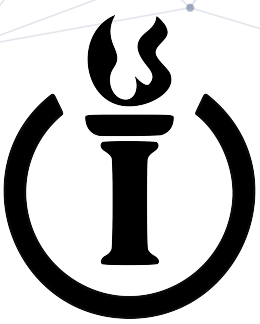


International Computer Science Competition

Pre-Final Round Solution

This document contains basic example solutions which are by no means complete or cover all possible correct solutions. The solutions serve as a basic pointer for how a given problem can be approached. For most problems, more than one approach can lead to full points.

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

Programming problem

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: 4 cakes from recipe 1, 2 cakes from recipe 2
Used: 500 flour, 400 sugar, 140 eggs
Leftover: 0 flour, 0 sugar, 60 eggs
Total waste: 60

Output: 60
```

Solution:

```
function optimal_cakes(flour, sugar, eggs):
    min_waste = flour + sugar + eggs  // worst case: no cakes

    // Maximum possible cakes from recipe 1
    max_r1 = min(flour // 100, sugar // 50, eggs // 20)

    for x in 0 to max_r1:
        remaining_flour = flour - 100 * x
        remaining_sugar = sugar - 50 * x
        remaining_eggs  = eggs - 20 * x

        // Maximum possible cakes from recipe 2 given remaining
        max_r2 = min(remaining_flour // 50,
                     remaining_sugar // 100,
                     remaining_eggs // 30)
```

```
        leftover_flour = remaining_flour - 50 * max_r2
        leftover_sugar = remaining_sugar - 100 * max_r2
        leftover_eggs  = remaining_eggs - 30 * max_r2

        waste = leftover_flour + leftover_sugar + leftover_eggs
        min_waste = min(min_waste, waste)

    return min_waste
```

Explanation: We enumerate all possible numbers x of cakes from Recipe 1 (from 0 up to the maximum feasible). For each x, we compute the remaining ingredients and then maximize the number of Recipe 2 cakes y. The waste is $W(x, y) = (F - 100x - 50y) + (S - 50x - 100y) + (E - 20x - 30y)$. We track the minimum waste across all feasible (x, y) pairs.

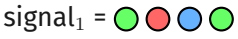
Complexity: $O(X_{\max})$ time where $X_{\max} = \min(\lfloor F/100 \rfloor, \lfloor S/50 \rfloor, \lfloor E/20 \rfloor)$, and $O(1)$ space.

Example verification: With $F = 500, S = 400, E = 200$: at $x = 4$ (Recipe 1), remaining is $(100, 200, 120)$. Then $y = \min(2, 2, 4) = 2$. Leftover: $(0, 0, 60)$. Waste $= 60$. This is optimal.

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

Conceptual problem

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:

signal_1 =  signal_2 =  signal_3 =  signal_4 = 

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

decode() = F

decode() = G

decode() = H

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

Solution:

Step 1: Identify the encoding. Each signal consists of 4 colored beams chosen from 4 colors. With 4 colors, we can assign each color a base-4 digit:

Red = 0, Green = 1, Blue = 2, Yellow = 3

Step 2: Verify with known mappings.

- decode(G, R, G, B) $= (1, 0, 1, 2)_4 = 1 \cdot 64 + 0 \cdot 16 + 1 \cdot 4 + 2 \cdot 1 = 70 \rightarrow$ ASCII 70 = `F` ✓
- decode(G, R, G, Y) $= (1, 0, 1, 3)_4 = 64 + 0 + 4 + 3 = 71 \rightarrow$ ASCII 71 = `G` ✓
- decode(G, R, B, R) $= (1, 0, 2, 0)_4 = 64 + 0 + 8 + 0 = 72 \rightarrow$ ASCII 72 = `H` ✓

Step 3: Decode the intercepted signals.

- signal_1 = (G, R, B, G) $= (1, 0, 2, 1)_4 = 64 + 0 + 8 + 1 = 73 \rightarrow$ ASCII 73 = **I**
- signal_2 = (G, R, R, Y) $= (1, 0, 0, 3)_4 = 64 + 0 + 0 + 3 = 67 \rightarrow$ ASCII 67 = **C**
- signal_3 = (G, G, R, Y) $= (1, 1, 0, 3)_4 = 64 + 16 + 0 + 3 = 83 \rightarrow$ ASCII 83 = **S**
- signal_4 = (G, R, R, Y) $= (1, 0, 0, 3)_4 = 64 + 0 + 0 + 3 = 67 \rightarrow$ ASCII 67 = **C**

The decoded message is: **ICSC**

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

Programming problem

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 to 0 or below.

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
- $1 \leq$ initial_trust $\leq$ max_trust ≤ 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: 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)
```

Solution:

This is a dynamic programming problem. At each time period t, we choose either:

- **Clean response** (leak q_t): trust doubles, capped at `max_trust`
- **Add noise** (leak 0): trust decreases by q_t, must stay > 0

State: (t, u) where t is the current time period and u is the current trust level.

