International Computer Science Competition 2026

Qualification Round Submission

Name: Nabil Bin Billal
Email: nabilbinbillal@gmail.com

Country: Bangladesh

Problem A: A History of Computing

---

1. (1936): Alan Turing publishes "On Computable Numbers," introducing the theoretical Turing Machine.

2. (1947): Bardeen, Brattain, and Shockley invent the transistor at Bell Labs, replacing vacuum tubes.

3. (1958): Jack Kilby builds the first integrated circuit at Texas Instruments.

4. (1969): ARPANET connects four university nodes, forming the first packet-switched network.

5. (1981): IBM releases the Personal Computer with Intel 8088 and

MS-DOS, founding the PC industry.

6. (1991): Tim Berners-Lee opens the World Wide Web to the public beyond CERN.

7. (2000): Intel Pentium 4 launches with 42 million transistors, crossing the 1 GHz barrier.

8. (2007): Apple releases the first iPhone, beginning the smartphone era.

9. (2012): AlexNet wins ImageNet on NVIDIA GPUs, starting the modern deep learning era.

10. (2022): OpenAI releases ChatGPT, bringing large language models to public use.

Problem B: Pixel Art

```python
def generate_shape(n, shape):
    if shape == "checkerboard":
        return [[(i + j) % 2 for j in range(n)] for i in range(n)]

    c = n // 2
    return [[1 if abs(i - c) + abs(j - c) <= c else 0 for j in range(n)] for i in range(n)]
```

Problem C: Moore's Law

(a) Predicted Transistor Count in 2026

Given:

- $T_0 = 2{,}300$
- $t = 2026 - 1971 = 55$ years
- Formula: $T(t) = T_0 \times 2^{(t/2)}$

Calculation:

```
T(55) = 2,300 × 2^(55/2)
      = 2,300 × 2^27.5
```

Breaking down 2^27.5:

```
2^27   = 134,217,728
2^0.5  = 1.41421...
2^27.5 = 134,217,728 × 1.41421 ≈ 189,812,531
```

Final:

```
T(55) = 2,300 × 189,812,531
      ≈ 436,568,000,000
      ≈ 4.37 × 10^11
```

Predicted count: ~ 436.6 billion transistors

(b) Comparison with Apple M5

Apple M5 estimated transistor count: ~30 billion transistors

	Value
Moore's Law prediction (2026)	~ 436.6 billion
Apple M5 actual estimate	~ 30 billion
Ratio (predicted / actual)	~ 14.6x

Moore's Law predicts roughly **14.6 times more** transistors than the M5 contains. The 2-year doubling period held reasonably well from the 1970s through the early 2000s. After around 2010, physical limits in lithography, heat dissipation, and transistor leakage caused the pace to slow noticeably. Growth remained exponential, but the doubling period stretched beyond 2 years.

Moore's Law significantly overestimates modern transistor counts (by more than an order of magnitude). While the exponential growth trend still holds qualitatively, the strict 2 year doubling period has slowed in recent decades due to physical and manufacturing limits.

(c) Effective Doubling Period

Starting from the general form with doubling period d:

$$T = T_0 \times 2^{(t/d)}$$

Solving for d:

$$T / T_0 = 2^{(t/d)}$$

$$\log_2(T / T_0) = t / d$$

$$d = t / \log_2(T / T_0)$$

Substituting values ($T = 30 \times 10^9$, $T_0 = 2{,}300$, $t = 55$):

$$T / T_0 = 30{,}000{,}000{,}000 / 2{,}300 \approx 13{,}043{,}478$$

$$\log_2(13{,}043{,}478) = \log(13{,}043{,}478) / \log(2) = 7.1154 / 0.30103 \approx 23.63$$

$$d = 55 / 23.63 \approx 2.33 \text{ years}$$

Effective doubling period: ~2.33 years

Note: the problem states the formula as $t = 2 \times \log_2(T / T_0)$, which assumes $d = 2$ fixed and solves for the elapsed time. Here we generalise to solve for d directly from the real data. Both are equivalent when rearranged correctly.

Final Answers

(a) T(55) = 2,300 × 2^27.5 = **~436.6 billion transistors**

(b) Apple M5 contains approximately ~30 billion transistors. Moore's Law overestimates by a factor of **~14.6x**. The exponential trend holds but the actual doubling period is longer than 2 years.

(c) Effective doubling period = **~2.33 years**

Final Summary

Item		Value
Moore's Law predicted (2026)	~	436.6 billion transistors
Apple M5 real estimate (2026)	~	30 billion transistors
Effective doubling period	~	2.33 years

The real doubling period of ~2.33 years is close to but slower than Moore's original 2 year figure, which explains why the prediction overshoots by roughly 14-15x over 55 years.

Problem D: The Simultaneous Strike

(a) Explanation

The bug:

The KO check at lines 3-4 runs before every individual event, not after finishing all events on the same frame. This means the loop can break mid-frame, skipping attacks that arrived on the same frame but were processed later in the sorted order.

