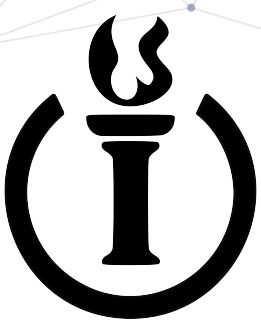


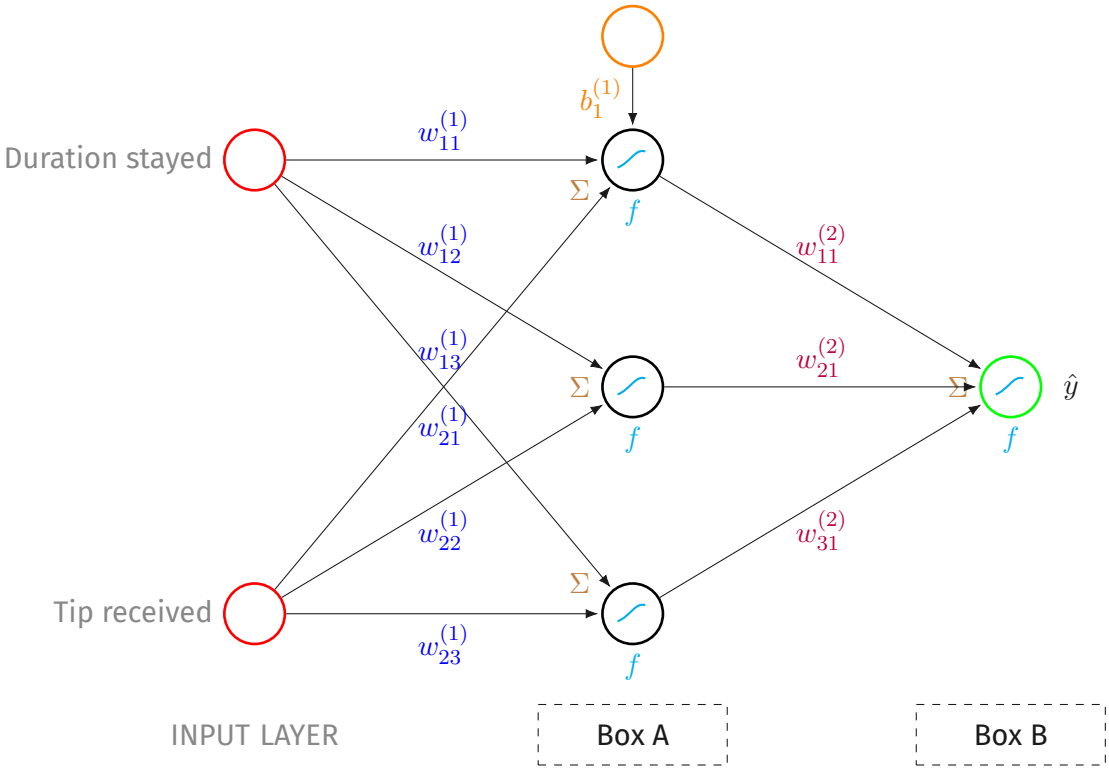
International Computer Science Competition

Qualification Round – Solution

The following solutions are example solutions and are by no means complete or cover all possible correct solutions. Not all detailed steps are elaborated in this solution document. For most problems more than one approach can lead to full points.

Problem A **[5 points]**

Consider this illustration of a neural network (Multi-layer Perceptron), that predicts whether a restaurant customer was satisfied with their visit:



Identify and label the following components based on the figure above:

$w_{21}^{(1)}$: ..... Σ: ..... f: .....

 : .....  : .....  : .....

Box A: ..... Box B: ..... $\hat{y}$: .....

Solution:

- $w_{21}^{(1)}$: Weight – a numerical parameter that scales the input signal from one neuron to another. Specifically, $w_{21}^{(1)}$ is the weight connecting the second input neuron (Tip received) to the first hidden neuron, in layer 1.
- Σ: Summation function – computes the weighted sum of all inputs to a neuron: $z = \sum_i w_i x_i + b$.
- f: Activation function – a non-linear function applied to the weighted sum to determine the neuron's output (e.g., sigmoid, ReLU).
- **Red circle**: Input neuron – receives raw input data (Duration stayed, Tip received) and passes it to the next layer.
- **Orange circle**: Bias neuron – provides a constant offset b added to the weighted sum, allowing the model to shift the activation function.
- **Green circle**: Output neuron – produces the final prediction $\hat{y}$ of the network.
- **Box A**: Hidden layer – the intermediate layer of neurons between input and output that learns internal representations.
- **Box B**: Output layer – the final layer that produces the network's prediction.
- $\hat{y}$: Predicted output – the network's prediction of whether the customer was satisfied (e.g., a value between 0 and 1 after applying a sigmoid activation).

Problem B [5 points]

A baker has a favorite cake recipe that requires precise amounts of ingredients: **100 units of flour** and **50 units of sugar**. Your task is to help the baker to calculate:

- How many cakes can be made from a given inventory of flour and sugar.
- How much flour and sugar will remain unused after making as many cakes as possible.

