $T(n) = 4t(n/2) + n^2\sqrt{n} \Rightarrow \Theta(n^{2.5})$ (MT case 3)
- $T(n) = 3T(n/3 - 2) + n/2 \Rightarrow \Theta(n \lg n)$ (bounds)
- $T(n) = 2T(n/2) + n/\lg n \Rightarrow \Theta(n \lg \lg n)$ (tree)
- $T(n) = T(n/2) + T(n/4) + T(n/8) + n \Rightarrow \Theta(n)$ (bounds)
- $T(n) = T(n-1) + 1/n \Rightarrow \Theta(\lg n)$ (substitution)
- $T(n) = T(n-1) + \lg n \Rightarrow \Theta(n \lg n)$ (bounds)
- $T(n) = T(n-2) + 1/\lg n \Rightarrow \Theta(n/\lg n)$ (bounds)

- $T(n) = \sqrt{n}T\left(\sqrt{n}\right) + n \Rightarrow T(n) = \Theta(n \lg\lg n)$ (tree)

Useful math:

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$
- $a^{\lg_b c} = c^{\lg_b a}$
- Harmonic series: $\sum_{k=1}^{n} \frac{1}{k} = \Theta(\lg n)$
- $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$
- $S_n = \frac{n(a_1+a_n)}{2}$
- $\sum_{k=0}^{n} a \cdot q^k = a\frac{(q^{n+1}-1)}{q-1}$ when $n = \infty$: $\frac{a}{1-q}$ $(|q| < 1)$

Sorting algorithms:

Insertion Sort:

- Each element is pushed as far as possible to the left (sorted) part of the array.
- Running time: $O(n^2)$ worst-case (when sorted in reverse), $\Omega(n)$ best-case (when already sorted).
- Sorts **in-place**, space complexity: $O(n)$.

Merge-sort:

- Divide the sequence into two $n/2$ sequences, apply the algorithm recursively and merge solutions.
- $T(n) = \underbrace{2T(n/2)}_{divide} + \underbrace{O(n)}_{merge} = O(n \lg n)$

Quick Sort:

- **Partition**$(\boldsymbol{A}, \boldsymbol{p}, \boldsymbol{r})$: linear procedure that partitions A in-place around $x = A[r]$ s.t.: $\boxed{\leq x \quad |x| \quad > x}$ and returns the index of x – the pivot.
- Preferable over merge-sort, as it sorts in-place.
- General recurrence: $T(n) = T(k) + T(n-k-1) + \Theta(n)$
- Worst-case: array already sorted, partition at the beginning: $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$
- For any n-proportional division we get $\Theta(n \lg n)$:
  $T(n) = T(n/\alpha) + T\left((\alpha-1)n/\alpha\right) + \Theta(n)$
  For $\alpha = 2$ we get a balanced tree.
- Expected running time: $E[T(n)] = \sum_k \Pr[k - split] \cdot T(n|k - split) = 2/n \cdot \sum_{k=1}^{n-1}\left(T(k) + \Theta(n)\right) = \boxed{\Theta(n \lg n)}$

Heap:

- **Left filled** structure that satisfies **partial order** (node > its children, no particular relation between children).
- For node i: left child at $2i$, right at $2i+1$, parent $\lfloor i/2 \rfloor$
- Height is $\lg n$: the cost of **insertion, extract-max, delete**.
- Every level of height h has $\leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes.

Operations:

- **Insert**: put node x at the end and heapify up (switch with parent(x) until $x <$ parent(x) or x is root.
- **Remove-max**: remove root and put last node of the heap in its place. Heapify it down, each level switching with the greater child of the two until satisfies heap properties.
- **Build-heap**: for node $n/2$ down to 1, heapify down. Amortized analysis – each level k but lowest will be visited at most $k \cdot (n+1)/2^k$. Total: $(n+1)\sum_{k=0}^{\lg(n+1)}(k/2^k)$. The sum part is $O(1) \Rightarrow$ total: $O(n)$.

Heap-sort:

- Build a heap and then run $\sim n$ times: Swap the max with the last element, decrease heap size by 1 and heapify the new root down. Result: the array is sorted.
- Running time: $O(n \lg n)$, sorts in-place.

Selection problems:

The Select algorithm for finding the ith o.s.:

$Select(A, p, q, i)$:

- Divide A into $n/5$ groups of size 5.
- Find the median for each group and store in A'.
- Call $select(A', 1, n/5, n/10)$ to find $x =$ median(A').

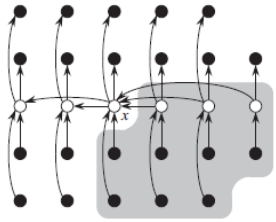
Let $k = partition(A, x)$:

- If $i = k$ return x.
- If $i < k$ return $Select(A, p, k-1, i)$
- If $i > k$ return $Select(A, k, q, i-k)$

Running time: $O(n)$.

Select's recurrence: $T(n) = T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + O(n) = O(n)$ (proof by induction for upper bound – substitution method).

