Quick Disclaimer: this is meant to be a reference cheatsheet and **NOT** a comprehensive and thorough explanation of each concept. If there are factual inaccuracies, let me know!

Textbooks will be referenced by author last name. The following textbooks are the ones I drew from:
- Kleinberg, and Tardos. Algorithm Design. Addison Wesley, 2006
- Jeff Erickson. Algorithms. jeffe.cs.illinois.edu/teaching/algorithms/

1. Mini-Topics and Other Notes

For Recurrence Relations: Remember the **BASE CASE**!

BFS/DFS $\in O(m + n)$

2. Greedy

Induction is your best friend!

Exchange Proof of Correctness
1. Assume some non-greedy optimal solution
2. Find the first difference between the greedy and optimal
3. Prove that we can exchange the optimal choice for the greedy choice without making the solution worse. We do not need to prove that its better.

Stays-Ahead Proof of Correctness
1. Establish a notion of time/steps. Define discrete points.
2. Using induction, show that over each time step, the greedy solution can only be the same or better than any other given optimal solution.
3. Use this property proven via induction to prove that the greedy solution is optimal.

3. Divide and Conquer

All Divide and Conquer is either like Binary-Search or MergeSort. Prove via induction.

4. Dynamic Programming

Recommended Reading: **Erickson**, Chapter 3

Below is an excerpt from **Erickson**, Chapter 3.4:

Dynamic programming is *not* about filling in tables. It's about smart recursion!

Dynamic programming algorithms are best developed in two distinct stages.

1. **Formulate the problem recursively.** Write down a recursive formula or algorithm for the whole problem in terms of the answers to smaller subproblems. This is the hard part. A complete recursive formulation has two parts:
   (a) **Specification.** Describe the problem that you want to solve recursively, in coherent and precise English—not *how* to solve that problem, but *what* problem you're trying to solve. Without this specification, it is impossible, even in principle, to determine whether your solution is correct.
   (b) **Solution.** Give a clear recursive formula or algorithm for the whole problem in terms of the answers to smaller instances of *exactly* the same problem.
2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:
   (a) **Identify the subproblems.** What are all the different ways your recursive algorithm can call itself, starting with some initial input? For example, the argument to RecFibo is always an integer between 0 and n.
   (b) **Choose a memoization data structure.** Find a data structure that can store the solution to *every* subproblem you identified in step (a). This is usually *but not always* a multidimensional array.
   (c) **Identify dependencies.** Except for the base cases, every subproblem depends on other subproblems—which ones? Draw a picture of your data structure, pick a generic element, and draw arrows from each of the other elements it depends on. Then formalize your picture.
   (d) **Find a good evaluation order.** Order the subproblems so that each one comes *after* the subproblems it depends on. You should consider the base cases first, then the subproblems that depends only on base cases, and so on, eventually building up to the original top-level problem. The dependencies you identified in the previous step define a partial order over the subproblems; you need to find a linear extension of that partial order. ***Be careful!***
   (e) **Analyze space and running time.** The number of distinct subproblems determines the space complexity of your memoized algorithm. To compute the total running time, add up the running times of all possible subproblems, *assuming deeper recursive calls are already memoized*. You can actually do this immediately after step (a).
   (f) **Write down the algorithm.** You know what order to consider the subproblems, and you know how to solve each subproblem. So do that! If your data structure is an array, this usually means writing a few nested for-loops around your original recurrence, and replacing the recursive calls with array look-ups.

Of course, you have to prove that each of these steps is correct. If your recurrence is wrong, or if you try to build up answers in the wrong order, your algorithm won't work!

Proving correctness for Dynamic Programming algorithms is done best with **Strong Induction**

