# Final Exam: Algorithm Analysis - CS/DSA 4413

**Nima Najafian**
**May 1, 2025**
**Question 1: Solve the recurrence relation** $T(n) = 2T(n-1) + 3T(n-2)$ with $T(0) = 1, T(1) = 2$.
**Solution:** $T(n) = \frac{3^{n+1}+(-1)^n}{4}$.
**Steps:**

1. Identify the recurrence as a second-order linear homogeneous recurrence with constant coefficients.
2. Form the characteristic equation: $r^2 - 2r - 3 = 0$.
3. Solve the quadratic: $r = \frac{2 \pm \sqrt{4+12}}{2} = \frac{2 \pm 4}{2}$, yielding roots $r = 3, -1$.
4. Write the general solution: $T(n) = A \cdot 3^n + B \cdot (-1)^n$.
5. Use initial conditions: $T(0) = A + B = 1$, $T(1) = 3A - B = 2$.
6. Solve the system: Add equations to get $4A = 3$, so $A = \frac{3}{4}$. Then, $B = 1 - \frac{3}{4} = \frac{1}{4}$.
7. Simplify: $T(n) = \frac{3}{4} \cdot 3^n + \frac{1}{4} \cdot (-1)^n = \frac{3^{n+1}+(-1)^n}{4}$.
8. Verify: For $n = 2$, compute $T(2) = 2T(1) + 3T(0) = 4 + 3 = 7$, and check $\frac{3^3+1}{4} = \frac{28}{4} = 7$.

**Question 2: Binary Addition and Prefix Computation**

**Problem:** Prove that binary addition can be reduced to prefix computation.
**Solution:** Binary addition can be reduced to prefix computation because each bit in the sum depends on the preceding bits, similar to how prefix sums operate. This dependency enables efficient computation using a structure akin to full adders, creating a cascading effect that allows parallel processing in digital systems.
**Steps to Obtain the Answer:**

1. **Binary Addition:** When summing two binary numbers, each bit of the sum is calculated based on the corresponding bits of the input numbers and any carry from the prior position. For example, consider adding $A = 7$ (0111 in binary) and $B = 4$ (0100 in binary), resulting in 11 (1011 in binary):
   - Bit 0 (LSB): $1 + 0 = 1$, carry $= 0$.
   - Bit 1: $1 + 0 + 0 = 1$, carry $= 0$.
   - Bit 2: $1 + 1 + 0 = 0$, carry $= 1$.
   - Bit 3: $0 + 0 + 1 = 1$, carry $= 0$.

- Result: 1011 (11 in decimal).

2. **Prefix Computation:** This technique, often used in parallel algorithms, involves operations where each output depends on all preceding inputs. In binary addition, each sum bit $S_i$ depends on $A_i$, $B_i$, and the carry from the previous bit, resembling a prefix operation.

3. **Parallel Dependencies:** The addition process can be structured for parallel computation. In a binary adder circuit with full adders, each adder takes inputs $A_i$, $B_i$, and the carry-in from the previous adder, producing a sum bit and a carry-out. This creates a cascading effect, similar to prefix sums.

4. **Binary Addition as Prefix:** The carry propagation in binary addition mirrors prefix computation. For the example above:

   - The carry from each bit influences the next, forming a chain of dependencies.
   - This can be parallelized using generate ($G_i = A_i \wedge B_i$) and propagate ($P_i = A_i \vee B_i$) signals, where carry $C_{i+1} = G_i \vee (P_i \wedge C_i)$.

5. **Efficient Implementation:** By structuring binary addition as a prefix computation, carries can be computed in parallel, as seen in carry-lookahead adders, enhancing performance in digital circuits.

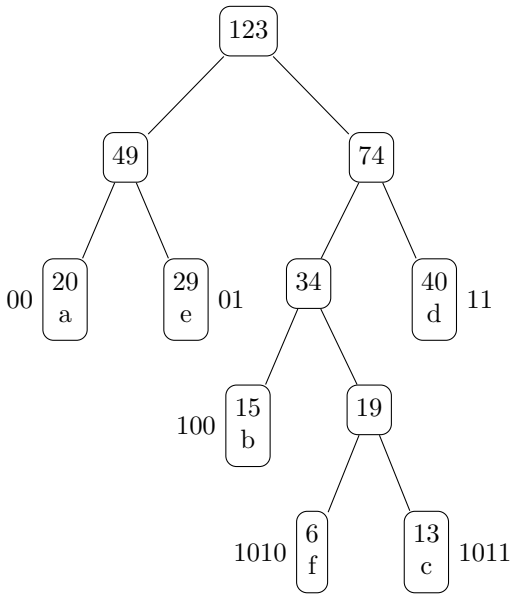
**Question 3: Huffman Coding**

(a) **Define fixed and variable length codes:**
Fixed length codes assign the same number of bits to each symbol (e.g., 3 bits for 6 symbols). Variable length codes assign different bit lengths, typically shorter codes for more frequent symbols, as in Huffman coding.