A general recurrence (for constants other than 5):



Calculate the **minimal** number of elements we know for sure are greater than x, the medians' median. Then we take the complement of that number:

For any constant k (the default in the selection algorithm is $k = 5$), where we divide into $\left\lceil \frac{n}{k} \right\rceil$ sets, the number of elements that are **definitely** greater than x:

$$\underbrace{\left\lceil \frac{n}{2} \right\rceil}_{\substack{\#\ elements\\ every\ set\\ contributes}} \cdot \underbrace{\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{k} \right\rceil \right\rceil - 2\right)}_{\#\ contributing\ sets} \geq \left\lceil \frac{n}{2} \right\rceil \cdot \left(\frac{n}{2k} - 2\right) \Rightarrow$$

The recursion over the complement is:

$$T(n) = T\left(\left\lceil \frac{n}{k} \right\rceil\right) + T\left(n - \left\lceil \frac{n}{2} \right\rceil \cdot \left(\frac{n}{2k} - 2\right)\right) + O(n)$$

Then show by substitution that it is bounded / not bounded from above by $c \cdot n$ (would be bounded for any $k \geq 4$). Note that we get $c \geq 0$ if it is bounded above, and $c < 0$ otherwise.

Binary Search Trees:

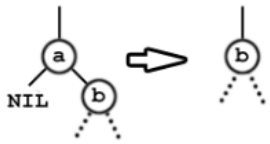
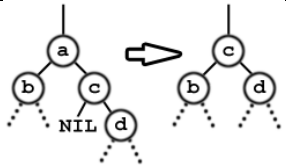
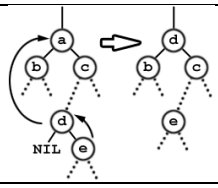
- Trees that satisfy: left(x) $\leq$ key(x) $<$ right(x)
- Unbalanced trees: all operations cost $O(h)$, where h is the height of the tree, between $\lg n$ and n.

Successor of x:

- Left most leaf of x's right subtree.
- If doesn't have one, then the lowest ancestor with left subtree.

Sub-tree rooted at x has $\geq 2^{bh(x)} - 1$ internal nodes.

Deletion:

Case 1,2: No left/right child. Put sole child instead.	Case 3: Successor is right child. Put successor instead.	Case 4: Successor is not direct right child. Swap successor with its sole (right) child, and then swap deleted node with successor.
		

Red-Black Trees:

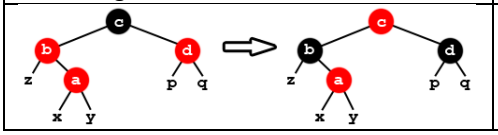
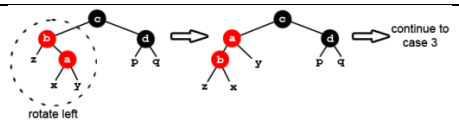
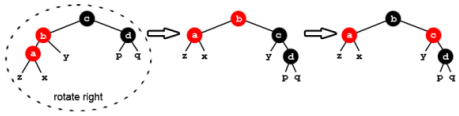
- Every node is either red or black
- Root and leafs (NIL) are black
- If a node is red, its children will be black
- All black paths have same # of blacks (black height)

Height: maximum $2 \lg n$

Operations:

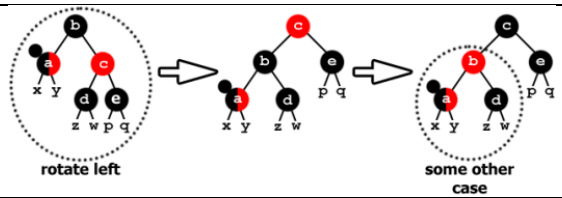
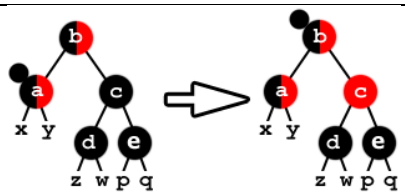
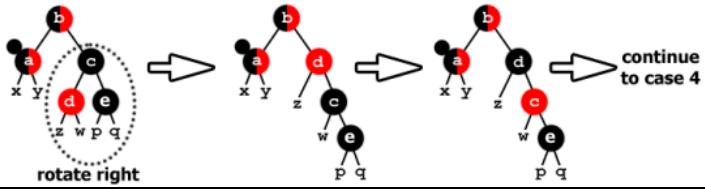
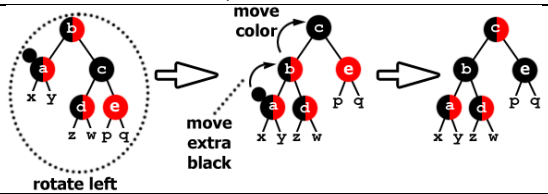
Insertion:

Create new red node with 2 black leafs and put in place. Corrections (a inserted):

Case 1: a's uncle is red: Color father and uncle black, and grandfather red. Problem moved up 2 levels to grandfather.	Case 2: a's uncle is black, a is a right child: Rotate left around father and continue to case 3.	Case 3: a's uncle is black, a is left child: Rotate right around grandfather, switch colors between father and new sibling.
		

