Final Exam Cheat Sheet
CS 310: Algorithms

By: Darien Nouri

P vs. NP

- **P:** The set of all search problems that are solvable in a reasonable amount of time (Polynomial time).
- **NP (Nondeterministic Polynomial):** The set of all search problems whose solution can be checked in Polynomial time (includes P) - there might exist search problems whose solutions can not be checked in Polynomial time. A solution may not necessary be found in a reasonable amount of time (2^n and $n!$ algorithms can be in NP). Called "Guessing in P", nondeterministic. Should be called "Guessing in P".
- **NP-Hard:** Any problem in NP can be reduced to this problem in Polynomial time, so it is at least as difficult or "hard" as any NP problem. The Halting Problem is NP hard and also impossible to solve in a finite amount of time, so this idea is not always practically useful for reductions. Most NP-Hard problem are NOT in NP, but those that are, are NP-Complete.
- **NP-Complete:** Problem that is not only as hard as every problem in NP, but is also in NP. (NP-Hard and in NP). Any NP problem can be reduced to one of these in Polynomial time. It is often useful to prove the difficulty of problems. These are the hardest problems in NP and the reason why P = NP can revolutionize things.

$O(f(n))$
on the order of at most $f(n)$
$\Theta(f(n))$
tightly on the order of $f(n)$
$\Omega(f(n))$
on the order of at least $f(n)$

- in NP
  - at most as hard as an NP-complete problem
  - NP-complete
  - exactly as hard as any other NP-complete problems
  - NP-hard
  - at least as hard as an NP-complete problem

Common NP-Complete Problems

NP-complete	Polynomial (P)
3SAT	2SAT, HORN SAT
Traveling Salesman Problem	Minimum Spanning Tree
Longest Path	Shortest Path
3D Matching	Bipartite Matching
Knapsack	Unary Knapsack
Independent Set	Independent Set on trees
Integer Linear Programming	Linear Programming
Rudrata Path	Euler Path
Balanced Cut	Minimum Cut

Satisfiability (SAT)

This is the prototypical NP-Complete problem that everything started from. Say you have some Boolean expressions written using only AND, OR, NOT, variables, and parentheses (Example: $x_1 \wedge x_2 \vee x_3$). The SAT problem is given any one of these expressions, is there some assignment of TRUE and FALSE values to the variables that will make the entire expression TRUE?

3SAT

This is a stricter version of the SAT problem in which the statement is divided into clauses where each clause can have exactly 3 literals. Example: $(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6)$. For these you want to find whether there exists values for $x_1 ... x_6$ such that the boolean evaluates to TRUE.

CircuitSAT

Given a circuit of logic gates with a single output and no loops find there a setting of the inputs that causes the circuit to output 1.

Integer Linear Programming

Solve a problem using linear objective function and linear inequalities, WHILE constraining the values of the variables to integers.

Graph Based Algorithms

- **Topological Sort:** Linearly orders the vertices of a directed acyclic graph respecting directional dependencies.

Undirected Graphs

- **Prim's Algorithm:** Finds a minimum spanning tree for a weighted undirected graph by greedily adding the cheapest connection.
- **Kruskal's Algorithm:** Finds a minimum spanning tree for a weighted undirected graph by adding the least weight edges that do not form cycles.
- **Dijkstra's Algorithm:** Finds the shortest path from a single source to all other vertices in a graph with non-negative edge weights.

Shortest Path Problems

- Single Source Shortest Path
- All Pairs Shortest Paths
- **Dijkstra's Algorithm:** Finds the shortest paths from a single source vertex to all other vertices, capable of handling graphs with non-negative edge weights.
- **Floyd-Warshall Algorithm:** Finds shortest paths between all pairs of vertices in a graph.
- **Bellman-Ford Algorithm:** Finds the shortest paths from a single source vertex to all other vertices, capable of handling graphs with negative weight edges.

Minimum Spanning Tree

- Minimum Spanning Tree
- **Prim's Algorithm:** Finds a minimum spanning tree for a weighted undirected graph by greedily adding the cheapest connection.
- **Kruskal's Algorithm:** Finds a minimum spanning tree for a weighted undirected graph by adding the least weight edges that do not form cycles.