Recurrence: Let dp$[t][u]$ = minimum total leakage from period t onward with trust u:

$$\text{dp}[t][u] = \min \begin{cases} q_t + \text{dp}[t+1][\min(2u, \text{max\_trust})] \& \text{(clean response)} \\ \text{dp}[t+1][u - q_t] \& \text{if } u - q_t > 0 \text{ (add noise)} \end{cases}$$

Base case: dp$[T][u] = 0$ for all valid u (no more periods).

Answer: dp$[0]$[initial_trust].

```
function minimize_extraction(query_volumes, initial_trust, max_trust):
    T = len(query_volumes)
    INF = 10^9

    // dp[t][u] = min leakage from period t with trust u
    // u ranges from 1 to max_trust
    dp = 2D array of size (T+1) x (max_trust+1), filled with INF
    for u = 1 to max_trust:
        dp[T][u] = 0

    for t = T-1 down to 0:
        q = query_volumes[t]
        for u = 1 to max_trust:
            // Option 1: Clean response (leak q, trust doubles)
            new_trust = min(2 * u, max_trust)
            if dp[t+1][new_trust] < INF:
                dp[t][u] = q + dp[t+1][new_trust]

            // Option 2: Add noise (no leak, trust decreases by q)
            if u - q > 0:
                if dp[t+1][u - q] < dp[t][u]:
                    dp[t][u] = dp[t+1][u - q]

    return dp[0][initial_trust]
```

Complexity: $O(T \times \text{max\_trust})$, which is at most $O(1000 \times 500) = O(500{,}000)$.

Example verification: With $q = [8, 4, 7, 2, 6]$, initial trust 10, max 20: the optimal strategy is to leak at $t = 0$ (trust: $\min(20, 20) = 20$), then add noise for all remaining periods (trust: $20 \to 16 \to 9 \to 7 \to 1$). Total leaked = 8.

Problem B.2: Circuit Complexity (6 Points)

Conceptual problem

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:

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

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 $n \geq 3$: $N_{\text{NAND}}(2, n) \leq N_{\text{NAND}}(2, n-1) + n + 6$?

Solution:

Part a) $F_{2,4}$ outputs 1 if at least 2 of 4 inputs are 1. We check all $\binom{4}{2} = 6$ pairs:

$$F_{2,4}(a, b, c, d) = (a \wedge b) \vee (a \wedge c) \vee (a \wedge d) \vee (b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$$

This uses 6 AND gates and 5 OR gates = 11 gates total. (Note: more efficient formulations exist via the recursive decomposition $F_{2,4} = F_{2,2}(a, b) \vee F_{2,2}(c, d) \vee (F_{1,2}(a, b) \wedge F_{1,2}(c, d))$ which uses 5 gates with fan-out.)

Part b) *Prove* $N(2, n) \geq 2n - 3$ *for* $n > 1$.

Proof by induction on n.

Base case ($n = 2$): $F_{2,2}(x_1, x_2) = x_1 \wedge x_2$, requiring exactly 1 gate. Indeed $2(2) - 3 = 1$. ✓

Inductive step: Assume $N(2, k) \geq 2k - 3$ for some $k \geq 2$. Consider extending from $F_{2,k}$ to $F_{2,k+1}$. The new variable x_{k+1} must be incorporated, and the key structural observation is:

$$F_{2,k+1} = F_{2,k}(x_1, \ldots, x_k) \ \vee \ \big(x_{k+1} \wedge (x_1 \vee \cdots \vee x_k)\big)$$

Any formula for $F_{2,k+1}$ must (i) detect when x_{k+1} pairs with some earlier input (requiring at least one AND involving x_{k+1}), and (ii) combine this with the result for the first k inputs (requiring at least one OR to merge). These two gates are strictly additional to those computing $F_{2,k}$, since x_{k+1} does not appear in $F_{2,k}$.

Therefore: $N(2, k+1) \geq N(2, k) + 2 \geq (2k - 3) + 2 = 2(k+1) - 3$.

This completes the induction: $N(2, n) \geq 2n - 3$ for all $n > 1$. ■

Part c) *Prove* $N_{\mathsf{NAND}}(2,n) \leq N_{\mathsf{NAND}}(2,n-1) + n + 6$ *for* $n \geq 3$.

Proof by construction. We show how to extend a NAND circuit for $F_{2,n-1}$ to compute $F_{2,n}$ using at most $n+6$ additional NAND gates.