Deletion:

Delete as for any BST. If successor was black, leave behind "extra" black and correct:

Case 1: a's sibling is red: Rotate left around father, switch colors between father and grandfather and continue with circled sub-tree to other case.	Case 2: a's sibling and nephews are black: Take blacks from a and c and move problem up.
	
Case 3: a's sibling is black with left red and right black: Rotate right around sibling, switch colors between new sibling and old sibling and continue to case 4.	Case 4: a's sibling is black with right red: Rotate left around father, color grandfather with father's color, color father with extra black, color uncle black.
	

Graph Theory:

Let $G = (V, E), |V| = n, |E| = m$

Some definitions:

- For $v \in V$ the degree $\deg(v) \leq n - 1$ is the number of edges connected to v. $\sum_{v \in V} \deg(v) = 2m$.
- K_n: a complete graph with n edges.
- Connected component: a sub-graph in which there's a path between any 2 nodes. If the graph is directed, it is called strongly connected component.
- Shortest path: the path with smallest sum of edge-weights in it.
- Bipartite graph: if $V = A \cup B$ such that $\forall (u, v) \in E: u \in A, v \in B$ or vice-versa.

  Theorem: a graph is bipartite ⇔ it doesn't have an odd cycle.
- Tree: an acyclic connected graph. For a tree with n nodes: $m = n - 1$.
  - ⇒ Any undirected tree is a bipartite graph (doesn't contain odd, or any, cycles).
  - Rooted-tree: directed tree; Forest: a set of trees
- DAG: directed acyclic graph – graph with no cycles "forward connected". Maximum degree: $\sum_{(i=0)}^{n-1} i$ – dense.

Minimum Spanning Trees (MSTs):

An MST is a tree with minimal sum of edge-weights $w(T) = \sum_{e \in T} w(e)$. There could be more than one spanning tree in a graph.

Greedy MST:

- Sort the set of edges by weights. Takes $O(m \lg m) = O(m \lg n)$.
- Go over every edge in weight-ascending order, and if it doesn't create a cycle, add it to the MST. Use union-set:

Disjoint union sets:

$S = (U, \{S_1, \dots, S_m\}), U = \{v_1, \dots, v_n\}$ where:

- $S_i \subset U$
- $i \neq j \Rightarrow S_i \cap S_j = \emptyset$
- $\cup_i S_i = U$

The data structure will support the following:

- Make-set(v): create a set containing v: $\{v\}$. Cost: $O(1)$.
- Find-set(v): returns the set to which v belongs. Cost: $O(1)$.
- Union(u, v): creates a set which is the union of the set containing v and the set containing u. Cost: $O(\min\{|S_u|, |S_v|\})$ – the cost of changing the pointers of all elements in the smaller set to the root of the larger set.

Every set S_i is represented by some canonical element $u \in S_i$ such that $\forall v \in S_i: find - set(v) = u$.

Back to greedy MST:

We start with $S = \left(V, \{\{v_1\}, \dots, \{v_n\}\}\right)$ and for every $(u, v) \in E$ we check if $find - set(u) = find - set(v)$ (i.e. creates a cycle).

- Number of find operations (w.c.): $2m$ (2 ends of every edge), $O(1)$ each. Total cost: $O(2m)$.
- Number of union operations: for every S_1, S_2, if $S_1 > S_2$ we point $v \in S_2$ to the root of S_1. Every pointer change of v, it becomes belonged to a set at least twice the size of the set it was previously belonged to. Therefore the maximum changes of pointer per node is $\lg n$. Total cost therefore is: $O(n \lg n)$.

That's Kruskal's algorithm.

Kruskal's algorithm for finding MST (greedy):

1. Start with $A = \emptyset$
2. For every $v \in V$ do $make - set(v)$
3. Sort edges in E
4. For each $(u, v) \in E$ in increasing order:
5. If $find - set(u) \neq find - set(v)$:
    a. $A \cup \{(u, v)\} \rightarrow A$
    b. $union(u, v)$
6. Return A

Notes:

- Doesn't work for negative weights.
- Running time: make-set takes $O(n)$; the sorting takes $O(m \lg n)$; find-set and union take a total of $O(n \lg n)$. The total running time: $O\big((m + n) \lg n\big)$.

Prim's Algorithm for finding MST:

Safe cut: A division of V into two sets of nodes $(S, V - S)$ such that no edge of the current iteration of the MST crosses the cut. The edge that would be safe for the MST would be the lightest edge that crosses the cut.

Prim's algorithm starts with an arbitrary node r, and using a priority queue it extracts the minimum weight node u, adds it to A (the MST in construction) and updates the keys of $Adj[u]$.

