Problem 1: Karnaugh Maps

Karnaugh maps exploit geometric patterns in truth tables to visually minimize Boolean functions. Adjacent cells differ by exactly one variable, enabling efficient grouping of *minterms* into simpler *product terms*. In this problem, you'll work with a 5-variable function, exploring both standard minimization and optimization with *don't-care conditions*.

Part (a): Generate the Function

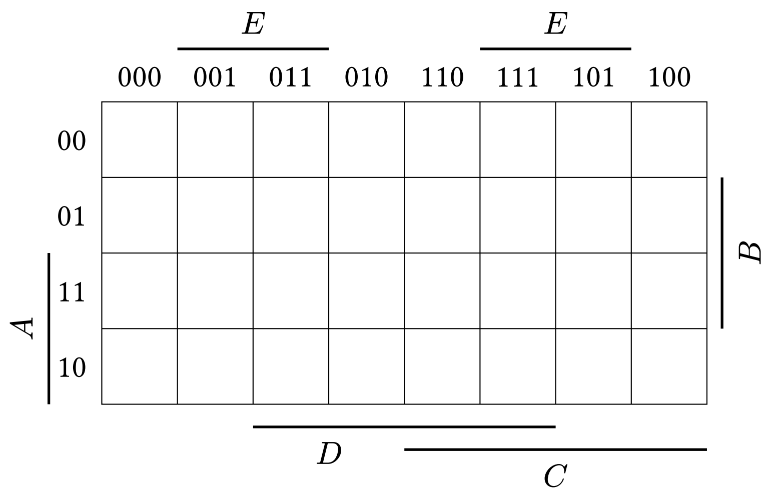
1. Compute[1] the SHA-256 hash h of string $s =$ "DM Fall 2025 HW3" (without quotes).
2. Split h into eight 32-bit blocks: $h = h_0 \parallel h_1 \parallel \ldots \parallel h_7$ where each h_i is 32 bits.
3. XOR all blocks together: $d = h_0 \oplus h_1 \oplus \cdots \oplus h_7$.
4. Apply the mask: $w = d \oplus$ `0x71be8976`, giving the 32-bit result $w = (w_0 w_1 \ldots w_{31})_2$.

The resulting bit string w encodes the 5-variable Boolean function $f^{(5)}$: bit w_0 (MSB) represents $f(0,0,0,0,0)$, and bit w_{31} (LSB) represents $f(1,1,1,1,1)$.

Check: Hash h ends with …`00010101`; the value d (before masking) begins with `0110`…

Part (b): Draw the K-Map

Construct a 5-variable Karnaugh map for your function using the template below:



Part (c): Find Minimal Forms

1. Find minimal DNF and CNF for f using your K-map.
2. Count all *prime implicants*.
3. If inputs where $A \land B \land C = 1$ are *don't-care conditions*, how do the minimal forms change?[2]

Part (d): Analysis and Limitations

1. How many fewer *literals* does your minimal DNF use compared to full minterm expansion?
2. Which is more compact for your function: DNF or CNF? Why?
3. What is the size of a 6-variable K-map? Why do K-maps become impractical beyond 5-6 variables?

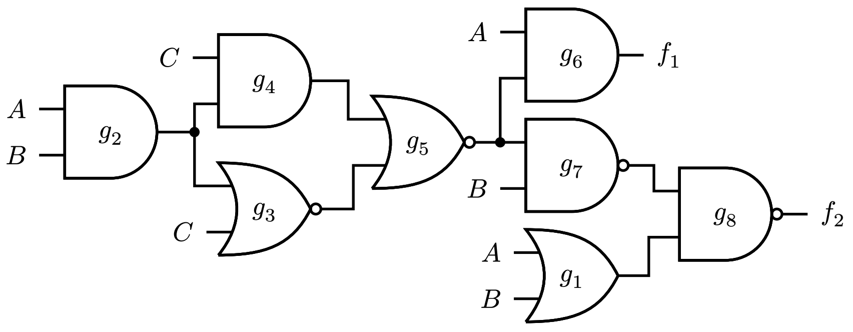
[1] You can use any programming language, tool or online service to compute SHA-256 hashes.

[2] *Don't-care conditions* are input combinations whose output values don't matter, allowing flexibility in minimization. They can be treated as either 0 or 1 to produce smaller expressions.

