

## Appendix A Supplementary Examples of Findings

### A Examples of Findings 1: Some LLMs demonstrate highly professional mathematical question-design capabilities

LLM: gemini-2.5-pro-exp-03-25

#### Question

- 1 Let  $S_3(n)$  denote the sum of the digits of a positive integer  $n$  when
- ↪ written in base 3. Define a sequence  $a_n = (-1)^{S_3(n)}$  for  $n \geq 1$ .
  - ↪ Consider the Dirichlet series  $F(s) = \sum_{n=1}^{\infty} a_n / n^s$ ,
  - ↪ defined for complex numbers  $s$  where the series converges. Determine
  - ↪ if  $F(s)$  can be analytically continued to a region including  $s=0$ , and
  - ↪ if so, find the value  $F(0)$ .

#### Reference Answer

- 1 Step 1: Region of Convergence.
- 2 The sequence  $a_n = (-1)^{S_3(n)}$  consists of  $+1$  and  $-1$ . The Dirichlet
- ↪ series  $F(s) = \sum_{n=1}^{\infty} a_n / n^s$  converges absolutely
  - ↪ when  $\sum_{n=1}^{\infty} |a_n / n^s| = \sum_{n=1}^{\infty} 1 / n^{\text{Re}(s)}$  converges. This occurs for  $\text{Re}(s) > 1$ .
- 3
- 4 Step 2: Convergence for  $\text{Re}(s) > 0$ .
- 5 Let  $T_k = \sum_{n=0}^{3^k-1} a_n$ . Using the property  $S_3(3q+r) = S_3(q) +$
- ↪  $S_3(r)$  (for  $r=0,1,2$ ) and  $a_n = (-1)^{S_3(n)}$ , we have  $a_{3q+r} = a_q$
  - ↪  $\cdot a_r$  (with  $a_0=1$ ). This leads to  $T_k = T_{k-1} \cdot (a_0 + a_1 + a_2)$ .
  - ↪ Since  $a_0=1$ ,  $a_1=(-1)^1=-1$ ,  $a_2=(-1)^2=1$ , we have  $(a_0 + a_1 + a_2) =$
  - ↪  $1 - 1 + 1 = 1$ . As  $T_0 = a_0 = 1$ , we get  $T_k = 1$  for all  $k \geq 0$ . The
  - ↪ partial sums  $S(N) = \sum_{n=1}^N a_n$  can be shown to be bounded
  - ↪ based on this property ( $\sum_{n=1}^{3^k-1} a_n = T_k - a_0 = 0$ ).
  - ↪ Bounded partial sums imply that the Dirichlet series  $F(s)$  converges
  - ↪ for all  $s$  with  $\text{Re}(s) > 0$ .
- 6
- 7 Step 3: Analytic Continuation.
- 8 Split the sum  $F(s)$  based on  $n \bmod 3$ :
- 9  $F(s) = \sum_{m=1}^{\infty} a_{3m} / (3m)^s + \sum_{m=0}^{\infty} a_{3m+1} /$
- ↪  $(3m+1)^s + \sum_{m=0}^{\infty} a_{3m+2} / (3m+2)^s$ .
- 10 Using  $a_{3m}=a_m$ ,  $a_{3m+1}=-a_m$ ,  $a_{3m+2}=a_m$  (for  $m \geq 0$ , with  $a_0=1$ ):
- 11  $F(s) = \sum_{m=1}^{\infty} a_m / (3^s \cdot m^s) + (a_1/1^s + \sum_{m=1}^{\infty} a_m /$
- ↪  $(3m+1)^s) + (a_2/2^s + \sum_{m=1}^{\infty} a_m / (3m+2)^s)$
- 12  $F(s) = 3^{-s} \cdot F(s) + (-1)/1^s + 1/2^s + \sum_{m=1}^{\infty} a_m \cdot [$
- ↪  $1/(3m+2)^s - 1/(3m+1)^s]$
- 13 Rearranging gives:

$F(s) * (1 - 3^{(-s)}) = -1 + 2^{(-s)} + \sum_{m=1 \text{ to } \infty} a_m * [ (3m+2)^{(-s)} - (3m+1)^{(-s)} ]$   
 $\hookrightarrow (3m+1)^{(-s)} ]$   
 Let  $K(s) = -1 + 2^{(-s)} + \sum_{m=1 \text{ to } \infty} a_m * [ (3m+2)^{(-s)} - (3m+1)^{(-s)} ]$ . The series defining  $K(s)$  converges for  $\text{Re}(s) > 0$ , thus  $K(s)$  is analytic in this region. The equation  $F(s) = K(s) / (1 - 3^{(-s)})$  provides an analytic continuation of  $F(s)$  to the region  $\text{Re}(s) > 0$ , except for potential simple poles where  $1 - 3^{(-s)} = 0$ , i.e.,  $s = 2\pi k i / \ln(3)$  for non-zero integers  $k$ .

Step 4: Value at  $s=0$ .  
 We investigate the limit as  $s$  approaches 0. We evaluate  $K(0)$ :  
 $K(0) = -1 + 2^0 + \sum_{m=1 \text{ to } \infty} a_m * [ (3m+2)^0 - (3m+1)^0 ] = -1 + 1 + \sum_{m=1 \text{ to } \infty} a_m * (1 - 1) = 0$ .  
 Since the numerator  $K(s)$  and the denominator  $(1 - 3^{(-s)})$  are both 0 at  $s=0$ , we have a removable singularity. We can find  $F(0)$  using L'Hopital's rule:  
 $F(0) = \lim_{s \rightarrow 0} K(s) / (1 - 3^{(-s)}) = K'(0) / (d/ds (1 - 3^{(-s)}))|_{s=0}$ .  
 The derivative of the denominator is  $\ln(3) * 3^{(-s)}$ , which equals  $\ln(3)$  at  $s=0$ .  
 The derivative of the numerator is  $K'(s) = d/ds [ -1 + 2^{(-s)} + \sum_{m=1 \text{ to } \infty} a_m * ( (3m+2)^{(-s)} - (3m+1)^{(-s)} ) ]$   
 $K'(s) = -\ln(2) * 2^{(-s)} + \sum_{m=1 \text{ to } \infty} a_m * [ -\ln(3m+2) * (3m+2)^{(-s)} + \ln(3m+1) * (3m+1)^{(-s)} ]$   
 Evaluating at  $s=0$ :  
 $K'(0) = -\ln(2) * 2^0 + \sum_{m=1 \text{ to } \infty} a_m * [ -\ln(3m+2) * 1 + \ln(3m+1) * 1 ]$   
 $K'(0) = -\ln(2) + \sum_{m=1 \text{ to } \infty} a_m * \ln( (3m+1)/(3m+2) )$   
 This infinite series converges (conditionally). Therefore,  $F(s)$  can be analytically continued to  $s=0$ , and its value is:  
 $F(0) = K'(0) / \ln(3) = [ -\ln(2) + \sum_{m=1 \text{ to } \infty} (-1)^{S_3(m)} * \ln( (3m+1)/(3m+2) ) ] / \ln(3)$ .

Final Answer:  $F(s)$  can be analytically continued to a region including  $s=0$ .  
 $\hookrightarrow$  The value at  $s=0$  is  $F(0) = [ -\ln(2) + \sum_{m=1 \text{ to } \infty} (-1)^{S_3(m)} * \ln( (3m+1)/(3m+2) ) ] / \ln(3)$ .

## Principle:

The core principle is the deep connection between the properties of a sequence defined by discrete arithmetic rules (e.g.,  $a_n$  depends on the sum of digits of  $n$  in base 3) and the analytic properties (such as convergence and analytic continuation) of the complex Dirichlet series  $F(s) = \sum a_n/n^s$  constructed from that sequence. This reflects a broader pattern seen in analytic number theory, including connections between prime distribution and the Riemann zeta function.