5. Network Flow

Recommended Reading: **Kleinberg and Tardos**, Chapter 7

Name	Problem Description	Construction
Max Flow	Given a weighted directed graph $G = (E, V)$ where $s, t \in V$, what is the maximum flow of weights from s to t.	Use Ford-Fulkerson (DFS $O(C \cdot \|E\|)$) or Edmonds-Karp (BFS $O(\|E\|^2 \cdot \|V\|)$), where you find augmenting paths and reversing them until exhaustion, and adding weight of augmenting path as a sum. A scaled version of FF exists where you take the max path available that is at least more than a specific power of 2.
Min-Cut	Given a weighted directed graph, partition the nodes into sets A, B. $s \in A, t \in B$. Min-cut is the sum of the weights for all edges that go from partition A to partition B.	Min-Cut = Max Flow. Run a BFS on the residual graph to retrieve the partitions.
Bipartite Matching	A bipartite graph where some nodes on one side connect to some, but not all, nodes on the other partition. No nodes within the same partition connect to each other. We want to see if a feasible matching between partitions is possible.	Attach a source to all nodes in one partition, and a sink to the other. Set each edge weight to 1. Run FF or EK, and use the residual graph to retrieve matchings
Disjoint Paths	Given a directed graph, how many distinct paths exist from start s to end t?	Set all edge weights to 1, then run FF or EK. Use the residual graph to retrieve distinct paths.
Circulation With Demand	Given a graph where each node has a demand d, where demand is negative if its a source, and it is positive if its a sink.	Append a super-source s' and super-sink node t', where the s' connects to v if demand is negative with an edge weight equal to $\|d_v\|$. Every node with a positive demand connects to t' where the edge weight is equal to d_v. Running a max-flow algorithm will reveal if the demand for each node can be met and the max-flow from all sources to all sinks.
Circulation With Demand and Lower Bound	Given a graph where some edges have lower bounds, convert this to a Circulation-With-Demand problem by take every directed edge $e = (u, v)$ with lower bound ℓ and capacity/weight c.	The edge now has capacity $c' = c - \ell$. The new demand of node u, $d'_u := d_u + \ell$, and the new demand for node v, $d'_v := d_v - \ell$. This is now a Circulation-With-Demand Problem and can be solved like such.
Image Segmentation	Given an image E we want to classify each pixel as being in the foreground or background. We have a grid where nodes have a likelihood a_i that it belongs in the foreground and b_i for the background. There is also a seperation penalty between neighboring nodes to "smooth out" the segmentation. That is, for every pixel/grid/node i and j, penalty $p_{ij} \geq 0$.	Create a Network Flow graph where neighboring grids have an edge going both ways each with the same penalty p for the pair. Every grid connects to an added sink node t with weight b_i and the source connects to every node with weight a_i. Note we want to maximize the score: $$q(A, B) = \sum_{i \in A} a_i + \sum_{i \in B} b_i - \sum_{(i,j) \in E \text{ s.t. } \|A \cup \{i,j\}\| = 1} p_{ij}$$ Maximizing q is the same as minimizing q' (Note the Sets!): $$q'(A, B) = \sum_{i \in A} b_i + \sum_{i \in B} a_i + \sum_{(i,j) \in E \text{ s.t. } \|A \cup \{i,j\}\| = 1} p_{ij}$$ Running a Max-Flow algorithm on this construction graph will yield an optimal score and segmentation. To retrieve the sets, use the residual graph/min-cut partition. Every element reachable from s in the residual is in the foreground, background otherwise.
Airline Scheduling	Given a list of flights a limited number of k planes, where each flight has a origin and destination and takeoff time and landing time, is there a feasible deployment of k planes to satisfy all flights. Additionally, planes need a constant maintainance time between flights to prep for the next flight.	1. Every flight i will have a departure node u_i and v_i 2. The Graph will have a source s with demand $-k$ and a sink t with demand k. 3. For every flight i, there is an edge $u_i \to v_i$ with a lower bound of 1 and capacity of 1. 4. For every flight j that begins late enough that the plane can service it after completing flight i, there is an edge $v_i \to u_j$ with lower bound 0 and capacity of 1. 5. Every departure node is connected from the source with lower bound 0 and capacity 1. 6. Every landing node connects to the sink; lower bound 0 and capacity of 1. 7. Connect $s \to t$ with capacity k, so extra planes are discarded.
Project Selection	Given a skill-tree style projects each with some real number profit p_i. Some projects are prerequsites of others. How do we select a set of projects to maximize profit? Note that this encoded in a graph G, where each project is an edge (i, j) where j is a prerequsite to i.	Connect all projects with negative profit to t, and connect all nodes with positive profit from the source s. Then, all edges have a capacity equal to slightly more than the total profit of all projects.

6. Intractability and NP-Completeness

Recommended Reading: **Kleinberg and Tardos**, Chapter 8 ; **Erickson**, Chapter 12

Proof Framework

Adapted from **Kleinberg and Tardos**.

Given a problem X, we do the following to prove its $\mathcal{NP}$-Completeness:

1. Prove that $X \in \mathcal{NP}$–There exists an efficient (polynomial time) certifier.
2. Prove that $X \in \mathcal{NP}$-Hard:
    1. Let problem Y be a known $\mathcal{NP}$-Complete problem.
    2. For any instance s_Y of Y, construct a polynomial time reduction to s_X, an instance of X.
        - Using this reduction, prove that s_Y is a yes instance $\iff s_X$ is a yes instance.
        - You **must** prove that s_X is a yes instance **if and only if** s_Y is a yes instance.