Problem 2: Circuit Analysis

Real-world digital circuits often contain redundancies from design iterations or automated synthesis. Your task is to reverse-engineer a given circuit to understand its logical behavior and performance characteristics.

Given a combinational circuit with 3 inputs (A, B, C) and 2 outputs (f_1, f_2):



Part (a): Truth Table

Build the truth table mapping $\langle A, B, C \rangle \mapsto \langle f_1, f_2 \rangle$ for all 8 inputs.

Part (b): Boolean Expressions

1. Derive minimal DNF for both f_1 and f_2.
2. Verify against your truth table.
3. Count literals in each expression. Which output is more complex?

Part (c): Optimization

1. Identify all *redundant gates* that can be removed without changing the circuit's functionality.
2. Draw the simplified circuit with redundant gates eliminated.

Part (d): Timing Analysis

The *delay* of a gate is the time between input change and output stabilization. The *critical path* is the longest path from any input to any output, determining the circuit's maximum operating frequency.

1. Compute the maximum delay from inputs to each output, assuming unit delay per gate.[3]
2. Identify the critical path for each output.

Problem 3: Boolean Function Analysis

This problem explores multiple representations of Boolean functions that reveal different properties: K-maps show groupings, DNF/CNF expose clause structure, ANF reveals algebraic degree.

Consider the following four functions[4]:

1. $f_1 = f^{(4)}_{47541}$
2. $f_2 = \sum m(1, 4, 5, 6, 8, 12, 13)$
3. $f_3 = f^{(4)}_{51011} \oplus f^{(4)}_{40389}$
4. $f_4 = A\overline{B}D + \overline{A}\,\overline{C}D + \overline{B}C\overline{D} + A\overline{C}D$

[3]In reality, gate delays vary based on fan-in (number of inputs), fan-out (load), and implementation technology. NAND and NOR gates are typically faster than AND and OR gates.

[4]Notation: $f^{(n)}_k$ is the k-th Boolean function of n variables, where k is the decimal truth table value, which, represented in binary as $(f_0 f_1 \ldots f_{2^n-1})_2$, corresponds to values of f: MSB f_0 is $f(0, \ldots, 0)$, LSB f_{2^n-1} is $f(1, \ldots, 1)$.

Part (a): Karnaugh Maps

Draw and fill the K-map for each function.

Part (b): Prime Implicants

1. Find *all prime implicants* for each function (Blake Canonical Form).
2. Show each prime implicant as a *maximal rectangle* on your K-map.
3. Identify which are *essential* (cover at least one minterm no other prime implicant covers).

Part (c): Minimal DNF and CNF

1. Derive the minimal DNF using the prime implicants from Part (b).
2. Derive the minimal CNF using either dual K-map analysis or algebraic methods.
3. For each function, state whether the DNF or CNF is more *compact* (fewer literals).

Part (d): Algebraic Normal Form

Construct the ANF for each function using any of the methods below:
- *Indeterminate coefficients*: Solve linear system of equations
- *Tabular*: Pascal's triangle-like XOR structure
- *K-map*: Identify XOR patterns geometrically

Requirement: Use each method at least once across the four functions.

Note: WolframAlpha reverses bit order! For $f_{10}^{(2)} = (1010)_2$, query "5th Boolean function of 2 vars", since $\mathrm{rev}(1010_2) = 0101_2 = 5$.

Problem 4: CNF Conversion

CNF is essential for SAT solvers and theorem provers. Direct conversion often causes exponential blow-up, while the Tseitin transformation introduces auxiliary variables to keep complexity linear.

Part (a): Basic Conversions

Convert to CNF:

1. $X \leftrightarrow (A \land B)$
2. $Z \leftrightarrow \bigvee_{i=1}^{n} C_i$
3. $D_1 \oplus D_2 \oplus \cdots \oplus D_n$
4. $\mathrm{majority}(X_1, X_2, X_3)$[5]
5. $R \to \left(S \to \left(T \to \bigwedge_{i=1}^{n} F_i\right)\right)$
6. $M \to \left(H \leftrightarrow \bigvee_{i=1}^{n} D_i\right)$

Part (b): Tseitin Transformation

The Tseitin transformation produces *equisatisfiable* CNF by introducing *auxiliary variables*. The transformed formula has different models, but they naturally correspond to the original's.