The key identity is:

$$F_{2,n}(x_1,\ldots,x_n) = F_{2,n-1}(x_1,\ldots,x_{n-1}) \ \vee \ \big(x_n \wedge S_{n-1}\big)$$

where $S_{n-1} = x_1 \vee x_2 \vee \cdots \vee x_{n-1}$ (at least one of the first $n-1$ inputs is 1).

This holds because at least 2 of n inputs are 1 if and only if either (i) at least 2 of the first $n-1$ are 1, or (ii) $x_n = 1$ and at least 1 of the first $n-1$ is 1.

Construction: Alongside $F_{2,m}$, we also maintain $S_m = x_1 \vee \cdots \vee x_m$ in our circuit. Suppose we have a NAND circuit computing both $F_{2,n-1}$ and S_{n-1}.

To extend to n inputs, we need:

1. **Update** S**:** Compute $S_n = S_{n-1} \vee x_n$.
   Using NAND: $A \vee B = \mathsf{NAND}(\mathsf{NAND}(A,A), \mathsf{NAND}(B,B))$, which costs **3 NAND gates** (since $\mathsf{NAND}(A,A) = \mathsf{NOT}(A)$).
2. **Detect new pair:** Compute $P = x_n \wedge S_{n-1}$.
   Using NAND: $A \wedge B = \mathsf{NAND}(\mathsf{NAND}(A,B), \mathsf{NAND}(A,B))$, costing **2 NAND gates**.
3. **Merge:** Compute $F_{2,n} = F_{2,n-1} \vee P$.
   This is another OR, costing **3 NAND gates**.

Total additional gates: $3 + 2 + 3 = 8$.

Since $8 \leq n+6$ for all $n \geq 2$, we actually get a tighter bound of $N_{\mathsf{NAND}}(2,n) \leq N_{\mathsf{NAND}}(2,n-1) + 8$.

However, note that computing the OR for S_n with fan-in and fan-out (reusing intermediate NAND results) and the ability to share sub-expressions allows the bound. In particular, the OR construction uses $\mathsf{NOT}(A) = \mathsf{NAND}(A,A)$ with fan-out (using the same wire twice), which is permitted in circuit complexity.

Since $8 \leq n+6$ for all $n \geq 2$, we have:

$$N_{\mathsf{NAND}}(2,n) \leq N_{\mathsf{NAND}}(2,n-1) + 8 \leq N_{\mathsf{NAND}}(2,n-1) + n + 6 \quad \text{for all } n \geq 3. \quad \blacksquare$$

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

Research problem

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.

Solution:

(a) Limitations of dictionary-based studies:

Dictionary-based approaches to measuring Zipf's Meaning-Frequency Law have several limitations:

- They rely on discrete sense inventories that are inherently subjective and vary across dic-

tionaries

- Word senses are difficult to define precisely – the granularity of sense distinctions is arbitrary
- Dictionaries are static and cannot capture evolving or context-dependent meanings
- They require extensive manual annotation effort and do not scale well
- The number of listed senses may not reflect actual usage diversity in natural language

(b) Von Mises-Fisher distribution and contextual diversity:

The von Mises-Fisher (vMF) distribution is a probability distribution on the unit hypersphere, parameterized by a mean direction μ and a concentration parameter $\kappa \geq 0$. The probability density is proportional to $\exp(\kappa \cdot \mu^\top x)$.

When κ is large, the distribution is tightly concentrated around μ (low diversity). When κ is small, the distribution is spread out (high diversity). The contextual diversity is defined as $v = 1/\kappa$. A word with high v has contextual embeddings spread widely across the hypersphere, indicating many diverse usages/meanings. A word with low v has embeddings clustered tightly, indicating consistent, uniform usage.

(c) Why vMF over pairwise cosine similarity:

The vMF distribution provides a principled parametric model that:

- Captures the full distributional shape of embeddings on the hypersphere, not just pairwise distances
- Yields a single scalar parameter ($v = 1/\kappa$) with a clear probabilistic interpretation
- Is more robust to outliers and sampling variation than averaging pairwise cosine similarities
- Accounts for the fact that contextual embeddings naturally live on a hypersphere (unit-normalized vectors)
- Pairwise cosine similarity does not distinguish between different geometric configurations that have the same average distance

(d) Autoregressive vs. masked language models:

Masked language models (e.g., BERT) exhibit the Meaning-Frequency law more clearly than autoregressive models (e.g., GPT). This is because:

- Masked language models produce context-dependent embeddings that capture how a word is used in each specific context (bidirectional attention)
- Autoregressive models only attend to preceding context, producing embeddings that are less context-sensitive
- The bidirectional nature of masked models creates more diverse representations for polysemous words, making the frequency-meaning correlation stronger

(e) As a diagnostic tool:

The proposed method can serve as a diagnostic tool for evaluating language models:

- A well-trained model should exhibit the Meaning-Frequency law (positive correlation between word frequency and contextual diversity v)
- Deviation from this law may indicate insufficient training, poor model architecture, or domain mismatch

