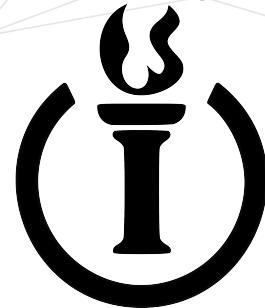


International Computer Science Competition

Pre-Final Round Problem Sheet

Important: Read all the information on this page carefully!



General Information

- The six problems are separated into three categories: 2x basic problems (A; 4 points), 2x advanced problems (B; 6 points), 2x research problems (C; 8 points). The problem type is indicated by the following symbols:
 - 💡 Conceptual problem: Logical or mathematical reasoning, solveable by hand.
 - 💻 Programming problem: Implementing an algorithmic solution in code.
 - 🔬 Research problem: Reading a short scientific article to answer the questions.
- You must submit a single solution document that provides a written description of your solution for all problems (conceptual, programming, and research problems).
- You may submit handwritten or typed solutions (e.g., using TeX). For programming problems, please include any code as text on your solution document with a description and in-line comments. The code must be written in pseudocode, Python (3.9), Java (OpenJDK 11), C (C99), or C++ (GCC 15.1). Only standard libraries are allowed unless otherwise specified in the problem. If unsure, please check with us.
- You receive points for the correct solution as well as for the performed steps. That means: You will not get full points for a correct value if the reasoning steps are missing.
- You can reach up to 36 points in total. You qualify for the Final Round if you reach at least 16 Points (Junior), 20 points (Youth), or 24 points (Senior).
- It is not allowed to work in groups. Assistance from teachers, friends, family, or the internet is prohibited. Textbooks and calculators are allowed.
- (Optional) Alongside your solution document, you can optionally upload your runnable code file for programming problems. Please use the provided skeletons so that your code can be evaluated on our server. Submitted code qualifies for a feature on our leaderboard.

Uploading Your Solution

- Please upload a PDF document containing your solutions, either handwritten or upload a digitally typed document.
- You can upload your solution in your account: <https://icscompetition.org/login>
- Only upload **one single PDF file!** If you have multiple pictures, please merge them into one PDF file. Solution in a different format are not accepted (e.g., Word or Zip files.)
- The submission deadline is Sunday, 19 October 2025.
- The results of the Pre-Final Round will be announced on Monday, 27 October 2025.

Good luck!

</> Problem A.1: Optimal Cake Production (4 Points)

For this problem please use our **code skeletons**. You can download them for [Python](#), [C](#), [C++](#), and [Java](#).

In Problem B from the Qualification Round, you helped a baker calculate how many cakes could be made from available ingredients using a single recipe. Now, the baker has expanded their business and needs your help with a more complex scenario.

The baker now has **two different cake recipes**:

- **Recipe 1** requires: 100 units of flour, 50 units of sugar, 20 units of eggs
- **Recipe 2** requires: 50 units of flour, 100 units of sugar, 30 units of eggs

The baker wants to optimize production. Your task is to determine the optimal combination of cakes from both recipes that **minimizes the total waste** (sum of all leftover ingredient units).

Write the following function:

```
list optimal_cakes(flour, sugar, eggs)
    • flour: An integer representing available flour units
    • sugar: An integer representing available sugar units
    • eggs: An integer representing available eggs units
    • The function should return an integer: total waste
```

Example

```
flour = 500, sugar = 400, eggs = 200
Optimal solution: 2 cakes from recipe 1, 3 cakes from recipe 2
Used: 350 flour, 400 sugar, 130 eggs
Leftover: 150 flour, 0 sugar, 70 eggs
Total waste: 220

Output: 220
```

💡 Problem A.2: The Lighthouse Code (4 Points)

You have intercepted a series of light signals from an old lighthouse. The lighthouse keeper seems to be using the light to transmit coded messages by changing the color of the light. The lighthouse uses four colours: red, blue, green, and yellow. You see the following color beams:

$$\text{signal}_1 = \text{green} \text{ red} \text{ blue} \text{ green}$$

$$\text{signal}_2 = \text{green} \text{ red} \text{ red} \text{ yellow}$$

$$\text{signal}_3 = \text{green} \text{ green} \text{ red} \text{ yellow}$$

$$\text{signal}_4 = \text{green} \text{ red} \text{ red} \text{ yellow}$$

Based on prior analysis, the following sequences are known to map to letters:

$$\text{decode}(\text{green} \text{ red} \text{ green} \text{ blue}) = F$$

$$\text{decode}(\text{green} \text{ red} \text{ green} \text{ yellow}) = G$$

$$\text{decode}(\text{green} \text{ red} \text{ blue} \text{ red}) = H$$

You suspect that the sequence mapping **involves a binary code in ASCII**¹. Can you decode the message sent by the lighthouse?

¹<https://www.asciitable.com/>

</> Problem B.1: Defense Against Model Extraction (6 Points)

For this problem please use our **code skeletons**. You can download them for [Python](#), [C](#), [C++](#), and [Java](#).

You're protecting a valuable machine learning model from extraction attacks. Attackers query your API to steal the model, and you must decide when to inject noise into responses to prevent theft while maintaining service quality for legitimate users.

Over T time periods, you observe query volumes q_t . At each period, you can **add noise** to prevent q_t points of information leakage (but degrade service quality) or **provide clean responses** (maintain perfect service but leak information).

