



ICSC 2025 Submission

Solutions to the Qualification Round

DARIO LOI

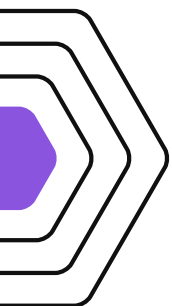
2025

Problem A: Neural Network Components.....	2	Problem C: School Messaging App	5
Problem B: Cake Calculator	4	Problem D: Word Choice Puzzle ..	7
		Problem A: NAND Universality...	10

Solutions

This document presents the solutions for the qualification round of the International Computer Science Competition (ICSC) 2025. Each section corresponds to a problem from the problem sheet. For programming problems, the source code is provided with a description and in-line comments, as per the submission guidelines.




Student: Dario Loi



Problem A: Neural Network Components

This section provides the identification and description of the neural network components as requested in Problem A.

Solution: Solution to Problem A

$w_{21}^{(1)}$:	Weight multiplying input feature 2 (Tip received) and feeding into hidden feature 1.
Σ :	Summation of all incoming weighted features, aggregates all of the contributions of the input features as a linear combination, before it's fed to the activation function f .
f :	Activation Function (typically ReLU: $f(x) = \max(0, x)$, historically sigmoid: $f(x) = \frac{1}{1+e^{-x}}$), ensures that the network can compute non-linear functions, allowing it to act as a universal approximator (according to the Universal Approximation Theorem).
 :	Input Neuron, receives raw input features (in this case Duration Stayed and Tip Received) and passes them to the first hidden layer in a fully-connected fashion.
 :	Bias, a scalar summed to the linear combination of incoming features to ensure that the linear projection computed by the neuron is not constrained to pass through the origin.
 :	Output Neuron, outputs final prediction \hat{y} .
Box A:	Hidden Layer, where the input features are transformed through weighted connections and non-linear activation functions to create higher-level representations.
Box B:	Output Layer, where the final predictions are made based on the transformed features from the hidden layer.
\hat{y} :	Final prediction: the probability of a restaurant customer being satisfied with their visit. Since \hat{y} is a probability and f is the same across the MLP, it is likely that $f(x) = \frac{1}{1+e^{-x}}$ (sigmoid function) is used, which outputs values in the range $[0, 1]$.

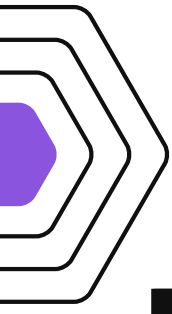


Problem B: Cake Calculator

Solution: Source Code: CakeCalculator

This section provides the implementation of the `CakeCalculator` as requested in Problem B. We provide a $O(1)$ solution that directly calculates the required quantities without need for a while loop.

```
1  from typing import Tuple
2
3  def cake_calculator(flour: int, sugar: int) -> list:
4      """
5          Calculates the maximum number of cakes that can be made and the leftover
6          ingredients.
7
8          Args:
9              flour: An integer larger than 0 specifying the amount of available
10             flour.
11             sugar: An integer larger than 0 specifying the amount of available
12             sugar.
13
14             Returns:
15                 A list of three integers:
16                 [0] the number of cakes that can be made
17                 [1] the amount of leftover flour
18                 [2] the amount of leftover sugar
19
20             Raises:
21                 ValueError: If inputs flour or sugar are not positive.
22             """
23             FLOUR_NEEDED = 100
24             SUGAR_NEEDED = 50
25
26             # This ensures robustness of our solution to invalid inputs,
27             # as our type hint only specifies "int"
28             assert flour >= 0 and sugar >= 0, "Flour and sugar must be non-negative."
29
30
31             # We use integer division (//) to calculate the maximum number of cakes
32             # we can bake given the available flour and sugar.
33             # We then take the min to ensure we don't exceed either ingredient.
34             cakes = min(flour // FLOUR_NEEDED, sugar // SUGAR_NEEDED)
35
36             # We calculate the spare amount of each ingredient by subtracting
37             # the amount we actually used
38             spare_flour = flour - (cakes * FLOUR_NEEDED)
39             spare_sugar = sugar - (cakes * SUGAR_NEEDED)
40
41             return cakes, spare_flour, spare_sugar
```



Problem C: School Messaging App

In this section, we answer the various questions about the information-theory related problem C.

Solution: Question 1

As our message traffic analysis show, natural language messages do not exhibit a uniform distribution across their character set, rather, they tend to use some characters more frequently than others.

Given a character set of size N , and a probability distribution $P(c)$ over the characters, we can define the expected length of a character in bits as:

$$L = \sum_{c \in C} P(c) \cdot \text{size}(c) \quad (1.1)$$

This formula captures the intuition that more frequent characters (higher $P(c)$) and larger characters (higher $\text{size}(c)$) contribute more to the expected length.

To design an efficient encoding scheme, we should aim to assign shorter codes to more frequent characters and longer codes to less frequent characters, decreasing this expected value as much as possible.