Why this causes the problem:

After a stable sort by frame, both events (1, 512, 20) and (2, 512, 20) sit adjacent in the list. The loop processes event 1 first:

```
Start of frame 512:
  player1_hp = 10, player2_hp = 15

Iteration 1: event = (1, 512, 20)
  Check: player1_hp <= 0 OR player2_hp <= 0 → 10 <= 0 OR 15 <= 0 → False,
continue
  Player 1 attacks player 2: player2_hp = 15 - 20 = -5

Iteration 2: event = (2, 512, 20)
  Check: player1_hp <= 0 OR player2_hp <= 0 → 10 <= 0 OR -5 <= 0 → TRUE →
BREAK
```

Player 2's attack on frame 512 never executes. The function returns hp1 = 10, hp2 = -5, declaring Player 1 the winner. But both attacks happened on the same frame, so both should land before any KO check runs.

Correct behavior:

All events with the same frame number must be collected and applied together. Only after the entire frame is processed should the code check whether either player has reached 0 or below. In this example, the correct result is:

```
player2_hp = 15 - 20 = -5  (clamped to 0)
player1_hp = 10 - 20 = -10 (clamped to 0)
Result: [0, 0] → double KO
```

(b) Python Solution

```
def processGame(events, H):
    hp1 = H
    hp2 = H

    # Sort events by frame number (stable sort preserves relative order)
    events_sorted = sorted(events, key=lambda e: e[1])

    # Group events by frame and process frame by frame
    i = 0
    while i < len(events_sorted):

        # Collect all events belonging to this frame
        current_frame = events_sorted[i][1]
        frame_events = []
        while i < len(events_sorted) and events_sorted[i][1] ==
current_frame:
            frame_events.append(events_sorted[i])
            i += 1

        # Apply every event in this frame before any KO check
        for player, frame, attack in frame_events:
            if player == 1:
                hp2 -= attack
            else:
                hp1 -= attack

        # Clamp HP to 0 after the full frame is processed
        hp1 = max(0, hp1)
        hp2 = max(0, hp2)

        # Now check for KO, only after the entire frame is done
        if hp1 == 0 or hp2 == 0:
            break

    return [hp1, hp2]
```

Verification against the example:

```
events = [(1, 512, 20), (2, 512, 20)], H = 10 (using player1_hp = 10 as H)

frame 512 events: [(1, 512, 20), (2, 512, 20)]

Apply event (1, 512, 20): player 1 attacks player 2 → hp2 -= 20
Apply event (2, 512, 20): player 2 attacks player 1 → hp1 -= 20

Both attacks land before the KO check runs.
Result: double KO → [0, 0]
```

This is the correct result under the game's simultaneous frame rule.

Complexity

Time Complexity: O(N log N) for sorting, O(N) for processing, so O(N log N) overall.

Space Complexity: O(N) for the sorted event list and frame grouping buffer.

Problem E: PageRank in the Age of AI Slop

(a) Explanation

Random surfer interpretation: Imagine a user browsing the web by randomly clicking links. At each page, with probability d they follow a random outgoing link. With probability (1-d) they "teleport" to a completely random page from the entire web. PageRank of a page equals the long-run fraction of time this surfer spends on it.

Role of d: d = 0.85 is the damping factor. It controls the balance between following links and teleporting. Without teleportation (d = 1), pages with no incoming links would receive zero rank and the surfer could get trapped in loops. The teleportation term (1-d)/N ensures every page receives a minimum baseline of rank.

Why the equation is recursive: The PageRank of page p_i depends on the PageRank of pages linking to it, which themselves depend on other pages' ranks. The formula has no closed-form solution from a single pass. Every page's rank is defined in terms of other pages' ranks simultaneously.

How it is computed in practice: Start by assigning PR(p) = 1/N to every page. Then repeatedly apply the PageRank formula to all pages until the values converge (the change between iterations falls below a small threshold). This is the power iteration method. It converges because the update rule is a contraction on the space of rank vectors.

(b) Derivation of S_A

Setup:

Total PageRank across all pages sums to 1:

```
S_H + S_A = 1
```

There are h human pages and a = αN AI pages, with h = (1-α)N.

Summing PageRank over all AI pages:

Sum the PageRank formula over every page p_i in group A:

```
S_A = sum_{p_i in A} PR(p_i)
    = sum_{p_i in A} [ (1-d)/N + d * sum_{p_j in M(p_i)} PR(p_j)/L(p_j) ]
    = a*(1-d)/N + d * sum_{p_i in A} sum_{p_j in M(p_i)} PR(p_j)/L(p_j)
```

The first term:

```
a*(1-d)/N = αN*(1-d)/N = α(1-d)
```

Evaluating the link-flow term:

The double sum counts, for each AI page p_i, the PageRank contribution flowing into it from every page p_j that links to p_i. We can reverse the order of summation: for each page p_j, count how much of its rank flows to AI pages.