Prim-MST(G, w, r):

1. Create priority queue $Q \leftarrow V$ (initialized to V).
2. For each $u \in V$: $key[u] = \infty$
3. $key[r] = 0$
4. $p(r) = nil$
5. While $Q \neq \emptyset$:
6. $u \leftarrow extract - \min(Q)$
7. For each $v \in Adj[u]$:
8. If $v \in Q$ and $w(u, v) < key[v]$:
9. $p(v) \leftarrow u$
10. $decrease - key\big(Q, v, w(u, v)\big)$ – update v's key

Running time:

- The priority queue can be implemented using a binary heap.
- Steps 1-4 take $O(n)$ time.
- n extract-min operations will take in total $O(n \lg n)$.
- Worst case there will be $\sum_{u \in V} \deg(u) = 2m$ decrease-keys each $\lg n$, thus $O(m \lg n)$

The total running time is $O\big((m + n) \lg n\big)$ (with better constant than Kruskal's).

Graph Traversal Algorithms:

Breadth First Search (BFS):

- Given $G, s \in V$ calculates the shortest paths from s to any $v \in V$
- Frontier: the current active layer of search. At each iteration it is constructed of nodes that are **at most 1 level apart**.
- Node coloring:
  - White: pre visited nodes.
  - Gray: visited node, still haven't finished exploring – node in the frontier.
  - Black: done exploring this node and edges connected to it.
- For all $u \in V$: $p[u]$ is the parent node of u from on the shortest path from s; $d[u]$ is the distance of that path. Inverting p pointers gives the BFS tree.
- Edges:
  - Tree edges: the edges of the BFS tree (inverted p pointers).
  - Cross/back edges: all other edges.

  If a back edge is between 2 nodes with different d-value parity, it creates an odd cycle (⇔ the graph is not bipartite).
- Uses a queue for handling the nodes.
- Use example: calculate the diameter of a tree – run BFS on some starting node, and get the farthest node from it, say u. Then run BFS on u and get the maximum distant v from u – u, v are the two diameter edges.

The algorithm:

- Initialize $\forall u \in V$: white color, $d[u] = \infty$, $p[u] = nil$. Initialize source: $d[s] = 0, color[s] = gray$, Queue: $Q = \{s\}$.
- While Q is not empty, extract the top of the queue u, and for any undiscovered (white) neighbor $v \in Adj[u]$:
  - Color it gray and add to Q
  - Set $d[v] = d[u] + 1, p[v] = u$

  When done with u, set its color to black.

Running time:

- The initialization takes $\Theta(n)$.
- Every node is colored 3 times (white, gray and black), and every edge is looked at once from each end.

Total running time: $\Theta(m + n)$.

Depth First Search (DFS):

It is also $\Theta(m + n)$. For every node we hold:

- Color (as before) and parent $p[u]$.
- $d[u]$ is the time of discovery, $f[u]$ is the time the processing is finished.

The algorithm:

- Initialize every $u \in V$ to white color and $p[u] = nil$. Initialize the beginning time to 0.
- For each $u \in V$, if u is white – run the recursive DFS-visit method:
  - Color u gray, increase time by 1 and update $d[u]$ to the new time.
  - Run DFS-visit recursively on every white $v \in Adj[u]$
  - Color u black, increase time by 1 and update $f[u]$ to the new time.

DFS creates a forest of DFS trees. Types of edges:

- Tree edges: edges that are part of a DFS tree.
- Back edges: (u, v) where v is an ancestor of u in the tree.
- Forward edges: (u, v) where v is a proper descendant of u.
- Cross edges: (u, v) where u, v are not ancestors / descendants of one another (could cross trees in the forest).

Uses:

- Determine if the graph contains cycles: given a directed graph G, it contains cycles ⇔ the DFS forest has a back edge. This is found easily: (u, v) is a back edge ⇔ $f[u] \leq f[v]$

Topological Sort of a DAG:

A linear ordering of V such that $\forall (u, v) \in E$, u appears before v.

The algorithm is similar to DFS:

- Initialize the color of every $v \in V$ to white, and L an empty linked list.
- For every $u \in V$ that is white run the recursive method Top-visit(u):
    - Color u gray and for each white $v \in Adj[u]$ run Top-visit(v).
    - Append u to the front of L.

Strongly connected components:

To find G^{SCC}, the directed graph of the strongly connected components, the algorithm is:

- Call $DFS(G)$ to compute $f[u]$ for all $u \in V$.
- Compute G^T – the graph G where every edge is reversed.
- Call $DFS(G^T)$ over the nodes by $f[u]$ decreasing order.
- Each set of nodes of every tree in the forest calculated for G^T is a strongly connected components, and the G^{SCC} is a DAG.

Running time: $\Theta(n + m)$.

Shortest Paths:

- A minimal path $\pi^*(u, v) = \text{argmin}_{\pi: u \to v} w(\pi(u, v))$
- Notations:
    - $\delta(u, v) = w(\pi^*(u, v))$ – the weight of the shortest path between u and v.
    - $\Delta G = A_{n \times n}$ such that $(a_{ij}) = \delta(v_i, v_j)$
- Metric: $\forall v, u, w \in V$: $\delta(v, v) = 0$, $\delta(u, v) = \delta(v, u)$, $\delta(u, v) \leq \delta(u, w) + \delta(w, v)$
- MST is insufficient to calculate shortest path (consider $v_1 \to \cdots v_n \to v_1$ where $w(e) = 1$ except $w(v_n, v_1) = 2$. $\delta(v_n, v_1) = 2$ but using the MST we will get $n - 1$).
- There could be more than 1 SP.
- Values used in algorithms for finding SPs:
    - $d[u]$: the current known shortest distance of u from s.
    - $\pi[u]$: the parent of u on the currently known SP from s.