Legitimate users have a trust score that decreases by q_t when you add noise at time t , recovers by doubling (capped at max_trust) when you provide clean responses, and causes service failure if it drops below 0.

Your task is to minimize total information leaked while keeping trust above 0.

Write the following function:

```
int minimize_extraction(query_volumes, initial_trust, max_trust)
```

- `query_volumes`: A list of integers representing information that would leak at each time period if no defense is applied
- `initial_trust`: An integer representing the starting user trust score
- `max_trust`: An integer representing the maximum possible trust score (trust is capped at this value)
- The function should return an integer representing the minimum information that must be leaked to keep trust positive throughout all time periods

Constraints

- $1 \leq T \leq 1000$ time periods
- $1 \leq \text{initial_trust} \leq \text{max_trust} \leq 500$
- $1 \leq q_t \leq 100$ for each query volume

Example

```
query_volumes = [8, 4, 7, 2, 6] # Information leaked if no defense
initial_trust = 10 # Starting user trust
max_trust = 20 # Maximum possible trust score

Time 1: Add noise (prevent 8 leakage, trust: 10 - 8 = 2)
Time 2: Clean (leak 4, trust: 2 * 2 = 4) # Trust doubles when clean
Time 3: Add noise (prevent 7 leakage, trust: 4 - 7 = -3) → FAILURE!
```

Better:

```
Time 1: Clean (leak 8, trust: 10 * 2 = 20, capped at max)
Time 2: Add noise (prevent 4, trust: 20 - 4 = 16)
Time 3: Add noise (prevent 7, trust: 16 - 7 = 9)
Time 4: Add noise (prevent 2, trust: 9 - 2 = 7)
Time 5: Add noise (prevent 6, trust: 7 - 6 = 1)
Total leaked: 8 (only from time 1)
```

💡 Problem B.2: Circuit Complexity (6 Points)

You've proven that `NAND` gates are functionally complete: any Boolean function can be built using only `NAND` gates. However, while we can build any function with enough gates, we often don't know the *minimum* number of gates required.

Consider the “at least k ” function:

$$F_{k,n}(x_1, \dots, x_n) = 1 \quad \text{if and only if} \quad \sum_{i=1}^n x_i \geq k$$

This function appears everywhere: fire alarms (trigger if at least 2 detectors activate) or redundant systems (proceed if at least 3 of 5 sensor measurements in an airplane agree).

a) Express $F_{2,4}$ (outputs 1 if at least 2 of 4 inputs are 1) as a Boolean formula using `AND` (\wedge), `OR` (\vee), and `NOT` (\neg) operations.

b) Let $N(k, n)$ denote the circuit complexity of $F_{k,n}$ as the minimum number of gates needed to compute it. The number of gates is the sum of operators in a boolean formula.

Prove that any Boolean formula computing $F_{2,n}$ has a complexity of at least $N(k, n) = 2n - 3$ for $n > 1$.

c) Define $N_{\text{NAND}}(k, n)$ as the `NAND`-complexity of a Boolean function, that is the minimum number of `NAND` gates needed to compute $F_{k,n}$.

Can you prove that for all $n \geq 3$: $N_{\text{NAND}}(2, n) \leq N_{\text{NAND}}(2, n - 1) + n + 6$?

⚠ Problem C.1: Zipf's Meaning-Frequency Law (8 Points)

This problem requires you to read the following recently published scientific article:

A New Formulation of Zipf's Meaning-Frequency Law through Contextual Diversity.

Ryo Nagata and Kumiko Tanaka-Ishii (2025).

Link: <https://aclanthology.org/2025.acl-long.744.pdf>

Answer the following questions related to this article:

- (a) What are the limitations of dictionary-based studies on measuring Zipf's Meaning-Frequency Law?
 - (b) Explain the von Mises-Fisher distribution and how $v = 1/\kappa$ measures contextual diversity.
 - (c) Why do the authors use the von Mises-Fisher distribution instead of simpler measures like average pairwise cosine similarity between word vectors?
 - (d) How do autoregressive models compare to masked language models for observing the Meaning-Frequency law?
 - (e) How can the proposed method serve as a diagnostic tool for language models?
 - (f) What does the observation that meaning-frequency law breaks down for small models and out-of-domain data suggest?
- (Bonus) What factors may lead more frequent words to have more meanings? What factors may lead to fewer meanings? Give examples of each.

⚠ Problem C.2: Self-Improvement Capabilities of LLMs (8 Points)

This problem requires you to read the following recently published scientific article:

Mind the Gap: Examining the Self-Improvement Capabilities of Large Language Models.

Y. Song, H. Zhang, C. Eisenach, S. M. Kakade, D. Foster, and U. Ghai (2025).

Link: <https://openreview.net/pdf?id=mtJSMcF3ek>

Answer the following questions related to this article:

- (a) Describe the term self-improvement using the author's framework. What key assumption are the authors making that allows for self-improvement?
- (b) What is the generation-verification gap (GV-Gap)? Why is it a better metric than measuring performance differences after model updates?
- (c) What is greedy decoding and why is self-improvement with greedy decoding impossible?
- (d) Explain why the relative GV-Gap scales monotonically with pre-training FLOPs for certain verification methods but not others.
- (e) Why do most models fail to self-improve on Sudoku puzzles despite the exponential computational complexity separation between generation and verification?
- (f) Propose a task domain where you would expect self-improvement to improve performance and explain why.