Advanced Graph Concepts

- **Kosaraju-Sharir Algorithm:** Finds all strongly connected components (SCCs) in a directed graph.
- **Vertex Cover:** Finds the smallest set of vertices such that every edge is connected to at least one vertex from the set.
- **Independent Set:** Finds the largest set of vertices, no two of which are adjacent.

Topological Sort

- **Purpose:** To linearly order the vertices of a directed acyclic graph such that for every directed edge UV from vertex U to vertex V, U comes before V in the ordering.
- **Idea:**
  1. Use DFS to process each vertex.
  2. On finishing exploration of a vertex, add it to the front of a list.
  3. The resulting list is the topological ordering.
- **Correctness:** The algorithm ensures no vertex is placed before its predecessors, respecting all directional dependencies.
- **Runtime:** $O(|V| + |E|)$, dominated by the DFS used in the process.

Prim's Algorithm

- **Purpose:** To find a minimum spanning tree for a weighted undirected graph.
- **Idea:**
  1. Start with any vertex and construct the spanning tree by adding the cheapest available connection from the tree to a vertex not yet in the tree.
  2. Repeat until all vertices are included.
- **Correctness:** Prim's algorithm is greedy, and at each step, it chooses the least weight edge ensuring the MST property.
- **Runtime:** $O(|E| \log |V|)$ using a binary heap and adjacency list.

Kruskal's Algorithm

- **Purpose:** To find a minimum spanning tree for a weighted undirected graph.
- **Idea:**
  1. Sort all the edges from low weight to high.
  2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge creates a cycle, then discard this edge.
  3. Repeat until there are $|V| - 1$ edges in the spanning tree.
- **Correctness:** Kruskal's algorithm uses a union-find structure to detect cycles and ensures that the least weight edge that does not form a cycle is chosen.
- **Runtime:** $O(|E| \log |E|)$ or $O(|E| \log |V|)$, dominated by the sorting of edges.

Dijkstra's Algorithm

- **Purpose:** To find the shortest path from a single source to all other vertices in a graph with non-negative edge weights.
- **Idea:**
  1. Assign to every vertex a tentative distance value: set it to zero for the initial vertex and to infinity for all other vertices.

2. Set the initial vertex as current. For the current vertex, consider all of its unvisited neighbors and calculate their tentative distances. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.
3. Once all neighbors have been considered, mark the current vertex as visited and not to be checked again.
4. Select the unvisited vertex with the smallest tentative distance, set it as the new "current vertex," and repeat.

- **Correctness:** Dijkstra's algorithm is guaranteed to find the shortest path from the start vertex to any other vertex in the graph, provided all edge weights are non-negative.
- **Runtime:** $O((|V| + |E|) \log |V|)$ using a priority queue.

Kosaraju-Sharir Algorithm

- **Purpose:** To find all strongly connected components (SCCs) in a directed graph.
- **Idea:**
  1. Perform a Depth-First Search (DFS) on the original graph to calculate finishing times for each vertex.
  2. Reverse the directions of all the edges in the graph to obtain the transpose graph.
  3. Perform a DFS on the transpose graph, starting with the vertex with the highest finishing time from the first DFS. Continue the process for each unvisited vertex in decreasing order of their finishing times.
  4. Each tree formed by the DFS in the second step represents a strongly connected component.
- **Correctness:** The algorithm relies on the fact that the transpose of a strongly connected component is also strongly connected. By processing vertices in the order of their finishing times from the original graph, it ensures that each SCC is identified correctly.
- **Runtime:** $O(|V| + |E|)$, as the algorithm essentially involves two DFS traversals of the graph, and thus is linear in terms of the number of vertices and edges.

Vertex Cover

- **Purpose:** To find the smallest set of vertices in a graph such that every edge in the graph is incident to at least one vertex from this set.
- **Idea:**
  1. A vertex cover of a graph is a set of vertices such that every edge of the graph is connected to at least one vertex from the set.
  2. This problem does not have a known efficient polynomial-time solution as it is NP-hard, implying it's computationally intensive to solve exactly for large graphs.
  3. Approximation algorithms or heuristic methods are commonly used in practical applications, including greedy algorithms.
- **Correctness:** The correctness of any solution can be verified by checking that every edge in the graph has at least one endpoint in the vertex cover.
- **Runtime:** Exact solutions often use exponential time algorithms due to the problem's NP-hard nature. Approximate solutions can be polynomial, e.g., $O(|E|)$ for a simple greedy approximation.