Properties:

- Sub-path of an SP is also an SP: If $v_1 \to \cdots v_i \to v_j \to \cdots v_n$ us an SP, then $v_i \to v_j$ is also an SP.
- 2 SPs don't necessarily combine into an SP (triangular inequality).

Relax operation:

$Relax(u, v, w)$:

$if\ d[v] > d[u] + w(u, v)$

$d[v] = d[u] + w(u, v)$

$\pi[v] = u$

Initial values: $d[s] = 0$, for all others $d[u] = \infty$. $\pi[s] = s$, for all others $\pi[u] = nil$. Properties:

- For every $u \in V$: $d[u] \geq \delta(s, u)$. If there's no path from s to u then $d[u] = \delta(s, u) = \infty$.
- If $d[u] = \delta(s, u)$ before relaxation, it stays that way after relaxation.

Single Source Shortest Paths:

Bellman-Ford:

1. Initialize for all $u \in V$: $d[u] = \infty$ and $d[s] = 0$
2. Do the following n times:
3. For every edge $(u, v) \in E$:
4. Relax(u, v, w)
5. For every edge $(u, v) \in E$:
6. If $d[v] > d[u] + w(u, v)$ then output: **No solution!** (or: there's a negative cycle).

Claim:

- After n iterations of relaxing all edges, d-values are δ, and if not – there's a negative cycle in the graph (that derives infinite improvement during relax).
- Every path is at most of size $n - 1$, thus requires at most $n - 1$ relaxation rounds of the edges on its path. Therefore if after n rounds the path still improves, there must be a negative cycle on it.

Running time: $O(mn)$.

Dijkstra's Algorithm:

Assumes all weights are non-negative.

1. Initialize d-values like for BF.
2. Initialize $S = \emptyset$ and a priority queue $Q \leftarrow V$
3. While $Q \neq \emptyset$:
4. $u = extract - \min(Q)$.
5. $S = S \cup \{u\}$
6. For every $v \in Adj[u]$:
7. Relax(u, v, w)

Properties:

- Like Prim's MST, only here we look at the total weight of the path and not just of one edge.
- Running time: n extract-min operations cost $\lg n$ each, and there are a total m relaxations $\lg n$ each. Total: $O\big((m + n) \lg n\big)$.
- Correctness: if u is at the top of the queue then $d[u] = \delta(s, u)$ (for every $u \in S$: $d[u] = \delta(s, u)$). Proof by contradiction: if u is the first node such that $d[u] \neq \delta(s, u)$ on extraction, then there's $s \to \cdots x \to y \to \cdots u$ such that $x \in S, y \in Q$. When x was added to S, $d[x]$ was $\delta(s, x)$ (u is the first with $d \neq \delta$), and (x, y) was relaxed $\Rightarrow d[y] = \delta(x, y)$. Also since y appears on the SP

from s to u before u then $y[d] = \delta(s,y) \leq \delta(s,u) \leq d[u]$. But since u is chosen by extract-min then $d[u] \leq d[y] \Rightarrow d[y] = \delta(s,y) = \delta(s,u) = d[u]$.

DAG SSSPs:

- Initialize all d-values to ∞ except s.
- Use DFS to apply topological-order.
- For each $u \in V$ taken in topological order:
  - For each $v \in adj[u]$: $Relax(u,v,w)$

Properties:

- Running time: $O(DFS + \sum_{u \in V} \deg(u)) = O(n+m)$.
- Because there are no cycles, once we pass a node, its d-value cannot be changed anymore.

All Pairs Shortest Paths:

Basic APSS:

- Assumption: no negative cycles, thus every shortest path has $\leq n-1$ edges.
- Define a weight matrix: $w_{ij} = \begin{cases} 0, & i = j \\ w(i,j), & i \neq j, (i,j) \in E \\ \infty, & (i,j) \notin E \end{cases}$
- Define $d_{ij}^{(m)}$ as the shortest path between i and j with at most m edges. For $m = 0$: $d_{ij}^{(0)} = \begin{cases} 0, & i = j \\ \infty, & i \neq j \end{cases}$
- Recursively define: $d_{ij}^{(m)} = \min_{1 \leq k \leq n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}$
- We run the algorithm n times to achieve $D^{(n)}$ – the SP matrix.
- Like matrix multiplication: $min \rightarrow \Sigma$, $\cdot \rightarrow +$ (i.e. instead of $d_{ij} = \sum_{k=1}^{n} d_{ij} \cdot w_{kj}$ we calculate $d_{ij}^{(m)} = \min_{1 \leq k \leq n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}$).
- Running time:
  - One iteration is $O(n^3)$, for n iterations the total is $O(n^4)$.
  - Improvement: take advantage of associativity, do $\lg n$ calculations: $D^{(2m)} = D^{(m)} \cdot D^{(m)}$. For any $m \geq n$, $D^{(m)}$ doesn't change anymore. Total running time: $\Theta(n^3 \lg n)$.
- Doing an extra round can detect negative cycles, like in BF.