Consider $(A \lor B) \land (C \lor (D \land E))$:

1. Apply Tseitin: introduce variables for subformulae, encode each as CNF clauses.
2. Show that: (i) if the original formula is satisfiable, then the CNF has a satisfying assignment, and (ii) if the CNF is satisfiable, then the original formula is satisfiable.
3. Compare sizes (clauses, variables, literals) with direct CNF. When is each method better?[6]

[5] The majority function outputs 1 if more than half (*e.g.*, 2 of 3) of its inputs are 1.

[6] Direct CNF conversion can cause exponential blow-up for nested formulas, while Tseitin transformation keeps size linear but introduces auxiliary variables.

Part (c): Resource Allocation

A system manages five resources $\{R_1, R_2, R_3, R_4, R_5\}$ with constraints:

- Either R_1, or (exclusive) both R_2 and R_3
- If R_1, then R_4 or R_5 (inclusive)
- Not both R_2 and R_4
- At least two of $\{R_1, R_2, R_3\}$

1. Encode constraints as Boolean formulae.
2. Convert to CNF and combine.
3. Find all satisfying assignments.
4. Which resource appears in the most valid configurations?

Problem 5: Functional Completeness

Not all operator sets can express every Boolean function — some are fundamentally limited. Post's criterion systematically determines *functional completeness* by checking preservation of *closed classes*.

Part (a): Apply Post's Criterion

For each system, check Post classes (T_0, T_1, S, M, L)[7] and determine functional completeness:

1. $F_1 = \{\wedge, \vee, \neg\}$
2. $F_2 = \{f_{14}^{(2)}\}$
3. $F_3 = \{\rightarrow, \nrightarrow\}$
4. $F_4 = \{1, \leftrightarrow, \wedge\}$

Part (b): Express Majority

The majority function $\text{majority}(A, B, C)$ outputs 1 when at least two inputs are 1.

For each *complete* basis from Part (a):

1. Express $\text{majority}(A, B, C)$ using only operations from that basis.
2. Draw the corresponding circuit diagram with clearly labeled gates.
3. Count the total number of gates required.
4. Verify your expression by checking the test cases: $(0, 0, 0)$, $(1, 1, 0)$, and $(1, 1, 1)$.

Part (c): NAND Gates

1. Express $\neg A$, $A \wedge B$, and $A \vee B$ using only NAND.
2. Build a NAND-only circuit for $\text{majority}(A, B, C)$. Count gates.
3. Find a complete basis from Part (a) that yields the smallest majority circuit.

Part (d): Custom Basis

1. Design a single 3-input Boolean function that is by itself functionally complete.
2. Prove completeness using Post's criterion.
3. Express $\neg A$ and $A \wedge B$ using only your function.
4. Compare gate count for $\text{majority}(A, B, C)$ vs NAND implementation.

[7] T_0 — functions preserving 0; T_1 — preserving 1; S — self-dual; M — monotone; L — linear (affine).

Problem 6: Zhegalkin Polynomials

Every Boolean function has a unique polynomial representation over $\mathbb{F}_2$ using XOR and AND, called the Algebraic Normal Form (ANF) or *Zhegalkin polynomial*. The *algebraic degree* of this polynomial measures cryptographic strength: low-degree functions are vulnerable to linear attacks, while high-degree functions resist algebraic cryptanalysis.

Part (a): Functional Completeness

1. Prove $\{\oplus, \land, 1\}$ is functionally complete by expressing $\neg x$ and $x \lor y$.
2. Express majority(x, y, z) using only $\{\oplus, \land, 1\}$. Count operations.

Part (b): Computing ANF

Find the ANF for each function (using any construction method):

1. $f_1(a, b, c) = ab \lor bc \lor ac$
2. $f_2(a, b, c, d) = \sum m(1, 4, 6, 7, 10, 13, 15)$
3. $f_3(a, b, c) = (a \to b) \oplus (b \to c)$
4. $f_4(a, b, c, d) = (a \oplus b)(c \oplus d)$

Part (c): Algebraic Degree

1. Identify the algebraic degree of each function from Part (b).
2. Which has highest degree? Is it *maximal* for that number of variables?
3. Prove $\deg(f \oplus g) \leq \max(\deg(f), \deg(g))$. Find an example with strict inequality.