Independent Set

- **Purpose:** To find the largest set of vertices in a graph, none of which are adjacent to each other.
- **Idea:**
  1. An independent set in a graph is a set of vertices, no two of which are adjacent. In other words, there is no edge connecting any two vertices within this set.
  2. Like vertex cover, finding the largest independent set is an NP-hard problem, indicating a lack of efficient polynomial-time solutions for large instances.
  3. Various heuristic and approximation methods are used to find reasonable solutions within practical time limits.
- **Correctness:** A solution is correct if none of the vertices in the set share an edge, which can be easily verified by checking all pairs of vertices in the set against the graph's adjacency matrix or list.
- **Runtime:** As with vertex cover, exact algorithms typically run in exponential time. Approximation or heuristic solutions may vary widely in complexity, often depending on the specific approach and graph structure.

Floyd-Warshall Algorithm

- **Purpose:** Find shortest paths between all pairs of vertices in a graph, including paths that pass through intermediate vertices.
- **Method:**
  - Initialize a distance matrix with direct distances between vertices or infinity where no direct edge exists.
  - Iteratively update the matrix, considering each vertex as an intermediate point to potentially shorten the path between vertex pairs.
- When we pick vertex number k as an intermediate vertex, we already have considered vertices $\{0, 1, 2, ..k-1\}$ as intermediate vertices.
- For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
  - k is not an intermediate vertex in shortest path from i to j. We keep the value of $dist[i][j]$ as it is.
  - k is an intermediate vertex in shortest path from i to j. We update the value of $dist[i][j]$ as $dist[i][k] + dist[k][j]$, if $dist[i][j] > dist[i][k] + dist[k][j]$.

```
For k = 0 to n { 1;
For i = 0 to n { 1;
For j = 0 to n { 1
    Distance[i, j] = min(Distance[i, j],
        Distance[i, k] +
        Distance[k, j])
```

  where i = source Node, j = Destination Node, k = Intermediate Node
- **Runtime:** $O(|V|^3)$, where $|V|$ is the number of vertices, reflecting the triple nested loop over vertices.
- **Characteristics:** Efficient for dense graphs or when paths between all vertex pairs are needed; handles negative weights but not negative cycles.

Bellman-Ford Algorithm

- **Purpose:** Find the shortest paths from a single source vertex to all other vertices in a graph, capable of handling graphs with negative weight edges.
- **Method:**
  - Initialize the distance to the source vertex as zero and to all other vertices as infinity.
  - Relax all edges $|V| - 1$ times where $|V|$ is the number of vertices. Each relaxation step involves checking if the current distance to a vertex v can be reduced by taking an edge u to v.
  - Check for negative weight cycles by attempting to relax the edges one more time; if any distance decreases, a negative cycle exists.
- Each iteration over all edges ensures that the algorithm progressively finds shorter paths by building on the shorter paths found in previous iterations.
- For every edge $u \to v$ with weight w, if the distance to u plus w is less than the current distance to v, update the distance to v.
- This process repeats for $|V| - 1$ iterations because in a graph with $|V|$ vertices, the longest path without a cycle has at most $|V| - 1$ edges, ensuring that all shortest paths are correctly calculated without prematurely stopping due to a cycle.

```
for i in range(|V| - 1):
    for u, v, w in edges:
        if distance[u] + w < distance[v]:

            distance[v] = distance[u] + w
```

- After $(|V| - 1)$ relaxations, a final check for any decrease in distances indicates a negative weight cycle:

```
for u, v, w in edges:
    if distance[u] + w < distance[v]:
        print("Graph contains a negative weight cycle")
```

- **Runtime:** $O(|V| * |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges. The algorithm is less efficient for dense graphs compared to algorithms like Floyd-Warshall, which is designed for all-pairs shortest path problems.
- **Characteristics:** Particularly useful for applications that may involve negative weights and need to detect negative cycles, such as in network flow analysis or currency arbitrage scenarios.

Matching and Stability

Overview

Stable Matching solves the problem of finding a stable matching between two sets of elements given individual preferences, with applications in college admissions, job assignments, and more.