Floyd-Warshall:

- Here at every iteration the path is allowed to use more and more nodes (enumerated between 1 to n): $\emptyset, \{1\}, \{1,2\}, \ldots, \{1, \ldots, n\}$. Not just **how many** nodes, but **which nodes** can be used.
- $D^{(0)} = W$ (no intermediate nodes are allowed between i,j).
- Calculations of $D^{(k)}$:
  - $d_{ij}^{\{1\}} = \min\{ d_{ij}^{\emptyset} = w_{ij}, d_{i1}^{\emptyset} + d_{1j}^{\emptyset} \}$
  - $d_{ij}^{\{1,2\}} = \min\left\{ d_{ij}^{\{1\}}, d_{i2}^{\{1\}} + d_{2j}^{\{1\}} \right\}$
  - $d_{ij}^{\{1,2,\ldots,k\}} = \min\left\{ d_{ij}^{\{1,2,\ldots,k-1\}}, d_{ik}^{\{1,2,\ldots,k-1\}} + d_{kj}^{\{1,2,\ldots,k-1\}} \right\}$
- Running time: each calculation of $D^{(k)}$ does $O(1)$ operations per entry, total of $\Theta(n^2)$. Total running time: $\Theta(n^3)$.
- Cannot speed-up as before (2 instances of k are using the same set of nodes $\Rightarrow$ they are the same).

Transitive Closure:

Given a graph $G = (V, E)$, the graph $G^* = (V, E^*)$ is the transitive closure of G and is defined:

$E^* = \{(i,j) |\ there\ is\ a\ path\ from\ i\ to\ j\ in\ G\}$.

Computing TC by using modified FW, updating t-values:

- Initialization: $t_{ij}^{(0)} = \begin{cases} 0, i \neq j\ or\ (i,j) \notin E \\ 1, i = j\ or\ (i,j) \in E \end{cases}$
- Iteration computation over all i, j: $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$

The running time is $\Theta(n^3)$. This is like the FW algorithm, substituting $\min$ by $\vee$ and $+$ by $\wedge$.

Dynamic Programming:

- Bottom-up optimization technique: solution is based on solutions to sub-problems.
- Sub-problems are overlapping.

Longest Common Subsequences (LCS):

- Given a sequence $X = x_1 x_2 \dots x_m$ we say $Z = z_1 z_2 \dots z_k$ is a subsequence of X if there are indices $i_1, i_2, \dots, i_k$ such that $\forall j: x_{i_j} = z_{i_j}$ and $i_1 < \dots < i_k$.
- Given two sequences $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$ we want to find $Z = z_1 z_2 \dots z_k$ where $Z \subset X$ and $Z \subset Y$, and k is maximal.
- Denote $X_i = x_1 x_2 \dots x_i$ – the prefix of X of size i.
- Denote c_{ij} the **length** of the LCS of X_i, Y_j:

$$c_{ij} = \begin{cases} 0, & i = 0\ or\ j = 0 \\ c_{(i-1)(j-1)} + 1, & i,j > 0\ and\ x_i = y_j \\ \max\{c_{i(j-1)}, c_{(i-1)j}\}, & i,j > 0\ and\ x_i \neq y_j \end{cases}$$

- Filling a matrix of size $(m+1) \times (n+1)$. Cases:
  - First case: initialization of first row and first column (any LCS with an empty string is of length 0).
  - Second case: the last elements of X_i, Y_j match, so the length is $LCS(X_{i-1}, Y_{j-1}) + 1$.
  - Third case: the last elements don't match, so take the maximum out of $LCS(X_{i-1}, Y_j), LCS(X_i, Y_{j-1})$.
- The total running time is: $\Theta(mn)$. The result is c_{mn}.

Reductions and NP-Completeness:

Definitions:

- Problem: any problem P is defined by a language L_P over an alphabet Σ (usually $\{0,1\}$). $L_P \subseteq \Sigma^*$, where Σ^* is the set of all possible strings over alphabet Σ. For any input $x \in \Sigma^*$, the decision problem is: is $x \in L_P$?
- Polynomial-time languages: we say a language L is poly-time decidable, i.e. $L \in P$, if there's a poly-time algorithm that decides for any $x \in \{0,1\}^*$ whether $x \in L$. Formally: $L \in P \Leftrightarrow \exists A_L . T_{A_L} = n^c$ where n is the input length and c is some constant.
  The computational model: random access memory and a Turing machine.
- Decision problem: check if an input is in L.
- Optimization problem: find minimum / maximum / etc.
- Verification problem: given an input $x \in \{0,1\}^*$ and a certificate/witness w with length polynomial to the size of x ($|w| = |x|^c$ for some constant c), verifies whether $x \in L$ using w.

