International Computer Science Competition pre-final 2025 Round Problem Submission
Chi Nguyen (10/10/2006)
Boston University Class of 2028 majoring in Biomedical Engineering/Computer Engineering
Applicant Senior (current sophomore)

</> 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
```

```python
import sys

def optimal_cakes(flour: int, sugar: int, eggs: int) -> int:
    """
    Determines the optimal combination of cakes from two recipes that maximizes
    total cakes and minimizes waste.

    Recipe 1: 100 flour, 50 sugar, 20 eggs
    Recipe 2: 50 flour, 100 sugar, 30 eggs
```

```
    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.
        eggs: An integer larger than 0 specifying the amount of available eggs.

    Returns:
        An integer representing the total waste (sum of leftover ingredients)

    Raises:
        ValueError: If inputs are not positive.
    """
    # WRITE YOUR CODE HERE
    # Checks for invalid input. If any ingredient amount is negative, the function
returns -1
    if flour < 0 or sugar < 0 or eggs < 0:
        raise ValueError("Inputs must be nonnegative integers.")

    # Assign variable names
    F = flour
    S = sugar
    E = eggs

    total_min_waste = F + S + E
    max_cakes_found = 0 #record largest total numbers of cakes found
    total_used_ingredients = 0 # record total ingredients amt used in the best case

    #Calculate max cakes if each ingredient is the limiting factor
    max_cakes_recipe1 = min(F // 100, S // 50, E // 20)

    # Try every possible number of Recipe 1 cakes and compute remaining flour, sugar,
and eggs for each case
    for x in range(max_cakes_recipe1 + 1):
        flour_leftover = F - 100 * x
        sugar_leftover = S - 50 * x
        eggs_leftover  = E - 20 * x

        # Check that none of the leftover amounts are negative
        validleftovers = (flour_leftover >= 0 and sugar_leftover >= 0 and eggs_leftover
>= 0)
        #Checks if enough ingredients remain
        if validleftovers:
            y = min(flour_leftover // 50, sugar_leftover // 100, eggs_leftover // 30)
```

```
        #Calculates how many Recipe 2 cakes can be made with leftovers
            flour_used = 100 * x + 50 * y
            sugar_used = 50 * x + 100 * y
            eggs_used  = 20 * x + 30 * y
        #Compute total leftover (unused) ingredients
            waste = (F - flour_used) + (S - sugar_used) + (E - eggs_used)
            total_cakes = x + y #total cakes made
            total_use = flour_used + sugar_used + eggs_used #total ingredients used

            # Compare current plan with best so far
            if waste < total_min_waste:
                better_plan = True #pick less waste
            elif waste == total_min_waste and total_cakes > max_cakes_found:
                better_plan = True #if same waste, pick plan makes more cakes
            elif waste == total_min_waste and total_cakes == max_cakes_found and
total_use > total_used_ingredients:
                better_plan = True #if same waste & cakes, pick plan more total
ingredients used
            else:
                better_plan = False #none conditions above met
            #update results
            if better_plan:
                total_min_waste = waste
                max_cakes_found = total_cakes
                total_used_ingredients = total_use
    #return smallest total leftover (waste) found
    return total_min_waste


# --- Main execution block. DO NOT MODIFY  ---
if __name__ == "__main__":
    try:
        # 1. Read input from stdin
        flour_str = input().strip()
        sugar_str = input().strip()
        eggs_str = input().strip()

        # 2. Convert inputs to appropriate types
        flour = int(flour_str)
        sugar = int(sugar_str)
        eggs = int(eggs_str)
```

```
        # 3. Call the optimal cakes function
        result = optimal_cakes(flour, sugar, eggs)

        # 4. Print the result to stdout in the required format
        print(result)

    except ValueError as e:
        # Handle errors during input conversion or validation
        print(f"Input Error or Validation Failed: {e}", file=sys.stderr)
        sys.exit(1)
    except EOFError:
        # Handle cases where not enough input lines were provided
        print("Error: Not enough input lines provided.", file=sys.stderr)
        sys.exit(1)
    except Exception as e:
        # Catch any other unexpected errors
        print(f"An unexpected error occurred: {e}", file=sys.stderr)
        sys.exit(1)
```

Output testing without Main Execution block:

```
print(optimal_cakes(500, 400, 200))  # expect 60
```

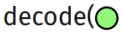
Output:
60

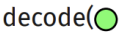
Problem A.2: The Lighthouse Code (4 Points)

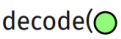
You have intercepted a series of light signals from an old lighthouse. The lighthouse keeper seems to be using the light to transmit coded messages by changing the color of the light. The lighthouse uses four colours: red, blue, green, and yellow. You see the following color beams:

$signal_1$ =  $signal_2$ =  $signal_3$ =  $signal_4$ = 

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**[1]. Can you decode the message sent by the lighthouse?

decode() = F

F (G,R,G,B) ⇒ split to 2 bits pairs: 01 00 01 10 = 01000110

F = 70 → binary: `01000110`

green = `01`

red = `00`

green = `01`

blue = `10`

decode() = G

G (G,R,G,Y) ⇒ split to 2 bits pairs: 01 00 01 11 = 01000111

G = 71 → binary: `01000111`

green = `01`

red = `00`

green = `01`

yellow=`11`

decode() = H

H (G,R,B,R) ⇒ split to 2 bits pairs: 01 00 10 00 = 01001000

H = 72 → binary: `01001000`

green = `01`

red = **00**

blue = **10**

red = **00**

signal$_1$ = G R B G → 01 00 10 01 = 01001001 = ASCII 73 → I
signal$_2$ = G R R Y → 01 00 00 11 = 01000011 = ASCII 67 → C
signal$_3$ = G G R Y → 01 01 00 11 = 01010011 = ASCII 83 → S
signal$_4$ = G R R Y → 01 00 00 11 = 01000011 = ASCII 67 → C

Message: **ICSC**

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 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
- $1 \leq initial\_trust \leq max\_trust \leq 500$
- $1 \leq q_t \leq 100$ 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)
```

```python
import sys
#This returns the minimum total information leaked while keeping trust positive throughout all time periods
def minimize_extraction(query_volumes: list, initial_trust: int, max_trust: int) -> int:
    """
    Determines the minimum information leaked while keeping trust above 0.

    Args:
        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

    Returns:
        An integer representing the minimum information that must be leaked

    Raises:
        ValueError: If inputs are invalid.
    """
    # Check valid input and raise ValueError: If inputs are invalid and not sastify conditions below
    #Checks that query_volumes is a valid list of integers between 1 and 100
    if not query_volumes or not all(isinstance(q, int) and 1 <= q <= 100 for q in query_volumes):
        raise ValueError("query_volumes must be a non-empty list of integers between 1 and 100.")
    #initial_trust and max_trust are valid within allowed bounds
    if not (1 <= initial_trust <= max_trust <= 500):
        raise ValueError("initial_trust and max_trust must satisfy 1 ≤ initial_trust ≤ max_trust ≤ 500.")

    #Dynamic Programming setup
    T = len(query_volumes) #Total number of time periods
    INF = float('inf')    #placeholder for unreachable states
    #Create dp[t][trust] table where each cell stores the minimum leaked info at time t with trust level trust
    dp = [ [INF] * (max_trust + 1) for _ in range(T + 1) ]
    dp[0][initial_trust] = 0  #Initializes the starting state: time 0 with initial_trust and 0 leakage
    #Loops through each time period t and for each trust level, checks if it's reachable and skips unreachable states.
```

```
    for t in range(T):
        qt = query_volumes[t]
        for trust in range(1, max_trust + 1):
            if dp[t][trust] == INF:
                continue

            # Option 1: Add noise (trust decreases by qt, prevent leakage but reduce
trust)
            #Only proceed if trust stays positive and update the next time step with
the same leakage (no new leakage)
            new_trust = trust - qt
            if new_trust > 0:
                dp[t + 1][new_trust] = min(dp[t + 1][new_trust], dp[t][trust])

            # Option 2: Clean response → trust doubles (capped) but leak qt
            #Update the next time step with increased leakage.
            new_trust = min(trust * 2, max_trust)
            dp[t + 1][new_trust] = min(dp[t + 1][new_trust], dp[t][trust] + qt)

    # Final result: Return the minimum leakage among all trust levels that are still
positive
    return min(dp[T][trust] for trust in range(1, max_trust + 1))



# --- Main execution block. DO NOT MODIFY ---
if __name__ == "__main__":
    try:
        # 1. Read input from stdin
        query_volumes_str = input().strip()
        initial_trust_str = input().strip()
        max_trust_str = input().strip()

        # 2. Convert inputs to appropriate types
        query_volumes = list(map(int, query_volumes_str.split()))
        initial_trust = int(initial_trust_str)
        max_trust = int(max_trust_str)

        # 3. Call the minimize_extraction function
        result = minimize_extraction(query_volumes, initial_trust, max_trust)

        # 4. Print the result to stdout
```

```
        print(result)

    except ValueError as e:
        print(f"Input Error or Validation Failed: {e}", file=sys.stderr)
        sys.exit(1)
    except EOFError:
        print("Error: Not enough input lines provided.", file=sys.stderr)
        sys.exit(1)
    except Exception as e:
        print(f"An unexpected error occurred: {e}", file=sys.stderr)
        sys.exit(1)
```

Output for testing WITHOUT Main Execution block code

```
query_volumes = [8, 4, 7, 2, 6]
initial_trust = 10
max_trust = 20

result = minimize_extraction(query_volumes, initial_trust, max_trust)
print("Total leaked:", result)
```

Output:

```
Total leaked: 8
```

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:

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

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

a)We want the function that outputs 1 if at least 2 of the 4 inputs are 1.