Part (d): Cryptographic S-Box

S-box $S : \mathbb{B}^3 \to \mathbb{B}$ has truth table $(0, 1, 1, 0, 1, 0, 0, 1)$.

1. Find ANF and degree.
2. Is S *balanced*?[8] *Affine*?[9]
3. Construct a different function $T : \mathbb{B}^3 \to \mathbb{B}$ that is balanced and has degree 2.
4. For both S and T, compute distance to the nearest affine function.

Problem 7: Gray Code Circuits

When a rotary encoder transitions from 3 (011) to 4 (100) in binary, all three bits flip — the sensor might read *anything* during the transition. *Gray code* solves this: consecutive values differ in exactly one bit, eliminating ambiguous readings.

Example: 4-bit: $0 \to$ `0000`, $1 \to$ `0001`, $2 \to$ `0011`, $3 \to$ `0010`, …, $15 \to$ `1000`

Part (a): Binary-to-Gray Conversion

1. Build the 4-bit binary-to-Gray truth table.
2. Prove that for your conversion formula, consecutive binary values always produce Gray codes differing in exactly one bit.
3. Derive the conversion formula for each g_i in terms of (b_3, b_2, b_1, b_0).

[8] A function is *balanced* if it outputs 0 and 1 equally often.
[9] A function is *affine* if it has degree ≤ 1.

Part (b): Gray-to-Binary Conversion

1. Derive the inverse formula for each b_i in terms of (g_3, g_2, g_1, g_0).
2. Prove "binary → Gray → binary" is the identity.
3. Test with examples: `1001` and `0110`.

Part (c): Circuit Implementation

1. Design both converters using XOR gates.
2. Count gates and compute *propagation delay* for each.
3. Compare complexity and explain structural differences.

Part (d): NAND Implementation

Redesign binary-to-Gray using only NAND gates. Compare gate count and delay with XOR design. Which is more practical?

Part (e): Timing Analysis

A 4-bit rotary encoder outputs Gray code at 10 MHz (10 million steps/second). With 10ps gate delay[10], is your Gray-to-binary converter fast enough? What is the maximum step rate it can handle?

Part (f): Glitch Analysis

Mechanical encoders don't switch bits *atomically*. During transitions where multiple binary bits change (*e.g.*, $3 \to 4$), different bits flip at different times, creating intermediate *glitch states*. Demonstrate this with a concrete example and explain why Gray code eliminates this problem.

Problem 8: Arithmetic Circuits

In this problem, you will design and analyze subtraction and comparison circuits. Unlike addition, subtraction requires careful *borrow* handling, flowing backward through bit positions.

Part (a): Half Subtractor

Design a half subtractor for $x - y$ (outputs: difference d, borrow b_{out}). Build truth table, derive expressions for outputs, construct circuit using AND/OR/NOT gates. Show how the difference output can be expressed using XOR.

Part (b): Full Subtractor

A full subtractor handles three inputs: *minuend* x, *subtrahend* y, and borrow-in b_{in}.[11]

1. Build truth table for $x - y - b_{\text{in}}$.
2. Design using two half subtractors. Explain *borrow propagation*.
3. Calculate *critical path delay*.
4. Verify on $(0, 1, 1)$ and $(1, 0, 1)$.

Part (c): 4-bit Saturating Subtractor

A *saturating subtractor* computes $d = \max(0, x - y)$, clamping negatives to zero.

1. Design using four full subtractors plus saturation logic. Which signal detects $x < y$?

[10] 1 picosecond (ps) = 10^{-12} seconds. Modern gates have delays in the range 10-100 ps depending on technology.
[11] The *minuend* is the number being subtracted from, and the *subtrahend* is the number being subtracted.

2. Test: $5 - 3$, $3 - 5$ (saturates to 0), $15 - 8$.

Part (d): 2-bit Comparator

1. Design a 2-bit comparator for inputs $x, y \in \{0, 1, 2, 3\}$ with three binary outputs: $>, =, <$.
2. Verify on test inputs: $(3, 2)$, $(2, 2)$, $(1, 3)$.
3. Explain *cascading* for 4-bit comparator.[12]

Part (e): 2-bit Multiplier

Design a multiplier for 2-bit integers $x, y \in \{0, 1, 2, 3\}$ producing 4-bit product $p = x \times y$.