- Class NP: class of polynomial-time verifiable languages. Alternatively: problems that can be poly-time decided given an access to a non-deterministic Turing machine (i.e. polynomial access to an oracle).

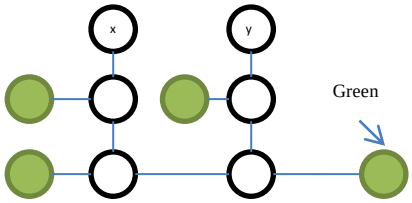
  To show $A \in NP$ give a verifying algorithm and show it is correct and polynomial in the input size.
- Class coNP: $L \in coNP \Leftrightarrow \bar{L} \in NP$, where $\bar{L} = \{x \in \{0,1\}^* | \, x \notin L\}$.
- Reduction: let A, B be two languages. We say A is poly-reducible to B, denoted $A \leq_P B$ iff there exists a polynomial-time reduction R such that $\forall x: x \in A \Leftrightarrow R(x) \in B$. If $A \leq_P B$:
  - B is at least as hard as A – otherwise we can reduce x to $R(x)$ and solve $B\big(R(x)\big)$ more easily than solving $A(x)$.
  - If B is easy then A is easy.

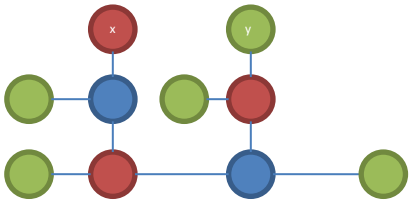
  To prove NP-hardness of A take some problem $P \in NP - hard$ and show: $P \leq A$. E.g.: $CNF - SAT \leq_P A$.

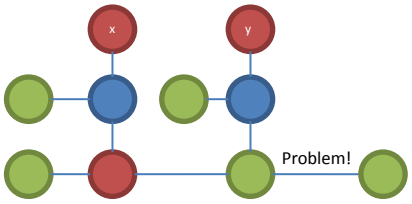
  Reducibility is transitive, and: $A \leq_P B \wedge B \leq_P A \Rightarrow A \equiv_P B$.
- Class NP-Hard: class of all problems H such that $\forall L \in NP: L \leq_P H$.
- Class NP-Complete: $NP \cap NP - hard$

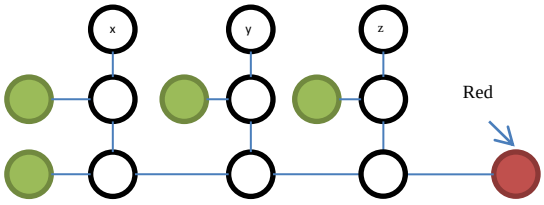
Problems:

Problem	Properties	
CNF-SAT	Definition	All formulas ϕ over variables $\{x_1, \dots, x_n, \overline{x_1}, \dots, \overline{x_n}\}$ in conjunctive normal form: $\wedge C_i$ where $C_i = \vee b_j$ ($b_j \in \{x_1, \dots, \overline{x_n}\}$ that have a satisfying assignment for $x_1, \dots, x_n$.
	NP	Given an assignment, assign it and check that all clauses are satisfied. polynomial
	NPC	Cook's theorem
Clique	Definition	$Clique = \{< G, k > \mid G = (V, E)$ is an undirected graph, $k \in \mathbb{N}, G$ contains a clique of size $\geq k\}$
	NP	The witness: the set of vertices $\{v_1, \dots, v_k\}$ that form a clique of size k. Verification is polynomial: check that all pairs $< v_i, v_j >$ of the set are connected ($\big(v_i, v_j\big) \in E$).
	NPC	$CNF - SAT \leq_P Clique$: Create a graph G: • Create a node for each literal in every clause in ϕ. • For every 2 nodes that are not in the same clause or complements of one another, create an edge. k is the number of clauses. If $\phi \in CNF - SAT$ then there's a clique of size k in G: the one constructed from the vertices that represent one literal that satisfies a clause for all k clauses. Same the other way around.
IS (Independent Set)	Definition	$IS = \{< G, k > \mid$ exists $U \subseteq V, \lvert U\rvert \geq k, \forall u, v \in U: (u, v) \notin E\}$
	NP	The witness: a set $U \subseteq V$ of size k. Verify that for any pair $u, v \in U: (u, v) \notin E$.
	NPC	$Clique \leq_P IS$: For $< G, k >$ an input for Clique create the input $< \bar{G}, k >$ for IS, where $\bar{V} = V, \bar{E} = V \times V - E$. G contains a clique of size $k \Leftrightarrow \bar{G}$ contains an IS of size k.
VC (Vertex Cover)	Definition	$VC = \{< G, k > \mid$ exists $U \subseteq V, \lvert U\rvert \leq k$ such that $\forall (u, v) \in E: u \in U \vee v \in U\}$
	NP	The witness: a set $U \subseteq V$ of size k. Verify that $\forall (u, v) \in E: u \in U \vee v \in U$
	NPC	$IS \leq_P VC$: For $< G, k >$ an input for IS create $< G, \lvert V\rvert - k >$ for VC.
1-CNF-SAT	Definition	All satisfyable CNF formulas with 1 literals per clause: $\phi(x_1, \dots, x_n, \overline{x_1}, \dots, \overline{x_n}) = \wedge x_j$
	P	Assign all variables a satisfying assignment. If encountered a conflict ($x_i = T$ & $\overline{x_i} = T$), the input is not satisfyable.
2-CNF-SAT	Definition	Clauses are of the form $(l_1 \vee l_2)$
	P	$(l_1 \vee l_2)$ is satisfyable iff $\big(\overline{l_1} \Rightarrow l_2\big) \vee \big(\overline{l_2} \Rightarrow l_1\big)$. Build a graph where: • There's a node for every literal. • There's a directed edge for every $\Rightarrow$ We can say ϕ is NOT satisfyable $\Leftrightarrow$ the graph contains a cycle with some variable and its complement. To do that we build G^{SCC} and check every SCC in it. To find a satisfying assignment: apply topological sort (it's a DAG), satisfy the last component and go backwards, satisfying the $\Rightarrow$'s.