Stable Matching

- **Purpose:** To solve the problem of finding a stable matching between two sets of elements given individual preferences.
- **Gale-Shapley Algorithm:**
  - **Idea:** Initialize all participants as unmatched. Let each unmatched participant propose to the most preferred member of the opposite set who has not yet rejected them. Recipients accept the "best" proposal and reject others. Repeat until all are matched.
  - **Correctness:** Guarantees a stable matching with no two participants preferring each other over current partners.
  - **Runtime:** $O(n^2)$, where n is the number of participants in one set.

Overview

Sorting algorithms are a fundamental class of algorithms that put elements of a list in a certain order, with applications in searching, databases, and more. They can be classified into comparison-based (e.g., Quick Sort, Merge Sort) and non-comparison based (e.g., Counting Sort, Radix Sort).

Comparison Based Sorting

- **Quick Sort: Purpose:** To sort by partitioning into subarrays. **Idea:** Choose pivot, partition so elements less than pivot come before and greater after. Recursively apply to subarrays. **Correctness:** Each pivot in final position, each partition independently sorted. **Runtime:** $O(n \log n)$ avg, $O(n^2)$ worst.
- **Merge Sort: Purpose:** To sort by dividing, sorting halves, and merging. **Idea:** Recursively split array until subarrays have one element, then merge subarrays. **Correctness:** Dividing and conquering ensures resulting array is sorted. **Runtime:** $O(n \log n)$ always.
- **Insertion Sort: Purpose:** To efficiently sort small or nearly sorted arrays. **Idea:** Build final sorted array by inserting each element into its correct position. **Correctness:** Each insertion maintains sorted order. **Runtime:** $O(n^2)$ worst case, efficient for small or nearly sorted data.

Decision Trees in Sorting

- **Purpose:** Visualize decision-making in comparison-based sorting where nodes are comparisons, leaves are input permutations.
- **Leaves:** A tree sorting n elements has $\geq n!$ leaves to represent every permutation.
- **Tree Height:** Minimum height to sort n elements is $\Omega(n \log n)$ based on $\log_2(n!)$ comparisons needed.
- **Implications:** Tree height is minimum comparisons needed, so no comparison-based sort is faster than $O(n \log n)$ worst case.

Non-Comparison Based Sorting

- **Counting Sort: Purpose:** To sort objects with small integer keys. **Idea:** Count occurrences of each key, use counts to determine positions. **Correctness:** Counting and computing positions correctly places all elements. **Runtime:** $O(n + k)$, k is range of input data.
- **Radix Sort: Purpose:** To sort integers by processing digits. **Idea:** Sort by each digit using stable sub-sort, from least to most significant. **Correctness:** Each pass sorts by digit, preserving order of previous digits. **Runtime:** $O(d(n + k))$, d digits, n elements, k digit range.

Algorithmic Strategies

Overview

Algorithmic strategies are general approaches to solving problems, such as Divide and Conquer (breaking problem into subproblems) and Dynamic Programming (solving subproblems and combining solutions).

Dynamic Programming

- **Knapsack Problem: Purpose:** To find most valuable subset fitting weight constraint. **Idea:** Use table to store max value for each weight, consider items one by one. **Correctness:** Table entries are optimal subproblem solutions. **Runtime:** $O(nW)$, n items, W max weight.
- **Longest Common Subsequence: Purpose:** To find longest subsequence common to all sequences. **Idea:** 2D table where cell (i, j) is LCS length of substrings up to i and j. **Correctness:** Considering characters one by one builds LCS incrementally. **Runtime:** $O(mn)$, m and n are sequence lengths.
- **Floyd-Warshall:** Described previously for finding all-pairs shortest paths.

Divide and Conquer

- Previously described sorting algorithms (e.g., Merge Sort, Quick Sort) also apply.
- **Karatsuba Algorithm: Purpose:** To multiply large numbers faster than conventional algorithm. **Idea:** Split numbers into halves to perform fewer multiplications on smaller numbers. **Correctness:** Based on math reducing four multiplications to three. **Runtime:** $O(n^{\log_2 3}) \approx O(n^{1.585})$, faster than standard $O(n^2)$ for large n.