Write the following function:

`list<cake, remaining_flour, remaining_sugar> cake_calculator(flour, sugar).`

- `flour`: An integer larger 0 specifing the amount of available flour.
- `sugar`: An integer larger 0 specifiying the amount of available sugar.
- The function should return a list (or array) of three integers (`cake`, `remaining_flour`, `remaining_sugar`) with the following indices: (0) the number of cakes that can be made, (1) the amount of leftover flour, (2) the amount of leftover sugar.

Hint: There are several valid ways to approach this problem. To help you with the problem, we show you a possible solution below in pseudocode.[1] *You may use the pseudocode for your program, or come up with a different solution altogether.*

CakeCalculator

```
Input: Ingredients: flour, sugar
Output: cakeCount, flourLeft, sugarLeft
flourNeeded ← 100                          /* Recipe requires 100 flour */
sugarNeeded ← 50                           /* Recipe requires 50 sugar */
cakeCount ← 0                              /* Number of cakes made */
while true do
    if flour < flourNeeded OR sugar < sugarNeeded then
        break                              /* Stop if we can't make any more cakes */
    end if
    flour ← flour - flourNeeded
    sugar ← sugar - sugarNeeded
    cakeCount ← cakeCount + 1
end while
flourLeft ← flour
sugarLeft ← sugar
return cakeCount, flourLeft, sugarLeft
```

[1] Pseudocode is an abstract way of describing algorithms without focusing on the specifics of any particular programming language. It is a useful form of abstraction. Learn how to read pseudocode here: https://www.geeksforgeeks.org/what-is-pseudocode-a-complete-tutorial/

Solution:

```
function cake_calculator(flour, sugar):
    flour_needed = 100
    sugar_needed = 50
    cakes = min(flour // flour_needed, sugar // sugar_needed)
    remaining_flour = flour - cakes * flour_needed
    remaining_sugar = sugar - cakes * sugar_needed
    return [cakes, remaining_flour, remaining_sugar]
```

Explanation: The maximum number of cakes is determined by the limiting ingredient: $\text{cakes} = \min\left(\left\lfloor \frac{\text{flour}}{100} \right\rfloor, \left\lfloor \frac{\text{sugar}}{50} \right\rfloor\right)$. The remaining flour and sugar are computed by subtracting the amounts used.

Example: With 250 flour and 200 sugar: $\min(2, 4) = 2$ cakes, leaving 50 flour and 100 sugar.

Problem C [5 points]

You and your friends have developed a messaging app that operates over the school network. To transmit messages, the app encodes characters into binary sequences (combinations of zeros and ones). However, due to the school's strict data policy, each student is allocated only a small amount of data per day. To make the most of this limited bandwidth, you aim to implement a more efficient encoding strategy.

Information entropy helps determine the theoretical minimum number of bits needed on average to encode messages. For a set of symbols with probabilities $p_1, p_2, \ldots, p_n$, the entropy H is given by:

$$H = -\sum_{i=1}^{n} p_i \times \log_2(p_i)$$

You know from previous message traffic that characters have the following frequencies:

Character	Probability	Character	Probability
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

Question 1: Standard text encodings use the same number of bits for each character. Why would using different length codes for characters with different probabilities help you transmit more messages within your data limit? Give an example.

Question 2: Calculate the entropy for the 12-character set above. What does this value represent in terms of optimal encoding?

The Fano method is one way for creating efficient variable-length prefix codes.[2] You have implemented this algorithm and generated the following code for your character set:

Character	Probability	Fano Code	Character	Probability	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

[2] Details on Fano Codes can be found on: https://cs.stanford.edu/people/eroberts/courses/soco/projects/1999-00/information-theory/fano_codes_4.html

Question 3: Calculate the average code length of your Fano code and compare it to the theoretical entropy limit. How efficient is your Fano code?

Solution:

Question 1: Fixed-length encoding assigns the same number of bits to every character. With 12 characters, this requires $\lceil \log_2(12) \rceil = 4$ bits per character. Variable-length encoding assigns shorter codes to more frequent characters and longer codes to less frequent ones. Since common characters like A (20%) use fewer bits, the average bits per character decreases. For example, if A uses 3 bits instead of 4, and A appears 20% of the time, that alone saves $0.20 \times 1 = 0.2$ bits per character on average.

Question 2:

$$\begin{aligned} H &= -\sum_{i=1}^{12} p_i \log_2(p_i) \\ &= -(0.20 \log_2 0.20 + 0.15 \log_2 0.15 + 0.12 \log_2 0.12 + 0.10 \log_2 0.10 \\ &\quad + 0.10 \log_2 0.10 + 0.08 \log_2 0.08 + 0.06 \log_2 0.06 + 0.05 \log_2 0.05 \\ &\quad + 0.05 \log_2 0.05 + 0.04 \log_2 0.04 + 0.03 \log_2 0.03 + 0.02 \log_2 0.02) \\ &\approx 3.32 \text{ bits per character} \end{aligned}$$

This is the theoretical minimum average number of bits needed per character for optimal encoding. No prefix code can achieve a lower average code length.

