ICSC 2025 — Qualification Round: Solutions

Problem A: Neural Network Components

$w_{21}^{(1)}$: Weight from input feature 1 to hidden neuron 2.
Σ : Weighted sum (linear combination + bias).
f : Activation function applied to Σ.
Box A : Hidden layer.
Box B : Output layer.
$\hat{y}$: Predicted output.

Problem B: Cake Calculator

Each cake needs 100 flour and 50 sugar.

$$\text{cakes} = \min\left(\left\lfloor \frac{\text{flour}}{100} \right\rfloor, \left\lfloor \frac{\text{sugar}}{50} \right\rfloor\right)$$

C++ Implementation

```cpp
#include <iostream>
#include <vector>
#include <stdexcept>

std::vector<int> cake_calculator(int flour, int sugar) {
    /**
     * Calculates the maximum number of cakes that can be made and
        the leftover ingredients.
     *
     * Args:
     *     flour: An integer larger than 0 specifying the amount of
        available flour.
     *     sugar: An integer larger than 0 specifying the amount of
        available sugar.
     *
     * Returns:
     *     A vector of three integers:
     *     [0] the number of cakes that can be made
     *     [1] the amount of leftover flour
     *     [2] the amount of leftover sugar
     */

    const int flourNeeded = 100;
```

```
    const int sugarNeeded = 50;

    // Calculate max cakes possible
    int cakes = std::min(flour / flourNeeded, sugar / sugarNeeded)
        ;

    // Calculate leftovers
    int rem_flour = flour - cakes * flourNeeded;
    int rem_sugar = sugar - cakes * sugarNeeded;

    return {cakes, rem_flour, rem_sugar};
}


// --- Main execution block. DO NOT MODIFY ---
int main() {
    try {
        // 1. Read input from stdin
        std::string flour_str, sugar_str;
        std::getline(std::cin, flour_str);
        std::getline(std::cin, sugar_str);

        // 2. Convert inputs to appropriate types
        int flour = std::stoi(flour_str);
        int sugar = std::stoi(sugar_str);

        // 3. Call the cake calculator function
        std::vector<int> result = cake_calculator(flour, sugar);

        // 4. Print the result to stdout in the required format
        std::cout << result[0] << " " << result[1] << " " << result
            [2] << std::endl;

    } catch (const std::invalid_argument& e) {
        // Handle errors during input conversion or validation
        std::cerr << "Input Error or Validation Failed: " << e.what
            () << std::endl;
        return 1;
    } catch (const std::out_of_range& e) {
        // Handle out of range errors during stoi conversion
        std::cerr << "Input Error: Number out of range" << std::
            endl;
        return 1;
    } catch (const std::ios_base::failure& e) {
        // Handle cases where reading input failed
```

```
        std::cerr << "Error: Failed to read input" << std::endl;
        return 1;
    } catch (const std::exception& e) {
        // Catch any other unexpected errors
        std::cerr << "An unexpected error occurred: " << e.what()
          << std::endl;
        return 1;
    }

    return 0;
}
```

Problem C: The School Messaging App

Question 1

Standard fixed-length encodings waste bits on common symbols. Using variable-length prefix codes assigns *shorter* codes to frequent symbols and *longer* codes to rare ones, reducing the *average* bits per character and allowing more messages within the same data limit.

Example: If $\Pr(A) = 0.9$ and $\Pr(B) = 0.1$, a variable-length code like $A \mapsto 0$ (1 bit), $B \mapsto 10$ (2 bits) yields

$$\bar{L} = 0.9 \cdot 1 + 0.1 \cdot 2 = 1.1 \text{ bits/char},$$

which is better than giving both symbols the same length in many practical alphabets.

Question 2

Entropy (theoretical lower bound on average lossless code length) is

$$\begin{aligned} H &= -\sum_{i=1}^{12} p_i \log_2 p_i \\ &= -\big(0.20\log_2 0.20 + 0.15\log_2 0.15 + 0.12\log_2 0.12 \\ &\quad + 0.10\log_2 0.10 + 0.08\log_2 0.08 + 0.06\log_2 0.06 \\ &\quad + 0.05\log_2 0.05 + 0.05\log_2 0.05 + 0.04\log_2 0.04 \\ &\quad + 0.03\log_2 0.03 + 0.02\log_2 0.02 + 0.10\log_2 0.10\big) \\ &\approx 3.3240 \text{ bits/char}. \end{aligned}$$

Interpretation: No lossless encoding (on average) can beat 3.3240 bits per character for this source.

Question 3

Given the Fano codes:

A: 000 (3), B: 100 (3), C: 010 (3), D: 1100 (4), E: 0110 (4), F: 1010 (4),
G: 001 (3), H: 1011 (4), I: 0111 (4), J: 1101 (4), K: 1111 (4), L: 1110 (4),

the average code length is

$$\begin{aligned}\bar{L} &= \sum_i p_i \, \ell_i \\ &= 0.20 \cdot 3 + 0.15 \cdot 3 + 0.12 \cdot 3 + 0.10 \cdot 4 + 0.08 \cdot 4 + 0.06 \cdot 4 \\ &\quad + 0.05 \cdot 3 + 0.05 \cdot 4 + 0.04 \cdot 4 + 0.03 \cdot 4 + 0.02 \cdot 4 + 0.10 \cdot 4 \\ &= 3.48 \text{ bits/char.}\end{aligned}$$

Efficiency relative to the entropy bound:

$$\text{Efficiency} = \frac{H}{\bar{L}} \times 100\% = \frac{3.3240}{3.48} \times 100\% \approx 95.5\%.$$

Final results: $H \approx 3.3240$ bits/char, $\bar{L} = 3.48$ bits/char, Efficiency $\approx 95.5\%$.