(b) **Design Huffman code for the given frequencies:**

| Alphabets | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| Occurrences ($\times 10^2$) | 20 | 15 | 13 | 40 | 29 | 6 |
| Huffman Code | 00 | 100 | 1011 | 11 | 01 | 1010 |

**Huffman Tree Diagram:**




(c) **Compression ratio:** Fixed length coding uses 369 bits; Huffman coding uses 299 bits. Compression ratio: $\frac{369-299}{369}(100) \approx 18.9$.

(d) **Encode and decode a five-letter word:** Word: "faced".
Encoded: 1010 00 1011 01 11.
Decoded: "faced".

**Question 4: Iterated Logarithm and Function**

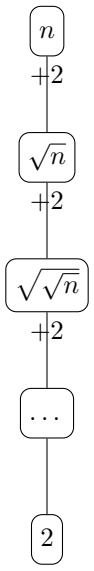
(a) **Define** $\log^* n$ **and find for** $n = 10^{12}$**:** $\log^* n$ is the number of times the base-2 logarithm is applied until the result is $\leq 1$. For $n = 10^{12}$, $\log^* n = 5$.
**Let** $n = 2^k$**,** $f(n) = n/2$**, find** $f^*(n)$**:** $f^*(n) = \log_2 n$.

(b) **Solve** $T(n) = T(\sqrt{n}) + 2$**, where** $n = 2^{2^k}$ **and** $T(2) = 1$**:**

**Solution:**

1. Let $n = 2^{2^k}$, so $\sqrt{n} = 2^{2^{k-1}}$.
2. Define $T_k = T(2^{2^k})$. The recurrence becomes:
$$T_k = T_{k-1} + 2, \quad \text{with base case } T_0 = T(2) = 1.$$
3. Solve by unrolling:
$$T_k = 1 + 2k.$$
4. Since $\log_2 n = 2^k$, we have $k = \log_2 \log_2 n$.
5. Thus, the final solution:
$$\boxed{T(n) = O \log_2 \log_2 n.}$$

**Recursion Tree Diagram:**



1 (base case)

**Question 5: Six-City TSP Approximation**

**Solution:**

For this problem, I tackled both the exact and approximate solutions to the Traveling Salesman Problem (TSP). I started by computing the exact optimal tour using brute force—evaluating all possible permutations of the cities and selecting the tour with the lowest total cost.

To approximate the TSP solution, I implemented the Christofides algorithm manually, applying it as follows:

- Built a minimum spanning tree (MST) of the graph using Kruskal's and Prim's algorithms.
- Identified the vertices with odd degrees within the MST.
- Found a minimum-weight perfect matching among those odd-degree vertices.
- Combined the MST and the matching to form an Eulerian circuit.
- Finally, traversed the Eulerian circuit while skipping repeated vertices (shortcutting) to produce a Hamiltonian circuit—this is the approximate tour.

**Results:**

1. **Optimal tour:**
    - Path: [a, e, f, b, c, d, a]

- Total cost: $\approx 17.954$
- Denoted as $C(H^*) \approx 17.954$.

2. **Approximate tour (Christofides):**
   - Path: [a, e, b, c, f, d, a]
   - Total cost: $\approx 20.428$
   - Denoted as $C(H) \approx 20.428$.

**Verification:**
To confirm the effectiveness of Christofides' algorithm, I checked that the approximate tour's cost satisfies the 2-approximation guarantee. We have:

$$20.428 < 2 \times 17.954,$$

which shows that the approximate solution is less than twice the cost of the optimal tour, as expected:

$$C(H) \leq 2 \times C(H^*).$$

**Conclusion:** This experiment validates the theoretical guarantee of the Christofides algorithm—the approximate tour remains within twice the optimal cost, demonstrating reliable performance for this six-city TSP instance.