For a page p_j with out-degree L(p_j), it contributes PR(p_j)/L(p_j) to each page it links to. The total flow from p_j to the AI group equals:

```
PR(p_j) * (number of p_j's links pointing to AI pages) / L(p_j)
```

Now split by group:

Human pages (p_j in H): Each human page links to k pages chosen uniformly at random from all N pages. The expected number of those k links pointing to AI pages is k * (a/N) = k*α. So the fraction of a human page's rank flowing to AI is:

```
k*α / k = α
```

Flow from all human pages to AI:

```
sum_{p_j in H} PR(p_j) * α = α * S_H
```

AI pages (p_j in A): AI pages link only to other AI pages, so all of their outgoing rank stays within A. The fraction flowing to AI is 1.

Flow from all AI pages to AI:

```
sum_{p_j in A} PR(p_j) * 1 = S_A
```

Combining:

```
S_A = α(1-d) + d*(α*S_H + S_A)
```

Substitute S_H = 1 - S_A:

```
S_A = α(1-d) + d*(α*(1 - S_A) + S_A)
S_A = α(1-d) + d*α - d*α*S_A + d*S_A
S_A = α(1-d) + dα + d*S_A*(1 - α)
S_A = α(1-d+d) + d*S_A*(1-α)
S_A = α + d*(1-α)*S_A
```

Collect S_A terms on the left:

```
S_A - d*(1-α)*S_A = α
S_A * [1 - d*(1-α)] = α
```

Closed-form result:

```
S_A = α / [1 - d*(1-α)]
```

Verification:

When α = 0 (no AI pages): S_A = 0/(1-d) = 0. Correct.

When α = 1 (all pages are AI): S_A = 1/(1-d*0) = 1/1 = 1. Correct.

When α = 0.5, d = 0.85:

```
S_A = 0.5 / [1 - 0.85*0.5] = 0.5 / [1 - 0.425] = 0.5 / 0.575 ≈ 0.870
```

Thus, if AI pages constitute 50% of all pages, they collectively receive approximately 87% of the total PageRank.

Total check: S_H = 1 - S_A. Since S_A is derived consistently from the system, S_H + S_A = 1 holds by construction.

(c) Tipping Fraction

Setting S_A = S_H:

Since S_H + S_A = 1, the condition S_A = S_H gives:

```
S_A = 0.5
```

Set the formula equal to 0.5:

```
α* / [1 - d*(1 - α*)] = 0.5
```

Solve for α*:

```
α* = 0.5 * [1 - d*(1 - α*)]
α* = 0.5 - 0.5*d + 0.5*d*α*
α* - 0.5*d*α* = 0.5 - 0.5*d
α* * (1 - 0.5*d) = 0.5*(1 - d)
α* = 0.5*(1-d) / (1 - 0.5*d)
α* = (1-d) / (2 - d)
```

Numerical result at d = 0.85:

```
α* = (1 - 0.85) / (2 - 0.85)
   = 0.15 / 1.15
   ≈ 0.1304
```

Tipping fraction: α ≈ 13.04%

Verification:

Plug α* = 0.15/1.15 back into S_A:

```
S_A = (0.15/1.15) / [1 - 0.85*(1 - 0.15/1.15)]
    = (0.15/1.15) / [1 - 0.85*(1.00/1.15)]
    = (0.15/1.15) / [1 - 0.85/1.15]
    = (0.15/1.15) / [(1.15 - 0.85)/1.15]
    = (0.15/1.15) / (0.30/1.15)
    = 0.15 / 0.30
    = 0.5
```

Confirmed. S_A = 0.5 at α* = (1-d)/(2-d).

Discussion:

At d = 0.85, once AI pages make up just ~13% of the web, they collectively hold as much total PageRank as the remaining 87% of human pages. This represents a strong amplification effect, and it happens for two structural reasons.

First, AI pages form a closed link cluster. All of their outgoing rank stays within the AI group. Human pages, by linking uniformly across all N pages, leak a fraction α of their rank to AI pages on every iteration. AI pages return none of it.

Second, the teleportation term distributes rank proportionally to the number of pages. As α grows, more of the base rank (1-d)/N is seeded directly into AI pages simply because there are more of them.

The scenario reveals a fundamental weakness of PageRank: it measures link structure, not content quality. A coordinated set of pages that interlink densely and never link out can accumulate disproportionate rank. In the original web this was hard to engineer at scale, but AI-generated content can produce millions of interlinked pages cheaply, making this attack vector practically relevant.

Final Answers

(a) PageRank models a random surfer who follows links with probability d and teleports with probability (1-d). The equation is recursive because each page's rank depends on the ranks of pages linking to it. In practice, power iteration from a uniform starting distribution converges to the fixed point.

(b)

```
S_A = α / [1 - d*(1-α)]
```

(c)

```
α* = (1-d) / (2-d)

At d = 0.85:  α* = 0.15/1.15 ≈ 0.1304  (approximately 13%)
```

When just 13% of pages are adversarial AI pages that interlink exclusively with each other, they collectively equal the total PageRank of all human pages. PageRank has no mechanism to distinguish link quality from link quantity, making it structurally vulnerable to coordinated inlinking at scale.