1. Build 16-row truth table for inputs (x_1, x_0, y_1, y_0) and outputs (p_3, p_2, p_1, p_0).
2. Find minimal expressions for output bits (p_3, p_2, p_1, p_0) using K-maps.
3. Draw optimized circuit exploiting *shared sub-expressions*.[13]
4. Verify: $3 \times 2 = 6$, $3 \times 3 = 9$, $1 \times 1 = 1$.

Problem 9: Binary Decision Diagrams

BDDs provide *canonical representations* enabling efficient equivalence checking, but size varies exponentially with variable ordering. The same function might need 10 nodes with one ordering and 1000 with another. Finding optimal orderings is NP-complete, but good heuristics exist.

Part (a): If-Then-Else Function

$\mathrm{ITE}(c, x, y)$ returns x when $c = 1$, and y when $c = 0$.

1. Express ITE using $\{\wedge, \vee, \neg\}$. Verify with truth table.
2. Prove $\{\mathrm{ITE}\}$ alone is functionally complete via Post's criterion.
3. Express $x \oplus y$ and $\mathrm{majority}(x, y, z)$ using only ITE. Count ITE calls.

Part (b): Construct ROBDDs with Natural Ordering

Use natural order $x_1 \prec x_2 \prec x_3 \prec \cdots$ for:

1. $f_1(x_1, x_2, x_3, x_4) = x_1 \oplus x_2 \oplus x_3 \oplus x_4$
2. $f_2(x_1, \ldots, x_5) = \mathrm{majority}(x_1, \ldots, x_5)$
3. $f_3(x_1, \ldots, x_4) = \sum m(1, 2, 5, 12, 15)$
4. $f_4(x_1, \ldots, x_6) = x_1 x_4 + x_2 x_5 + x_3 x_6$

For each function:
1. Build the complete *binary decision tree*.
2. Apply *reduction rules* to get the ROBDD: eliminate duplicate terminals, merge *isomorphic subgraphs*, remove redundant nodes.
3. Draw the final ROBDD. Count nodes and compare to unreduced tree size.
4. Verify correctness by tracing two test inputs through your ROBDD.

Part (c): Variable Ordering Impact

For each function from Part (b):

1. Find an ordering producing smaller ROBDD than natural order.

[12] *Cascading* means connecting multiple smaller comparators to build larger ones, using the outputs of one stage as inputs to the next.

[13] *Shared sub-expressions* are intermediate computations used by multiple outputs, reducing total gate count.

2. Construct the smaller ROBDD and count nodes.
3. For f_4, explain why *interleaving*[14] paired variables helps. For others, discuss patterns leading to size reduction, or why the order does not matter at all.

Part (d): BDD Operations

Using the ROBDDs for f_1 and f_3 from Part (b):

1. Construct the ROBDD for $f_1 \wedge f_3$ using the *apply algorithm*. Describe how this algorithm combines two ROBDDs without enumerating truth tables.
2. Draw the resulting ROBDD and count its nodes. Compare to the sum of individual ROBDD sizes.
3. What is the worst-case size for the resulting BDD when combining two ROBDDs?

Problem 10: Reed–Muller Codes

In 1971, Mariner 9 transmitted the first close-up Mars images using Reed–Muller codes. These error-correcting codes are built from Boolean functions with restricted algebraic degree.

The Reed–Muller code $\mathrm{RM}(r,m)$ consists of all Boolean functions $f : \mathbb{F}_2^m \to \mathbb{F}_2$ with $\deg(f) \leq r$.

Parameters: length $n = 2^m$, dimension $k = \sum_{i=0}^{r} \binom{m}{i}$, minimum *Hamming distance*[15] $d = 2^{m-r}$.

Generator matrix construction: Rows correspond to *monomials* of degree $\leq r$ in lexicographic order. Columns correspond to inputs $(x_1, ..., x_m) \in \mathbb{F}_2^m$ in lexicographic order. Entry (i,j) is the value of monomial i evaluated at input j.

Example: $\mathrm{RM}(1,2)$ has monomials $\{1, x_1, x_2\}$ and inputs $\{(0,0),(0,1),(1,0),(1,1)\}$:

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

Message $(1,0,1)$ encodes to $(1,0,1,0)$ via $\boldsymbol{c} = \boldsymbol{u}G$.