## B Examples of Findings 2: “Memorization-based answering” behavior does indeed exist in LLMs

LLM: deepseek-r1

### Answer

```
1 The function  $F(s)$  can be analytically continued to include  $s=0$ . The value
  ↳  $F(0)$  is determined by representing  $F(s)$  as an infinite product over
  ↳ base-3 digit positions. Each term in the product corresponds to the
  ↳ sum over digits  $(0,1,2)$  in each position, contributing  $(-1)^d /$ 
  ↳  $(d \cdot 3^k)^s$ . At  $s=0$ , each factor simplifies to  $1 + (-1) + 1 = 1$ . The
  ↳ infinite product of 1s converges to 1. Thus,  $F(0) = 1$ .
```

## C Examples of Findings 4: The consistency between LLMs’ question-setting and question-solving abilities

LLM: gpt-4.1

### Question

```
1 Cube Garden
2
3 In the magical land of Polycubia, gardeners grow structures using magical
  ↳ cubes. A Garden is a rectangular prism (cuboid) formed from  $1 \times 1 \times 1$ 
  ↳ magical cubes stacked on a regular 3D grid. Each of the three
  ↳ dimensions (length, width, height) must be at least 1.
4
5 A Garden Keeper’s task is to maintain the gardens such that every cuboid in
  ↳ Polycubia is well-balanced.
6
7 Definition: A cuboid of dimensions  $(L, W, H)$  is well-balanced if the number
  ↳ of its positive divisors is strictly greater than the combined number
  ↳ of positive divisors of  $L$ ,  $W$ , and  $H$  taken separately (counted without
  ↳ multiplicity, i.e. no double-counting even if  $L=W$ , etc).
8
9 Given an integer  $K$ , find the number of ordered triple  $(L, W, H)$  of positive
  ↳ integers such that  $1 \leq L, W, H \leq K$  and the cuboid formed by  $(L, W, H)$ 
  ↳ is well-balanced.
10
11 Input
12 The first line contains the integer  $K$  ( $1 \leq K \leq 2500$ ).
13
14 Output
```

```

15 Output a single integer: the number of ordered triples (L, W, H) that form
    ↪ a well-balanced cuboid.
16
17 Constraints
18 1 <= K <= 2500
19 All calculations fit within 64-bit signed integers.

```

## Reference Answer

```

1 # Key Concepts:
2 # The number of positive divisors of N is found by counting the exponent
    ↪ increments
3 # in its prime factorization. For a cuboid with sides (L, W, H), the cuboid
    ↪ as a
4 # whole has L * W * H cubes and its number of divisors is d(L * W * H).
5 # Collect the number of divisors for all numbers up to K.
6 # Iterate for all possible (L, W, H), compute d(L * W * H),
7 # and compare it to d(L) + d(W) + d(H).
8
9 # Algorithm Steps:
10 # 1. Precompute d(n), the number of divisors for all 1 <= n <= K^3,
11 #    using divisor count by minimally factorizing up to K^3.
12 #    But for memory efficiency (since K^3 can be ~1.6e10),
13 #    precompute up to K, and perform prime factorizations and divisor counts
14 #    on the fly for L*W*H, caching if needed.
15 # 2. For each (L, W, H), calculate L*W*H and get d(L*W*H).
16 # 3. For all L, W, H in [1, K], if d(L*W*H) > d(L) + d(W) + d(H), increment
    ↪ answer.
17 #    - For performance, precompute d(n) for n = 1..K,
18 #      use fast divisor-counting by merging prime exponents for products
19 #      (since L*W*H can be factorized by adding exponents of L, W, H's
    ↪ prime factors).
20
21 # Implementation (Python):
22 def divisor_count(n, spf):
23     cnt = 1
24     while n > 1:
25         p = spf[n]
26         c = 0
27         while n % p == 0:
28             n //= p
29             c += 1
30         cnt *= (c + 1)
31     return cnt
32

```

```

33 K = int(input())
34
35 # Precompute smallest prime factor (spf) up to K
36 MAX = K + 1
37 spf = [0] * MAX
38 for i in range(2, MAX):
39     if spf[i] == 0:
40         for j in range(i, MAX, i):
41             if spf[j] == 0:
42                 spf[j] = i
43 for i in range(2, MAX):
44     if spf[i] == 0:
45         spf[i] = i
46
47 # Precompute divisors count and factorizations for 1..K
48 from collections import defaultdict
49
50 div_count = [1] * MAX
51 divisors_fact = [{}] for _ in range(MAX)]
52 div_count[0] = 0 # d(0) is undefined, set to 0
53
54 for i in range(2, MAX):
55     x = i
56     d = defaultdict(int)
57     while x > 1:
58         d[spf[x]] += 1
59         x //= spf[x]
60     divisors_fact[i] = d.copy()
61     cnt = 1
62     for e in d.values():
63         cnt *= (e + 1)
64     div_count[i] = cnt
65
66 result = 0
67
68 # Iterate through all (L, W, H)
69 for L in range(1, K + 1):
70     facts_L = divisors_fact[L]
71     for W in range(1, K + 1):
72         facts_W = divisors_fact[W]
73         for H in range(1, K + 1):
74             facts_H = divisors_fact[H]
75
76             # Merge facts_L, facts_W, facts_H
77             merged = defaultdict(int)
78             for p in facts_L:

```

```

79         merged[p] += facts_L[p]
80     for p in facts_W:
81         merged[p] += facts_W[p]
82     for p in facts_H:
83         merged[p] += facts_H[p]
84
85     # Compute d(L * W * H)
86     cnt = 1
87     for e in merged.values():
88         cnt *= (e + 1)
89
90     # Check condition
91     if cnt > div_count[L] + div_count[W] + div_count[H]:
92         result += 1
93
94 print(result)
95
96 # Explanation:
97 # - Precompute smallest-prime-factor up to K for fast factorization.
98 # - Precompute divisor counts and prime exponent factorizations for all
99   ↪ 1..K.
100 # - For each (L, W, H), merge prime exponents to construct L*W*H's prime
101   ↪ decomposition;
102 #   divisor count is product of (exponent + 1) for each prime.
103 # - Check the well-balance condition as defined.
104
105 # Time Complexity:  $O(K^3 * \log K)$ 
106 #   - Triple loop over (L, W, H):  $O(K^3)$ 
107 #   - Each merging operation:  $O(\log K)$ 
108 #     (since typical number of prime factors is  $\leq \log K$ )
109
110 # Space Complexity:  $O(K * \log K)$  for storage of factorizations and divisor
111   ↪ count

```

## Scoring Criteria

- 1 Accuracy and Correctness (40):
  - 2 - 40: Produces correct output under all valid inputs and expected edge
    - ↪ cases.
  - 3 - 30: Correct for majority, minor ( $\leq 2$ ) edge cases fail.
  - 4 - 20: Only simple cases correct, many edge cases fail.
  - 5 - 0: Incorrect output for most cases.
- 6 Execution Speed (20):
  - 7 - 20: Finishes in  $< 2s$  for  $K=2500$ .

```

8   - 15: Finishes in <5s for K=2500.
9   - 7: Only handles K up to 500 efficiently.
10  - 0: Too slow on K=100 and above.
11  Readability & Code Quality (15):
12  - 15: Clear structure, good variable naming, modular functions, properly
    ↪ commented.
13  - 10: Reasonable but could be improved on modularity or clarity.
14  - 5: Messy, some unclear variable names or missing comments.
15  - 0: Not understandable without serious effort.
16  Memory Efficiency (10):
17  - 10: Memory usage well within 500 MB for all K.
18  - 7: Slightly wasteful, may use up to 1GB.
19  - 2: Unsafe; crashes or uses >1GB unnecessarily, possible leaks.
20  - 0: OOM on sample inputs.
21  Code Modularity and Organization (15):
22  - 15: Divided into logical functions where needed, DRY, scalable
    ↪ structure.
23  - 10: Mostly in main, some structure but could improve.
24  - 5: Giant monolithic function; hard to adapt.
25  - 0: No discernible structure

```

LLM: gemini-2.5-pro-exp-03-25

### Answer

```

1  #include <iostream>
2
3  int main() {
4      // Use faster I/O
5      std::ios_base::sync_with_stdio(false);
6      std::cin.tie(NULL);
7
8      long long k;
9      std::cin >> k;
10
11     // Calculate K^3 using long long type to prevent potential integer
    ↪ overflow.
12     // K is up to 2500, K^3 can be up to 15,625,000,000, which fits in a
    ↪ 64-bit signed integer.
13     long long k_cubed = k * k * k;
14
15     // The problem defines a cuboid (L, W, H) as well-balanced if
16     // d(L*W*H) > |Divisors(L) U Divisors(W) U Divisors(H)|,
17     // where d(n) is the number of positive divisors of n.
18     // We need to find the number of ordered triples (L, W, H) such that 1