Huffman Coding

- **Purpose:** To compress data using variable-length codes. More frequent characters get shorter codes.
- **Idea:** Count character frequencies. Create leaf node for each character, build priority queue on frequencies. Extract two lowest frequency nodes, create combined node as parent with frequency as sum, add back to queue. Repeat until one node left as root. Assign codes: '0' for left, '1' for right.
- **Correctness:** Ensures no code is prefix of another, crucial for unique decoding. Most frequent characters get shortest codes, optimizing size.
- **Runtime:** $O(n \log n)$, n unique characters, using priority queue and assigning codes root to leaves.
- **Average Bit Length:** Sum of frequency of each character times its encoding length, divided by sum of frequencies.

Dynamic Programming (DP)

General Steps to Solving DP Problems

1. **Define Subproblems:** Break main problem into smaller subproblems.
2. **Guess (part of solution):** Guess part of solution, recurse on subproblems based on guess.
3. **Recurrence Relation:** Express solution in terms of solutions to subproblems.
4. **Reconstruct Solution:** Start from base case, use computed subproblem values to build solution.
5. **Optimize:** Use memoization or tabulation to optimize recursive calls, avoid redundancy.

DP Techniques

Amortized Analysis

- **Key Techniques:**
  - **Aggregate Method:** Amortized cost of each operation over n operations is $T(n)/n$.
  - **Accounting Method:** Assigns theoretical cost to each operation. Some charged more or less to compensate, ensuring balanced overall cost.

Coin Change Problem

- **Recurrence Relation:**
  - **Equation:** `opt[k,x] = min(opt[k-1,x], opt[k,x - dk] + 1)`
  - **Base Cases:** `opt[1,x] = x` for smallest denomination covering up to x. `opt[k,0] = 0` for no coins needed for zero amount.

0/1 Knapsack Problem

- **Recurrence Relation:**
  - **Equation:** `opt[k,x] = max(vk + opt[k-1, x - wk], opt[k-1,x])`
  - **Base Cases:** `opt[0,x] = 0` (no items, zero value). `opt[k,0] = 0` (zero capacity, zero value).
  - **Condition:** `opt[k,x] = opt[k-1,x]` if $wk > x$ (item weight exceeds capacity).

Longest Common Subsequence (LCS)

- **Recurrence Relation:**
  - **Equation:** `LCS[i,j] = 1 + LCS[i-1,j-1]` if $s[i] = s[j]$. `LCS[i,j] = max(LCS[i-1,j], LCS[i,j-1])` if $s[i] \neq s[j]$.
  - **Base Cases:** `LCS[i,0] = 0` and `LCS[0,j] = 0` for all i and j.
- **Strongly Connected Components (SCCs):** Maximal set of vertices in directed graph, every pair has directed path both ways. Represents "clusters". Found using Kosaraju's algorithm.
- **Topological Sort:** Linear ordering of DAG vertices, for every directed edge (u, v), u before v. Applications: dependency resolution, scheduling. Algorithms: Kahn's, modified DFS.
- **Minimum Spanning Tree (MST):** Subset of edges forming tree connecting all vertices with minimum total weight. Applications: network design, clustering, approximation algorithms. Algorithms: Prim's, Kruskal's.
- **All Pairs Shortest Paths:** Shortest paths between all vertex pairs in weighted graph. Floyd-Warshall algorithm: considers all vertices as potential intermediates, incrementally improves distances. Works for directed/undirected graphs, negative weights, not negative cycles.
- **Stable Matching:** Matching between two equally sized sets, no pair prefers each other over current partners. Applications: college admissions, hospital residency, online ad allocation. Gale-Shapley algorithm finds stable matching in polynomial time.
- **Karatsuba Algorithm:** Fast multiplication, reduces n-digit numbers to $\leq 3n^{\log_2 3}$ single-digit multiplications. Divide-and-conquer, clever trick reduces recursive calls. Complexity: $O(n^{\log_2 3}) \approx O(n^{1.585})$, better than $O(n^2)$ grade-school algorithm.
- **Master Theorem:** Solves recurrences $T(n) = aT(n/b) + f(n)$, $a \geq 1$, $b > 1$. Analyzes divide-and-conquer time complexity. Three cases based on $f(n)$ and $n^{\log_b a}$ growth rates determine asymptotic behavior.
- **Reduction:** Transforms one problem into another. Shows solution to reduced problem yields solution to original. Proves hardness by reducing known hard problem. Example: 3SAT reduction proves NP-hardness.