Part (a): Code Construction

Consider $\mathrm{RM}(1,3)$:

1. Compute the code parameters: length n, dimension k, minimum distance d, and error correction capability $t = \left\lfloor \frac{d-1}{2} \right\rfloor$.
2. Construct the 4×8 generator matrix G. Rows correspond to monomials $\{1, x_1, x_2, x_3\}$. Columns are ordered: $(0,0,0), (0,0,1), ..., (1,1,1)$, where column i (for $i = 1, ..., 8$) corresponds to the binary representation of $i - 1$.
3. Encode the message $\boldsymbol{u} = (1,0,1,1)$ into codeword $\boldsymbol{c} = \boldsymbol{u}G$.

Part (b): Single-Error Correction

Reed–Muller codes support *majority-logic decoding*: each ANF coefficient is determined by XORing pairs of received bits that differ in the corresponding variable position. Each XOR result "votes" for that coefficient's value; the majority vote wins.

[14] *Interleaving* means alternating between related variable groups (*e.g.*, $x_1, x_4, x_2, x_5, x_3, x_6$ instead of $x_1, x_2, x_3, x_4, x_5, x_6$), which can dramatically reduce BDD size for functions with structural dependencies.
[15] The *Hamming distance* between two codewords is the number of positions in which they differ.

1. Flip *one* bit at position $i \in \{1, ..., 8\}$ of your codeword $\boldsymbol{c}$ to simulate a transmission error, obtaining received word $\boldsymbol{r} = (r_1, ..., r_8)$.
2. Apply majority-logic decoding to recover the ANF coefficients (a_0, a_1, a_2, a_3):
   - For each a_j where $j > 0$: identify all pairs of positions (p, q) whose input vectors differ only in variable x_j Each pair gives a vote $v = r_p \oplus r_q$. Set $a_j = \text{majority}(v_1, v_2, ...)$.
   - For a_0: after determining a_1, a_2, a_3, compute $a_0 = r_1 \oplus (a_1 \cdot 0) \oplus (a_2 \cdot 0) \oplus (a_3 \cdot 0) = r_1$, since position 1 corresponds to input $(0, 0, 0)$.[16]
3. Verify that the decoded message (a_0, a_1, a_2, a_3) equals your original $\boldsymbol{u} = (1, 0, 1, 1)$.
4. Show the detailed computation for coefficient a_1: list all pairs differing in x_1, compute each XOR vote, and determine the majority.

Part (c): Beyond Guaranteed Correction

1. Flip *two* bits at positions i and j (where $i \neq j$) of your original codeword $\boldsymbol{c}$ to simulate two transmission errors, obtaining received word $\boldsymbol{r}$.
2. Apply the same majority-logic decoding algorithm from Part (b).
3. Does decoding recover the correct message? If yes, explain why it succeeded despite exceeding the guaranteed correction capability. If no, identify which coefficients were decoded incorrectly.
4. Since $d = 4$, the code guarantees correction of only $t = 1$ error. Explain why majority-logic decoding might sometimes successfully correct 2 errors, and under what conditions it would fail.

Part (d): Rate and Efficiency

The *rate* of a code is k/n: the ratio of message bits to transmitted bits. Higher rate means less *redundancy* but weaker error correction.

1. Calculate the rate k/n for $\text{RM}(1, 3)$.
2. A *repetition code* encodes each bit by repeating it: for example, the 3-repetition code maps each bit $0 \to 000$ and $1 \to 111$, giving rate $1/3$. Compare with $\text{RM}(1, 3)$: which is more efficient?
3. For general $\text{RM}(r, m)$: as r increases from 0 to m, what happens to k (dimension) and d (distance)? Explain the fundamental trade-off between rate and error correction capability.

Submission Guidelines:
- Organize solutions clearly with problem numbers and parts.
- For circuit designs: Draw clear diagrams with labeled gates and signals.
- For proofs: State assumptions, show logical steps, and clearly mark conclusions.
- For truth tables: Use standard binary ordering (000, 001, 010, ..., 111).
- For K-maps: Show groupings clearly and indicate which groups form the minimal form.

Grading Rubric:
- Correctness of circuits and Boolean expressions: 40%
- Mathematical rigor and proof quality: 30%
- Clarity of diagrams and presentation: 20%
- Completeness and attention to detail: 10%

[16] In case of tied votes, choose 0 as the default value. Note that with a single error, ties should not occur for $\text{RM}(1, 3)$.