<table>
<tr><td>3-CNF-SAT</td><td>Definition</td><td>$\phi(x_1,\dots,x_n,\overline{x_1},\dots,\overline{x_n}) = \wedge_{i=1}^{N} C_N$, where the clauses $C_k = (l_1 \vee l_2 \vee l_3)$ where $l_j \in \{x_1,\dots,x_n,\overline{x_1},\dots,\overline{x_n}\}$
All such ϕ that are satisfyable.</td></tr>
<tr><td></td><td>NP</td><td>Given an assignment, assign it and verify the formula is satisfied.</td></tr>
<tr><td></td><td>NPC</td><td>$CNF-SAT \leq_P 3CNF-SAT$:
Input: $B(x_1,\dots,x_n,\overline{x_1},\dots,\overline{x_n}) = C_1 \wedge \dots \wedge C_i \wedge \dots \wedge C_M$, where $C_i = (l_1 \vee \dots \vee l_k)$
Output: $\phi_B = K_1 \wedge \dots \wedge K_j \wedge \dots \wedge K_N$, where $K_j = (y_1 \vee y_2 \vee y_3)$
To create it we use the mapping λ that maps clauses C_i to $3CNF$ as follows:
$\lambda(C) = (l_1 \vee l_2 \vee x_1) \wedge (\overline{x_1} \vee l_3 \vee x_2) \wedge (\overline{x_2} \vee l_4 \vee x_3) \wedge \dots \wedge (\overline{x_{m-3}} \vee l_{m-1} \vee l_m)$, where $x_1,\dots,x_{m-3}$ are new variables.
$\Rightarrow$: If C is satisfyable, w.l.o.g. $l_m = T$. Then we can assign $x_i = T$ to satisfy all other clauses (the last clause is satisfied by l_m so it's ok that $\overline{x_{m-3}} = F$).
$\Leftarrow$: If C is not satisfyable, for the first clause x_1 must be T, and that derives that all $x_i = T$, but then in the last clause all literals are false, thus $\lambda(C)$ is not satisfyable.
This is a polynomial time reduction.</td></tr>
<tr><td>2COL</td><td>Definition</td><td>All graphs G that their nodes can be colored in 2 colors such that $\forall (u,v) \in E: col(u) \neq col(v)$</td></tr>
<tr><td></td><td>P</td><td>Check if a graph is bipartite (for instance using BFS). If yes, every level can be colored in one of two colors.</td></tr>
<tr><td>K-COL
$(k \geq 3)$</td><td>Definition</td><td>$< G, C >$ where $G = (V, E)$ is an undirected graph and $C = \{c_1,\dots,c_k\}$ is a set of colors and there exists a map $X: V \to C$ such that $\forall (u,v) \in E: X(u) \neq X(v)$</td></tr>
<tr><td></td><td>NP</td><td>The witness: a coloring map. Verify by checking the colors of every $(u,v) \in E$.</td></tr>
<tr><td></td><td>NPC</td><td>$k-CNF-SAT \leq_P 3COL$:
For an input $\phi \in k-CNF$ we create a graph G and a set of k colors. The graph G:
• Create 3 connected sentinel nodes to enforce 3 colors are required.
• Create nodes for all $x_1,\dots,x_n,\overline{x_1},\dots,\overline{x_n}$
• Create edges between each $x_i, \overline{x_i}$ and connect them all to the BLUE sentinel. That way all $x_i, \overline{x_i}$ are colored either red (false) or green (true).
• For every clause we create a gadget in the graph as follows.
For an even clause, say $(x \vee y)$:
 

Has satisfying assignment $(x = red, y = green)$ Doesn't have satisfying assignment $(x, y = red)$
 

For an odd clause, say $(x \vee y \vee z)$:
 
</td></tr>
</table>