Analysis and Design of Advanced Algorithms

Chapters 2–3

Nezih N
Computer Science Department

September 8, 2025

Instructions

This evidence is to be developed individually, although team-based discussions are highly encouraged.

Extra Credit: You may earn up to **10 additional points** by adding well-documented simulations or experiments to support your algorithmic implementations. The quality, correctness, and clarity of these simulations will be evaluated by the professor (LaTeX will always be favored).

Benchmark Requirement: All algorithm implementations must include empirical benchmarking that compares execution time as a function of input size. Include plots to visualize the experimental runtime and to validate the expected asymptotic complexity. Clearly describe your experimental setup (hardware, compiler/interpreter, input generators, number of trials, aggregation method).

String Algorithms

1. Let T denote a large concatenation of a web app server logs (ASCII/UTF-8), and let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be a finite set of short exact signatures (strings), with at least two P_i having nontrivial borders. Assume an alphabet Σ of size ≥ 26. Given $(T, \mathcal{P})$, find all exact occurrences of each $P_i \in \mathcal{P}$ in T and compare the algorithmic approaches below on inputs that include both random-like logs and highly periodic logs.

   1.a. **Knuth–Morris–Pratt (KMP).** Compute all match positions for each P_i in T and report the prefix function (π-array) for two nontrivial patterns.

   1.b. **Z-Algorithm (via $P\#T$).** Use the standard concatenation trick to perform exact matching for each P_i against T.

1.c. **Rabin–Karp (Rolling Hash).** Implement single and double hashing for multi-pattern matching over $\mathcal{P}$; specify base(s), modulus/moduli, and collision handling.

How does applying and comparing string-matching algorithms on different types of security log data demonstrate the practical importance of foundational algorithms in real-world cybersecurity? You may add as many references as you want.

2. Let $A, B \in \{A, C, G, T\}^*$ be two DNA strings (FASTA or plain text). Let $\mathcal{M} = \{M_1, \ldots, M_k\}$ be a small set of exact motifs (length 4–12) to be queried for membership/positions. (i) Identify longest palindromic regions within each of A and B; (ii) compute the longest common substring(s) shared by A and B; and (iii) support exact substring membership queries for motifs in $\mathcal{M}$.

2.a. **Manacher's Algorithm.** Compute the longest palindromic substring of each sequence; provide a brief correctness justification based on radius propagation.

2.b. **Suffix Array with Kasai LCP.** Build a suffix array for $A\#B$ (with a unique sentinel $\#$) and use it to support exact substring membership queries for all $M_j \in \mathcal{M}$.

2.c. **Longest Common Substring ($\mathbf{LCS_{substr}}$).** Compute an $\mathrm{LCS}_{\mathrm{substr}}$ of (A, B) using the suffix array + LCP; additionally, implement one alternative method (either a suffix automaton or an $O(|A||B|)$ DP baseline) for methodological comparison.

How does applying algorithms such illustrate the practical importance of foundational algorithmic techniques in genomics research? You may add references to relevant bioinformatics studies or applications.

Graph Algorithms

Unless specified, assume simple graphs. State whether you treat inputs as directed/undirected, and justify that choice.

1. **Maximum Flow.**
   Investigate and implement a maximum-flow algorithm (e.g., Edmonds–Karp or Dinic). Construct a small capacitated directed graph (6–8 vertices) and compute the max flow *step by step*: show augmenting paths, residual graphs, and final flow values per edge. Provide complexity and a brief comparison between your chosen algorithm and one alternative.

2. **Second Minimum Spanning Tree (2nd MST).**
   Given a connected weighted graph $G = (V, E, w)$ and an MST T, design an algorithm to obtain the *second* MST. Discuss the *replacement edge* strategy (swap one edge of T with an edge outside T that creates a cycle, then drop the heaviest on that cycle) and analyze complexity. Address: "Is it necessary to compute the MST first?" Provide a reasoned answer and a counterexample or justification.

3. **Edge Types in Traversals.**
Explain/prove:

   - For DFS on an *undirected* graph: only tree and back edges appear (no cross/forward).
   - For BFS on an *undirected* graph: edges are tree or cross (no back/forward).
   - For BFS on a *directed* graph: tree, back, and cross edges may appear, but no forward edges.

   Provide minimal illustrative examples and discovery/finish (or level) labelings that justify each claim.

4. **Biconnected Components (Blocks) and Bridges.**
Generate an undirected graph with at least *five* biconnected components; ensure at least one *bridge* (an edge whose removal increases the number of connected components) appears as a singleton block (two vertices, one edge). Apply (step by step) the block/bridge-finding algorithm seen in class (Tarjan-based articulation-point/stack method). Show discovery/low arrays and the component output.

5. **MST via Kruskal and Prim.**
Randomly generate a connected weighted graph with $|V| = 10$ nodes and $|E| = 30$ edges (positive integer weights; specify seed for reproducibility). Compute its MST using *both* Kruskal's and Prim's algorithms. Report the total MST weight and the chosen edges for each method. Compare runtimes over multiple random instances.
NOTA BENE: Dijkstra's algorithm is *not* an MST algorithm; use Prim's for the heap-based greedy.

Can we combine all of this? Yes we can...

1. Design and implement a system that:

   1.a. Accepts logical expressions over up to three propositional variables (defined at the beginning with assigned Boolean values). Supports operators `NOT`, `AND`, `OR`, `XOR` and parentheses.

   1.b. Detects whether the input is in *Polish (prefix)* or *normal (infix)* notation using a string-matching approach (e.g., your Z/KMP-based recognizer simulating regex-like patterns); no external regex engines.

   1.c. Parses the expression into an Abstract Syntax Tree (AST) and evaluates it. Producing full truth tables and shows at least four test cases per notation (valid/invalid forms).

   You may use any graph traversal for AST evaluation (e.g., post-order DFS) or a heap-based construction for operator precedence in infix. Include (i) the AST for two exam-

ples in each notation, and (ii) a short discussion of correctness and complexity (parsing and evaluation).