Name	**Description**	**Decision Version**
Independent Set (IS)	Given a Graph, what is a set of vertices such that none of the vertices are adjacent to each other.	Does there exist a k-sized independent set?
Vertex Cover (VC)	Given a graph, is there a set of Vertices $S, \|S\| \leq k$ such that every edge e is incident to at least one vertex in S.	Does there exist a vertex cover with k vertices?
Set Cover (SC)	Given a set U, a collection $S = \{S_1 \subseteq U, ...S_m \subseteq U\}$, is there a union of less than or equal to k sets in S such that it encompasses all elements in U?	Does there exist a set cover of exactly k sets?
Set-Packing (SP)	Given a set U, a collection $S = \{S_1 \subseteq U, ...S_m \subseteq U\}$, is there an intersection of zero elements for more than or equal to k sets in S?	Is there a valid set-packing of k sets?
3SAT	Given a set of clauses $C_1, ..., C_k$, each of length 3, over a set of variables $X = \{x_1, ..., X_n\}$, does there exist a satisfying truth assignment? Example: $$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_4 \vee \neg x_2 \vee \neg x_5)$$	N/A
Circuit Satisfiabiliy (CSAT)	Given an arbitrarily long boolean statement encoded purely by $\vee$, $\wedge$, or $\neg$, is there a satisfying assignment to all variables?	N/A
Hamiltonian Cycle/ Path	1. A Hamiltonian Cycle on a Graph G exists if there is a path that starts and ends at the same node, but only traveses through each other vertex one. 2. A Hamiltonian Path is a Hamiltonian Cycle that does not need to start and end at the same node.	Whether A Hamiltonian Path/Cycle exists.
Traveling Salesman Problem (TSP)	Given a salesman who starts at their home city v_1 and must go through a tour $v_1, ...v_n$ and wants to minimize the distance travelled. This can be encoded into a graph where the distance between city v_i and v_j is $d(v_i, v_j)$. $d(v_i, v_j)$ is not neccesarily equal to $d(v_j, v_i)$.	Is there a tour of length of at most D?
Clique	What is the maximum n such that in graph G, there is a subgraph K_n, where K_n represents a graph of n nodes that are fully adjacent to every other node.	Is there a clique of size n?
3D-Matching	Given disjoint sets X, Y, and Z, each of size n, and given a set $T \subseteq XxYxZ$ of ordered triples, does there exist a set of n triples in T so that each element of $X \cup Y \cup Z$ is contained in exactly one of these triples?	N/A
3-Coloring	Given a graph G, is there a way to use 3 colors to color/label each vertex so that no vertices of the same color are adjacent to each other.	N/A
Subset Sum	Given a set X of positive integers and an integer T, determine whether X has a subset whose elements sum to T.	N/A
0-1 Knapsack	Given elements $X = \{1, ...n\}$, a set of weights W and values V, is there a subset of X such that the sum of the weights is less than the given capacity D and the value is more than some goal B.	This is a decision version of the knap-sack problem because the inputs are binary 1 or 0, for in or not in the knap sack, respectively.
Bin-Packing	Given an array of numbers S, is there a way to create k subsets of of S such that the sum of each subset is less than or equal to some constant B.	Is there a valid bin-packing using k bins?
Partition	Is there a 2-partition of some Set S, such that the sums of the partitions are equal to each other?	N/A

How do I choose the appropriate problem (From Erickson 12.14):

- If the problem asks how to assign bits to objects, or to choose a subset of objects, or to partition objects into two different subsets, try reducing from some version of SAT or PARTITION.
- If the problem asks how to assign labels to objects from a small fixed set, or to partition objects into a small number of subsets, try reducing from *k*COLOR or even 3COLOR.
- If the problem asks to arrange a set of objects in a particular order, try reducing from DIRECTEDHAMCYCLE or DIRECTEDHAMPATH or TRAVELINGSALESMAN.
- If the problem asks to find a *small* subset satisfying some constraints, try reducing from MINVERTEXCOVER.
- If the problem asks to find a *large* subset satisfying some constraints, try reducing from MAXINDSET or MAXCLIQUE or MAX2SAT.
- If the problem asks to partition objects into a large number of small subsets, try reducing from 3PARTITION.
- If the number 3 appears naturally in the problem, try 3SAT or 3COLOR or X3M or 3PARTITION. (No, this is not a joke.)
- If all else fails, try 3SAT or even CIRCUITSAT!

I do not recommend trying to reduce from TETRIS, SUPERMARIOBROS, or TRAINYARD. You really want to choose a starting problem that is as simple as possible, while still capturing *some* feature of your problem that makes it difficult to solve.