Solution: Question 2

We are then tasked to calculate the entropy for our 12-character set, which is defined as:

$$H = - \sum_{c \in C} P(c) \log_2 P(c)$$

We recall our character set to be:

Character	Probability $P(c)$	Character	Probability $P(c)$
A	0.20	G	0.05
B	0.15	H	0.05
C	0.12	I	0.04
D	0.10	J	0.03
E	0.08	K	0.02
F	0.06	L	0.10

Therefore we can calculate our entropy to be:

$$\begin{aligned} H &= - \sum_{c \in C} P(c) \log_2 P(c) \\ &= - \left(0.20 \log_2 0.20 + 0.15 \log_2 0.15 + 0.12 \log_2 0.12 \right. \\ &\quad + 0.10 \log_2 0.10 + 0.08 \log_2 0.08 + 0.06 \log_2 0.06 \\ &\quad + 0.05 \log_2 0.05 + 0.05 \log_2 0.05 + 0.04 \log_2 0.04 \\ &\quad \left. + 0.03 \log_2 0.03 + 0.02 \log_2 0.02 + 0.10 \log_2 0.10 \right) \\ &\approx 3.324 \end{aligned}$$

This corresponds to the theoretical lower bound for the expected number of *bits* needed to encode a message using this character set, assuming optimal encoding.

This means that, no matter how efficient, no coding scheme can exist that achieves a lower average code length than 3.324.

Solution: Question 3

We calculate the average code length of the provided Fano encoding according to Equation (1.1). First we recall our Fano Code:

Character	Probability $P(c)$	Fano Code	Character	Probability $P(c)$	Fano Code
A	0.20	000	G	0.05	001
B	0.15	100	H	0.05	1011
C	0.12	010	I	0.04	0111
D	0.10	1100	J	0.03	1101
E	0.08	0110	K	0.02	1111
F	0.06	1010	L	0.10	1110

We can now calculate the average code length of this Fano encoding using Equation (1.1):

$$\begin{aligned}
 L &= \sum_{c \in C} P(c) \cdot \text{size}(c) \\
 &= 0.20 \cdot 3 + 0.15 \cdot 3 + 0.12 \cdot 3 \\
 &\quad + 0.10 \cdot 4 + 0.08 \cdot 4 + 0.06 \cdot 4 \\
 &\quad + 0.05 \cdot 3 + 0.05 \cdot 4 + 0.04 \cdot 4 \\
 &\quad + 0.03 \cdot 4 + 0.02 \cdot 4 + 0.10 \cdot 4 \\
 &\approx 3.480
 \end{aligned}$$

Indeed the provided solution is quite efficient. We can calculate the efficiency of the encoding as follows:

$$\begin{aligned}
 \eta &= \frac{H}{L} \\
 &\approx \frac{3.324}{3.480} \\
 &\approx 0.955
 \end{aligned}$$

Meaning that our average message length under Fano encoding is 95.5% close to the theoretical lower bound.

Problem D: Word Choice Puzzle

Solution: Word Choice Puzzle description & source code

The fourth question asks us to provide a code that generates a 10×10 word choice puzzle. We attempt to do this in the most creative way possible, by implementing these optional features:

- Given a set of words W , we generate a set of distractors \hat{W} by removing the last character of each word in W , we only do this for words longer than 4 letters to ensure that the resulting distractor can be picked up by the human eye, we also discard any distractors that would make $|\hat{W}| > 3$.
- We maximize intersections of legitimate words and distractors, creating a dense mesh of words that make it trickier for the user to find the correct words. We do so by scanning every grid cell in three possible directions (horizontal, vertical and diagonal).
- We fill the rest of the grid randomly. To prevent these random letters from turning a distractor into a copy of a full word accidentally, we temporarily mark cells with lowercase letters to indicate that that letter should be excluded from the generation process.

Below is an example of a 10×10 grid generated by our algorithm, reproducible by setting seed to 42. We color target words and distractors for easy readability.

notice how distractors *can* overlap with completed words, meaning that letters belonging to a distractor can also be part of a full word, when this happens, we color that letter in yellow (full word takes priority).

L	O	V	E	L	A	C	I	P	J
O	Q	R	E	D	M	O	C	M	O
V	R	J	N	S	R	M	S	H	I
E	E	W	T	U	Q	P	C	I	Z
L	T	U	R	I	N	U	T	M	V
A	Y	Q	O	H	Z	T	U	T	*P
C	O	M	P	U	T	E	R	T	H
E	W	M	Y	C	E	D	I	G	Q
F	G	X	T	W	K	D	N	O	Z
F	K	J	I	N	J	B	G	B	Y

Legend: A = letter belonging to one of the target words (ICSC, TURING, LOVELACE, COMPUTER, ENTROPY);

A = letter belonging to a distractor.