```

```

19  ↪ <= L,W,H <= K
    // and the cuboid is well-balanced.
20
21  // Let's analyze the condition.
22  // Let S = Divisors(L) U Divisors(W) U Divisors(H).
23  // Let P = Divisors(L*W*H).
24  // The condition is |P| > |S|.
25
26  // Consider cases based on the number of 1s in (L, W, H).
27
28  // Case 1: (1, 1, 1)
29  // L=1, W=1, H=1. d(L*W*H) = d(1) = 1.
30  // S = Div(1) U Div(1) U Div(1) = {1}. |S| = 1.
31  // Condition: 1 > 1. False. (1,1,1) is not well-balanced.
32
33  // Case 2: Exactly two 1s. E.g., (1, 1, H) where H > 1.
34  // d(L*W*H) = d(H).
35  // S = Div(1) U Div(1) U Div(H) = Div(H). |S| = d(H).
36  // Condition: d(H) > d(H). False. Triples with exactly two 1s are not
    ↪ well-balanced.
37  // There are 3 positions for the non-1 value (H), and K-1 choices for H
    ↪ > 1.
38  // Total non-well-balanced triples of this type: 3 * (K - 1).
39
40  // Case 3: Exactly one 1. E.g., (1, W, H) where W > 1, H > 1.
41  // d(L*W*H) = d(W*H).
42  // S = Div(1) U Div(W) U Div(H) = Div(W) U Div(H).
43  // |S| = |Div(W)| + |Div(H)| - |Div(W) intersect Div(H)| = d(W) + d(H)
    ↪ - d(gcd(W, H)).
44  // Condition: d(W*H) > d(W) + d(H) - d(gcd(W, H)).
45  // It can be shown that this condition holds true for all W > 1, H > 1.
46  // One way to see this: d(W*H) = |Div(W*H)|. Div(W) U Div(H) is a
    ↪ subset of Div(W*H).
47  // The element W*H is in Div(W*H). However, W*H is not in Div(W) (since
    ↪ H > 1)
48  // and not in Div(H) (since W > 1).
49  // Thus, W*H is in Div(W*H) but not in Div(W) U Div(H).
50  // This implies |Div(W*H)| >= |Div(W) U Div(H)| + 1, so the condition
    ↪ holds.
51  // Therefore, all triples with exactly one 1 are well-balanced.
52
53  // Case 4: No 1s. L > 1, W > 1, H > 1.
54  // Condition: d(L*W*H) > |Div(L) U Div(W) U Div(H)|.
55  // Similar to Case 3, the element L*W*H is in Div(L*W*H).
56  // L*W*H is not in Div(L) (since W*H > 1),
57  // not in Div(W) (since L*H > 1),

```



```

58 // not in Div(H) (since  $L*W > 1$ ).
59 // So,  $L*W*H$  is in  $\text{Div}(L*W*H)$  but not in  $\text{Div}(L) \cup \text{Div}(W) \cup \text{Div}(H)$ .
60 // This implies  $d(L*W*H) = |\text{Div}(L*W*H)| \geq |\text{Div}(L) \cup \text{Div}(W) \cup \text{Div}(H)| +$ 
61  $\rightarrow 1$ .
62 // Thus, the condition holds, and all triples with  $L, W, H > 1$  are
63  $\rightarrow$  well-balanced.
64
65 // Summary:
66 // A triple  $(L, W, H)$  is well-balanced if and only if it does not
67  $\rightarrow$  contain two or more 1s.
68 // The non-well-balanced triples are those with at least two 1s.
69 // These are  $(1,1,1)$  [1 triple] and permutations of  $(1,1,n)$  where  $n > 1$ 
70  $\rightarrow$   $[3*(K-1) \text{ triples}]$ .
71 // Total number of non-well-balanced triples =  $1 + 3 * (K - 1) = 1 + 3K$ 
72  $\rightarrow - 3 = 3K - 2$ .
73
74 // Total number of triples =  $K * K * K = K^3$ .
75 // Number of well-balanced triples = Total triples - Non-well-balanced
76  $\rightarrow$  triples
77 //
78 //  $= K^3 - (3K - 2)$ 
79 //  $= K^3 - 3K + 2$ .
80
81 long long result = k_cubed - 3 * k + 2;
82
83 // Output the result
84 std::cout << result << std::endl;
85
86 return 0;
87 }

```