Question 3: Average Fano code length:

$$\begin{aligned} \bar{L} &= \sum_{i=1}^{12} p_i \cdot l_i \\ &= 0.20(3) + 0.15(3) + 0.12(3) + 0.05(3) \\ &\quad + 0.10(4) + 0.08(4) + 0.06(4) + 0.05(4) + 0.04(4) + 0.03(4) + 0.02(4) + 0.10(4) \\ &= 1.56 + 1.92 = 3.48 \text{ bits per character} \end{aligned}$$

Efficiency: $\eta = \frac{H}{\bar{L}} = \frac{3.32}{3.48} \approx 95.5\%$. The Fano code is quite efficient, using only about 4.5% more bits than the theoretical minimum.

Problem D [5 points]

A Word Search Puzzle is a common educational game in which players search for given words hidden in a grid of letters. Given a 2D grid of letters, one needs to find all valid words in that grid. Below is an example of such a puzzle containing the words *learning*, *science*, and *fun*:

Your task is to implement a function that generates a word search puzzle of **size 10 x 10** based on a list of input words. Specifically write the function:

```
list<list<char>{}> create_crossword(list<string> words)
```

- `words`: A list of words that should appear in the Word Search Puzzle.
- The function should return a 2D array (or list of lists) of characters representing the puzzle. Each word must appear as a continuous sequence in the grid.
- No specific positioning rules are required. Your words can be arranged horizontally, vertically, or diagonally. Be creative! Exceptional solutions will be awarded.

Hint: You may want to review the following programming concepts before solving this problem: (two-dimensional) arrays, string manipulation, iterations, and nested loops.

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

Solution:

```
function create_crossword(words):
    grid = 10x10 array filled with random letters

    for each word in words:
        placed = false
        while not placed:
            direction = random choice from
                        ["horizontal", "vertical", "diagonal"]
            if direction == "horizontal":
                row = random(0, 9)
                col = random(0, 10 - len(word))
                // Check if positions are available
                if all positions grid[row][col+i] are
                   empty or match word[i]:
                    for i in 0..len(word)-1:
                        grid[row][col + i] = word[i]
                    placed = true

            else if direction == "vertical":
                row = random(0, 10 - len(word))
                col = random(0, 9)
                if all positions grid[row+i][col] are
                   empty or match word[i]:
                    for i in 0..len(word)-1:
                        grid[row + i][col] = word[i]
                    placed = true

            else:  // diagonal
                row = random(0, 10 - len(word))
                col = random(0, 10 - len(word))
                if all positions grid[row+i][col+i] are
                   empty or match word[i]:
                    for i in 0..len(word)-1:
                        grid[row + i][col + i] = word[i]
                    placed = true

    // Fill remaining empty cells with random letters
    for each cell in grid:
        if cell is empty:
            cell = random letter A-Z

    return grid
```

The algorithm initializes a 10×10 grid, places each word in a random direction (horizontal, vertical, or diagonal) at a valid position, checking for boundary constraints and conflicts, then fills remaining cells with random letters.

Problem E [5 points]

Logic gates are a fundamental building blocks of computers, as they control how binary information is transmitted through a circuit. A logic gate performs a binary function:

$$f(a, b) \to \{0, 1\}, \quad \text{where } a, b \in \{0, 1\}.$$

Some gates are particularly important due to a property known as *functional completeness*. A logic gate (or set of gates) is said to be functionally complete if any Boolean function can be constructed using only that gate (or gates).

One such gate is the `NAND` (NOT AND) gate, which outputs 0 only when $a = b = 1$. Thus, it outputs 1 if at least one input is 0.

Prove that the `NAND` gate is functionally complete.

Hint: Can you express the `AND`, `OR`, and `NOT` operations using only `NAND`?

Solution:

To prove NAND is functionally complete, we show that NOT, AND, and OR can all be constructed using only NAND gates. Since {NOT, AND, OR} is a functionally complete set, this proves NAND alone is also functionally complete.

1. NOT from NAND:

$$\text{NOT}(a) = \text{NAND}(a, a) = \overline{a \cdot a} = \overline{a}$$

2. AND from NAND:

$$\begin{aligned}\text{AND}(a, b) &= \text{NOT}(\text{NAND}(a, b)) \\ &= \text{NAND}(\text{NAND}(a, b),\ \text{NAND}(a, b)) \\ &= \overline{\overline{a \cdot b}} = a \cdot b\end{aligned}$$

3. OR from NAND (using De Morgan's law):

$$\begin{aligned}\text{OR}(a, b) &= \text{NAND}(\text{NOT}(a),\ \text{NOT}(b)) \\ &= \text{NAND}(\text{NAND}(a, a),\ \text{NAND}(b, b)) \\ &= \overline{\overline{a} \cdot \overline{b}} = a + b\end{aligned}$$

Since NOT, AND, and OR can all be expressed using only NAND, the NAND gate is functionally complete. ■