Below we list the full source code for the Word Choice Puzzle generator, this is quite long and will end up being typeset across multiple pages. For a more readable version, check out the attached python file `problem_d.py`.

```

1  def create_crossword(words: list) -> list:
2      """
3      Generate a 10x10 word search puzzle containing the given words.
4
5      Args:
6          words: A list of words to include in the puzzle.
7
8      Returns:
9          A 2D array (list of lists) representing the word search puzzle.
10     """
11
12     import string
13     random.seed(42)
14
15     # WRITE YOUR CODE HERE
16     GRID_SIZE = 10
17     base = [w.upper() for w in words]
18     # generate distractors by removing the last character of each word
19     longer than 4 letters
20     # we only want at most 3 distractors
21     distractors = [w[:-1].upper() for w in base if len(w) > 4][:3]
22     full_map = {
23         d: [f for f in base if f.startswith(d) and len(f) == len(d) + 1]
24         for d in distractors
25     }
26     # map distractor to its full word
27     seq = sorted(base + distractors, key=len, reverse=True)
28     g = [[None] * GRID_SIZE for _ in range(GRID_SIZE)]
29
30     # easily extendable to generate reverse words by adding (0, -1), (-1, 0)
31     , (-1, -1)
32     DIRS = ((0, 1), (1, 0), (1, 1))
33     for wi, w in enumerate(seq):
34         is_distractor = w in full_map
35         best = None
36         best_i = -1
37         placements = []
38         L = len(w)
39         for r in range(GRID_SIZE):
40             for c in range(GRID_SIZE):
41                 for dr, dc in DIRS:
42                     # skip trivial placement (first row, first column,
43                     horizontal)
44
45                     # a bit hardcoded, but necessary for aesthetics
46                     if wi == 0 and r == 0 and c == 0 and dr == 0 and dc ==
47
48 1:
49
50                     continue
51                     rr, cc = r + (L - 1) * dr, c + (L - 1) * dc
52                     # ensure within bounds
53                     if not (0 <= rr < GRID_SIZE and 0 <= cc < GRID_SIZE):
54                         continue
55                     inter = 0
56                     new_cells = 0
57                     ok = True
58                     for i, ch in enumerate(w):
59                         R, C = r + i * dr, c + i * dc
60                         cell = g[R][C]
61
62                         # if already taken by different char, skip

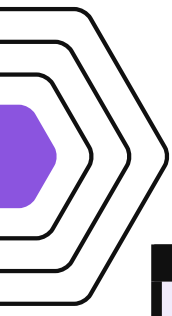
```



```

35         if cell and cell != ch and not cell.islower():
36             ok = False
37             break
38         # if already taken by same char, intersect
39         if cell == ch:
40             inter += 1
41         # overwrite constraint with different character
42         if (
43             cell is None
44             or cell.islower()
45             and not cell.upper() == ch
46         ):
47             new_cells += 1
48         if not ok:
49             continue
50         # skip fully embedded distractor (no new cells added)
51         if is_distractor and new_cells == 0:
52             continue
53         # mark placement as best one, maximizing intersections
54         if inter > best_i:
55             best_i, best = inter, (r, c, dr, dc)
56             placements.append((r, c, dr, dc))
57     if not best:
58         if not placements:
59             continue
60         # pick random best placement
61         best = random.choice(placements)
62     r, c, dr, dc = best
63     for i, ch in enumerate(w):
64         # write the word into the grid
65         g[r + i * dr][c + i * dc] = ch
66     # mark forbidden extension letters after distractor to avoid forming
67     full word
68     if is_distractor:
69         rr, cc = r + L * dr, c + L * dc
70         if (
71             0 <= rr < GRID_SIZE
72             and 0 <= cc < GRID_SIZE
73             and g[rr][cc] is None
74         ):
75             forb = "".join({f[-1].lower() for f in full_map[w]})
76             g[rr][cc] = forb
77     # fill remaining with random letters avoiding lowercase-forbidden
78     markers
79     for r in range(GRID_SIZE):
80         for c in range(GRID_SIZE):
81             cell = g[r][c]
82             if cell is None or cell.islower():
83                 # extract constraint
84                 banned = set(cell.upper()) if cell else set()
85                 choices = set(string.ascii_uppercase) - banned
86                 g[r][c] = random.choice(list(choices))
87     return g

```



Problem A: NAND Universality

Solution: Functional Completeness of NAND proof

To prove that NAND is functionally complete, we need to show that we can express all possible boolean functions using only the NAND operation. The NAND operation is defined as follows:

$$A \text{ NAND } B = \neg(A \wedge B)$$

We can express the basic logical operations NOT, AND, and OR using NAND, from these operators, any other boolean function can be constructed. The steps are as follows:

1. **NOT** operation:

$$\neg A = A \text{ NAND } A$$

2. **AND** operation:

$$A \wedge B = \neg(A \text{ NAND } B) = (A \text{ NAND } B) \text{ NAND } (A \text{ NAND } B)$$

3. **OR** operation:

$$A \vee B = \neg A \wedge \neg B = (A \text{ NAND } A) \text{ NAND } (B \text{ NAND } B)$$

This short constructive proof demonstrates that NAND is functionally complete, as we can express all boolean functions using only this operation.