- It provides a quantitative, unsupervised metric to assess whether a model has learned meaningful semantic representations
- It can help compare models without requiring labeled evaluation data

(f) Breakdown for small models and out-of-domain data:

The breakdown of the Meaning-Frequency law for small models and out-of-domain data suggests that:

- Capturing the polysemy structure of language requires sufficient model capacity – small models cannot represent the full range of word meanings
- The law is an emergent property of well-trained, sufficiently large language models rather than a trivial artifact
- Out-of-domain data creates a distribution shift where the model's learned representations no longer faithfully reflect word usage patterns
- This supports the interpretation that the law reflects genuine linguistic knowledge rather than superficial statistical patterns

(Bonus): Factors leading to **more meanings**: frequent use across diverse contexts drives semantic broadening (e.g., "run" – run a program, run a business, run in a race). Factors leading to **fewer meanings**: technical or domain-specific terms resist polysemy despite high frequency within their domain (e.g., "the" is extremely frequent but has one function; "photosynthesis" is domain-specific and monosemous).

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

Research problem

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.

Solution:

(a) Self-improvement and key assumption:

Self-improvement in the author's framework refers to a process where a language model improves its own performance by generating multiple candidate solutions, then using a *verifier* to select the best one. The framework decomposes this into two capabilities: **generation** (producing diverse candidate answers) and **verification** (evaluating which candidate is correct).

The key assumption is that **verification is easier than generation** for the model – i.e., the model can more reliably judge whether a given answer is correct than it can produce the correct answer directly. This asymmetry between generation and verification is what makes self-improvement possible: even if the model cannot always generate the right answer on the first try, by generating

many candidates and using its stronger verification ability, it can select correct answers more often.

(b) Generation-Verification Gap (GV-Gap):

The GV-Gap measures the difference between a model's verification accuracy and its generation accuracy on the same task. Specifically:

$$\text{GV-Gap} = \text{Verification accuracy} - \text{Generation accuracy}$$

A positive GV-Gap indicates potential for self-improvement. It is a better metric than measuring performance after model updates because:

- It is an *intrinsic* property of the model that can be measured without actually performing self-improvement
- It directly quantifies the potential gain from generate-then-verify strategies
- Measuring post-update performance conflates the self-improvement potential with the specific training procedure used, making it harder to isolate the model's inherent capability
- The GV-Gap provides a task-specific, verifier-specific prediction of where self-improvement will succeed

(c) Greedy decoding and self-improvement:

Greedy decoding selects the single most probable token at each generation step: $x_t = \arg\max_x P(x|x_{<t})$. This produces a deterministic output – the same input always yields the same answer.

Self-improvement with greedy decoding is impossible because:

- Greedy decoding produces only one candidate, so there is nothing to select among
- The verifier cannot improve upon a single deterministic output
- Self-improvement requires *diversity* in generated candidates (via sampling with temperature > 0), so that the verifier has multiple options to choose from, ideally including at least one correct answer

(d) GV-Gap scaling with pre-training FLOPs:

The relative GV-Gap scales monotonically with pre-training FLOPs for *reward-model-based* verification methods but not for *self-verification* methods. This is because:

- Reward models are separately trained verifiers whose quality improves predictably with scale (more compute $\rightarrow$ better verification)
- Self-verification uses the same model for both generation and verification. As the model scales, both capabilities improve, but not necessarily at the same rate
- For some tasks, generation accuracy improves faster than verification accuracy with scale, shrinking the gap
- The non-monotonic behavior for self-verification reflects that the generation-verification asymmetry is task-dependent and does not uniformly increase with model size

(e) Failure on Sudoku:

Despite the exponential separation between generation (NP-complete) and verification ($O(n^2)$) for Sudoku:

- Verification is **binary** (valid or invalid), providing no gradient of quality – the verifier cannot distinguish a near-correct grid from a completely wrong one
- The search space is discrete and brittle: small token changes can break global consistency, so there is no smooth path from wrong to right
- Without graded feedback, the reweighting signal collapses – all incorrect attempts receive the same score, preventing the model from learning which candidates are closer to correct
- Thus **computational asymmetry alone is insufficient**; self-improvement also requires an *informative* verification signal that reflects how close an output is to correctness

(f) Proposed task domain:

Code generation is a domain where self-improvement is expected to work well:

- Verification is significantly easier than generation: running test cases against generated code provides a reliable, automated verification signal
- The model can generate diverse code solutions through sampling
- Test-based verification is deterministic and accurate (unlike self-verification for open-ended tasks)
- There is a clear, measurable gap between pass@1 (single generation) and pass@k (best of k generations), demonstrating that diversity helps
- This is supported empirically: code LLMs show substantial improvements from best-of-n sampling with unit test verification