Let the inputs be x1,x2,x3,x4

That means F2,4=1 whenever any two or more inputs are 1.

We can write it as an OR of all possible pairs:

$$F_{2,4}(x_1, x_2, x_3, x_4) = (x_1 x_2) \vee (x_1 x_3) \vee (x_1 x_4) \vee (x_2 x_3) \vee (x_2 x_4) \vee (x_3 x_4)$$

b)Prove N(2,n) ≥ 2n − 3 $N(2, n) \geq 2n - 3 \quad \text{for } n > 1$

To combine n inputs by AND and OR to detect "≥2," at least all pairs that include a new variable must be formed. When you add a new variable x_n you must add terms that involve it with all n−1 prior variables (each requiring a new AND gate).

Each new input also adds one OR connection to merge with prior results.

Structure of $F_{2,n}$

$$F_{2,n} = F_{2,n-1} \vee \bigvee_{i=1}^{n-1} (x_i \wedge x_n)$$

For n =2

$$F_{2,2} = x_1 \wedge x_2$$

$N(2,2)=1$

$$N(2,3) = 1 + 2 = 3$$
$$N(2,4) = 3 + 2 = 5$$

Keeps going

So for all n>1

$$N(2,n) = 1 + 2(n-2) = 2n - 3$$

c)

Structure of $F_{2,n}$

$$F_{2,n} = F_{2,n-1} \vee \bigvee_{i=1}^{n-1} (x_i \wedge x_n)$$

Reuse previous circuit

That circuit costs $N_{NAND}(2,n-1)$ gates.

We now only need to add logic that detects the new pairs involve x_n

Build each new pair using NANDs. To get 2 NAND gates per pair:

$$x_i \wedge x_n = \text{NAND}(\text{NAND}(x_i, x_n), \text{NAND}(x_i, x_n))$$

Combine with existing output

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

So for all n≥3n

c) $N_{NAND}(2,n) \leq N_{NAND}(2,n-1) + n + 6$

Problem C.1: Zipf's Meaning-Frequency Law

(a) What are the limitations of dictionary-based studies on measuring Zipf's Meaning-Frequency Law?

In sec 2

Dictionary-based studies assume that word meanings are fixed and countable, cons are:

They rely on human-curated senses, which vary across dictionaries and languages. They ignore contextual variation—words may have different meanings which dictionaries don't capture. Dictionaries don't reflect how meanings evolve or shift in real-time language use, makes them unsuitable for analyzing meaning in dynamic corpora or neural models, where meaning is fluid and context-dependent. Dictionaries list fixed meanings, but real usage is more flexible. They miss how words shift meaning in different contexts. They don't reflect how language models learn meaning from data.

b)Explain the von Mises-Fisher distribution and how v = 1/κ measures contextual diversity.

von Mises-Fisher distribution & v=1/κv=1/κv=1/κ:

It measure how spread out word meanings, word-context vectors are on a sphere.Words used in many different contexts have more spread (more meanings). The formula v=1/κ tells how diverse the contexts are.
κ measures concentration: big κ = tight cluster; small κ= wide spread.

$v=1/\kappa$ “contextual diversity” quantifies how varied the contexts of a word are.

larger v implies the word appears in more semantically diverse contexts, suggesting it has more meanings. larger v= word used in many contexts = more meanings.

c)) Why do the authors use the von Mises-Fisher distribution instead of simpler measures like average pairwise cosine similarity between word vectors?

The authors prefer vMF because:

- Cosine similarity is pairwise and local, while vMF models the global distribution of word contexts. Average cosine similarity is crude and sensitive to noise

- vMF provides a single interpretable parameter (κ) that captures the overall spread of vectors. vMF gives a mathematically solid way to estimate overall spread.

-It focuses on direction (semantic variation) and gives consistent results across models and avoids the scaling issues and sensitivity to outliers that affect cosine-based measures.

(d)) How do autoregressive models compare to masked language models for observing the MeaningFrequency law?

Masked models (like BERT) less effective show a clear meaning-frequency law even when smaller, are trained to predict missing words, which may lead to less nuanced semantic modeling.
Autoregressive models (like GPT) need to be much larger to show the same pattern and show a stronger correlation with the Meaning-Frequency Law. They learn contextual diversity more efficiently, generate text sequentially and learn richer contextual representations. Autoregressive models better capture the diversity of word usage, aligning more closely with Zipf’s law.

e) How the method serves as a diagnostic tool for language models

> The proposed vMF-based method can quantify how well a model captures semantic diversity. The method checks if a model’s frequency-vs-diversity curve follows the expected power law. It helps identify whether a model’s representations reflect real-world meaning-frequency patterns. If it does, the model handles word meanings well.
> If not, it may be too small or poorly adapted to the domain.