Problem D: Word Search Puzzle

Algorithm

1. Start with a 10×10 grid filled with dots.
2. For each word, attempt random placements (horizontal, vertical, diagonal).
3. If valid, place; otherwise retry.
4. Fill unused cells with random letters.

C++ Implementation

```
#include <iostream>
#include <vector>
#include <string>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <stdexcept>
#include <sstream>
#include <cctype>

/**
```

```
 * Generate a 10x10 word search puzzle containing the given words.
 *
 * @param words A vector of strings to include in the puzzle.
 * @return A 2D vector of chars representing the word search
    puzzle.
 */
std::vector<std::vector<char>> create_crossword(const std::vector<
   std::string>& words) {
    const int N = 10;
    // initialize empty grid (0 means empty)
    std::vector<std::vector<char>> grid(N, std::vector<char>(N, 0)
       );

    // Only forward directions: right, down, diagonal down-right,
       diagonal down-left
    const std::vector<std::pair<int,int>> dirs = {
        {0, 1},   // right
        {1, 0},   // down
        {1, 1},   // diagonal down-right
        {1, -1}   // diagonal down-left
    };

    for (auto word : words) {
        // Convert word to uppercase
        for (auto &ch : word) ch = std::toupper(static_cast<
           unsigned char>(ch));

        int L = static_cast<int>(word.size());
        if (L > N) {
            throw std::runtime_error("Word too long to fit in
               puzzle: " + word);
        }

        bool placed = false;
        // try up to 1000 random placements
        for (int attempt = 0; attempt < 1000 && !placed; ++attempt
           ) {
            auto [dx, dy] = dirs[std::rand() % dirs.size()];
            int r = std::rand() % N;
            int c = std::rand() % N;
            int endR = r + dx * (L - 1);
            int endC = c + dy * (L - 1);

            // check bounds
            if (endR < 0 || endR >= N || endC < 0 || endC >= N)
```

```
                continue;

            // check overlap validity
            bool ok = true;
            for (int i = 0; i < L; ++i) {
                char existing = grid[r + dx * i][c + dy * i];
                if (existing != 0 && existing != word[i]) {
                    ok = false;
                    break;
                }
            }
            if (!ok)
                continue;

            // place the word
            for (int i = 0; i < L; ++i) {
                grid[r + dx * i][c + dy * i] = word[i];
            }
            placed = true;
        }

        if (!placed) {
            throw std::runtime_error("Failed to place word: " +
               word);
        }
    }

    // fill remaining empty cells with random uppercase letters
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            if (grid[i][j] == 0) {
                grid[i][j] = 'A' + (std::rand() % 26);
            }
        }
    }

    return grid;
}

// --- Main execution block. DO NOT MODIFY. ---
int main() {
    try {
        // Seed random number generator
        std::srand(static_cast<unsigned int>(std::time(nullptr)));
```

```
        // Read words from first line (comma-separated)
        std::string wordsInput;
        std::getline(std::cin, wordsInput);

        std::vector<std::string> words;
        std::stringstream ss(wordsInput);
        std::string word;

        // Parse comma-separated words
        while (std::getline(ss, word, ',')) {
            // Trim whitespace
            word.erase(0, word.find_first_not_of(" \t\r\n"));
            word.erase(word.find_last_not_of(" \t\r\n") + 1);

            if (!word.empty()) {
                words.push_back(word);
            }
        }

        // Generate the word search puzzle
        std::vector<std::vector<char>> puzzle = create_crossword(
            words);

        // Print the result as a 2D grid
        for (const auto& row : puzzle) {
            for (char c : row) {
                std::cout << c;
            }
            std::cout << std::endl;
        }

    } catch (const std::runtime_error& e) {
        std::cerr << "Input Error: " << e.what() << std::endl;
        return 1;
    } catch (const std::exception& e) {
        std::cerr << "An unexpected error occurred: " << e.what()
            << std::endl;
        return 1;
    }

    return 0;
}
```

Problem E: Functional Completeness of NAND

We want to show that the set $\{\text{NAND}\}$ is functionally complete.

Let $a \uparrow b$ denote $\text{NAND}(a, b) = \neg(a \wedge b)$.

$$\textbf{NOT:} \quad \neg a = a \uparrow a.$$

$$\textbf{AND:} \quad a \wedge b = (a \uparrow b) \ \uparrow \ (a \uparrow b).$$

$$\textbf{OR:} \quad a \vee b = (\neg a) \uparrow (\neg b) = (a \uparrow a) \ \uparrow \ (b \uparrow b).$$

Since $\{\text{AND}, \text{OR}, \text{NOT}\}$ is a functionally complete set, and each of these has been expressed using only NAND, it follows that NAND by itself is functionally complete. Therefore, any Boolean function can be expressed using only NAND gates.