It can be used to compare models, detect underfitting or overfitting, and assess domain adaptation. It's especially useful for evaluating small models trained.

(f) What does the observation that meaning-frequency law breaks down for small models and out-of-domain data suggest?

Small models = not enough capacity to represent many contexts.

Out-of-domain data = model hasn't seen similar usage patterns.
Breakdown means the model lacks generalization or semantic depth

When the law fails in small models or out-of-domain settings, it suggests: Insufficient capacity to model semantic variation; Poor generalization to unfamiliar contexts; Limited exposure to diverse usage patterns during training.

This highlights the need for larger, more diverse training data and better model architectures to capture real-world semantics.

(Bonus) What factors may lead more frequent words to have more meanings? What factors may lead to fewer meanings? Give examples of each

More meanings: very frequent, reused words (e.g. *run*, *set*).

- High frequency: frequent words appear in many contexts (e.g., "run" can mean sprint, operate, flow).
- Word from metaphor or idiom: common words get reused metaphorically (e.g., "head" → body part, leader, top).
- Words used as multiple parts of speech (e.g., "light" as noun, verb, adjective).

Fewer meanings: rare, technical, or fixed words (e.g. *microtome*, *approximately*).

- technical words have narrow usage (e.g., "neuron").
- Low frequency: rare words don't appear in varied contexts.
- words with fixed grammatical roles (e.g., "the" has one function).

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

(a) Describe the term self-improvement using the author's framework. What key assumption are the authors making that allows for self-improvement?

Self-improvement is when a model improves its output by verifying and filtering its own generations. It first *generates* candidate answers to tasks, then *verifies* them internally, and retains/improves on those that pass verification.
The key assumption is that verification is easier than generation, allowing the model to identify better outputs. The model (or an auxiliary verifier) can reliably check correctness of its own outputs better than random chance. That is, the verification step is cheaper or easier than generation–or at least feasible–so the model can refine itself. Without a good verifier, self-improvement cannot work. This enables learning from its own filtered data without external supervision

(b) What is the generation-verification gap (GV-Gap)? Why is it a better metric than measuring performance differences after model updates?

GV-Gap measures the performance difference between verified outputs and original generations, or difference between the success rate of *generation* (i.e., how often the model's first attempt is correct) and the success rate of *verification* (how often the verifier flags a correct answer). It reflects the model's ability to recognize good responses, not just generate them. This is more reliable than post-update performance, which may be confounded by other factors.

$$\text{GV-Gap} = P(\text{verifier accepts}) - P(\text{generator succeeded})$$

(c) What is greedy decoding and why is self-improvement with greedy decoding impossible?

Greedy decoding selects the highest-probability token at each step, producing deterministic outputs. It yields a deterministic output and does not explore multiple candidate answers. With greedy decoding, there's effectively one output and thus no alternative candidate to compare.

Self-improvement requires generating multiple candidates (or diverse outputs) so that some may succeed in verification that initial output didn't. Thus self-improvement fails under greedy decoding since self-improvement is impossible without variation in outputs.

(d) Explain why the relative GV-Gap scales monotonically with pre-training FLOPs for certain verification methods but not others.

The author states as model size/pre-training compute (FLOPs) increases, the gap between what the generator can do and what the verifier can detect shrinks or changes in predictable ways—but only for verification methods that are aligned with model capabilities. For verification methods like CoT-Score, GV-Gap increases with pre-training FLOPs, showing better verification. This scaling doesn't hold for all methods. When the verifier is too weak or mis-aligned, the GV-Gap does not scale smoothly with compute because the bottleneck becomes verification rather than generation. In contrast, when the verification method is sufficiently powerful and grows with model scale, then as FLOPs increase, generation improves and verification improves, so the GV-Gap behaves monotonically.

(e) Why do most models fail to self-improve on Sudoku puzzles despite the exponential computational complexity separation between generation and verification?

The model's generation output space is enormous and the model cannot reliably explore it enough. The model can't reliably check full correctness of a board). This is because verification still requires reasoning and planning, which small models lack of structural search algorithm or reasoning to find or verify valid Sudoku solutions.

f) Propose a task domain where you would expect self-improvement to improve performance and explain why.

Generation: the model proposes a corrected sentence.
Verification: checking grammar/spelling rules is easier
Math word problems are ideal for self-improvement. Models can generate diverse solutions and use reasoning-based verification to filter correct ones. the model can generate multiple candidate corrections, learn, improve over time